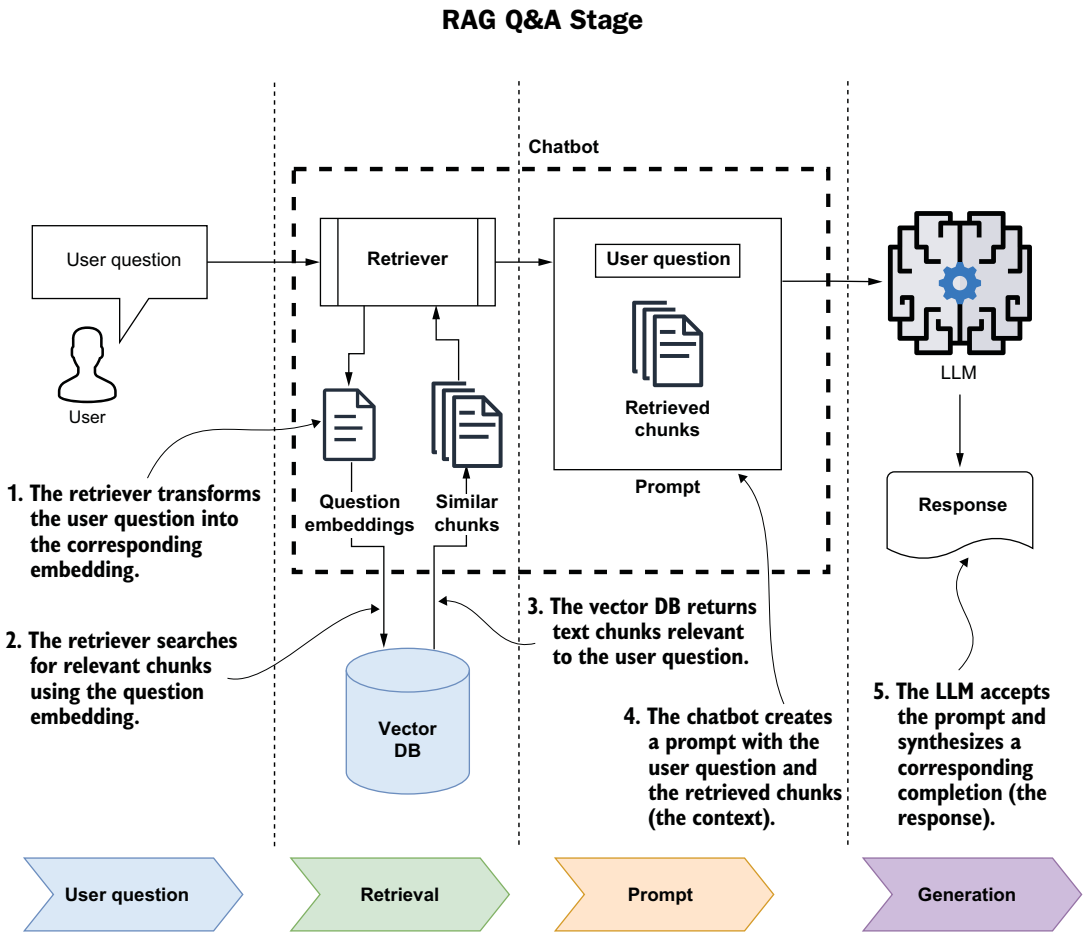
AI Agents and Applications

With LangChain, LangGraph and MCP

Roberto Infante

MANNING



Retrieval-Augmented Generation (RAG) Q&A stage: retrieval and generation

AI Agents and Applications

WITH LANGCHAIN, LANGGRAPH AND MCP

ROBERTO INFANTE

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor:	Dustin Archibald
Technical editors:	Keerthivasan Santhanakrishnan and Antowan Malik Batts
Review editor:	Kishor Rit
Production editor:	Kathy Rossland
Copy editor:	Julie McNamee
Proofreader:	Melody Dolab
Technical proofreader:	Andrew R. Freed
Typesetter and cover designer:	Marija Tudor

ISBN 9781633436541
Printed in the United States of America

To my mother and father

brief contents

PART 1 GETTING STARTED WITH LLMS 1

1 ■ Introduction to AI agents and applications 3
2 ■ Executing prompts programmatically 27

PART 2 SUMMARIZATION 53

3 ■ Summarizing text using LangChain 55
4 ■ Building a research summarization engine 69
5 ■ Agentic workflows with LangGraph 103

PART 3 Q&A CHATBOTS 119

6 ■ RAG fundamentals with ChromaDB 121
7 ■ Q&A chatbots with LangChain and LangSmith 143

PART 4 ADVANCED RAG 171

8 ■ Advanced indexing 173
9 ■ Question transformations 205
10 ■ Query generation, routing, and retrieval postprocessing 228

PART 5 AI AGENTS 265

11 ■ Building tool-based agents with LangGraph 267
12 ■ Multi-agent systems 293

13 ■ Building and consuming MCP servers 308
14 ■ Productionizing AI agents: Memory, guardrails, and beyond 327

appendix A Trying out LangChain 351
appendix B Setting up a Jupyter Notebook environment 357
appendix C Choosing an LLM 360
appendix D Installing SQLite on Windows 375
appendix E Open source LLMs 377

contents

preface xvi
acknowledgments xviii
about this book xx
about the author xxv
about the cover illustration xxvi

PART 1 GETTING STARTED WITH LLMS 1

1 Introduction to AI agents and applications 3

1.1 Building LLM-based applications and agents 4
LLM-based applications: Summarization and Q&A engines 4 ▪ LLM-based chatbots 7 ▪ AI agents 8

1.2 Introducing LangChain 11
LangChain architecture 12 ▪ LangChain's core object model 15

1.3 Typical LLM use cases 19

1.4 How to adapt an LLM to your needs 20
Prompt engineering 20 ▪ RAG 20 ▪ Fine-tuning 22

1.5 Which LLMs to choose 23

1.6 What you'll learn from this book 24

2 *Executing prompts programmatically 27*

2.1 Running prompts programmatically 28

Setting up the environment for this chapter 28 ▪ Minimal prompt execution 30

2.2 Running prompts with LangChain 31

2.3 Prompt templates 32

Implementing a prompt template with a Python function 32
Using LangChain's PromptTemplate 33

2.4 Prompt types 34

Text classification 34 ▪ Sentiment analysis 35 ▪ Text summarization 36 ▪ Composing text 36 ▪ Question answering 38 ▪ Reasoning 39

2.5 Reasoning in detail 40

One-shot learning 40 ▪ Two-shot learning 41 ▪ Providing steps 41 ▪ Few-shot learning 42 ▪ Implementing few-shot learning with LangChain 44 ▪ Chain of Thought 46

2.6 Prompt structure 48

PART 2 SUMMARIZATION 53

3 *Summarizing text using LangChain 55*

3.1 Summarizing a document bigger than the context window 56

Chunking the text into Document objects 57 ▪ Split 58
Map 59 ▪ Reduce 60 ▪ MapReduce combined chain 60
MapReduce execution 61

3.2 Summarizing across documents 61

Creating a list of Document objects 63 ▪ Wikipedia content 63
File-based content 64 ▪ Creating the Document list 65
Progressively refining the final summary 65

3.3 Summarization flowchart 67

4 *Building a research summarization engine 69*

4.1 Overview of a research summarization engine 70

4.2 Setting up the project 71

4.3 Implementing the core functionality 73

Implementing web searching 73 ▪ Implementing web scraping 74 ▪ Instantiating the LLM client 75
JSON to Python object converter 76

4.4 Enhancing the architecture with query rewriting 76
4.5 Prompt engineering 78
Crafting web search prompts 78 ▪ Crafting summarization prompts 81 ▪ Research report prompt 81
4.6 Initial implementation 82
Importing functions and prompt templates 82 ▪ Setting constants and input variables 83 ▪ Instantiating the LLM client 83 ▪ Generating the web searches and collecting the results 83 ▪ Scraping the web results 85 ▪ Summarizing the web results 86 ▪ Generating the research report 87
4.7 Reimplementing the research summary engine in LCEL 89
Assistant Instructions chain 91 ▪ Web Searches chain 93 Search and Summarization chain 94 ▪ Web Research chain 99

5 *Agentic workflows with LangGraph 103*
5.1 Understanding agentic workflows and agents 104
Workflows 105 ▪ Agents 106 ▪ When to use agent-based architectures 106 ▪ Agent development frameworks 106
5.2 LangGraph basics 107
5.3 Moving from LangChain chains to LangGraph 107
5.4 LangGraph core components 108
StateGraph structure 109 ▪ State management and typing 109 Node functions and edge definitions 110 ▪ Entry points and end conditions 110
5.5 Turning the web research assistant into an AI agent 110
Original LangChain implementation overview 111 ▪ Identifying components for conversion 112 ▪ Step-by-step transformation process 112 ▪ Code comparison and benefits realized 116

PART 3 Q&A CHATBOTS..... 119

6 *RAG fundamentals with ChromaDB 121*
6.1 Semantic search 122
A basic Q&A chatbot over a single document 122 ▪ A more complex Q&A chatbot over a knowledge base 126 ▪ The RAG design pattern 127
6.2 Vector stores 130
What's a vector store? 130 ▪ How do vector stores work? 131 Vector libraries vs. vector databases 131 ▪ Most popular vector

stores 132 ▪ Storing text and performing a semantic search using Chroma 133

6.3 Implementing RAG from scratch 136

Retrieving content from the vector database 137 ▪ Invoking the LLM 137 ▪ Building the chatbot 139 ▪ Recap of RAG terminology 140

7 *Q&A chatbots with LangChain and LangSmith 143*

7.1 LangChain object model for Q&A chatbots 144

Content ingestion (indexing) stage 144 ▪ Q&A (retrieval and generation) stage 146

7.2 Vector store content ingestion 148

Splitting and storing the documents 149 ▪ Removing duplication 150 ▪ Ingesting multiple documents from a folder 151

7.3 Q&A across stored documents 152

Querying the vector store directly 153 ▪ Asking a question through a LangChain chain 153 ▪ Completing the RAG chain setup 154 ▪ Follow-up question 156

7.4 Chatbot memory of message history 157

Amending the prompt 158 ▪ Updating the chat message history 159 ▪ Feeding the chat history to the RAG chain 160 Putting everything together 160

7.5 Tracing execution with LangSmith 163

PART 4 ADVANCED RAG 171

8 *Advanced indexing 173*

8.1 Improving RAG accuracy 174

Content ingestion stage 174 ▪ Question-answering stage 175

8.2 Advanced document indexing 177

8.3 Splitting strategy 177

Splitting strategies 178 ▪ Factors to consider 179 ▪ Choosing the right strategy 179 ▪ Splitting by HTML header 179

8.4 Embedding strategy 183

Embedding child chunks with ParentDocumentRetriever 184 Embedding child chunks with MultiVectorRetriever 187

Embedding document summaries 189 ▪ Embedding hypothetical questions 193

8.5 Granular chunk expansion 197

8.6 Semi-structured content 201

8.7 Multimodal RAG 202

9 *Question transformations 205*

9.1 Rewrite-Retrieve-Read 206

Retrieving content using the original user question 209 Setting up the query rewriter chain 210 ▪ Retrieving content with the rewritten query 211 ▪ Combining everything into a single RAG chain 212

9.2 Generating multiple queries 213

Setting up the chain for generating multiple queries 215 Setting up a custom multi-query retriever 216 ▪ Using a standard MultiQueryRetriever instance 218

9.3 Step-back question 218

Setting up the chain to generate a step-back question 220 Incorporating step-back question generation into the RAG chain 220

9.4 Hypothetical Document Embeddings (HyDE) 222

Generating a hypothetical document for the user question 223 Integrating the HyDE chain into the RAG chain 223

9.5 Single-step and multi-step decomposition 224

10 *Query generation, routing, and retrieval postprocessing 228*

10.1 Content database query generation 229

10.2 Self-querying (metadata query enrichment) 230

Ingestion: Metadata enrichment 232 ▪ Q&A on a metadata-enriched collection 234

10.3 Generating a structured SQL query 239

Installing SQLite 240 ▪ Setting up and connecting to the database 240 ▪ Generating SQL queries from natural language 242 ▪ Executing the SQL query 244

10.4 Generating a semantic SQL query 244

Standard SQL query 245 ▪ Semantic SQL query 246 Creating the embeddings 246 ▪ Performing a semantic SQL search 247 ▪ Automating semantic SQL search 247 Benefits of a semantic SQL search 247

10.5 Generating queries for a graph database 248
10.6 Chain routing 251
Setting up data retrievers 252 ▪ Setting up the query router 252
Integrating the chain router into a full RAG chain 254
10.7 Retrieval postprocessing 256
Similarity postprocessors 257 ▪ Keyword postprocessors 257
Time weighting 257 ▪ RAG fusion (Reciprocal Rank Fusion) 258

PART 5 AI AGENTS 265

11 Building tool-based agents with LangGraph 267
11.1 Starting simple: Building a single-tool travel info agent 268
Project setup 268 ▪ Loading environment variables 269
Preparing the travel information vector store 269
11.2 Enabling agents to call tools 271
From function calling to tool calling 271 ▪ How tool calling works with LLMs 273 ▪ Registering tools with the LLM 273
Agent state: Tracking the conversation 275 ▪ Executing tool calls 275 ▪ The LLM node: Coordinating reasoning and action 277
11.3 Assembling the agent graph 277
11.4 Understanding the agent graph structure 278
11.5 Running the agent chatbot: The Read-Eval-Print Loop 279
11.6 Executing a request 279
Step-by-step debugging 280
11.7 Expanding your agent: Adding a weather forecast tool 283
Implementing a mock weather service 283 ▪ Creating the weather forecast tool 284 ▪ Updating the agent for multi-tool support 284
11.8 Executing the multi-tool agent 284
Running the multi-tool agent (initial behavior) 285 ▪ Improving LLM tool usage with system guidance 285
11.9 Using prebuilt components for rapid development 288
Refactoring to use the LangGraph ReAct agent 288
Running the prebuilt agent 289 ▪ Observing and debugging with LangSmith 289 ▪ Enabling LangSmith tracing 289

12 *Multi-agent systems 293*

12.1 Building an accommodation booking agent 294

Hotel booking tool 294 ▪ B&B booking tool 295 ReAct accommodation booking agent 297

12.2 Building a router-based travel assistant 298

Designing the router agent 298 ▪ Routing logic 298 Building the multi-agent graph 300 ▪ Trying out the router agent 301

12.3 Handling multi-agent requests with a Supervisor component 302

The Supervisor pattern: An agent of agents 302 ▪ From "one-way" to "return ticket" interactions 304 ▪ Trying out the Supervisor agent 305

13 *Building and consuming MCP servers 308*

13.1 Introduction to MCP servers 309

The problem: Context integration at scale 309 ▪ The solution: The Model Context Protocol 310 ▪ The MCP ecosystem 311

13.2 How to build MCP servers 312

Essential resources for MCP server development 312 ▪ Official language-specific MCP SDKs 312 ▪ Consuming MCP servers in LLM applications and agents 313

13.3 Building a weather MCP server 314

Implementing the MCP server 314 ▪ Trying out the MCP server with MCP Inspector 316 ▪ Consuming the MCP server from a test MCP host 320

13.4 Integrating the Weather MCP tool into an agent 322

Preparing the travel agent for live weather data 323 Integrating the AccuWeather MCP tool 323 ▪ Updating the agent chat loop 323 ▪ Combining local and remote tools 324 Testing and verification 324 ▪ Using the agent for complex queries 325

14 *Productionizing AI agents: Memory, guardrails, and beyond 327*

14.1 Memory 328

Types of memory 328 ▪ Why short-term memory is needed 328 Checkpoints in LangGraph 329 ▪ Adding short-term memory to

our travel assistant 331 ▪ Executing the checkpointer-enabled assistant 334 ▪ Rewinding the state to a past checkpoint 334

14.2 Guardrails 337

Implementing guardrails to reject nontravel-related questions 338 Implementing more restrictive guardrails at the agent level 343

14.3 Beyond this chapter 346

Long-term user and application memory 346 ▪ Human-in-the-loop 346 ▪ Post-model guardrails 347 ▪ Evaluation of AI agents and applications 347 ▪ Deployment on the LangGraph platform and Open Agent Platform 348

appendix A Trying out LangChain 351

appendix B Setting up a Jupyter Notebook environment 357

appendix C Choosing an LLM 360

appendix D Installing SQLite on Windows 375

appendix E Open source LLMs 377

index 411

preface

In late 2022, something changed. Large language models (LLMs) stopped feeling like experimental demonstrations and started becoming genuinely useful. A quick attempt to summarize a paragraph evolved into a chatbot capable of answering questions, and a small script turned into a service that other teams wanted to try. Before long, LLMs were no longer a curiosity—they had become an essential part of the software development toolkit.

Here's why that's so exciting: LLMs allow software to "speak human." They can revise a contract, turn logs into meaningful answers, draft code, and plan the next step—and then invoke the right tools and data to actually accomplish the task. Combined with retrieval and tool use, an application stops feeling like a rigid machine and begins to feel more like a collaborative partner. The potential is significant, but turning that potential into production systems isn't simple. It still requires careful work to integrate data flows, design effective prompts, ground answers with retrieval, orchestrate multi-step workflows, and monitor how the system behaves once it's deployed.

My own journey into this field followed a similar path to many developers. I began by experimenting in Jupyter Notebooks, exploring APIs, and learning where the models succeeded and where they struggled. Those early explorations gradually evolved into small, agent-based side projects at work, built for a handful of early adopters seeking productivity gains. As the technology advanced—through improvements in OpenAI's APIs, the growing capabilities of LangChain, the orchestration power of LangGraph, and emerging techniques such as advanced Retrieval-Augmented Generation (RAG) and ReAct—those prototypes became more sophisticated. At the same time, I began writing this book. More than once, a chapter I had just completed felt outdated only a few weeks later. It was both exhilarating and, at times, exhausting.

That experience strongly influenced the way this book is written. Rather than chase every new parameter or short-lived "best practice," the focus here is on the concepts, architectures, and design patterns that have proven stable and that underpin reliable LLM applications. We'll build concrete systems—engines, chatbots, and agents—but the aim is to provide reusable foundations: how to structure retrieval, design prompts, compose chains, evaluate behavior, and orchestrate multi-step workflows with clarity and confidence.

Frameworks play a crucial role in this process. LangChain standardizes the essential components—loaders, splitters, embeddings, retrievers, vector stores, and prompts—so that you don't need to reinvent the plumbing for every project. LangGraph extends this by structuring workflows as graphs and coordinating agent loops, while LangSmith adds visibility for debugging and evaluation. Together, they enable developers to focus on the application itself rather than the underlying infrastructure.

Why this book, and why now? The foundations are finally stable enough to teach effectively, and the need is greater than ever. Development teams want practical guidance that bridges the gap between ideas and implementation without tying them to a single vendor or a fleeting technique. If this book succeeds, you'll come away with a clear mental model for building LLM-powered systems, a set of proven patterns you can depend on, and the confidence to keep building—even as the landscape continues to evolve beneath our feet.

acknowledgments

Writing a book like this is never a solo effort, and I feel profoundly grateful to have had so many talented and generous people alongside me on this journey. First, I would like to thank Juan-Mauro Bozzano, Romulus Corneanu, Carolyn Kao, and Ivan Lin for their thoughtful feedback at the very beginning, when the first chapters were still little more than outlines. Their comments and conversations helped shape the direction of the book in its early stages. I'm also indebted to David Bujan and Diego López-de-Ipiña from Deusto University, who encouraged me to bring greater precision and rigor to the work—especially in those formative drafts when many of the ideas were still taking shape.

A special word of thanks goes to Michael Stephens at Manning, who believed in this project from the very first conversation. At the time, LLMs were only beginning to capture the world's attention, and the road ahead was far from clear. His support, together with timely guidance and encouragement to adapt as the technology evolved, helped keep the book relevant and focused.

I'm particularly grateful to my editor, Dustin Archibald, whose steady guidance and thoughtful advice carried me through every stage of the writing process, and to Andrew Freed, whose sharp technical insights made the explanations clearer and more accurate. Many thanks also to technical editor Antowan Batts. Antowan is a financial systems analyst and applied economist with expertise in finance, data analysis, and global supply chains. He teaches economics and focuses on connecting analytical insights to real-world business challenges.

To the reviewers—Abdullah Al Imran, Abhishek Guha, Andres Mariscal, Antowan Malik Batts, Artur Guja, Aryan Jadon, Balaji Venkateswaran, Borko Djurkovic, Byron Galbraith, Carnell Greenfield, David Caswell, David Jacobs, Derek Morgan, Felipe P.

Coutinho, Ganesh Swaminathan, Geevarghese George, Guillaume Alleon, Heather S. Ward, Hobson Lane, James Black Jr., John Powell, Karl-Gustav Kallasmaa, Krisztián Boros, Lola Vicente, Lucian-Paul Torje, Luke Kupka, Manas Talukdar, Marco Massenzio, Meetu Malhotra, Naresh Vurukonda, Nicolas Modrzyk, Peter V. Henstock, Piotr Jastrzębski, Prashanth Josyula, Ramani Natarajan, Raul Ciotescu, Roman Fedytskyi, Shivendra Srivastava, Sriharsha Annamaneni, Stefano Lottini, Sumaira Afzal, Theo Despoudis, Vinoth Nageshwaran, Walter Alexander Mata López, and many others—thank you for your careful reading and valuable feedback. Your suggestions pushed me to think more deeply, write more clearly, and refine the ideas that underpin this book. It's a stronger work because of your contributions.

I would also like to acknowledge the many colleagues and collaborators with whom I've had the privilege to work on real-world LLM projects. The in-depth discussions, lively debates, and countless "What if we tried this?" moments all shaped my thinking about how to design and deploy AI agents effectively. Many of those lessons are woven into the pages that follow.

Finally, my deepest thanks go to my family. To my wife, Estrella, and my daughters, Bianca and Clio: thank you for your patience, your support, and your unwavering love. Thank you for understanding when I disappeared into writing or revisions for days on end. This book wouldn't exist without you.

about this book

AI Agents and Applications is a hands-on, project-focused guide. We begin with the foundational skills that make large language model (LLM) applications work effectively in real-world scenarios: designing reusable prompts and grounding model outputs in your own data using Retrieval-Augmented Generation (RAG). From there, we progress into agentic workflows and multi-agent systems capable of using tools, making decisions, and collaborating when a single prompt is insufficient.

You'll learn how to trace and debug your systems with confidence, transforming brittle prototypes into maintainable applications ready for deployment. We'll use LangChain to build composable components, LangGraph to create clear and testable control flows (especially for agent-based solutions), and the Model Context Protocol (MCP) to integrate external capabilities as easily as local ones. Throughout the book, we'll explore where these systems excel, where they fall short, and how to resolve common issues such as retrieval inconsistencies, imprecise queries, unreliable tool calls, and drifting behavior. The ultimate goal is straightforward: to help you build AI solutions your users can trust.

Who should read this book

This book is intended for developers and technically inclined readers who want to build production-grade AI agents and applications using LangChain, LangGraph, and MCP. If you're comfortable with basic Python and standard developer workflows—such as running commands in a terminal and managing virtual environments—you'll feel at home. Familiarity with launching a Jupyter Notebook or developing Python applications in Visual Studio Code (VS Code) will help you progress even faster. No prior experience with LLM application development is required, although having experimented with tools such as ChatGPT, Claude, or Gemini will be beneficial.

You may be a software engineer ready to go beyond simple API calls to LLMs, a data scientist or machine learning engineer looking to convert RAG prototypes into reliable applications or agents, or a technical lead or product manager evaluating feasibility and deployment strategies. Students, independent learners, researchers, and hobbyists will also find a clear, practical path from foundational concepts to fully functional systems.

Across the chapters, you'll advance from foundational concepts to fully functional applications: designing effective prompts; building, testing, and strengthening RAG pipelines with LangChain; implementing agentic workflows and multi-agent coordination with LangGraph; and integrating external tools through MCP to create secure, interoperable capabilities. Along the way, you'll develop the skills to trace, debug, and refine system behavior with confidence. If your goal is to turn ideas into reliable, maintainable AI products, this book will guide you every step of the way.

How this book is organized: A road map

This book is divided into 14 chapters across 5 parts. Each part builds on the previous one, progressing from fundamental LLM skills to production-ready agents using LangChain, LangGraph, and MCP. You can read it from start to finish or focus on the sections most relevant to your project—cross-references throughout the text will help you bridge any gaps:

- *Part 1: Getting started with LLMs (chapters 1–2)*—This part lays the foundation by exploring where LLMs excel and where they struggle, and why frameworks are essential for real-world applications. You'll learn about the main application patterns (engines, chatbots, agents) and the pillars of robust systems—prompt engineering and RAG:
  - Chapter 1 maps the LLM application landscape, from summarization and semantic search to chatbots and agents, and then examines common challenges. You'll see how LangChain, LangGraph, and LangSmith offer modular building blocks and why prompt engineering and RAG are key to grounded systems.
  - Chapter 2 offers a hands-on introduction to prompt design, covering persona, context, instructions, inputs, and examples, as well as one-shot/few-shot learning strategies and chain-of-thought (CoT) prompting. You'll use LangChain's `PromptTemplate` and `FewShotPromptTemplate` and the OpenAI API to generate, test, and iterate prompts automatically.
- *Part 2: Summarization (chapters 3–5)*—This part focuses on a fundamental LLM use case: distilling large volumes of text into actionable insights, while managing context constraints and preparing for agentic workflows:
  - Chapter 3 builds summarization chains for single large documents, multi-document corpora, and structured data. You'll load sources into `Document` objects, compare MapReduce and refine strategies, and construct LangChain

Expression Language (LCEL) pipelines that preserve key ideas without exceeding context limits.
  - Chapter 4 develops a research summarization engine that searches the web, retrieves sources, summarizes them, and produces a concise, defensible report. The process is broken into sub-chains (search, fetch, summarize, compose) and orchestrated with LCEL.
  - Chapter 5 transitions from linear chains to LangGraph, introducing explicit state, nodes, and edges. You'll implement conditional branches (e.g., re-querying when evidence is insufficient) and refactor the research assistant for greater reliability and debuggability.
- *Part 3: Q&A chatbots (chapters 6–7)*—This part shifts the focus from summarizing information to answering user questions, demystifying RAG, and instrumenting your system for observability:
  - Chapter 6 implements RAG from first principles with OpenAI and ChromaDB, covering ingestion, embeddings, semantic search, and a minimal Q&A chatbot that demonstrates the ingestion–query loop and the roles of vector stores and retrievers.
  - Chapter 7 rebuilds the chatbot using LangChain's RAG components and integrates LangSmith tracing. You'll work with loaders, transformers, vector stores, retrievers, chat memory, and specialized utilities for streamlined question answering.
- *Part 4: Advanced RAG (chapters 8–10)*—This part explores how to scale retrieval as your data and user base grow. You'll refine both indexing and querying strategies and learn to combine multiple backends:
  - Chapter 8 enhances indexing with smarter chunking, multiple embeddings for coarse and fine contexts, and targeted expansion. You'll use `ParentDocumentRetriever` and `MultiVectorRetriever`, tailoring strategies for semi-structured and multimodal content.
  - Chapter 9 improves retrieval by strengthening the queries themselves: using rewrite–retrieve–read patterns, step-back prompts, Hypothetical Document Embeddings (HyDE), and question decomposition to consistently surface richer evidence.
  - Chapter 10 routes queries to the appropriate backend (vector store, SQL database, document database, or knowledge graph), generates backend-specific queries (SQL/SPARQL), and postprocesses results with Reciprocal Rank Fusion (RRF) to deliver the most relevant context to the LLM.
- *Part 5: AI agents (chapters 11–14)*—This part explores how systems make decisions and take action. You'll build tool-using and multi-agent systems, extend them with MCP, and prepare them for production:
  - Chapter 11 constructs tool-based agents in LangGraph, demonstrating how to register and invoke tools step-by-step as the model selects actions. You'll

evolve from a single-tool travel assistant to a multi-tool agent, with full observability in LangSmith.

- Chapter 12 composes multi-agent systems using router and Supervisor patterns to ensure the right specialist handles each subtask. You'll connect agents to real data sources (SQL/REST) and learn practical techniques for debugging and testing cross-agent workflows.
- Chapter 13 introduces the Model Context Protocol (MCP): what it is, why it matters, and how to use it. You'll build and test an MCP server (e.g., a weather tool), consume third-party MCP tools, and integrate remote capabilities alongside local ones, cleanly and securely.
- Chapter 14 prepares agents for production by adding memory (via LangGraph checkpoints), layered guardrails, and human-in-the-loop controls, along with establishing continuous evaluation processes to maintain reliability over time.

How to read this book

If you're new to LLM applications, read parts 1 to 3 sequentially before exploring the more advanced topics in part 4. If your primary interest is in agents, skim chapters 1 and 2 for terminology, complete the summarization and RAG foundations in chapters 3 to 7, and then focus on part 5. Each chapter includes runnable examples and suggestions for next steps, enabling you to incorporate the material into real projects incrementally.

About the code

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text. Sometimes code is also **`in bold`** to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this wasn't enough, and listings include line-continuation markers (➥). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

All examples in this book are written in Python and can be executed on a typical developer laptop—no special hardware is required. Where helpful, I provide brief instructions for creating and activating virtual environments, along with occasional shell commands, but the focus remains firmly on Python. The complete source code for every chapter is available from Manning at www.manning.com/books/ai-agents-and-applications and mirrored on GitHub at https://mng.bz/rZlZ, where you'll also find updates and errata. You can get executable snippets of code from the liveBook

(online) version of this book at https://livebook.manning.com/book/ai-agents-and-applications.

Some chapters are presented as Jupyter Notebooks that follow the narrative step-by-step, while others are organized as standalone VS Code projects you can run end-to-end (or in Cursor, if you prefer). The notebooks emphasize explanation and experimentation, whereas the VS Code projects focus on structure and reusability. Both approaches rely on the same underlying modules, ensuring consistent behavior as you move between them.

The projects target modern versions of Python (3.11 or newer) and use standard tools such as pip and virtual environments. Each chapter directory includes a `requirements.txt` file. When an example depends on an external service—typically a model API or a web search—you'll need to supply your own credentials via environment variables or a local `.env` file. These instances are clearly indicated throughout the text.

Because you'll be building LLM-based applications and agents, most examples use OpenAI models, primarily from the GPT-5 family. Before running them, you'll need to create an OpenAI account and link a debit or credit card (instructions are provided in the book). Most examples run comfortably on GPT-5-nano, the least expensive model, and completing all examples should cost less than $5 in total. Of course, you're free to experiment with larger models if you wish.

Because agentic systems are inherently nondeterministic, your results may vary slightly from those shown in the book. Where reproducibility matters, I fix random seeds, constrain model temperature, or include sample transcripts so you can compare outputs. Treat the code repository as a living companion to this book—run it, extend it, and use it as a foundation for your own agents and applications.

liveBook discussion forum

Purchase of *AI Agents and Applications* includes free access to liveBook, Manning's online reading platform. Using liveBook's discussion features, you can attach comments to the book as a whole or to specific sections and paragraphs. It's easy to make personal notes, ask and answer technical questions, and receive assistance from the author and other readers. To access the forum, visit https://livebook.manning.com/book/ai-agents-and-applications/discussion.

Manning is committed to providing a platform where meaningful dialogue can occur among readers and between readers and the author. However, this doesn't imply any specific level of author participation, which remains voluntary and unpaid. We encourage you to ask the author challenging questions to maintain his engagement. The forum and the archives of past discussions will remain accessible on the publisher's website for as long as the book is in print.

about the author

ROBERTO INFANTE is a software engineer with more than 25 years' experience, largely in finance—spanning investment banks, asset managers, brokers, and exchanges. He currently leads quantitative development for a hedge fund's treasury function while also leading several generative-AI initiatives. Roberto is the author of *Building Ethereum Dapps*.

about the cover illustration

The figure on the cover of *AI Agents and Applications* is "Djiebedji, ou officier des munitionnaires," or "Djiebedji, or an ammunitions officer," taken from the *Album of Turkish Costume Paintings* (George Arents Collection, The New York Public Library, 1808–1826).

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture centuries ago, brought back to life by pictures from collections such as this one.

Part 1

Getting started with LLMs

This opening part lays the foundation for the rest of the book. We'll explore the "why" and "how" of building applications powered by large language models (LLMs)—what they do well, where they struggle, and why frameworks such as LangChain, LangGraph, and LangSmith are essential for creating reliable, real-world systems. Along the way, you'll discover the main architectural patterns—engines, chatbots, and agents—and learn the core techniques that make them work, including prompt engineering and Retrieval-Augmented Generation (RAG).

We'll also focus on one of the most important everyday skills in AI development: crafting and executing prompts. We'll look at how to design prompts for different kinds of tasks, improve them with one-shot and few-shot examples, and automate the entire process with LangChain's prompt templates and the OpenAI API. By the end of this part, you'll be ready to move beyond experimentation and start building purposeful, dependable LLM-powered applications.

1 Introduction to AI agents and applications

This chapter covers

- Core challenges in building applications powered by large language models
- LangChain's modular architecture and components
- Patterns for engines, chatbots, and agents
- Foundations of prompt engineering and Retrieval-Augmented Generation

Large language models (LLMs) such as GPT, Gemini, and Claude have moved from novelty to necessity. LLMs enable applications to answer complex questions, generate tailored content, summarize long documents, and coordinate actions across systems. More recently, LLMs have unlocked a new class of applications: AI agents. Agents take input in natural language, decide which tools or services to call, orchestrate multi-step workflows, and return results in a clear, human-friendly format.

AI applications and agent systems can be complex. They need to ingest and manage data, structure prompts, chain model calls together reliably, and integrate external APIs and services. Fortunately, frameworks such as LangChain, LangGraph, and LangSmith provide modular building blocks that eliminate boilerplate, promote best practices, and let you focus on application logic instead of low-level wiring. In this book, you'll learn how to design, build, and scale real LLM-based applications and agents using the best tools and frameworks.

1.1 Building LLM-based applications and agents

LLMs are great at handling natural language—they can understand text, generate it, and pull out information when you need it. That makes them useful for all kinds of applications: summarization, translation, sentiment analysis, semantic search, chatbots, and code generation. Because of this range, you'll now see LLMs showing up in fields as varied as education, healthcare, law, and finance.

Even with all of these different use cases, most LLM apps end up looking pretty similar under the hood. They take in natural language input, work with unstructured data, pull in extra context from one or more sources, and then package everything into a prompt for the model to process. At a high level, these systems generally fall into three main categories:

- *LLM-based applications or engines* are systems that deliver a specific, bounded capability (for example, summarization, search, or content generation). An engine is typically invoked to produce output and then it stops; it does not decide what to do next or carry work forward beyond the request.
- *Chatbots* are conversational interfaces that maintain context over multiple exchanges. They are primarily focused on dialogue and interactive Q&A rather than independently taking action.
- *AI agents* are autonomous or semi-autonomous systems that use LLMs to choose actions, plan, and execute multi-step work to reach a goal. Unlike engines, agents can loop (observe → decide → act → observe), coordinate multiple tools/APIs, and adapt their next step based on intermediate results, errors, or new information.

We'll examine each of these categories in turn before exploring how LangChain supports their development. Let's start with LLM-based applications and engines.

1.1.1 LLM-based applications: Summarization and Q&A engines

An LLM-based application acts as a backend tool that handles specific natural language requests for other systems. For example, a summarization engine condenses lengthy text passages into concise summaries. These summaries can be returned immediately to the client or stored in a database for later use by other applications, as shown in figure 1.1.

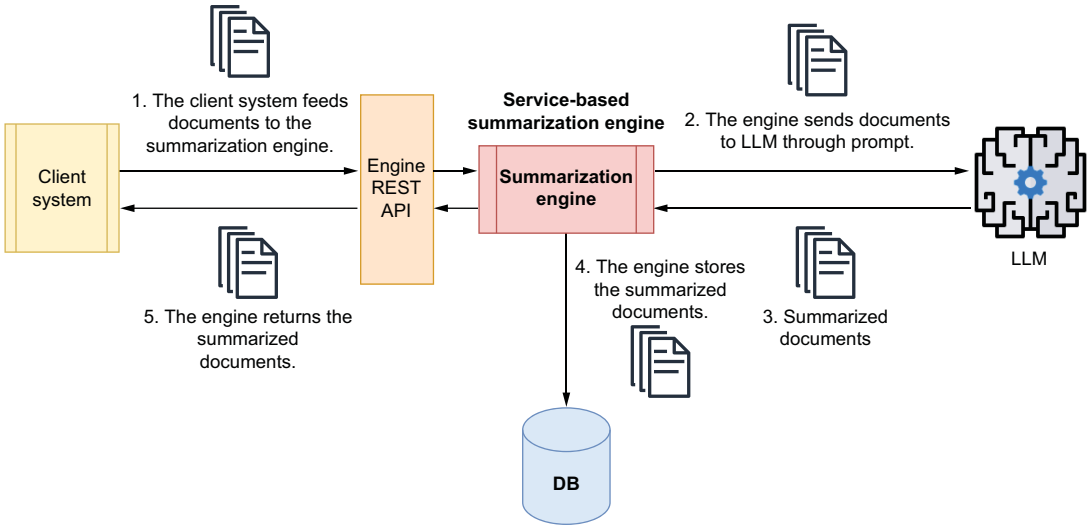


Figure 1.1 A summarization engine efficiently summarizes and stores content from large volumes of text and can be invoked by other systems through the REST API.

Summarization engines are often deployed as shared services, which are typically exposed through a REST API so multiple systems can call them on demand. Another common type is the Question & Answer (Q&A) engine, which answers user queries against a knowledge base. A Q&A engine works in two phases: ingestion and query.

In the content ingestion phase, the engine builds its knowledge base by pulling in text, splitting it into chunks, and converting those chunks into embeddings—mathematical vectors that capture meaning. Both the embeddings and the original chunks are stored in a vector store for efficient retrieval. Don't worry if *embedding model* or *vector store* sound unfamiliar; we'll cover them in detail later. For now, just think of this step as transforming raw text into a searchable representation.

DEFINITION *Embeddings* are vector representations of words, tokens, or larger text units—such as sentences, paragraphs, or document chunks—mapped into a continuous, high-dimensional space (typically with hundreds or thousands of dimensions). These vectors capture semantic and syntactic relationships, allowing language models to understand meaning, context, and similarity. Embeddings are typically learned during the pretraining phase, where models are trained on large-scale text corpora to predict tokens based on surrounding context. By encoding both individual words and broader semantic meaning, embeddings enable more effective reasoning, retrieval, and language understanding.

In the query phase, the engine takes a user's question, turns it into an embedding using the same model, and performs a semantic search over the vector store. It retrieves the most relevant chunks and combines them with the original question to form a prompt, which is then sent to the LLM. The model then uses the question and the retrieved context to generate an answer that is intended to be accurate and grounded in the provided sources—though the quality of the result ultimately depends on factors like retrieval relevance, chunking strategy, and the model's behavior.

This workflow is known as Retrieval-Augmented Generation (RAG), as shown in figure 1.2. RAG has quickly become a cornerstone of modern LLM applications. It's a foundational technique for making LLM outputs more useful, up to date, and grounded in relevant external information.

DEFINITION *Retrieval-Augmented Generation* (RAG) is a design pattern in which the LLM's text generation is augmented by incorporating additional context and then retrieved from a local knowledge base—often stored in a vector store—at query time.

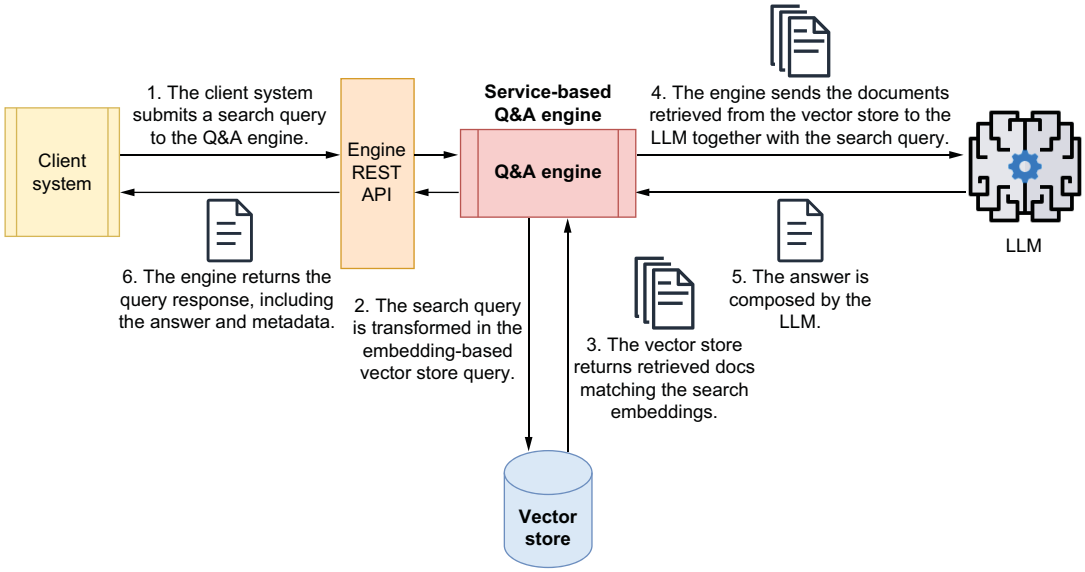


Figure 1.2 A Q&A engine implemented with RAG design. An LLM query engine stores domain-specific document information in a vector store. When an external system sends a query, it converts the natural language question into its embeddings (or vector) representation, retrieves the related documents from the vector store, and then gives the LLM the information it needs to craft a natural language response.

Engines aren't limited to Q&A. They can also call external tools by running predefined sequences of steps, often called chains. For example, an engine handling a user request might convert natural language instructions into API calls, pull data from outside systems, and then use an LLM to interpret and present the results in a clean, human-readable format.

At their core, these engines are designed for system-level work: automating processes, processing data intelligently, and stitching together different platforms. They simplify workflows that involve natural language by handling the messy parts—retrieval, transformation, orchestration—so you don't have to.

Later we'll see that AI agents build on these capabilities but differ in where the "decision-making" lives. An engine typically executes a predefined chain: the steps and branching logic are largely designed ahead of time, and the system runs that workflow to completion for a given request. An agent, by contrast, uses the LLM to plan and adapt its approach at runtime—choosing which tools to call, revising its plan based on intermediate results, and potentially looping through multiple actions until it reaches a stopping condition. Put simply: engines run workflows while agents manage workflows, trading some predictability for greater flexibility on open-ended, multi-step tasks.

Once a human is directly interacting with the system in real time, the game changes. At that point, the engine takes on a conversational role, shifting from pure automation into dialogue. This is where chatbots come into play.

1.1.2 LLM-based chatbots

An LLM-based chatbot acts as an intelligent assistant, enabling ongoing, natural conversations with a language model. Unlike simple question–answer scripts, these systems aim to keep interactions both useful and safe. They rely heavily on prompt design: clear instructions shape the model's behavior and help prevent irrelevant, inaccurate, or unsafe replies. Modern chat APIs—such as those from OpenAI—support role-based messaging formats (system, user, assistant), which let you define an assistant's persona and enforce consistent behavior across a conversation.

To improve accuracy, chatbots often pull in factual context from local knowledge sources such as vector stores. This lets them blend conversational fluency with domain-specific grounding, so the answers aren't just smooth but also relevant and reliable.

A key strength of chatbots is conversation memory. By tracking earlier turns, they can keep responses coherent and personalized. This memory is limited by the model's context window. Larger windows help, but many systems still compress or summarize conversation history to stay within limits—and to manage cost.

LLM-based chatbots are usually specialized for tasks such as summarization, question answering, or translation. They can either respond directly to user input or combine it with stored knowledge. For example, the architecture of a summarization chatbot (figure 1.3) builds on a basic summarization engine, but adds dialogue management and context-awareness layers.

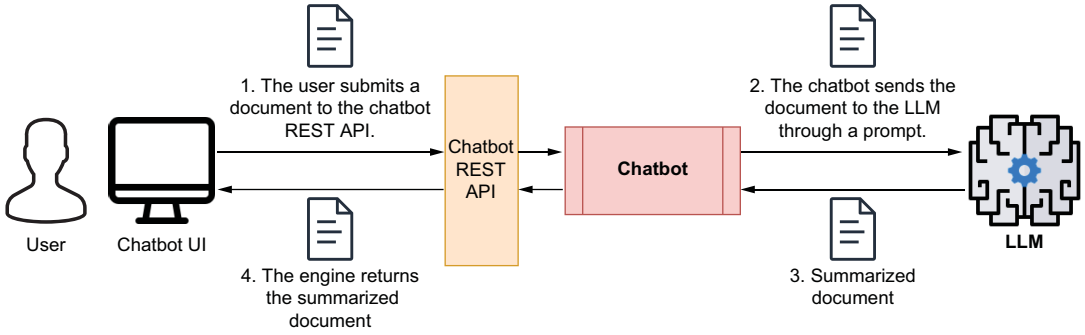


Figure 1.3 A summarization chatbot shares some similarities with a summarization engine, but it offers an interactive experience where the LLM and the user can work together to fine-tune and improve the results.

The crucial difference between a summarization engine and a summarization chatbot is interactivity. A chatbot lets you refine responses in real time: if you want a shorter or more casual summary, you can just ask, as illustrated in the sequence diagram in figure 1.4.

This back-and-forth makes the process collaborative—producing answers that feel more tailored and context-aware. In the next chapter, we'll explore role instructions, few-shot examples, and advanced prompt engineering to give you more control over chatbot behavior and output.

1.1.3 AI agents

An AI agent is a system that works with an LLM to carry out multi-step tasks—often involving multiple data sources, branching logic, and adaptive decision-making. Unlike simple pipelines, agents can operate with a degree of independence: they follow the constraints you set, but they make choices about what to do next.

At each step, the agent consults the LLM to decide which tools to use, runs those tools, and processes the results before moving on. This loop continues until the agent produces a complete solution. In practice, that might mean pulling information from both structured sources (e.g., databases or APIs) and unstructured ones (e.g., documents or web pages), combining the results, and presenting them in a coherent format.

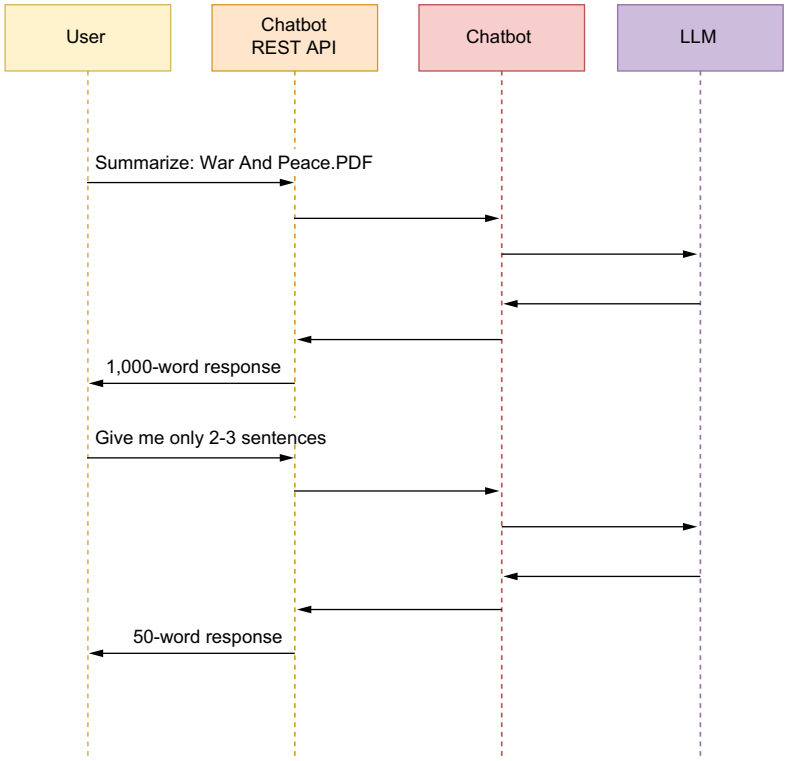


Figure 1.4 Sequence diagram that outlines how a user interacts with an LLM through a chatbot to create a more concise summary

Consider this example: a tour operator uses an AI agent to generate holiday packages based on natural language requests from a booking website. As shown in figure 1.5, the process could look like this:

1. The agent sends a prompt to the LLM asking it to choose the most relevant tools for the request—for example, flight, hotel, and car rental providers; weather forecast services; and internal holiday deals databases.
2. Guided by a developer-crafted prompt listing available tools and their descriptions, the LLM selects the appropriate ones and generates the required queries—such as SQL for the holiday deals database or REST API calls for external providers.
3. The agent executes the queries, gathers the results, and sends another prompt to the LLM containing both the original holiday request and the collected data.
4. The LLM responds with a summarized holiday plan that includes all bookings, which the agent then returns to the booking website.

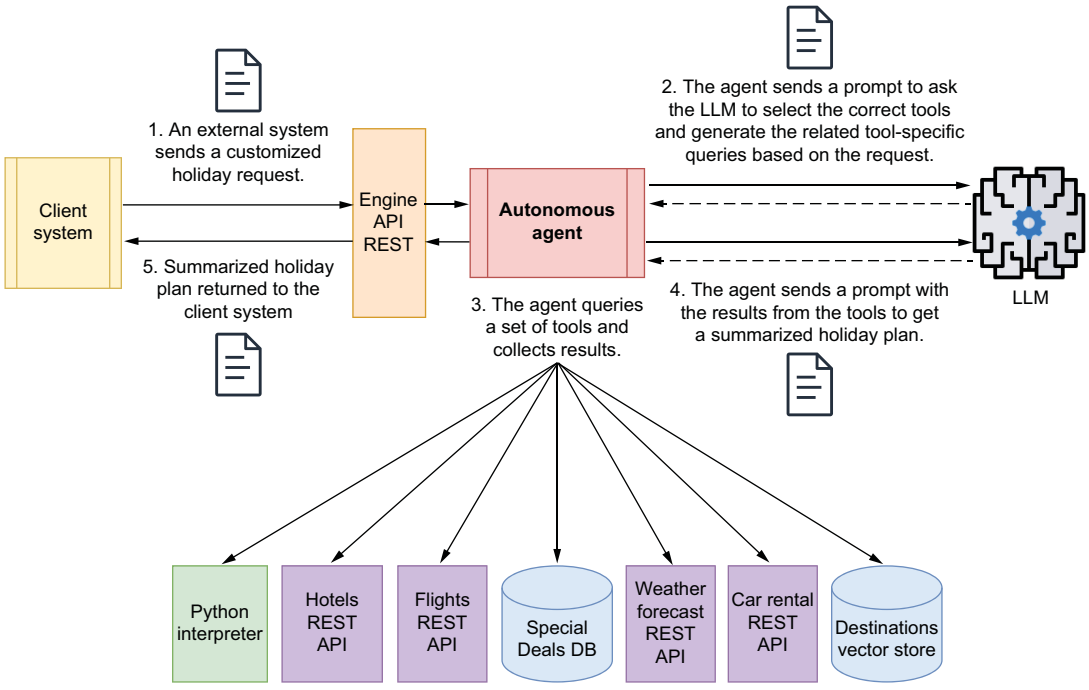


Figure 1.5 Workflow of an AI agent tasked with assembling holiday packages

The workflow can involve multiple iterations between the agent and the LLM before producing a final output, such as a complete holiday plan. In addition, the architecture shown in figure 1.5 is only one possibility. An alternative design could be based on a set of more granular agents coordinated by a Supervisor agent at the top.

In high-stakes domains such as finance or healthcare, it's common to include a human-in-the-loop step. This ensures that a human can review and validate critical actions—such as approving a financial transaction or confirming a complex medical recommendation—before finalizing them. In the holiday planning example, the agent could be programmed to pause and request human approval of the proposed itinerary before sending it to the client, adding an extra layer of oversight and trust.

There has been a surge of interest in AI agents, with major players such as OpenAI, Google, and Amazon, as well as independent developers such as LangChain, LlamaIndex, and Pydantic, all releasing agent SDKs to encourage broader adoption of the approach.

In this book, we'll focus on LangGraph, LangChain's dedicated agent framework. LangGraph provides prebuilt agent and orchestrator classes, along with ready-to-use tool integrations, so you don't have to reinvent the wheel. You'll also gain hands-on

experience building advanced agents with LangGraph, learning how to design, orchestrate, and refine them in practice.

In many ways, an AI agent represents the most advanced form of LLM-based application. It uses the full range of LLM capabilities—understanding, generating, and reasoning over text—to power complex, automated workflows. Agents can dynamically select and use multiple tools, guided by prompts you design, making them valuable across domains such as finance, healthcare, and logistics, where multi-step reasoning and data integration are essential.

Interest in agents has accelerated further with the introduction of the Model Context Protocol (MCP) by Anthropic. MCP defines a standard for services to expose tools through MCP servers, which agents can access via MCP clients as easily as if they were local components. This shifts the integration burden to the service itself, allowing developers to focus on building capable agents rather than maintaining custom connectors. Following its release in late 2024, MCP rapidly became a de facto standard—adopted by major LLM providers such as OpenAI and Google—and thousands of tools are now accessible through public MCP portals. In this book you'll learn how the protocol works, examine the ecosystem in practice, and build your own MCP server that integrates directly with an agent application.

1.2 Introducing LangChain

Imagine you've been asked to build a chatbot that can answer customer questions from your company's documentation, or a search engine that retrieves precise answers from thousands of internal reports. You quickly discover the same pain points:

- How do you get your own data into the model reliably, without pasting entire documents into prompts?
- How do you keep prompts, chains, and integrations maintainable as your features grow?
- How do you handle context limits and costs while still preserving accuracy?
- How do you orchestrate multi-step workflows and API calls without fragile glue code?
- Once your app is live, how do you evaluate, debug, and monitor its behavior?

Without a framework, developers end up reimplementing the same plumbing—load the data, and then split, embed, store, retrieve, and prompt it—again and again. That repetition is exactly what makes it hard to bring proprietary data into the model reliably without overstuffing prompts and keeping prompts, chains, and integrations from turning into an unmaintainable tangle. You also want to balance context limits and cost against answer quality. LangChain addresses these issues by providing a consistent set of building blocks—loaders to ingest data, splitters to chunk it, embeddings plus vector stores to index it, and retrievers to pull only the most relevant context at query time—so you're not pasting whole documents into prompts. Prompt templates

and structured chain composition help you standardize and reuse prompts and integrations as features grow, rather than duplicating logic across the codebase. And for multi-step workflows, LCEL and the `Runnable` interface give you a uniform way to orchestrate tool calls and processing steps, with clearer structure for tracing, debugging, and evaluating behavior once the application is live.

LangChain has also evolved rapidly, fueled by an active open source community. It continues to keep pace with advances in LLM architectures, new data sources, and retrieval technologies, while helping to establish shared best practices for building and deploying LLM-based systems.

Three principles guide LangChain's design: modularity, composability, and extensibility. Components follow standard interfaces, so you can swap an LLM, change a vector store, or add a new data connector without rewriting your entire application. Real-world tasks can be composed from multiple components, forming chains or agent workflows that dynamically select the right tools for the job. And while default implementations exist, you can always extend or replace them with custom logic or third-party integrations, avoiding lock-in and promoting interoperability.

By learning LangChain, you not only gain the ability to build production-grade LLM applications but also acquire transferable skills. Competing frameworks solve similar problems in similar ways, so once you understand these patterns, you can adapt them to whatever stack you choose in the future.

1.2.1 LangChain architecture

LangChain's documentation is thorough, but the best way to understand how it works is by building with it. This section gives you a high-level overview that we'll keep coming back to in later chapters. Think of it as the map you'll use while we dive into the details and code examples later on.

Most LLM frameworks follow a similar overall pattern, but each one defines its own components. In LangChain, the workflow looks roughly like what you see in figure 1.6. You start by pulling in text from different sources—files, databases, or websites—and wrapping it into `Document` objects. Those documents are often split into smaller chunks so they're easier to handle. Next, each chunk is passed through an embedding model, which turns the text into vectors that capture its meaning. Both the raw chunks and their embeddings are stored in a vector store, which lets you quickly retrieve the most relevant pieces of text based on similarity search.

When an LLM app runs a task—say, summarization or semantic search—it builds a prompt that combines the user's question with extra context. That context usually comes from document chunks pulled out of a vector store. Sometimes, though, you'll also want to bring in information from a graph database. Vector stores are still the backbone of most RAG workflows, but graph databases are becoming more common in apps that need to represent and reason about relationships between entities. This is especially relevant for agents: unlike engines (that typically execute a single retrieval

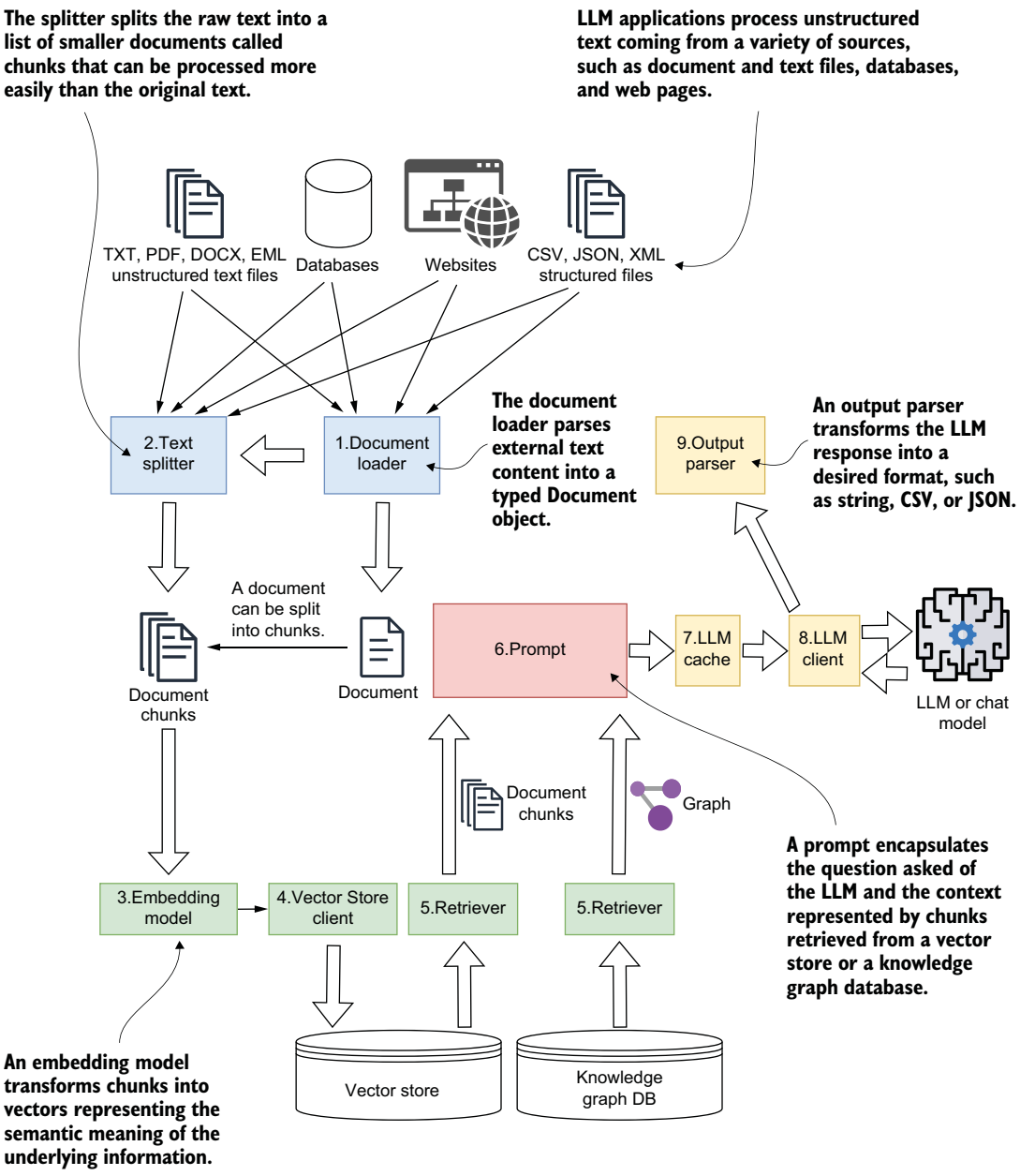


Figure 1.6 LangChain architecture. The document loader imports data, which the text splitter divides into chunks. These are vectorized by an embedding model, stored in a vector store, and retrieved through a retriever for the LLM. The LLM cache checks for prior requests to return cached responses, while the output parser formats the LLM's final response.

step and return an output), agents may need to maintain longer-lived memory, track entities and their relationships over time, and use those relationships to plan the next actions (for example, deciding which tool to call or which information to retrieve next). LangChain already integrates with popular options such as Neo4j, and it lets you use graph-based memory or planning components—features that are showing up more and more in advanced agent architectures.

To make all of this easier to wire together, LangChain introduced the `Runnable` interface and LCEL. These give you a clean, consistent way to chain components without writing piles of glue code. We'll go deeper into both later in this chapter.

It's also worth keeping in mind that LangChain's workflow isn't locked into a simple pipeline. You can arrange components as graphs to handle more complex, branching flows—a capability formalized in the LangGraph module. We'll dig into these patterns and architectural variations in the next section, where the bigger picture of LLM application design comes into focus.

A detailed description of each component follows here (it references the numbering in figure 1.6):

- *Document loaders (1)*— Extract content from sources (files, web pages, SaaS tools, and databases) and convert it into LangChain `Document` objects.
- *Text splitters (2)*— Break large texts into smaller chunks/`Document` objects so they fit within context limits and can be embedded, indexed, and retrieved effectively.
- `Document`— LangChain's core container for content + metadata (e.g., source, page number, author, and timestamp), used end-to-end from ingestion through to retrieval.
- *Embedding models (3)*— Convert text into vectors that represent semantic meaning, enabling similarity-based search and matching.
- *Vector stores (4)*— Specialized databases that store embeddings (and associated `Document` objects) and support efficient semantic retrieval for RAG.
- *Knowledge graph databases*— Graph databases that store entities and relationships. Useful when you need to model and query connections between concepts rather than just text similarity.
- *Retrievers (5)*— Components that query one or more backends (vector stores, SQL, and graphs) to return the most relevant Documents for a user question.
- *Prompts (6)*— Reusable prompt templates that combine the user input with retrieved context (and, optionally, examples) to form the final request sent to the LLM.

- *LLM cache (7)*— An optional layer that reuses prior responses to reduce latency and cost for repeated or similar queries.
- *LLM/chat model (8)*— The LLM interface used to generate outputs; LangChain supports multiple providers and also mock/fake models for testing.
- *Output parser (9)*— Transforms the model's response into a structured format (e.g., JSON), making downstream processing more reliable.

The components in the preceding list can be organized into a chain or can be structured around agents:

- *Chain*—This is a composite arrangement guiding LangChain's processing workflow, customized for specific use cases and based on a sequence of the described components.
- *Agent*—This component manages a dynamic workflow, extending a sequential chain. The agent's processing is flexible and can adapt based on user input or component output. Resources in the dynamic workflow are called *tools* or *plugins* in most frameworks' documentation, and the collection of all tools is referred to as the *toolkit*.

LangChain's comprehensive design supports three primary application types, which we'll examine in detail shortly: summarization and query services, chatbots, and agents. Although the framework can seem complex at first glance, its structure and core concepts will become clearer as we work through the examples and build up the pieces step by step.

1.2.2 LangChain's core object model

With a good understanding of LangChain's high-level architecture, we can now turn to its core object model for a clearer picture of how the framework operates. Understanding the key objects and their interactions will significantly improve your ability to use LangChain effectively. Just as importantly, these class families mirror the building blocks you'll encounter in most LLM applications—documents, chunking, retrieval, prompting, and model calls—so learning the LangChain object model also gives you a practical mental model for how LLM applications are commonly assembled.

The object model is organized into class hierarchies, beginning with abstract base classes from which multiple concrete implementations are derived. Figure 1.7 provides a visual overview of the main class families used across various tasks, all centered around the `Document` entity. It illustrates how loaders generate `Document` objects, how splitters divide them into smaller segments, and how these are then passed into vector stores and retrievers for downstream processing.

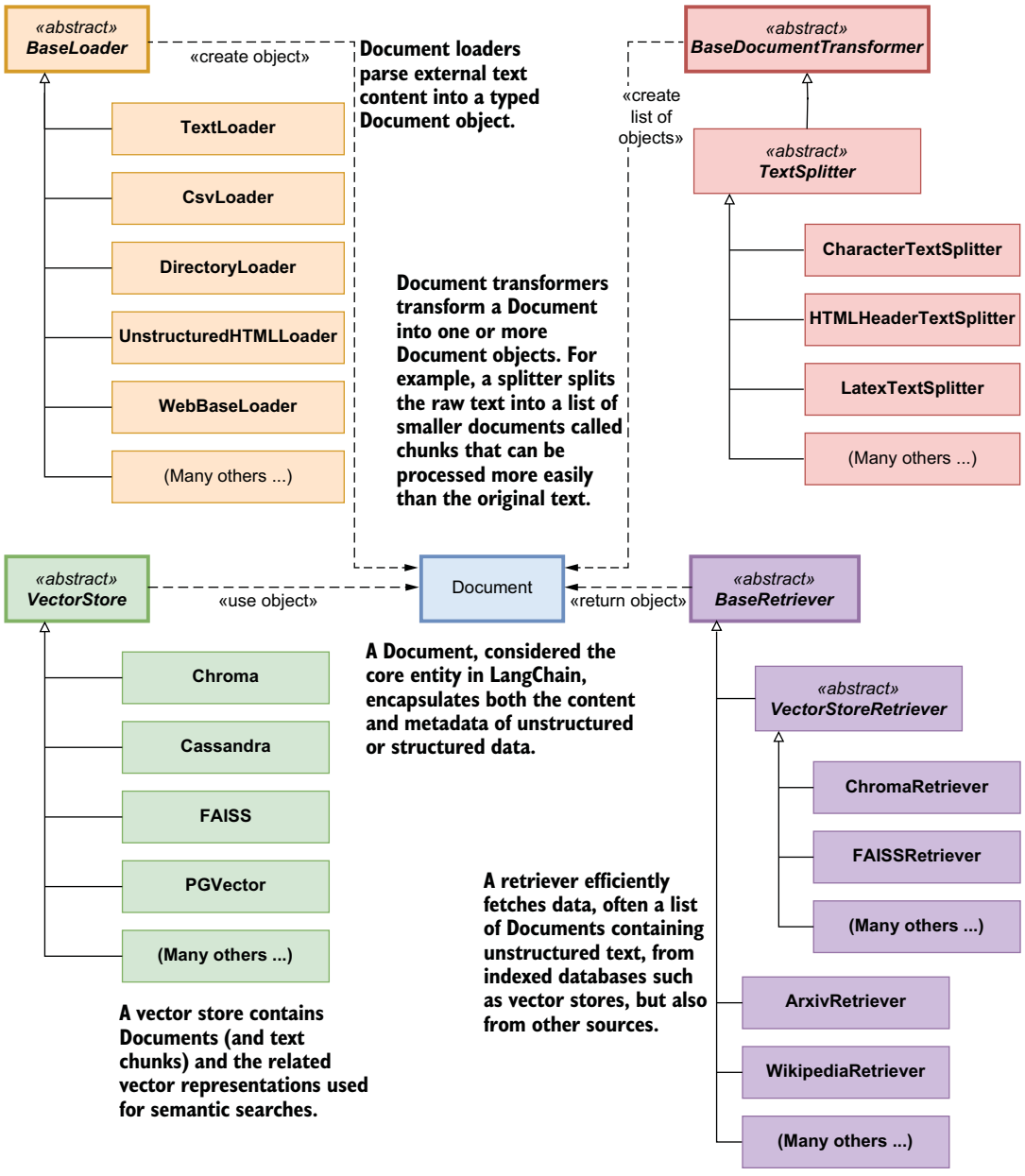


Figure 1.7 Object model of classes associated with the `Document` core entity, including `Document` loaders (create `Document` objects), splitters (create a list of `Document` objects), vector stores (store `Document` objects in vector stores), and retrievers (retrieve `Document` objects from vector stores and other sources)

As covered in section 1.2.1 on LangChain architecture, the primary classes include the following:

- `Document`
- `DocumentLoader`
- `TextSplitter`
- `VectorStore`
- `Retriever`

LangChain integrates with a wide variety of third-party tools and services across these components, offering great flexibility. You can find a full list of supported integrations in the official documentation. In addition, LangChain provides the LangChain Hub, a community-driven repository for discovering and sharing reusable components such as prompts, chains, and tools.

A key architectural feature shared by many of these components—from loaders to LLMs—is the `Runnable` interface. This common interface allows objects to be composed and chained together consistently, enabling highly modular workflows. We'll dive deeper into this feature, and into the LCEL, in a later section focused on building composable and expressive LLM pipelines.

In figure 1.8, you can see the object model related to LLMs, including `PromptTemplate` and `PromptValue`. This figure illustrates how these classes connect to the LLM interface, exposing a somewhat more complex hierarchy than the classes presented earlier.

Now that you've explored LangChain's purpose, architecture, and the core types of applications it supports, I encourage you to experiment with LangChain using the Jupyter Notebook described in appendix B. It's a quick and practical way to get a feel for the framework before we dive deeper.

> **NOTE** Because you'll be building LLM-based LangChain applications and LangGraph agents, most examples in this book use OpenAI models, primarily from the GPT-5 family. Before running them, you'll need to create an OpenAI account and link a debit or credit card (see appendix A for instructions). Most examples run comfortably on GPT-5-nano, the least expensive model, and completing all exercises should cost less than $5 in total. You're welcome to experiment with larger models if you wish.

At the heart of every LangChain application lies the power of the LLM. LLM-based applications can be thought of as specialized wrappers or interfaces around the model, tailored for specific use cases. Let's now explore the major types of use cases that LLMs are particularly well suited for.

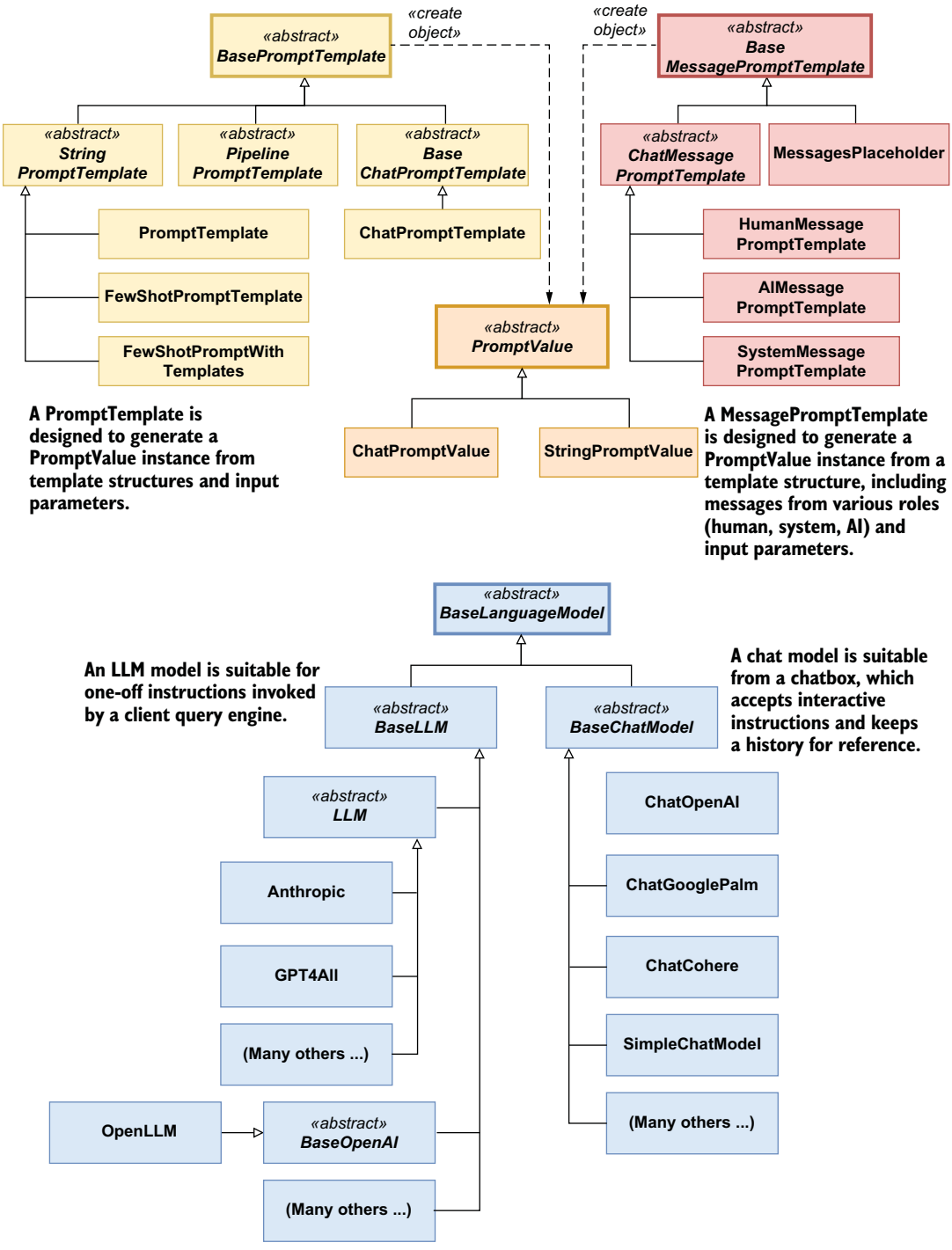


Figure 1.8 Object model of classes associated with LLMs, including `PromptTemplates` and `PromptValues`

1.3 Typical LLM use cases

LLMs are applied across a wide range of tasks, from text classification to code generation and logical reasoning. Following are some of the most common use cases, along with real-world examples for further exploration:

- *Text classification and sentiment analysis*—This includes categorizing news articles or recommending stocks based on sentiment analysis. For example, GoDaddy uses LLMs to automatically classify support tickets, as described in the article "LLM from the Trenches: 10 Lessons Learned Operationalizing Models at GoDaddy" (https://mng.bz/8X52).
- *Natural language understanding and generation*—LLMs can identify main topics in a text and generate summaries tailored by length, tone, or terminology. Duolingo uses AI to accelerate lesson creation, detailed in the blog post "How Duolingo Uses AI to Create Lessons Faster" (https://mng.bz/Ewpl).
- *Semantic search*—This involves querying a knowledge base based on the intent and context of a question, rather than relying on simple keywords. The Picnic supermarket app uses LLMs to enhance recipe search, as explained in the "Enhancing Search Retrieval with Large Language Models (LLMs)" post on Medium (https://mng.bz/NwW2).
- *Autonomous reasoning and workflow execution*—LLMs can handle tasks such as planning a complete holiday package by understanding requests and managing each step of the process.
- *Structured data extraction*—This involves pulling structured data—entities and their relationships, for example—from unstructured text, such as financial reports or news articles.
- *Code understanding and generation*—LLMs can analyze code to identify issues, suggest improvements, or generate new code components—ranging from simple functions and classes to entire applications—based on user instructions. This capability powers popular IDE extensions such as GitHub Copilot and Cline AI, and emerging AI-driven coding assistants such as Cursor and Windsurf, which provide real-time code suggestions and error detection as you work. In addition, CLI-based coding tools such as Anthropic's Claude Code and OpenAI's Codex allow developers to interact with AI through the command line, enabling code generation, refactoring, and debugging directly from terminal environments.
- *Personalized education and tutoring*—LLMs are increasingly used as interactive tutors, providing personalized help and feedback. For instance, Khan Academy's Khanmigo uses an LLM to assist students with interactive learning.

These use cases assume the LLM can competently handle user requests. However, real-world tasks often involve specific domains or scenarios that extend beyond the LLM's initial training. How can you ensure your LLM meets user needs effectively in these cases? That's exactly what you'll learn in the next section.

1.4 How to adapt an LLM to your needs

You can use several techniques to enhance the LLM's ability to respond to user requests, even without prior knowledge or training in a specific domain. These techniques are listed here:

- Prompt engineering
- Retrieval-Augmented Generation (RAG)
- Fine-tuning

Let's begin with the most basic approach: prompt engineering.

1.4.1 Prompt engineering

Prompts for LLMs can be as simple as a single command or as complex as a block of instructions enriched with examples and context. *Prompt engineering* is the practice of designing these inputs so that the model understands the task and produces useful, accurate responses. Done well, it allows developers to guide the model's behavior, adapt it to domain-specific needs, and even handle problems the model wasn't explicitly trained on. A common technique here is *in-context learning*, where the model infers patterns from examples embedded directly in the prompt—no fine-tuning required.

In practice, prompts are often organized as templates: a fixed instruction with variable fields that accept dynamic input. This makes prompts reusable and easier to manage across different parts of an application. A widely used pattern is *few-shot prompting*, where you include a handful of examples to teach the model how to generalize to similar inputs.

Well-crafted prompts are surprisingly powerful. They can coax high-quality, domain-aware output from an LLM without needing extra training data. For instance, chatbots often embed recent conversation turns into the prompt so the model maintains context and produces coherent, multi-turn replies. Prompt engineering can often take you very far, making it a lightweight yet effective tool for many real-world applications.

LangChain builds on this idea with abstractions for managing and reusing prompts consistently, which we'll explore in the next chapter. That said, prompt engineering alone has limits—especially when applications need to ground answers in user-specific or enterprise data. In those cases, the natural next step is RAG (mentioned earlier in this chapter), which augments prompts with knowledge pulled dynamically from external sources.

1.4.2 RAG

One of the most effective ways to improve LLM responses is to ground them in your own data. Instead of relying only on what the model learned during pretraining, you can implement RAG to retrieve relevant context from a local knowledge base—usually stored in a vector database—and add it to the prompt. RAG has become a core pattern in modern LLM applications.

The workflow begins with building a knowledge base. Documents are ingested, split into smaller chunks, and converted into vector representations using an embedding

model. These embeddings capture semantic meaning, making it possible to compare text by similarity rather than exact wording. LangChain provides tools to load documents in many formats, split them efficiently, generate embeddings, and then store everything in a vector database (see figure 1.9).

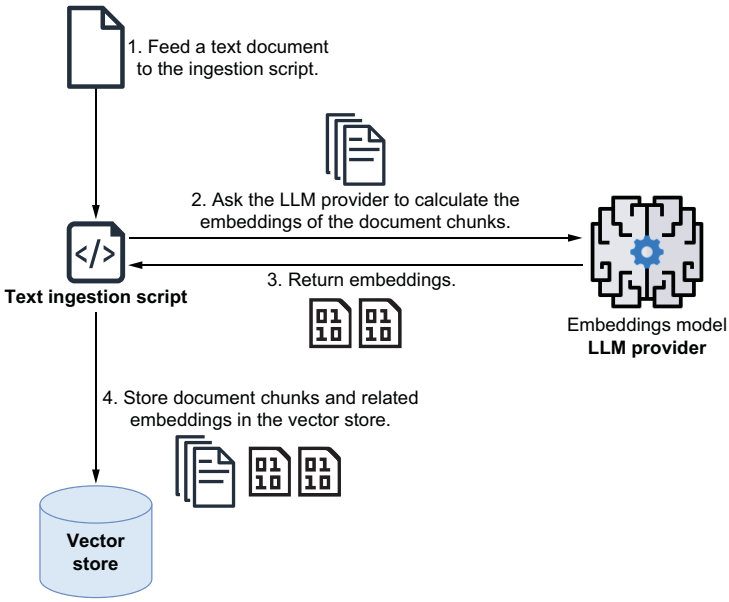


Figure 1.9 A collection of documents is split into text chunks and transformed into vector-based embeddings. Both text chunks and related embeddings are then stored in a vector store.

RAG offers multiple benefits:

- *Efficiency*—Instead of passing an entire document to the model, you retrieve only the key chunks—keeping inputs concise, reducing token costs, and working within context limits.
- *Accuracy*—Responses are *grounded* on real data, reducing the risk of *hallucinations*. Later in the book, you'll learn techniques for having the LLM cite its sources, further improving transparency and trust.
- *Flexibility*—By swapping embedding models, retrievers, or vector stores, you can adapt the same pattern to different domains and requirements.

DEFINITION *Grounding* an LLM involves crafting prompts that include context pulled from a trusted knowledge source—often stored in a vector store. This ensures that the LLM generates its response based on verified facts rather than relying solely on its pretrained knowledge, which may include outdated or unreliable data.

> **DEFINITION** A *hallucination* occurs when an LLM generates an incorrect, misleading, or fabricated response. This happens when the model draws from poor-quality data during training or lacks sufficient information to answer accurately. Due to their auto-regressive nature, LLMs will try to generate a response even when relevant content is missing—leading to hallucinations as they fill in the gaps with plausible-sounding but incorrect information.

To make RAG reliable, prompts should explicitly instruct the LLM to rely only on the retrieved context. LangChain also supports guardrails and validators to help enforce safe behavior. In high-stakes cases, human-in-the-loop review remains the best safeguard.

In short, RAG bridges the gap between static pretrained models and dynamic, domain-specific applications. By teaching your app to "speak the same language" as the LLM—vectors—you unlock a practical way to deliver grounded and cost-efficient answers. If prompt engineering and RAG still don't meet your needs, the next step is fine-tuning the model, which we'll cover in the next section. We'll return to RAG in depth and explore advanced patterns and techniques that extend its power for real-world applications.

1.4.3 Fine-tuning

Fine-tuning is the process of adapting a pretrained LLM to perform better in a specific task or domain. This is done by training the model on a curated dataset of examples that capture the style, terminology, and reasoning patterns you want it to master. Traditionally, fine-tuning required specialized machine learning frameworks and access to powerful hardware, but many platforms today—including OpenAI's fine-tuning API and several open source toolkits—make it possible with just a dataset upload and minimal configuration.

The main benefit of fine-tuning is efficiency: once a model has absorbed domain-specific knowledge, you don't need to stuff every prompt with long instructions or examples. Instead, the model "knows" how to respond in your context. The tradeoffs, however, are real. Preparing high-quality datasets takes time and expertise, and training runs can be costly because they often require GPUs.

Recent advances such as Low-Rank Adaptation (LoRA) and other parameter-efficient fine-tuning methods have lowered both cost and complexity, making fine-tuning more accessible. Related techniques, such as instruction tuning and *Reinforcement Learning from Human Feedback (RLHF)*, push models to follow directions more reliably, though they typically demand more engineering effort and infrastructure.

Whether fine-tuning is truly necessary is an ongoing debate. General-purpose LLMs are surprisingly strong out of the box, especially when paired with RAG. In fact, the research in "Fine-Tuning vs. RAG for Less Popular Knowledge" (by Heydar Soudani, Evangelos Kanoulas, and Faegheh Hasibi; https://arxiv.org/abs/2403.01432)

shows that RAG often outperforms fine-tuning by letting you provide context dynamically at runtime, reducing both costs and retraining needs.

That said, in highly specialized domains—such as medicine, law, or finance—fine-tuning remains invaluable. It allows models to capture domain-specific jargon and workflows in ways generic models struggle to match. Well-known examples include the following:

- BioMistral (biology and life sciences)
- LexiGPT (LexisNexis's legal-domain LLM)
- BloombergGPT (finance)
- Anthropic's Claude Code and similar code-focused models

In summary, fine-tuning customizes an LLM for domain expertise and specialized accuracy, but it comes at a cost in time, money, and complexity. As a developer, you'll need to weigh when it's truly worth it versus when RAG or prompt engineering will get you there faster. That said, we won't cover LLM fine-tuning in this book, as our focus is on building AI agents and applications—not on creating or modifying the models themselves.

1.5 Which LLMs to choose

When developing LLM-based applications, you'll find a wide range of models to choose from. Some are proprietary, accessible via subscription or pay-as-you-go APIs, while others are open source and can be self-hosted. Most modern LLMs offer REST APIs for easy integration and user-friendly chat interfaces. Many also come in different size variants, letting you choose the right balance of performance, speed, and cost.

LangChain makes it simple to integrate with different LLMs. Thanks to its standardized interface, you can switch models with minimal code changes—an essential feature in today's fast-evolving LLM landscape.

Following are the three major factors to weigh when selecting a model:

- *Accuracy*—Generally, larger models are more accurate because they've been trained on larger datasets. However, they're also slower and more expensive at inference time, as vendors charge a premium per million tokens.
- *Speed (latency)*—Smaller models respond faster. This makes them attractive for interactive applications such as chatbots where responsiveness is critical—but the tradeoff is lower accuracy.
- *Cost*—Inference cost is directly tied to model size. Bigger, more accurate models carry higher per-token charges, while smaller models are cheaper to use at scale.

In practice, you'll need to strike a balance across these three dimensions. The "best" model is rarely the largest or the cheapest; it's the one that matches your application's

requirements. A customer support chatbot, for example, might favor speed and cost, while a legal document analysis tool would prioritize accuracy even at higher expense. Beyond these core tradeoffs, there are additional considerations that can make one model family more suitable than another:

- *Model purpose*—For general tasks such as summarization, translation, or classification, most mainstream models (GPT, Gemini, Claude, Llama, Mistral) work well. For specialized needs such as code generation, choose models fine-tuned for that domain—for instance, Meta's Code Llama or Claude Code.
- *Context window size*—A larger context window allows processing of longer prompts and documents. Some models support millions of tokens, while many standard APIs cap at 128K to 256K. Larger windows expand possibilities but can also increase both cost and processing overhead.
- *Multilingual support*—If your application must handle multiple languages, look for models with strong multilingual training. Qwen and Llama are noted for broad coverage, while certain Gemma releases specialize in Asian languages.
- *Instruction versus reasoning models*—Instruction models (e.g., GPT-5-mini, Gemini Pro) excel when you know exactly what steps to follow; reasoning models (e.g., GPT-5 Thinking, Gemini Thinking) are designed to figure out how to solve a problem. The tradeoffs here are similar to accuracy/speed/cost: reasoning models are more powerful but slower and more expensive.
- *Open source versus proprietary*—Open source models (Llama, Mistral, Qwen, Falcon) give you privacy, control, and deployment flexibility. Proprietary APIs, on the other hand, are easier to set up and often provide best-in-class results with less effort, but can become costly in the long run.

By weighing these factors, you can select an LLM that fits your project's goals and constraints. In many applications, the most effective strategy is to use different models for different tasks. For example, one workflow might use GPT-5-nano for fast summarization, GPT-5-mini for answer synthesis, and full GPT-5 for routing complex queries—balancing accuracy, speed, and cost across the system.

1.6 What you'll learn from this book

If you've read this far, you already recognize the value of building applications powered by LLMs. LLM interaction is rapidly becoming a core feature of modern software—similar to how websites, in the early 2000s, opened up a new communication channel for server-based applications. This shift is only accelerating.

We'll begin with prompt engineering, the foundational skill for effective interaction with LLMs. You'll start by experimenting interactively with ChatGPT and then move to programmatic access via REST APIs. These exercises will establish the backbone for many projects in this book.

From there, we'll use LangChain to build two categories of applications: custom engines (e.g., summarization and Q&A systems) and chatbots that combine

conversational fluency with knowledge retrieval. Each project will be small and self-contained, but all will share a common background theme in the travel industry, so you can see how the ideas fit together in a coherent domain.

Once those foundations are in place, we'll move to LangGraph to build AI agents—applications that can orchestrate multi-step workflows, coordinate tools, and make adaptive decisions. This will be your introduction to the most advanced class of LLM-powered systems, where engines and tools come together into dynamic, autonomous applications. To make this approachable, I'll start with a simple Python script and then extend it step-by-step. Each extension will branch into a different capability—such as tool use, planning, or memory—so you can see how agents grow in sophistication without being overwhelmed all at once.

We'll also take a deep dive into RAG, the architectural pattern behind many real-world systems. RAG concepts will be introduced through short, focused scripts—some independent, others progressively refined into more advanced workflows—so you can master both fundamentals and cutting-edge techniques.

Although the examples reference OpenAI models for accessibility and quick wins, you'll also learn how to use open source models through the inference engines listed in appendix E. This way, you'll gain the skills to build and deploy applications that are cost-effective, privacy-conscious, and fully under your control.

Beyond building, you'll explore the entire life cycle of LLM applications: debugging, monitoring, and refining with LangSmith; orchestrating complex workflows with LangGraph; and applying best practices for production deployment to ensure scalability and maintainability.

By the end of this book, you'll have built a portfolio of working applications, learned the core architectural patterns, and developed the skills to design and implement LLM-powered systems with confidence. You won't just understand how these applications work—you'll be ready to keep innovating and pushing the boundaries of what's possible with LangChain, LangGraph, and LLMs.

Summary

- LLM-based systems come in three flavors: engines (specific tasks, e.g., summarization), chatbots (back-and-forth dialogue with memory), and agents (autonomous multi-step execution with tool selection).
- Embeddings convert text into mathematical vectors that capture meaning. Both text chunks and their embeddings are stored in vector databases for efficient similarity-based retrieval.
- Retrieval-Augmented Generation (RAG) enhances LLMs by pulling relevant information from your knowledge base before generating answers. This grounds responses in actual documents rather than relying solely on training data.
- LangChain packages common patterns into reusable components based on three principles: modularity (swap implementations easily), composability

(chain components via LangChain Expression Language [LCEL]), and extensibility (add custom loaders, retrievers, or tools).

- Documents are LangChain's basic building blocks that package text with metadata. Document loaders pull content from sources (PDFs, web pages, databases) into standardized `Document` objects.
- LangChain's core components include `BaseLoader` (extract content into documents), `TextSplitters` (chunk large texts), `Embedding` models (convert text to vectors), `VectorStores` (store and retrieve embeddings), `BaseRetriever` (query backends), and `BaseLanguageModel` (unified LLM interface).
- The `Runnable` interface makes all LangChain components work together smoothly through LCEL. Components can be chained using the pipe operator (`|`) to build complex workflows.
- Prompt engineering crafts inputs that help models understand what you want. Few-shot prompting includes examples in the prompt to teach the model desired patterns.
- Balance three dimensions when selecting models: accuracy (larger models trained on more data), speed (smaller models respond faster), and cost (bigger models charge more per token). Customer support chatbots favor speed and cost; legal analysis tools prioritize accuracy.
- Use different models for different tasks in the same workflow: for example, GPT-5-nano for fast summarization, GPT-5-mini for answer synthesis, and full GPT-5 for routing complex queries. This optimizes the accuracy/speed/cost tradeoff across your system.
- Choose between RAG and fine-tuning: RAG provides dynamic context at runtime (reducing costs and retraining needs), while fine-tuning embeds domain knowledge into the model (improving efficiency for specialized tasks).

2 Executing prompts programmatically

This chapter covers

- Prompts and prompt engineering
- Different kinds of prompts and how they're structured
- Enhancing prompt responses using one-, two-, or few-shot learning
- Examples of using prompts with ChatGPT and the OpenAI API

AI applications interact with large language models (LLMs) mainly through *prompts*—structured inputs that guide the model's behavior. It's a bit like giving directions to a talented but inexperienced colleague: the clearer and more specific you are, the better the results. To get accurate and relevant outputs, prompts need to be carefully crafted and tailored to the task. In practice, prompt design is one of the biggest factors in how well your application performs.

Prompt engineering—the practice of designing and refining prompts to guide an LLM's output—is a core skill in building LLM applications. You'll spend much of your time creating, testing, and iterating on prompts to make sure your system delivers reliable, high-quality results.

In this chapter, you'll begin with the basics of prompt design and gradually move to more sophisticated techniques, such as Chain of Thought (CoT). LangChain's suite of prompt engineering tools, including `PromptTemplate` and `FewShotPromptTemplate`, will be your key resource as you learn to harness the full power of LLMs in your applications.

2.1 Running prompts programmatically

LLM applications rely on well-crafted prompts to generate completions, which are then passed to the next component in the chain. Unlike prompts entered manually in interfaces such as ChatGPT, LangChain prompts are typically constructed and sent to the LLM programmatically as part of a larger workflow. The following sections start by setting up and executing prompts using the OpenAI API directly, and then we'll explore how to run and manage prompts within LangChain. Before diving in, make sure you've covered these basics:

- You have an OpenAI API key.
- You know how to create a Python Jupyter Notebook environment.

If you're not familiar with these tasks, check appendix A for guidance, and follow the instructions in appendix B on setting up a Jupyter Notebook environment.

2.1.1 Setting up the environment for this chapter

Assuming you've completed the prerequisites, this section will guide you through setting up a Jupyter Notebook environment for prompt engineering, which you'll use for all examples in this chapter. Throughout this chapter—and the rest of the book—I'll demonstrate how to set up Jupyter on Windows. If you're using a different operating system, refer to appendix B for platform-specific instructions.

Open your operating system's terminal (in this case, the Windows Command Prompt), and create a new project folder named `ch02`. Alternatively, you can clone the GitHub repository associated with this book: https://mng.bz/YZma. Then, navigate to the `ch02` folder:

```
C:\Github\building-llm-applications\ch02>
```

Create and activate a virtual environment with `venv`:

```
C:\Github\building-llm-applications\ch02>python -m venv env_ch02
C:\Github\building-llm-applications\ch02>.\env_ch02\Scripts\activate
```

You should now observe the updated operating system prompt, displaying the environment name in front as `(env_ch02)`:

```
(env_ch02) C:\Github\building-llm-applications\ch02>
```

Having activated the virtual environment, you're now prepared to implement a Jupyter Notebook for executing prompts with OpenAI models. If you cloned the repository

from GitHub (https://mng.bz/YZma) or downloaded the code as a zip file from the Manning website, you can install Jupyter and the OpenAI packages as follows:

```
(env_ch02) C:\Github\building-llm-applications\ch02>
➥pip install -r requirements.txt
```

Then, you can start the notebook:

```
(env_ch02) C:\Github\building-llm-applications\ch02>
➥jupyter notebook 02-prompt_examples.ipynb
```

If you decide to build everything locally from scratch, make sure to install the same package versions listed in `requirements.txt`—this will help you avoid version conflicts later on. For the remainder of the book, I'll assume you've either cloned the project repository from GitHub or downloaded the source code from the Manning website.

After about a minute, the installation of the notebook and OpenAI packages should be finished. You can start the Jupyter Notebook by executing the following command:

```
(env_ch02) C:\Github\building-llm-applications\ch02>jupyter notebook
```

After a few seconds, you should see some output in the terminal. Subsequently, a browser window will open at `http://localhost:8888/tree`. Create the notebook by choosing File > New > Notebook, as shown in figure 2.1, and then rename the file `prompt_examples.ipynb`.

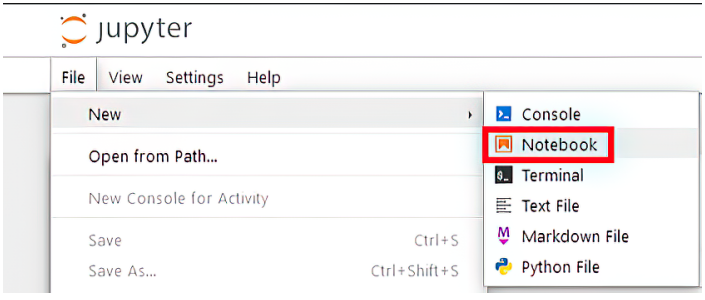


Figure 2.1 Creating a new Jupyter Notebook

Once you've created the notebook, go to the Jupyter menu, select File > Rename, and name the notebook file `02-prompt_examples.ipynb`. Now you're prepared to input code into the notebook cells (or simply execute them if you got the notebook from GitHub).

TIP If you're unfamiliar with Jupyter Notebook, remember to press Shift-Enter to execute the code in each cell.

Throughout the rest of the chapter, I assume you've set up a virtual environment, installed the OpenAI library, and launched a Jupyter Notebook instance, as outlined in this section.

2.1.2 Minimal prompt execution

In the first cell of your notebook, import the required libraries, and grab the OpenAI API key in a secure way (never hardcode the key, as it might get misused):

```
from openai import OpenAI
import getpass

OPENAI_API_KEY = getpass.getpass('Enter your OPENAI_API_KEY')
```

After entering your OpenAI API key (you only need to press Enter, without Shift), set up the OpenAI client as follows:

```
client = OpenAI(api_key=OPENAI_API_KEY)
```

In the next notebook cell, enter and execute the following code:

```
prompt_input = """Write a concise message to remind
users to be vigilant about phishing attacks."""
response = client.chat.completions.create(
  model="gpt-5-nano",
  messages=[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": prompt_input}
  ]
)

print(response)
```

> **NOTE** To keep execution costs low, we use GPT-5-nano, the smallest and most affordable model in the GPT-5 family. You'll see it used throughout most of this book. However, you're welcome to switch to GPT-5-mini or GPT-5 if you prefer a higher accuracy level. You can learn more about the OpenAI models at https://platform.openai.com/docs/models.

The output will look like this (though it may vary slightly due to the nondeterministic nature of LLMs):

```
ChatCompletion(id='chatcmpl-CFkN43Xs80ohhDJVIDeRUSID8RXNo', choices=[Choice(
finish_reason='stop', index=0, logprobs=None, message=ChatCompletionMessage(
content='Be vigilant against phishing: verify the sender, hover over links to
check URLs, don't click suspicious attachments or share passwords, and report
anything suspicious to IT.', refusal=None, role='assistant', annotations=[],
audio=None, function_call=None, tool_calls=None))], created=1757869206, model
='gpt-5-nano-2025-08-07', object='chat.completion', service_tier='default',
system_fingerprint=None, usage=CompletionUsage(completion_tokens=553,
prompt_tokens=32, total_tokens=585, completion_tokens_details=
CompletionTokensDetails(accepted_prediction_tokens=0, audio_tokens=0,
reasoning_tokens=512, rejected_prediction_tokens=0),
prompt_tokens_details=PromptTokensDetails(audio_tokens=0, cached_tokens=0)))
```

For a clearer output, explore the attributes and properties of the response object:

```
print(response.choices[0].message.content)
```

You'll see an output similar to the following:

```
Be vigilant against phishing: verify the sender, hover over links to check
URLs, don't click suspicious attachments or share passwords, and report
anything suspicious to IT.
```

> **NOTE** If you're not familiar, the three double quotes (`"""`) I've used to formulate the prompt allow you to enter a block of text spanning multiple lines in a very readable way without having to introduce newline characters (`\n`), which is ideal for capturing prompts.

The chat.completions.create function

The `chat.completions.create` function, part of the OpenAI Completions API, is the primary way to interact with OpenAI's LLMs. Understanding its signature is key:

```
client.chat.completions.create(
  model="gpt-5-nano",
  messages=[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": prompt_input}
  ]
)
```

Here's a description of the parameters:

- `model`—This refers to the OpenAI model you want to use. Options range from GPT-5, the most accurate but also the slowest and most expensive, to GPT-5-nano, the fastest and cheapest, though less accurate for some tasks. GPT-5-mini provides a solid middle ground, balancing speed, accuracy, and cost.
- `messages`—The prompt is expressed as a list of messages, each with a role (e.g., system, user, or assistant) and some content containing the instructions or text to be followed. This format follows the OpenAI convention for structuring conversational input.

Although the Responses API is now OpenAI's preferred interface—especially for agent-based workflows—this book primarily uses the Completions API, often accessed indirectly through the `ChatOpenAI` wrapper in LangChain. The reason is that Completions are supported not only by OpenAI's models but also by many open source LLMs, giving you more flexibility. We'll return to the Responses API in chapter 11 when we dive into AI agents.

Now that you've grasped the fundamentals of programmatically submitting prompts, you're ready to learn and work with more intricate prompts.

2.2 Running prompts with LangChain

Before moving to more advanced prompts, let me show you how to replicate the same example in LangChain to help you become familiar with its object model. In the following sections, I'll demonstrate how LangChain simplifies the implementation of more complex prompts that would be more challenging to implement from scratch.

If you followed the instructions in section 2.1 to install the required Python packages, you should already have the `langchain` and `langchain-openai` packages installed. You're now ready to use these libraries and instantiate a connection to the LLM:

```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(openai_api_key=OPENAI_API_KEY,
                 model_name="gpt-5-nano")
```

You can now instantiate and execute the prompt you saw earlier as follows:

```
prompt_input = """Write a concise message to remind
users to be vigilant about phishing attacks."""


response = llm.invoke(prompt_input)
print(response.content)
```

This will return output similar to the following:

```
Stay vigilant against phishing: verify the sender, hover over links to check
URLs before clicking, never share passwords, and report suspicious messages.
```

2.3 Prompt templates

When building LLM applications, creating flexible prompt templates that incorporate user input through parameters is crucial. LangChain makes this easier with its `PromptTemplate` class, which lets you manage and reuse parametrized prompts without the need for custom functions. This not only streamlines the creation of dynamic prompts but also enhances the efficiency and adaptability of your LLM interactions, as I'll demonstrate shortly.

2.3.1 Implementing a prompt template with a Python function

To illustrate the template concept, let's create a text summarization template that requests the text, desired summary length, and preferred tone. You could implement this using a simple Python function:

```
def generate_text_summary_prompt(text, num_words, tone):
    return f"You are an experienced copywriter. Write a
    ➥{num_words} words summary of the following text, using a
    ➥{tone} tone: {text}"
```

Let's use the prompt template to generate a prompt and then execute it through LangChain's `ChatOpenAI` wrapper as usual:

```
segovia_aqueduct_text = """The Aqueduct of Segovia (Spanish:
Acueducto de Segovia) is a Roman aqueduct in Segovia, Spain.
It was built around the first century AD to channel water from
springs in the mountains 17 kilometres (11 mi) away to the
city's fountains, public baths and private houses, and was in
use until 1973.
```

```
Its elevated section, with its complete arcade of 167 arches,
is one of the best-preserved Roman aqueduct bridges and the
foremost symbol of Segovia, as evidenced by its presence on the
city's coat of arms.
The Old Town of Segovia and the aqueduct, were declared a UNESCO
World Heritage Site in 1985. As the aqueduct lacks a legible
inscription (one was apparently located in the structure's attic,
or top portion[citation needed]), the date of construction cannot be
definitively determined. The general date of the Aqueduct's
construction was long a mystery, although it was thought to have
been during the 1st century AD, during the reigns of the Emperors
Domitian, Nerva, and Trajan. At the end of the 20th century,
Géza Alföldy deciphered the text on the dedication plaque by
studying the anchors that held the now missing bronze letters
in place. He determined that Emperor Domitian (AD 81–96) ordered
its construction[1] and the year 98 AD was proposed as the most
likely date of completion.[2] However, in 2016 archeological
evidence was published which points to a slightly later date,
after 112 AD, during the government of Trajan or in the
beginning of the government of emperor Hadrian,
from 117 AD."""

input_prompt = generate_text_summary_prompt(
    text=segovia_aqueduct_text,
    num_words=20,
    tone="knowledgeable and engaging")

response = llm.invoke(input_prompt)
print(response.content)
```

You should see output similar to the following:

```
The Aqueduct of Segovia, built in the 1st century AD, is a well-preserved
Roman structure and a UNESCO World Heritage Site.
```

2.3.2 Using LangChain's PromptTemplate

With LangChain, you don't need to implement a prompt template function manually. Instead, you can use the convenient `PromptTemplate` class to handle parametrized templates. Here's how you can use it:

```
from langchain_core.prompts import PromptTemplate

prompt_template = PromptTemplate.from_template(
"""You are an experienced copywriter.
Write a {num_words} words summary of the following text,
using a {tone} tone: {text}""")
```

To use the prompt template, create a prompt instance, and format it with your parameters:

```
prompt = prompt_template.format(
    text=segovia_aqueduct_text,
```

```
    num_words=20,
    tone="knowledgeable and engaging")
```

Then, invoke the `ChatOpenAI` client with the formatted prompt:

```
response = llm.invoke(prompt)
print(response.content)
```

This will generate output similar to what's shown here (as you now understand, the exact output may differ from what's printed in the book, so I won't repeat this point on the next page):

```
The Aqueduct of Segovia, a Roman marvel built in the 1st century AD, channels
water to Segovia's fountains and baths.
```

Let's take a short break from coding with LangChain. I want to delve deeper into prompt engineering. In the upcoming pages, you'll see that a good prompt can have various elements, depending on your goal, task complexity, and desired accuracy. By learning from different prompt variations, you'll be able to handle both simple and complex prompts, adjusting their complexity to fit specific cases.

2.4 Prompt types

Creating effective prompts is crucial for getting the best results in LangChain applications. Whether your app is focused on text classification, sentiment analysis, summarization, text generation, or question answering, each task requires a carefully designed prompt. In the following sections, we'll dive into how to craft prompts tailored to these specific tasks, ensuring your application functions as intended.

Although these prompting techniques are key to developing LangChain applications, we'll use ChatGPT for now to keep things simple and explain prompt engineering concepts more clearly. The patterns you learn here will directly apply to the LLM apps we build with LangChain in the upcoming chapters.

2.4.1 Text classification

In classification, the goal is to assign an input text to one of several predefined categories. The following prompt demonstrates this concept—try entering it into ChatGPT, which we'll use throughout this section:

INSTRUCTION Classify the following text into one of these categories: history, tech, gardening.

TEXT Headphones provide immersive audio experiences for music lovers and gamers alike.

> " The text "Headphones provide immersive audio experiences for music lovers and gamers alike." should be classified as "tech" because it discusses technology-related products and their functionality.

The output is overly detailed. Let's revise the prompt and specify to display the category only.

INSTRUCTION Classify the following text into one of these categories: history, tech, gardening.
TEXT Headphones provide immersive audio experiences for music lovers and gamers alike.
OUTPUT only the category

" Tech

Now categorize another sentence by applying the same prompt and only altering the text:

INSTRUCTION Classify the following text into one of these categories: history, tech, gardening.
TEXT Julius Caesar was a Roman general and statesman who played a pivotal role in the demise of the Roman Republic and the rise of the Roman Empire.
OUTPUT only the category

" History

In summary, in a standard text classification prompt, you find three components: an Instruction, the input Text, and an Output specification. Next, let's delve into a slightly specialized text classification: sentiment analysis.

2.4.2 Sentiment analysis

Sentiment analysis is a specific type of text classification that aims to determine whether a given text is perceived as positive, neutral, or negative. Following are three sample prompts (and expected responses) that you can experiment with independently:

INSTRUCTION Classify the following text as positive, neutral or negative
TEXT Apple has announced stellar quarterly results

" Positive

INSTRUCTION Classify the following text as positive, neutral or negative
TEXT this is the worst movie I have watched this month

" Negative

INSTRUCTION Classify the following text as positive, neutral or negative
TEXT The customer bought a green dress

" Neutral

If you have a list of texts and want to analyze their sentiments collectively, you can input them all into a single prompt:

INSTRUCTION classify the sentiment of following stock reports as positive, neutral or negative
Stock 1: Apple: the launch of the new iPhone has been a success
Stock 2: Nvidia: sales are being propelled by consumer demand on LLMs
Stock 3: GX oil: demand of carbon based energy is dropping thanks to the increase of renewable energy

OUTPUT output a table with the columns "stock name", "sentiment"

"
```
| Stock Name  | Sentiment |
|-------------|-----------|
| Apple       | positive  |
| Nvidia      | positive  |
| GX oil      | negative  |
```

As you can see, in this scenario, the output is formatted in a table and can be readily processed programmatically. Next, let's create a prompt for one of the most common LLM use cases: text summarization.

2.4.3 Text summarization

Creating a *text summarization* prompt is simple: you only need to ask for a summary of an input text with your preferred length.

INSTRUCTION write a 30 word summary for the following text

TEXT Home PCs from the 1980s were iconic pioneers of personal computing. Computers like the Apple II, IBM PC, and Commodore 64 revolutionized how we lived and worked. They introduced us to word processing, gaming, and the possibilities of home computing. The graphical user interface, as seen in the Apple Macintosh, changed the way we interacted with machines. These PCs, though less powerful by today's standards, ignited a digital revolution, shaping the technology landscape we know today. They hold a special place in history, symbolizing the dawn of a new era in computing that laid the foundation for the modern digital age.

" 1980s home PCs like Apple II, IBM PC, Commodore 64 pioneered personal computing, introducing word processing, gaming, and GUIs. They shaped today's tech, symbolizing a digital era's dawn.

You can also ask the LLM to provide a summary for a text it's already familiar with:

INSTRUCTION Write a one sentence summary of the Lord of the Rings

" "The Lord of the Rings" follows the epic journey of a diverse group of characters as they seek to destroy a powerful ring and save Middle-earth from the dark lord Sauron.

2.4.4 Composing text

LLMs are part of the AI category called *generative AI*. While LLMs can handle audio and video, when prompted with text, they can generate new content, like a paragraph or sentence, based on a list of facts, for example:

Write a piece on the diver watches, mentioning the following facts:

- the Rolex Oyster Perpetual was among the first to be water resistant
- Omega produced the Seamaster in 1948 as the first purpose designed diver watch was the Seamaster
- the Seamaster set a world record of immersion of 62.5m in 1955
- Rolex produced the first Submariner in 1953

- the Submariner became popular also because of the association with James Bond until 1995
- James Bond started to wear the Seamaster in 1995

> “ The Rolex Oyster Perpetual, introduced in the early 20th century, played a pioneering role in the evolution of diver watches. Among its notable features was its remarkable water-resistant design, which set the stage for the development of truly waterproof timepieces. This innovation marked a crucial step in ensuring that watches could withstand the rigors of underwater exploration.
>
> Omega, a respected Swiss watchmaker, made its mark on the diver watch scene in 1948 with the introduction of the Seamaster. This timepiece was a game-changer as it was purposefully designed for divers, combining style and functionality. The Seamaster didn’t stop at design; it also showcased its capabilities in the depths. In 1955, it set a remarkable world record for immersion, descending to an astonishing 62.5 meters underwater.
>
> Not to be outdone, Rolex joined the fray by producing the first Submariner in 1953. This iconic watch quickly became synonymous with underwater exploration and adventure. Its success was further boosted by its association with the fictional British spy, James Bond. However, the tides of popularity shifted in 1995 when James Bond, portrayed by Pierce Brosnan, made a bold move by strapping on an Omega Seamaster in “GoldenEye.” This marked the beginning of a new era for the Seamaster, as it became the watch of choice for the world’s most famous secret agent.”

You may notice that ChatGPT has used the limited information to create a quite lengthy article. On the other hand, you might have expected a shorter piece and may not be a fan of the complex vocabulary or somewhat formal tone it used. In such a situation, try restructuring the prompt more systematically to get the generated text to match your preferred length, vocabulary, and tone:

PERSONA you’re a watch enthusiast, experienced in writing high quality articles for watch magazines for a general audience

INSTRUCTION Write a piece on the diver watches

Facts to mention:

- the Rolex Oyster Perpetual was among the first to be water resistant
- Omega produced the Seamaster in 1948 as the first purpose designed diver watch was the Seamaster
- the Seamaster set a world record of immersion of 62.5m in 1955
- Rolex produced the first Submariner in 1953
- the Submariner became popular also because of the association with James Bond until 1995
- James Bond started to wear the Seamaster in 1995

TONE use a sober, knowledgeable but not too formal tone and use everyday vocabulary

OUTPUT the length of the article should be 150 words at most

> “ **Diver Watches: Timepieces of Exploration and Style**
>
> Diver watches have long been cherished for their blend of functionality and aesthetics, a perfect companion for both the adventurous spirit and the style-conscious.

Rolex deserves recognition for pioneering water resistance with the Oyster Perpetual, setting the stage for a new era in watchmaking.

Omega, however, made a significant splash in 1948 with the Seamaster, the world's first purpose-designed diver watch. Its 1955 record dive of 62.5 meters underlines its prowess in the deep.

Rolex didn't lag behind, introducing the Submariner in 1953, a watch that became synonymous with exploration, partly due to its affiliation with James Bond on-screen.

Speaking of James Bond, it's worth noting that in 1995, the iconic secret agent started sporting the Omega Seamaster, a move that reinforced the Seamaster's reputation as a sophisticated yet rugged timepiece for those who dare to explore both land and sea. These diver watches are more than instruments; they're symbols of adventure and style.

By adding more details to the prompt, the article has been generated exactly as you wanted it.

NOTE Creating a well-structured prompt with defined sections (Persona, Instruction, Facts, Tone, Output Format) helps the LLM understand your requirements better and generate the expected text.

2.4.5 Question answering

The ability of LLMs to "understand" text makes them suitable not only for summarizing but also for retrieving specific information, known as the *question answering* use case. Now let's go back to the same text we used in the summarization example and ask focused questions about its content.

INSTRUCTION read the following text, and tell me which home computers were popular in the 80s.

TEXT Home PCs from the 1980s were iconic pioneers of personal computing. Computers like the Apple II, IBM PC, and Commodore 64 revolutionized how we lived and worked. They introduced us to word processing, gaming, and the possibilities of home computing. The graphical user interface, as seen in the Apple Macintosh, changed the way we interacted with machines. These PCs, though less powerful by today's standards, ignited a digital revolution, shaping the technology landscape we know today. They hold a special place in history, symbolizing the dawn of a new era in computing that laid the foundation for the modern digital age.

OUTPUT just output the names of the computers in a list

" The popular home computers in the 1980s were:

1. Apple II
2. IBM PC
3. Commodore 64
4. Apple Macintosh

Question answering is such a common use case that most LLMs understand Q and A as shorthand for Question and Answer. For example:

Text Java is a popular programming language that compiles code into bytecode, which is executed by the Java Virtual Machine (JVM) for platform-independent application development.

Q: Where is Java code executed?

A:

“ Java code is executed by the Java Virtual Machine (JVM).

LLMs are skilled not just at finding requested information in a text but also at making logical deductions. This will become apparent in the next section.

2.4.6 Reasoning

While reasoning is a powerful capability, it’s an area where LLMs are still improving. Therefore, you might need more advanced prompt engineering techniques to get the results you want. Here’s a simple example involving mathematical reasoning:

INSTRUCTION Add the square numbers in this sequence: 19, 13, 1, 17, 4, 64, 900

“ The square numbers in the sequence 19, 13, 1, 17, 4, 64, and 900 are 1, 4, 64, and 900. To find their sum:

1 + 4 + 64 + 900 = 969

So, the sum of the square numbers in this sequence is 969.

ChatGPT identified the square numbers, picked them out, and then calculated their total. The next example is more challenging, involving palindromes—numbers that read the same forward and backward, such as 1661 or 27872:

INSTRUCTION Sum the palindromes in this sequence: 13, 1331, 121, 73, 99, 56, 232, 7

I got this output:

“ The palindromes in the sequence 13, 1331, 121, 73, 99, 56, 232, and 7 are 1331, 121, 232, and 7. To find their sum:

1331 + 121 + 232 + 7 = 1691

So, the sum of the palindromes in this sequence is 1691.

This is incorrect! A palindrome is a number that reads the same forward and backward. Therefore, 99 should have been considered a palindrome, but it wasn’t.

I’ll be frank with you: this output is what I encountered in the early stages of writing this book. However, you should now see a different and, more importantly, accurate output. It’s still valuable to explain how I corrected ChatGPT’s mistake before OpenAI addressed this in a later version of the model.

So how did I fix it? Because the earlier LLM version didn’t fully grasp the concept of a palindrome, I “taught” ChatGPT to correctly identify all relevant cases, as I’ll

explain in the next section. Before moving on, take a moment to review the types of prompts you've learned as summarized in table 2.1.

Table 2.1 Prompt types and examples

Prompt type	Prompt example
Text classification	Classify the following text into one of these categories: history, tech, gardening.
Sentiment analysis	Classify the following text as positive, neutral, or negative.
Text summarization	Write a 30-word summary for the following text.
Composing text	Write a piece on the diver watches, mentioning the following facts: ▪ The Rolex Oyster Perpetual was among the first to be water resistant. ▪ Omega produced the Seamaster in 1948 as the first purpose designed diver watch was the Seamaster.
Question answering	Read the following text, and tell me which home computers were popular in the 80s.
Reasoning	Add the square numbers in this sequence: 19, 13, 1, 17, 4, 64, 900.

2.5 Reasoning in detail

When developing LLM applications, particularly those requiring complex reasoning, it's important to understand the limitations of different prompt techniques. For instance, *zero-shot learning* is a method in which you provide no examples, expecting the LLM to grasp and perform the task based purely on the instructions given. While this approach can work well in some scenarios, it doesn't always yield accurate results for more intricate tasks, such as teaching an LLM to identify palindromes. In the following sections, I'll demonstrate how to go beyond zero-shot learning to refine LLM prompts for more reliable performance in reasoning tasks.

2.5.1 One-shot learning

When you provide just one example in the prompt, it's called *one-shot learning*. Here's how I attempted to address the palindrome problem by giving a single example:

INSTRUCTION Sum the palindromes in this sequence: 13, 1331, 121, 73, 99, 56, 232, 7
EXAMPLE 33 is a palindrome

> " I apologize for the previous oversight. Let's correctly identify and sum the palindromes in this sequence: 13, 1331, 121, 73, 99, 56, 232, and 7.
>
> The palindromes in the sequence are 1331, 121, 232, and 7. To find their sum:
>
> 1331 + 121 + 232 + 7 = 1691
>
> So, the sum of the palindromes in this sequence is indeed 1691.

Unfortunately, even after using one-shot learning, the issue persisted. So I adjusted the prompt and included two examples.

2.5.2 Two-shot learning

Here is the amended prompt I used, followed by the output it produced. This output illustrates how the model interpreted the revised instructions and applied them within the context of this two-shot learning example.

INSTRUCTION Sum the palindromes in this sequence: 13, 1331, 121, 73, 99, 56, 232, 7
Examples:
33 is a palindrome
44 is a palindrome

" Certainly, let's identify and sum the palindromes in the sequence: 13, 1331, 121, 73, 99, 56, 232, and 7.
The palindromes in this sequence are 1331, 121, 232, and 7. To find their sum:

1331 + 121 + 232 + 7 = 1691

So, the sum of the palindromes in this sequence is 1691.

Still incorrect: 99 got overlooked once again. Following another unsuccessful attempt, I explored different approaches and found a standard solution. It suggested breaking down the problem into steps and providing clear instructions for each step. This is similar to guiding a colleague who may be struggling with a specific step in a procedure.

2.5.3 Providing steps

This is the amended prompt I entered, along with the resulting output it generated. This output demonstrates how the model followed the revised instructions and applied the step-by-step reasoning approach required for this example.

INSTRUCTION Sum the palindromes in this sequence: 13, 1331, 121, 73, 99, 56 232, 7
Examples:
33 is a palindrome
44 is a palindrome
Steps: 1) identify the palindromes; 2) add them up

" Certainly, let's follow your instructions step by step:
Step 1: Identify the palindromes in the sequence.
Palindromes in the sequence:

- 1331
- 121
- 99
- 232
- 7

Step 2: Add them up.

1331 + 121 + 99 + 232 + 7 = 1790

So, the sum of the palindromes in the sequence is 1790.

Finally, I nailed it! Interestingly, it appears that ChatGPT has undergone a similar training process (on a much larger scale, of course) to fix this issue. To show this, if you enter the original prompt as

INSTRUCTION Sum the palindromes in this sequence: 13, 1331, 121, 73, 99, 56, 232, 7

you'll now see the following output:

> Let's identify the palindromes in the given sequence:
>
> - 1331 is a palindrome
> - 121 is a palindrome
> - 99 is a palindrome
> - 232 is a palindrome
> - 7 is a palindrome
>
> Now, let's add these palindromes together:
>
> 1331 + 121 + 99 + 232 + 7 = 1790
>
> Therefore, the sum of the palindromes in the sequence is 1790.

It appears that ChatGPT is tackling the problem step-by-step, much like I did. Reasoning models such as GPT-5 Thinking often decompose problems on their own, without being told what steps to follow. However, in more complex cases—especially when the steps are deterministic—it's still useful to spell them out explicitly in the prompt. This ensures the LLM follows the intended sequence rather than inventing its own, which could lead to errors.

I found it interesting to share an example that's no longer relevant. This is something you might experience while working with LLMs: you could invest time in teaching the LLM certain tasks using advanced prompt engineering techniques, only to find that a newer model version can handle the task without any additional training. For instance, OpenAI's GPT-5 model performed reasoning tasks with minimal prompts. These tasks previously required more complex prompts such as the few-shot techniques you're about to learn.

2.5.4 Few-shot learning

In some cases, providing only two examples may not be enough to teach the LLM a new concept. In such instances, you need to provide several examples, known as *few-shot learning*. To demonstrate this, let me create a game where a number can be classified as Abra, Kadabra, or Abra Kadabra. Let's see what happens when no information about the classification rules is provided.

Classify the following numbers as Abra, Kadabra or Abra Kadabra:
3, 5, 8, 2, 7, 12, 13

> The classification of numbers into "Abra," "Kadabra," or "Abra Kadabra" doesn't follow a conventional mathematical pattern. These are just words, not related to

numbers. Therefore, it's not possible to classify the numbers you provided as "Abra," "Kadabra," or "Abra Kadabra" based on any mathematical or logical criteria.

ChatGPT hasn't experienced any hallucinations but rightfully pointed out that you haven't clarified the classification rules. Now I'll show you a prompt that, instead of directly explaining the rules, offers an interpretation based on a specific list of numbers, different from the ones I originally asked to be addressed:

```
Classify the following numbers as Abra, Kadabra or Abra Kadabra:
3, 4, 5, 7, 8, 10, 11, 13, 35
Examples:
6 // not divisible by 5, not divisible by 7 // None
15 // divisible by 5, not divisible by 7 // Abra
12 // not divisible by 5, not divisible by 7 // None
21 // not divisible by 5, divisible by 7 // Kadabra
70 // divisible by 5, divisible by 7 // Abra Kadabra
```

Observe the format I've employed for every example: I've delineated each field with a double forward slash (//), although I haven't explicitly outlined the meaning of each field. I'm trusting the LLM to decipher it. Here's the output:

> Let's classify the numbers as Abra, Kadabra, or Abra Kadabra based on whether they are divisible by 5 and/or 7:
>
> 3: Not divisible by 5, not divisible by 7 - None
> 4: Not divisible by 5, not divisible by 7 - None
> 5: Divisible by 5, not divisible by 7 - Abra
> 7: Not divisible by 5, divisible by 7 - Kadabra
> 8: Not divisible by 5, not divisible by 7 - None
> 10: Divisible by 5, not divisible by 7 - Abra
> 11: Not divisible by 5, not divisible by 7 - None
> 13: Not divisible by 5, not divisible by 7 - None
> 35: Divisible by 5, divisible by 7 - Abra Kadabra
>
> So, here's the classification:
>
> Abra: 5, 10, 35
> Kadabra: 7
> Abra Kadabra: 35
> None: 3, 4, 8, 11, 13

ChatGPT not only provided accurate results but also explained the reasoning behind them. It successfully deduced the general rules from the given examples and articulated them well. Quite impressive, isn't it?

NOTE You may be familiar with a similar number classification game or "divisibility test algorithm," as it's called on Stack Overflow (https://mng.bz/GwzJ), where a number divisible by 3 is labeled "Foo," a number divisible by 5

is labeled "Bar," and a number divisible by both 3 and 5 is labeled "Foo Bar." I haven't used this as an example because ChatGPT already knows it and can correctly classify numbers without examples, using zero-shot learning.

2.5.5 Implementing few-shot learning with LangChain

Let's implement the AbraKadabra game with LangChain. First, establish a connection to the LLM, as you did previously:

```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(openai_api_key=OPENAI_API_KEY,
                 model_name="gpt-5-mini")
```

Now define and run the same few-shot prompt you executed previously directly against ChatGPT:

```
prompt_input = """Classify the following numbers as Abra, Kadabra
➥or Abra Kadabra:

3, 4, 5, 7, 8, 10, 11, 13, 35

Examples:
6 // not divisible by 5, not divisible by 7 // None
15 // divisible by 5, not divisible by 7 // Abra
12 // not divisible by 5, not divisible by 7 // None
21 // not divisible by 5, divisible by 7 // Kadabra
70 // divisible by 5, divisible by 7 // Abra Kadabra
"""

response = llm.invoke(prompt_input)
print(response.content)
```

The output is as expected:

```
3 // not divisible by 5, not divisible by 7 // None
4 // not divisible by 5, not divisible by 7 // None
5 // divisible by 5, not divisible by 7 // Abra
7 // not divisible by 5, divisible by 7 // Kadabra
8 // not divisible by 5, not divisible by 7 // None
10 // divisible by 5, not divisible by 7 // Abra
11 // not divisible by 5, not divisible by 7 // None
13 // not divisible by 5, not divisible by 7 // None
35 // divisible by 5, divisible by 7 // Abra Kadabra
```

The output is accurate, but the implementation isn't ideal because the examples are hardcoded in the prompt. LangChain provides a cleaner solution for creating a few-shot prompt. It lets you separate the training examples from the prompt template and inject them later, as shown in the following listing.

Listing 2.1 Few-shot prompt using `FewShotPromptTemplate`

```
from langchain_core.prompts.few_shot import FewShotPromptTemplate
from langchain_core.prompts.prompt import PromptTemplate
```

```
examples = [
  {
      "number": 6,
      "reasoning": "not divisible by 5 nor by 7",
      "result": "None"
  },
  {
      "number": 15,
      "reasoning": "divisible by 5 but not by 7",
      "result": "Abra"
  },
  {
      "number": 12,
      "reasoning": "not divisible by 5 nor by 7",
      "result": "None"
  },
  {
      "number": 21,
      "reasoning": "divisible by 7 but not by 5",
      "result": "Kadabra"
  },
  {
      "number": 70,
      "reasoning": "divisible by 5 and by 7",
      "result": "Abra Kadabra"
  } ]

example_prompt = PromptTemplate(input_variables=["number",
➥"reasoning", "result"],
➥template="{number} \\ {reasoning} \\ {result}")
few_shot_prompt = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    suffix="Classify the following numbers as Abra, Kadabra
    ➥or Abra Kadabra: {comma_delimited_input_numbers}",
    input_variables=["comma_delimited_input_numbers"]
)

prompt_input = few_shot_prompt.format(
➥comma_delimited_input_numbers="3, 4, 5, 7, 8, 10, 11, 13, 35.")

response = llm.invoke(prompt_input)
print(response.content)
```

The result is still correct:

```
3 \ not divisible by 5 nor by 7 \ None
4 \ not divisible by 5 nor by 7 \ None
5 \ divisible by 5 but not by 7 \ Abra
7 \ divisible by 7 but not by 5 \ Kadabra
8 \ not divisible by 5 nor by 7 \ None
10 \ divisible by 5 but not by 7 \ Abra
11 \ not divisible by 5 nor by 7 \ None
13 \ not divisible by 5 nor by 7 \ None
35 \ divisible by 5 and by 7 \ Abra Kadabra
```

2.5.6 Chain of Thought

Finally, I want to introduce you to the CoT technique, which blends providing logical steps with few-shot learning. Here, I'm making up a new mathematical definition in which a sequence of numbers is called "strange" if it contains at least two odd numbers and the sum of all odd numbers is divisible by 3. Initially, I check whether the LLM is unfamiliar with the rules of strange sequences:

> Is the following sequence "strange"?
> 3, 4, 5, 7, 10, 18, 22, 24

> " The sequence you provided appears to be increasing, but it doesn't follow a common mathematical pattern like arithmetic or geometric progressions. Whether it's considered "strange" depends on the context or criteria you're using to define "strange." In a mathematical sense, it might be considered an unusual or nonstandard sequence. If you've specific criteria or rules that define what a "strange" sequence is, please provide more details for a more precise classification.

As expected, ChatGPT doesn't seem to understand the concept of a "strange" sequence, reaffirming that, luckily, it hasn't developed psychic abilities—yet.

I'll now create a prompt using the CoT technique, as introduced by Wei et al. (https://arxiv.org/abs/2201.11903). This prompt, an extension of a few-shot learning prompt, will include the following:

- Several sample sequences for few-shot learning
- An explanation for each sample sequence of the logical steps to determine if a sequence is strange

> Q: Is the following sequence "strange"?
> 3, 4, 5, 7, 10, 18, 22, 24
> Examples:
> Q: is the following sequence strange: 1, 4, 6, 8, 20
> A: 1 is an odd number; I need at least two odd numbers // Not Strange
> Q: is the following sequence strange: 5, 6, 7, 8, 20
> A: 5 and 7 are odd numbers; the sum of 5 and 7 is 12; 12 is divisible by 3 // Strange
> Q: is the following sequence strange: 1, 5, 6, 7, 8, 20
> A: 1, 5 and 7 are odd numbers; the sum of 1, 5 and 7 is 13; 13 is not divisible by 3 // Not Strange
> Q: is the following sequence strange: 5, 6, 7, 8, 9, 20
> A: 5, 7, 9 are odd numbers; the sum of 5, 7 and 9 is 21; 21 is divisible by 3 // Strange

> " Thank you for providing the examples and clarifications. Based on the criteria you've presented:
>
> The sequence "3, 4, 5, 7, 10, 18, 22, 24" has 3 odd numbers: 3, 5, and 7. The sum of these odd numbers is 15. 15 is divisible by 3. So, according to the criteria, this sequence is **Strange**.

ChatGPT gave the right answer and explained its thought process. To reinforce your understanding, try using a similar prompt with a different set of numbers. In addition, observe how ChatGPT responds when the sequence contains just one odd number.

In this section, you've discovered how to introduce ChatGPT to a new task by offering various examples or outlining the necessary steps. This approach is known as in-context learning, as you're instructing the LLM using information provided within the context of a prompt.

DEFINITION *In-context learning* refers to any method that instructs an LLM on an unfamiliar task or instruction requested by the user, using examples within the prompt context. Common techniques include one-shot, two-shot, or few-shot learning, as well as providing step-by-step guidance, often referred to as a chain of thought. Compared to fine-tuning—which involves creating a specialized LLM from a general one by training it on domain-specific text—in-context learning is a more cost-effective and less resource-intensive approach. It doesn't require high-end hardware such as GPUs or an in-depth understanding of transformer architectures.

This section covered a range of in-context learning prompts, which are summarized in table 2.2.

Table 2.2 In-context learning prompts

In-context learning technique	Explanation
Zero-shot learning	No example is provided in the prompt.
One-shot learning	One example is provided in the prompt.
Two-shot learning	Two examples are provided in the prompt.
Few-shot learning	A number of examples are provided in the prompt.
Chain of Thought (CoT)	A number of examples are provided in the prompt, and for each example, all the logical steps to achieve an objective are clearly explained.

Beyond Chain of Thought

Chain of Thought (CoT) improves how language models handle complex reasoning by breaking problems into smaller, logical steps. However, it has limitations, such as missing deeper exploration or struggling with messy contexts. Two advanced techniques address these gaps—Tree of Thought (ToT) and Thread of Thought (ThoT):

- *Tree of Thought*—ToT improves problem-solving by allowing language models to explore multiple reasoning paths. Traditional models make token-by-token decisions, which limits their ability to plan ahead or revisit earlier choices. ToT structures the process into coherent steps, enabling models to evaluate and refine options as they go (see https://arxiv.org/abs/2305.10601). This method is highly effective in tasks requiring strategic thinking, such as Game of 24, creative

(continued)

writing, and mini crosswords. For example, ToT helped GPT-4 solve 74% of Game of 24 problems, compared to just 4% using CoT. This approach improves deliberate decision-making, backtracking, and strategic foresight.

- *Thread of Thought*—ThoT addresses challenges with chaotic contexts (see https://arxiv.org/abs/2311.08734), where irrelevant details distract models or lead to errors. Inspired by human cognition, ThoT systematically breaks down and analyzes large chunks of information while focusing on relevant details. It works as a modular add-on that integrates with various models and prompts. Experiments on datasets such as PopQA, EntityQ, and a Multi-Turn Conversation Response dataset show ThoT significantly enhances reasoning accuracy. It excels in filtering noise and maintaining focus in complex contexts.

Both techniques push language models beyond simple response generation, improving reasoning, planning, and decision-making in structured and chaotic scenarios. For technical details and examples, refer to the published research and code repositories.

Before closing this chapter, let's try to extract a standard prompt format from our examination of various use cases up to this point.

2.6 Prompt structure

Combining all the prompt elements from earlier sections results in the following generalized structure:

- *Persona*—Specify the role you want the LLM to embody. Whether a skilled marketing manager, an award-winning sci-fi author, or a proficient copy editor for a computing magazine, set the expectations high.
- *Context*—Provide detailed background information to help the LLM comprehend the context surrounding your request. Precision enhances the likelihood of aligning with your expectations.
- *Instruction*—Clearly define the action you want the LLM to take with your input text. Whether it's translation, writing, rewriting, improving, classifying, ordering, or finding, be explicit.
- *Input*—This refers to specific context details, which could take the form of a paragraph, a query, or a list of key points. You may choose to label it accordingly, such as "Text," "Question," and so on.
- *Steps*—Outline the processing steps the LLM should follow to generate the output.
- *Tone*—Specify the desired tone of the LLM's answer—formal, informal, witty, enthusiastic, sober, friendly, and so on. Combinations are possible as well.
- *Output format*—Optionally specify the desired output format. If not specified, the LLM assumes a text answer. However, you can request formats such as a table, CSV file (with column names), JSON (with attribute names), or other structured formats.

- *Examples*—In cases where the LLM may lack sufficient training for certain tasks, especially in text classification and reasoning, provide mini-inline training through examples. Even a few examples can significantly enhance answer quality.

This list acts as a guide or checklist for organizing prompts. Customize it based on your requirements, choosing relevant sections or making modifications. Here's a prompt that includes many of these elements, structured with clear sections:

PERSONA You're an experienced Large Language Model (LLM) developer and renowned speaker.

CONTEXT You've been invited to give a keynote speech for a LLM event.

INSTRUCTION Write the punch lines for the speech.

INPUT Include the following facts:

- LLMs have become mainstream with the launch of ChatGPT in November 2022
- many popular LLMs and LLM based chatbots have been launched since then, such as LLAMA-2, Falcon180B, Bard.
- LLMs becoming as popular as search engines
- many companies want to integrate LLMs in their applications

TONE Use a witty but entertaining tone.

OUTPUT FORMAT Present the text in two paragraphs of 5 lines each.

> " Ladies and gentlemen, as we embark on this linguistic odyssey, remember this: In November 2022, ChatGPT didn't just enter the scene; it was the linguistic equivalent of a rockstar's encore, setting the stage for an epic word-tastic journey! Today, with LLAMA-2, Falcon180B, and Bard, it's safe to say that LLMs have become the Beatles of the digital era, making search engines feel like yesterday's news.
>
> But here's the kicker – companies are now falling over themselves, desperately trying to get these LLMs into their applications. It's like the gold rush, but instead of panning for precious metal, we're sifting through bytes of brilliant language. So, my friends, in this age where words wield the power, and LLMs are the mighty pens, we're not just scripting the future; we're penning a linguistic saga that's bound to be a best-seller! Welcome to the age of LLMs, where we're rewriting the rules, one sentence at a time, and words are the currency of change!

A more reliable way to enforce prompt structure—recommended in both OpenAI's and Anthropic's prompt engineering guidelines (see the resource list at the end of this section)—is to mark different sections of the prompt with XML-style tags. For example, the previous prompt could be written as

```
<Persona>
You're an experienced Large Language Model (LLM) developer and renowned
speaker.
</Persona>
<Context>
You've been invited to give a keynote speech for a LLM event.
</Context>
```

<Instruction>
Write the punch lines for the speech.
</Instruction>
<Input>
Include the following facts:

- LLMs have become mainstream with the launch of ChatGPT in November 2022
- many popular LLMs and LLM based chatbots have been launched since then, such as LLAMA-2, Falcon180B, Bard.
- LLMs becoming as popular as search engines
- many companies want to integrate LLMs in their applications

</Input>
<Tone>
Use a witty but entertaining tone.
</Tone>
<OutputFormat>
Present the text in two paragraphs of 5 lines each.
</OutputFormat>

NOTE Studies (e.g., "The Prompt Report: A Systematic Survey of Prompting Techniques," https://arxiv.org/abs/2406.06608) have found that explicitly naming different parts of a prompt tends to improve results. However, you don't have to name every section. Most LLMs can figure out the purpose of the text in the prompt on their own. So you can mix things up by naming some parts (e.g., Question or Examples) and leaving out names for others (e.g., Context or Tone). Experiment and see what works best for your case and the results you're getting.

If you want to delve deeper into prompt engineering, I highly recommend the following resources:

- https://mng.bz/z2aA
- https://mng.bz/0zVv
- https://mng.bz/KwmO
- https://mng.bz/9ynr
- www.promptingguide.ai/techniques
- https://github.com/dair-ai/Prompt-Engineering-Guide
- https://mng.bz/jZ8e

Summary

- A prompt guides the LLM by giving it instructions and context (background information, e.g., domain knowledge or constraints). Clear, specific prompts produce better results than vague ones.
- Different instruction patterns suit different tasks. Classification prompts request category labels or sentiment scores; generation prompts ask for original creative content; and extraction prompts pull specific facts from input text.
- When prompts fail to produce expected outputs, the LLM may lack training examples for that task format. Adding few-shot examples or adjusting instruction clarity usually fixes this.
- Few-shot learning demonstrates the desired response pattern. One-shot uses a single example, two-shot uses two, and so on—more examples generally improve consistency but increase token costs.
- Standard prompts include persona (role), context (background), instructions (task), input (data), style (tone/format), and examples (patterns). Not all sections are required for every task.
- You can omit, rearrange, or extend prompt sections with custom elements. Add constraints, output format specifications, or multi-step reasoning instructions as needed.
- API-based prompt automation injects variables into prompt templates and executes requests programmatically. This scales prompting from single requests to thousands of variations. For instance, you can produce thousands of product descriptions by populating a single template with different product attributes (name, category, features) from a database or CSV file.
- Prompt templates use placeholders (typically `{variable_name}` in curly braces) that get replaced with actual values at runtime. This separates prompt structure from dynamic content.
- LangChain offers `PromptTemplate` for basic variable substitution. `FewShotPromptTemplate` dynamically selects examples based on input similarity, and `ChatPromptTemplate` structures multi-turn conversations.
- Chain-of-thought (CoT) prompting asks the LLM to show step-by-step reasoning before providing the final answer. Include instructions such as ““Let’s think step-by-step” or explicitly list the reasoning steps required.
- Structure prompts with XML-style tags to enforce sections: `<persona>`, `<context>`, `<instruction>`, `<input>`, `<examples>`. This follows OpenAI and Anthropic prompt engineering guidelines for clarity.
- Format messages for OpenAI API as a list with role and content: `messages=[{"role": "system", "content": "..."}, {"role": "user", "content": "..."}]`. Roles include system, user, and assistant.

Part 2

Summarization

This part is all about transforming too much information into something clear, concise, and useful. You'll start by learning practical techniques for summarizing long or numerous documents—from massive reports to folders full of mixed file types—while staying within an LLM's context limits. You'll explore when to use strategies such as MapReduce or the refine technique and how to connect everything with the LangChain Expression Language (LCEL) to produce fast, accurate, and maintainable summaries.

From there, you'll move beyond simple "summarize this text" tasks to more advanced research-oriented applications. You'll build a summarization engine that searches the web, collects relevant information, and composes a coherent, well-grounded report—all powered by prompt engineering and modular chains. Finally, you'll evolve your summarization workflows into agentic systems using LangGraph, introducing explicit state management and conditional branching so your pipelines can adapt intelligently, scale smoothly, and serve as a foundation for more autonomous AI agents later in the book.

3 Summarizing text using LangChain

This chapter covers

- Summarization of large documents exceeding the LLM's context window
- Summarization across multiple documents

In chapter 1, we explored three major LLM application types: summarization engines, chatbots, and AI agents. In this chapter, you'll begin building practical summarization chains using LangChain, with a particular focus on the LangChain Expression Language (LCEL) to handle various real-world scenarios. A chain is a sequence of connected operations where the output of one step becomes the input for the next—ideal for automating tasks like summarization. This work lays the foundation for constructing a more advanced summarization engine in the next chapter.

Summarization engines are essential for automating the summarization of large document volumes, a task that would be impractical and costly to handle manually, even with tools such as ChatGPT. Starting with a summarization engine is a practical entry point for developing LLM applications, providing a solid base for more

complex projects and showcasing LangChain's capabilities, which we'll further explore in later chapters.

Before we start building, we'll look at different summarization techniques, each suited to specific scenarios, including large documents, content consolidation, and handling structured data. You've already worked with summarizing small documents using a `PromptTemplate` in chapter 2, so we'll skip that and focus on more complex examples.

3.1 Summarizing a document bigger than the context window

As mentioned in chapter 1, each LLM has a maximum prompt size, also referred to as the *context window*. As the context window for popular LLMs continues to grow, you may still encounter situations where a document exceeds the token limit of your chosen model. In these cases, you can use a MapReduce approach, as illustrated in figure 3.1.

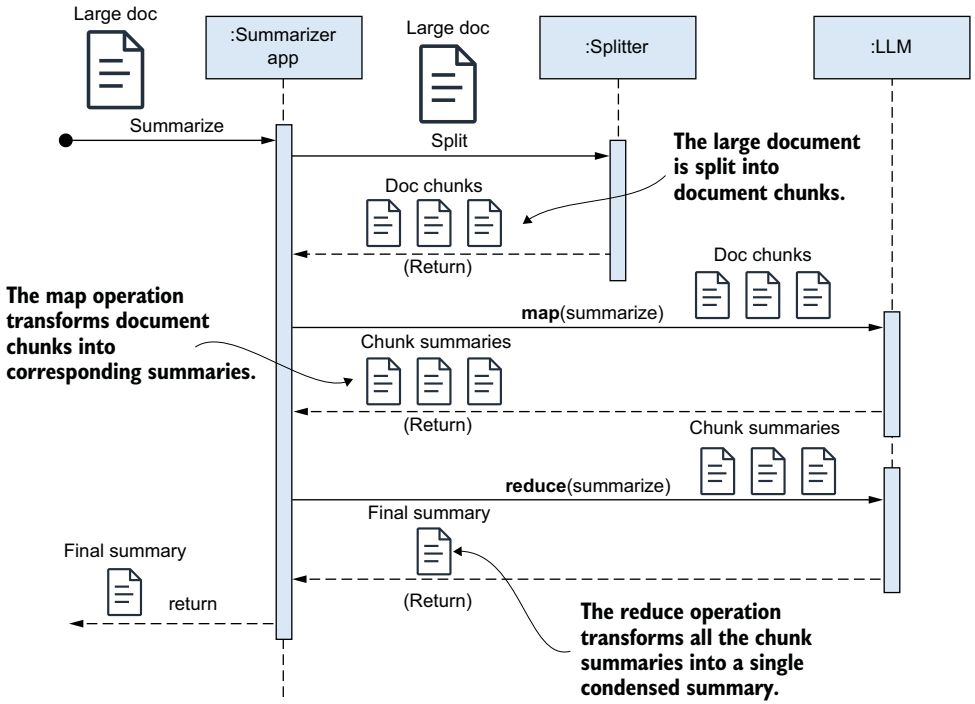


Figure 3.1 Summarizing a document bigger than the LLM's context window involves splitting the document into smaller chunks, summarizing each chunk, and then summarizing the combined chunk summaries.

DEFINITION The LLM context window refers to the maximum amount of text—comprising both instructions and context—that can be provided to a language model in a single prompt. Different LLMs support different token limits for this context window, where one token roughly corresponds to one

word. For example, GPT-3.5 could process up to 16,000 tokens, GPT-4 and Claude 3 supported up to 100,000 tokens, and newer models such as GPT-5 and Gemini can handle more than 1 million tokens.

When working with documents that exceed an LLM's context window, a common strategy is to break the text into manageable chunks, generate summaries for each chunk, and then produce a final summary from those intermediate results. To start, you need to split the text into chunks using a tokenizer. A tokenizer reads the text and breaks it into tokens, which are the smallest units of text, often parts of words. After tokenizing the document, the tokens are grouped into chunks of a specific size. This lets you control the content size being processed by the LLM and ensures the token count stays within your LLM's prompt limit. Here, I'll show you how to use `TokenTextSplitter`, part of the `tiktoken` package, a tokenizer developed by OpenAI.

Begin by opening a terminal and creating a new folder named `ch03` for this chapter's code. Then, create and activate a virtual environment:

```
C:\Github\building-llm-applications>md ch03
C:\Github\building-llm-applications>cd ch03
C:\Github\building-llm-applications\ch03>python -m venv env_ch03
C:\Github\building-llm-applications\ch03>.\env_ch03\Scripts\activate
(env_ch03) C:\Github\building-llm-applications\ch03>
```

Next, install the required packages—`tiktoken`, `notebook`, and `langchain`. If you've cloned the repository from GitHub (https://mng.bz/WwvW) or downloaded the code zip file from the Manning website, use the following:

```
C:\Github\building-llm-applications\ch03>pip install -r requirements.txt
```

Once the installation is complete, start the Jupyter Notebook:

```
(env_ch03) C:\Github\Building-llm-applications\ch03>jupyter notebook
```

Now open or create a notebook, and name it `03-summarization_examples.ipynb`. Finally, save the notebook.

3.1.1 Chunking the text into Document objects

Let's summarize the book *Moby Dick* using its text file, `Moby-Dick.txt`, downloaded from the Project Gutenberg site (www.gutenberg.org). You can locate the `Moby-Dick.txt` file in the chapter's subfolder on my GitHub page (https://mng.bz/DwyV). Place this file in your `ch03` folder, and load the text into a variable:

```
with open("./Moby-Dick.txt", 'r', encoding='utf-8') as f:
    moby_dick_book = f.read()
```

NOTE Keep in mind that running the code on the full *Moby Dick* text can get expensive. The provided `Moby-Dick.txt` file is a shorter version, containing only five chapters and about 18,000 tokens. Running the code a few times with this version shouldn't cost much. However, if you plan to run many tests, you may want to reduce the file size even more to save money. Each time you

execute a chain with an `LLMChain` block, you'll be charged. If budget allows, you can use the full version of the book, labeled `Moby-Dick_ORIGINAL_EXPENSIVE.txt`. The entire *Moby Dick* text has around 300,000 words, or about 350,000 tokens. Using GPT-5, which costs $1.25 per million tokens, processing the full text would cost about $0.37. Running it multiple times will add up. If you switch to GPT-5-nano, which costs $0.05 per million tokens, the cost drops to around $0.015, making it much cheaper.

I'll cover chunking strategies in detail—such as chunking by size and content structure—in chapter 8. For now, let's split the text into chunks of about 3,000 tokens each to simulate a context window shorter than the GPT-5-nano model we'll be using.

3.1.2 Split

To start the split process, you need to do a little prep work. First, import the necessary libraries:

```
from langchain_openai import ChatOpenAI
from langchain_text_splitters import TokenTextSplitter
from langchain_core.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnableLambda, RunnableParallel
import getpass
```

Next, retrieve your OpenAI API key using the `getpass` function:

```
OPENAI_API_KEY = getpass.getpass('Enter your OPENAI_API_KEY')
```

You'll be prompted to enter your `OPENAI_API_KEY`. After that, instantiate the OpenAI model in a new notebook cell:

```
llm = ChatOpenAI(openai_api_key=OPENAI_API_KEY,model_name="gpt-5-nano")
```

Now you're ready to set up the first chain, which will break the document into chunks of the specified size. In this chapter, you'll learn the basics of LCEL, with a more detailed exploration in the next chapter:

```
text_chunks_chain = (
    RunnableLambda(lambda x:
        [
            {
                'chunk': text_chunk,
            }
            for text_chunk in
               TokenTextSplitter(chunk_size=3000,
               ➥chunk_overlap=100).split_text(x)
        ]
    )
)
```

NOTE LangChain chains are made up of components that implement the `Runnable` interface, an abstract Python class that defines how a component takes input, processes it, and returns output. Any class implementing `Runnable`

can be part of a chain. `RunnableLambda` lets you turn any Python callable into a `Runnable`, making it easy to include custom functions in a LangChain chain. It's similar to Python's lambda expression, where you run code with a parameter and optionally return output without defining a full function. With `RunnableLambda`, you can create a chain component without writing a separate class to implement the `Runnable` interface. In this example, the code wrapped by `RunnableLambda` takes input text as a string through the `x` parameter and passes it to the `split_text()` function, which breaks the text into chunks.

3.1.3 Map

The next step is to set up the map chain, which will run a summarization prompt for each document chunk and ensure that every piece of the text is processed independently before being combined later:

```
summarize_chunk_prompt_template = """
Write a concise summary of the following text, and include the main details.
Text: {chunk}
"""

summarize_chunk_prompt =
➥PromptTemplate.from_template(summarize_chunk_prompt_template)
summarize_chunk_chain = summarize_chunk_prompt | llm

summarize_map_chain = (
    RunnableParallel (
        {
            'summary': summarize_chunk_chain | StrOutputParser()
        }
    )
)
```

In this code, the pipe operator (`|`) is used to chain components together, passing the output of one object as the input to the next. For instance, the `summarize_chunk_prompt` is piped into the `llm`, meaning the generated prompt is sent directly to the model. Similarly, the model's output is piped into `StrOutputParser()`, which converts the model's response into a clean text string.

NOTE In this chain, I've used `RunnableParallel`, which is similar to `RunnableLambda`, but it operates on a sequence, processing each element in parallel. In this case, we'll feed the sequence of text chunks to the `summarize_map_chain`, and each chunk will be summarized in parallel by the inner `summarize_map_chain`.

MapReduce

MapReduce is a programming model for processing large datasets in two steps. First, the map operation splits the data into smaller subsets, each processed independently by the same function. This step typically returns a list of results, grouped

(continued)

by a key. Next is the reduce operation, where results for each key are aggregated into a single outcome. The final output is a list of key–value pairs, where the keys come from the map step and the values are the result of aggregating the mapped data in the reduce step.

3.1.4 Reduce

Setting up the reduce chain, which summarizes the summaries from each document chunk, follows a process similar to the map chain but requires a bit more setup. Start by defining the prompt template:

```
summarize_summaries_prompt_template = """
Write a concise summary of the following text,
which joins several summaries, and include the main details.
Text: {summaries}
"""

summarize_summaries_prompt =
➥PromptTemplate.from_template(summarize_summaries_prompt_template)
```

Next, you can configure the reduce chain:

```
summarize_reduce_chain = (
    RunnableLambda(lambda x:
        {
            'summaries': '\n'.join([i['summary'] for i in x]),
        })
    | summarize_summaries_prompt
    | llm
    | StrOutputParser()
)
```

The reduce chain includes a lambda function that combines the summaries from the map chain into a single string. This string is then processed by the `summarize_summaries_prompt` prompt, which generates a final summary of the combined content.

3.1.5 MapReduce combined chain

Finally, we combine the document-splitting chain, the map chain, and the reduce chain into a single MapReduce chain:

```
map_reduce_chain = (
   text_chunks_chain          <-- Split chain
   | summarize_map_chain.map()          <-- Map chain
   | summarize_reduce_chain   <-- Reduce chain
)
```

This setup efficiently splits the input document into chunks, summarizes each chunk, and then compiles those summaries into a final summary. The `map()` function on `summarize_map_chain` is essential to enable parallel processing of the chunks.

3.1.6 MapReduce execution

Everything is now set up. Start the MapReduce summarization of the large document with this command (if you're on the OpenAI free tier, this might fail with a `429 Rate Limit` error):

```
summary = map_reduce_chain.invoke(moby_dick_book)
```

If you run `print(summary)`, you'll get output similar to the following:

```
The introduction to the Project Gutenberg eBook of "Moby-Dick; or The Whale"
by Herman Melville outlines the book's availability and updates, with the
first eBook release in June 2001 and the latest in August 2021. The narrative
begins with Ishmael, the narrator, who seeks solace at sea to escape his
melancholic state, showcasing the ocean's allure compared to city life. He
reflects on his reasons for joining a whaling voyage, driven by a fascination
with whales and a thirst for adventure. After arriving in New Bedford,
Ishmael faces challenges finding lodging, ultimately settling at "The Spouter
Inn," where he encounters a chaotic environment and a mysterious harpooneer
named Queequeg.

As Ishmael shares a bed with Queequeg, whom he initially fears to be a
cannibal, he gradually overcomes his apprehensions. The morning after their
first night together highlights their strange yet developing bond, as Ishmael
observes Queequeg's unique customs and politeness, emphasizing themes of
fate, choice, and the allure of the unknown in the whaling industry.
```

> **WARNING** As previously mentioned, running `map_reduce_chain` will incur costs; the longer the text you want to summarize, the higher the cost. Therefore, consider further shortening the input text (in our case, the `Moby-Dick.txt` e-book file) if you want to limit expenses. Additionally, ensure your OpenAI API credit balance remains positive to avoid errors such as the following: `RateLimitError: Error code: 429 - {'error': {'message': 'You exceeded your current quota, please check your plan and billing details [...].'` If necessary, log in to the OpenAI API page, navigate to Settings > Billing, and set a credit of at least $5.

Let's now proceed to the next use case: summarizing across documents. In this scenario, the goal is to combine insights from multiple sources rather than working with a single long text.

3.2 Summarizing across documents

You can easily learn how to summarize information across various data sources, such as Wikipedia or local files in Microsoft Word, PDF, and text formats. This process, as shown in figure 3.2, is similar to the MapReduce technique used in the previous section.

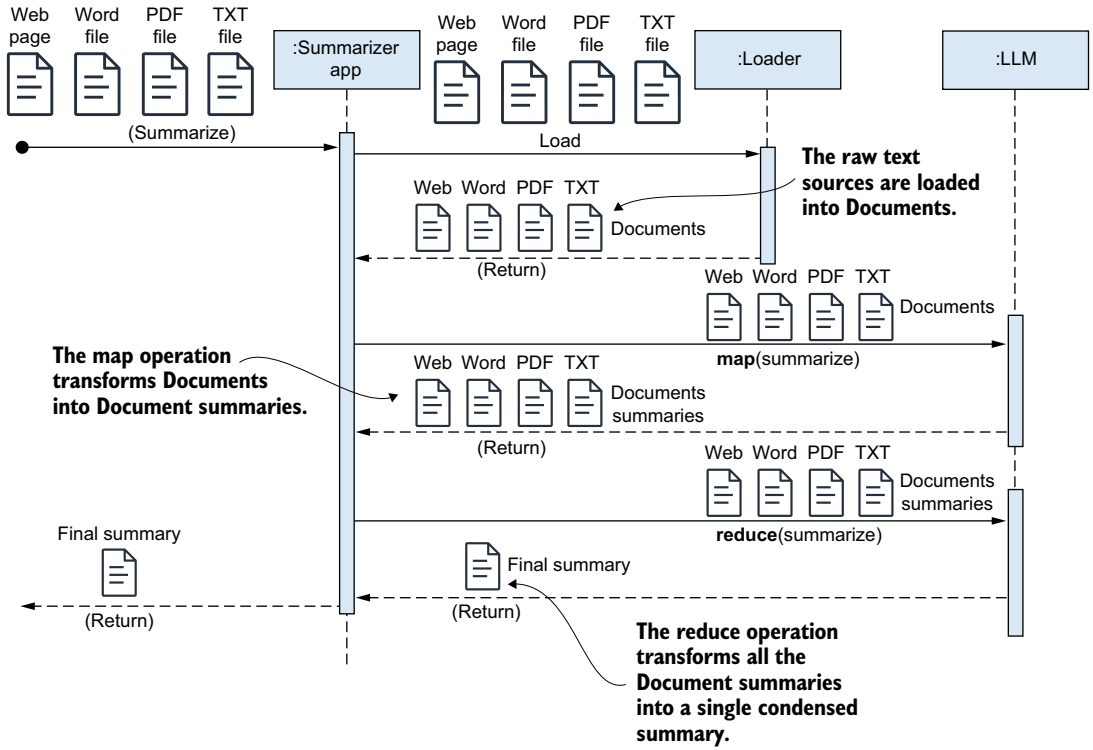


Figure 3.2 Summarizing across documents using the MapReduce technique seen earlier. In this method, each document chunk undergoes a `map` operation to generate a summary. These individual summaries are then further condensed into a single summary through the `reduce` operation.

In the sequence diagram in figure 3.2, content from each raw text source is loaded into a corresponding `Document` instance. During the `map` operation, these `Document` objects are converted into individual summaries, which are then combined into a single summary during the `reduce` operation.

Next, I'll introduce you to an alternative *refine technique*, as illustrated in figure 3.3. With this approach, a final summary is constructed incrementally by iteratively summarizing the combination of the current final summary and one of the document chunks. This process continues until all document chunks have been processed, resulting in the completion of the final summary.

In this method, you progressively build the final summary by refining it with each step. Each document is sent to the LLM for summarization, along with the current draft of the summary. This continues until all documents are processed, leading to the final summary. MapReduce works well for summarizing large volumes of text, where some content loss is acceptable to manage the processing load. In contrast, the refine technique is better when you want to ensure that the essence of each part is fully captured in the final summary.

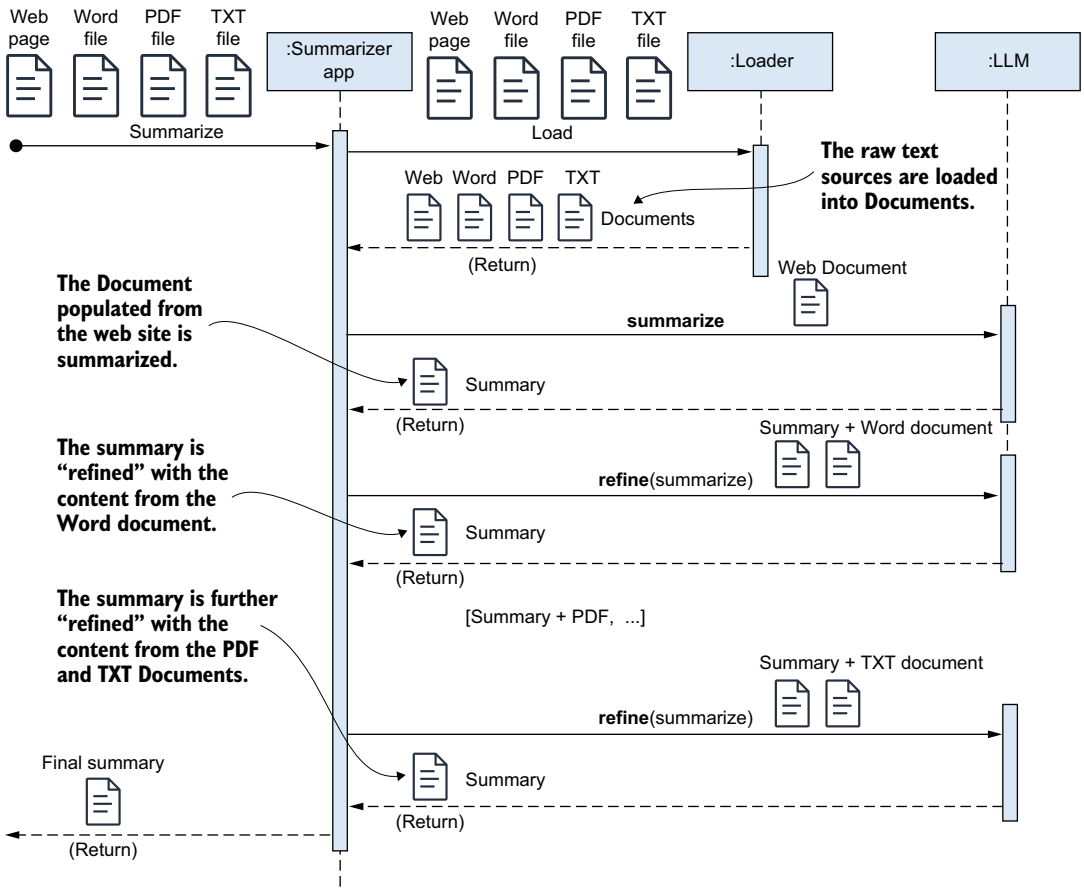


Figure 3.3 Summarizing across documents using the refine technique

3.2.1 Creating a list of Document objects

When summarizing a large document, you typically start by breaking it into smaller chunks, treating each chunk as a separate document. In this case, we're beginning with a set of existing documents, so there's no need to split anything. How you create each `Document` object will depend on the source of the text. I'll show you how to summarize content from four different sources: a Wikipedia page and a set of files in various formats (TXT, DOCX, PDF) stored in a local folder. All the content is related to Paestum, a Greek colony on the Cilento coast in southern Italy around 500 BC. You'll use the appropriate `DocumentLoader` for each data source, selecting from the many options introduced earlier in chapter 1, section 1.3, on LangChain's `Document` object model.

3.2.2 Wikipedia content

Let's begin with the Wikipedia content. While you can create a document from web-based data content using the `WebBaseLoader`, specific loaders are customized to

retrieve content from particular websites, such as the `IMSDbLoader` for the Internet Movie Script Database (IMSDb) website, the `AZLyricsLoader` for the AZLyrics website, and the `WikipediaLoader` for the Wikipedia website.

If you followed the package installation instructions at the beginning of section 3.1, you should already have the necessary loader packages installed, including `docx-2txt` (used by `Docx2txtLoader` for Word files) and `pypdf` (used by `PyPDFLoader` for PDFs). You can now import the content from the Paestum Wikipedia page:

```
from langchain.document_loaders import WikipediaLoader

wikipedia_loader = WikipediaLoader(query="Paestum", load_max_docs=2)
wikipedia_docs = wikipedia_loader.load()
```

> **NOTE** The `WikipediaLoader` may load content from other Wikipedia hyperlinks referenced in the requested article. For example, the Paestum article references the National Archeological Museum of Paestum, the Lucania region, Lucanians, and the temples of Hera and Athena, resulting in additional content loaded. Thus, it returns a `Document` list rather than a single `Document` object. I've set the maximum number of documents returned to `2` to save on summarization costs, but you can adjust it as needed.

3.2.3 File-based content

To get started, download or pull the `Paestum` folder from GitHub, and place it in your local `ch03` directory (if you didn't clone the entire repository). The `Paestum` subfolder within `ch03` contains three files:

- `Paestum-Britannica.docx`—Content sourced from the Encyclopedia Britannica website.
- `PaestumRevisited.pdf`—An excerpt from "Paestum Revisited," a master thesis submitted at Stockholm University. The extract comprises only four pages, but you have the option to use the full document located in the same folder (`PaestumRevisited-StocholmsUniversitet.pdf`).
- `Paestum-Encyclopedia.txt`—Content taken from Encyclopedia.com.

Following is the process to load these files into corresponding documents:

```
from langchain.document_loaders import Docx2txtLoader
from langchain.document_loaders import PyPDFLoader
from langchain.document_loaders import TextLoader

word_loader = Docx2txtLoader("Paestum/Paestum-Britannica.docx")
word_docs = word_loader.load()

pdf_loader = PyPDFLoader("Paestum/PaestumRevisited.pdf")
pdf_docs = pdf_loader.load()

txt_loader = TextLoader("Paestum/Paestum-Encyclopedia.txt")
txt_docs = txt_loader.load()
```

The document variables (`word_docs`, `pdf_docs`, `txt_docs`) are in plural mode because a loader always returns a list of documents, even if the list contains only one item.

NOTE You may have noticed the direct creation of an array of `Document` objects from `Paestum-Encyclopedia.txt` using a `TextLoader` and wonder why the `Moby-Dick.txt` file was read with the Python file reader previously. In that case, the intention was to split the content into a specific number of tokens to fit the LLM prompt, requiring manual creation of a `Document` object for each.

3.2.4 Creating the Document list

You can now merge all the documents from various sources into a single `Document` list. Use the following code to do so:

```
all_docs = wikipedia_docs + word_docs + pdf_docs + txt_docs
```

With everything compiled, you're ready to summarize the content using the refine technique. The next step is to create a chain that generates the final summary.

Document loaders

Along with document loaders for specific data sources, I encourage you to explore the `UnstructuredLoader` as well. It enables you to import content from various file types, including Word, PDF, and TXT files, among others.

Another option is the `DirectoryLoader`, which uses the `UnstructuredLoader` internally. It allows you to load content from files of different formats located in the same folder in a single operation.

As an exercise, I recommend re-creating the documents from the Word, PDF, and TXT Paestum content using either the `UnstructuredLoader` or the `DirectoryLoader`. If you choose to do so, you'll need to install the related package and refer to the documentation on the LangChain website:

```
pip install "unstructured[all-docs]"
```

The LangChain framework provides numerous loaders for retrieving content from diverse data sources. I highly encourage you to explore the list and experiment with any loaders that pique your interest: https://mng.bz/EwmR.

3.2.5 Progressively refining the final summary

Now that everything is set up, you can create a chain to generate the final summary step-by-step. Begin by importing the necessary modules:

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import PromptTemplate
import getpass
```

Next, capture the `OPENAI_API_KEY`, and set up the LLM model as before:

```
OPENAI_API_KEY = getpass.getpass('Enter your OPENAI_API_KEY')

llm = ChatOpenAI(openai_api_key=OPENAI_API_KEY,model_name="gpt-5-nano")
```

Now define the chain with the related prompt for summarizing individual documents:

```
doc_summary_template = """Write a concise summary of the following text:
{text}
DOC SUMMARY:"""
doc_summary_prompt = PromptTemplate.from_template(doc_summary_template)

doc_summary_chain = doc_summary_prompt | llm
```

Next, set up the chain for refining the summary by iteratively combining the current summary with the summary of an additional document:

```
refine_summary_template = """
You must produce a final summary from the current refined summary
which has been generated so far and from the content of an
additional document.
This is the current refined summary generated so far:
{current_refined_summary}
This is the content of the additional document: {text}
Only use the content of the additional document if it is useful,
otherwise return the current full summary as it is."""

refine_summary_prompt =
➥PromptTemplate.from_template(refine_summary_template)

refine_chain = refine_summary_prompt | llm | StrOutputParser()
```

Finally, define a function that loops over each document, summarizes it using the `doc_summary_chain`, and refines the overall summary using the `refine_chain`:

```
def refine_summary(docs):

    intermediate_steps = []
    current_refined_summary = ''
    for doc in docs:
        intermediate_step = \
           {"current_refined_summary": current_refined_summary,
            "text": doc.page_content}
        intermediate_steps.append(intermediate_step)

        current_refined_summary = refine_chain.invoke(intermediate_step)

    return {"final_summary": current_refined_summary,
            "intermediate_steps": intermediate_steps}
```

You can now start the summarization process by calling `refine_summary()` on your prepared document list:

```
full_summary = refine_summary(all_docs)
```

Printing the `full_summary` object will show the final summary under `final_summary` and the intermediate steps under `intermediate_steps`. Although the results steps are shortened for convenience, I encourage you to observe how the summary evolves at each stage:

```
print(full_summary )
```

Here's an excerpt from the output:

```
{'final_summary': "**Final Summary:**\n\nPaestum, an ancient Greek city
located on the coast of the Tyrrhenian Sea in Magna Graecia, was established
around 600 BC by settlers from Sybaris and originally named Poseidonia. The
city flourished as a Greek settlement […SHORTENED …] those interested in
ancient Greek culture and architecture.", 'intermediate_steps': [{'current_
refined_summary': '', 'text': 'Paestum ( PEST-əm, US also  PEE-stəm )was a
major ancient Greek city on the coast of the Tyrrhenian Sea, in Magna
Graecia. The ruins of Paestum are famous for their three ancient Greek
temples in the Doric order dating from about 550 to 450 BC that are
in an excellent state of preservation. The city walls and amphitheatre
[… SHORTENED …]
```

We've covered a few summarization techniques, so let's pause briefly to reflect on what we've learned. This is a good point to step back and consider how these methods relate to one another and when each approach is most effective.

3.3 Summarization flowchart

To wrap up this chapter, I've included a flowchart to help you choose the most appropriate summarization technique for your specific needs, as shown in figure 3.4. Here, the first key decision is whether you're summarizing one or multiple documents. If it's just one document and it fits within the context window, you can "stuff" the entire

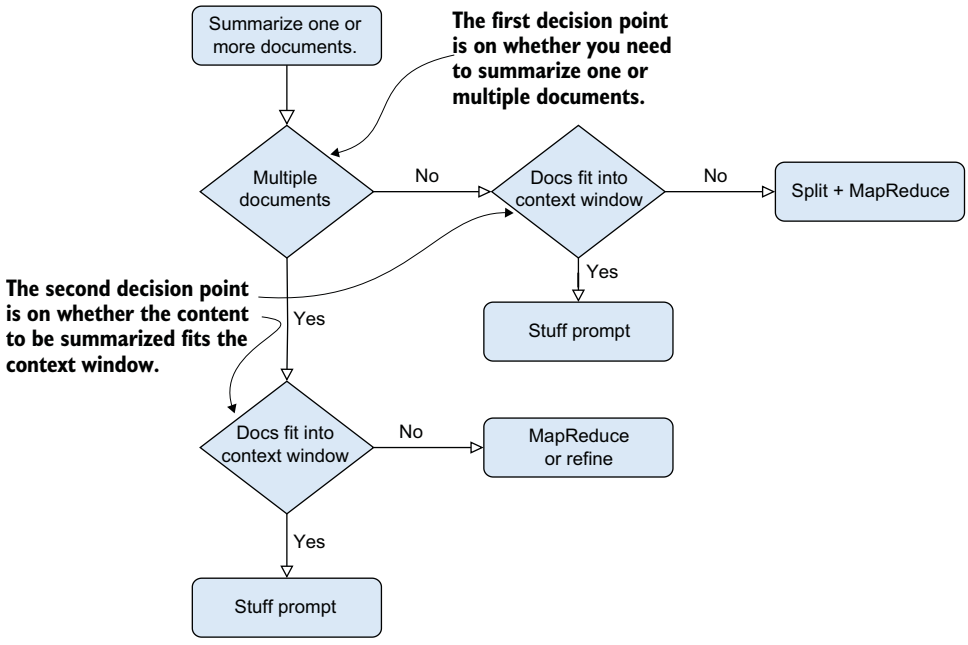


Figure 3.4 Summarization flowchart. This flowchart guides you in selecting the right summarization approach based on whether you need to summarize multiple unrelated documents and whether the input text fits within the context window.

document into a single prompt for summarization. If it doesn't fit, use the MapReduce method. For multiple documents, if they all fit within the context window, you can also stuff them into a single prompt. If not, use MapReduce for a large number of documents, or use the refine technique if you want to ensure that the core of each document is included in the final summary.

Summary

- Text summarization condenses documents into shorter versions while preserving key information. Use it for executive reports, article abstracts, and content previews. The approach varies by document count and size. Direct prompting works for single short documents, chaining handles medium-length texts, and MapReduce or Refine strategies tackle large corpora exceeding context windows.
- LangChain `Document` objects wrap raw text with metadata (source, page numbers, timestamps) to preserve provenance through processing pipelines.
- MapReduce summarization processes chunks independently and in parallel during the map stage. The reduce stage combines partial summaries into a final output. This approach handles documents exceeding context window limits, such as 100-page reports. Use MapReduce when parallel processing speed matters more than preserving all contextual connections.
- The refine summarization technique iteratively updates a running summary by incorporating each new document sequentially. Each step sees both the current summary and the next chunk.
- MapReduce sacrifices summary completeness for parallel processing speed and lower token costs. The refine technique preserves more context but processes sequentially, increasing latency and total tokens.
- Create prompt templates with `PromptTemplate.from_template(template_string)`. Chaining components using the pipe operator—`summarize_chunk_chain = summarize_chunk_prompt | llm`—passes the prompt output directly to the model.
- To build map chains for parallel processing, you define `summarize_map_chain` that processes each chunk and then use `.map()` to apply it across all chunks simultaneously with `RunnableParallel`.

4 Building a research summarization engine

This chapter covers

- Understanding research summarization engines
- Using prompt engineering for creating web searches and summarizing results
- Structuring the process into individual LangChain chains
- Integrating sub-chains into a main chain
- Advanced LCEL for parallel processing

Building on the content summarization techniques from chapter 3, this chapter guides you through creating a research summarization engine. This LLM application will process user queries, perform web searches, and compile a comprehensive summary of the findings. We'll develop this project step-by-step, starting with the basics and gradually increasing in complexity. Along the way, you'll deepen your knowledge of LangChain as I introduce creating LLM chains with the LangChain Expression Language (LCEL).

4.1 Overview of a research summarization engine

Imagine you're researching various topics, such as a specific NBA player, a tourist destination, or whether to invest in a stock. Manually, you'd perform a web search, sift through results, read related pages, take notes, and compile a summary. A modern approach is to let an LLM handle this work. You could copy text from each web page, paste it into a ChatGPT prompt for summarization, and repeat for multiple pages. Then, combine these summaries into a final prompt for a consolidated summary (see figure 4.1).

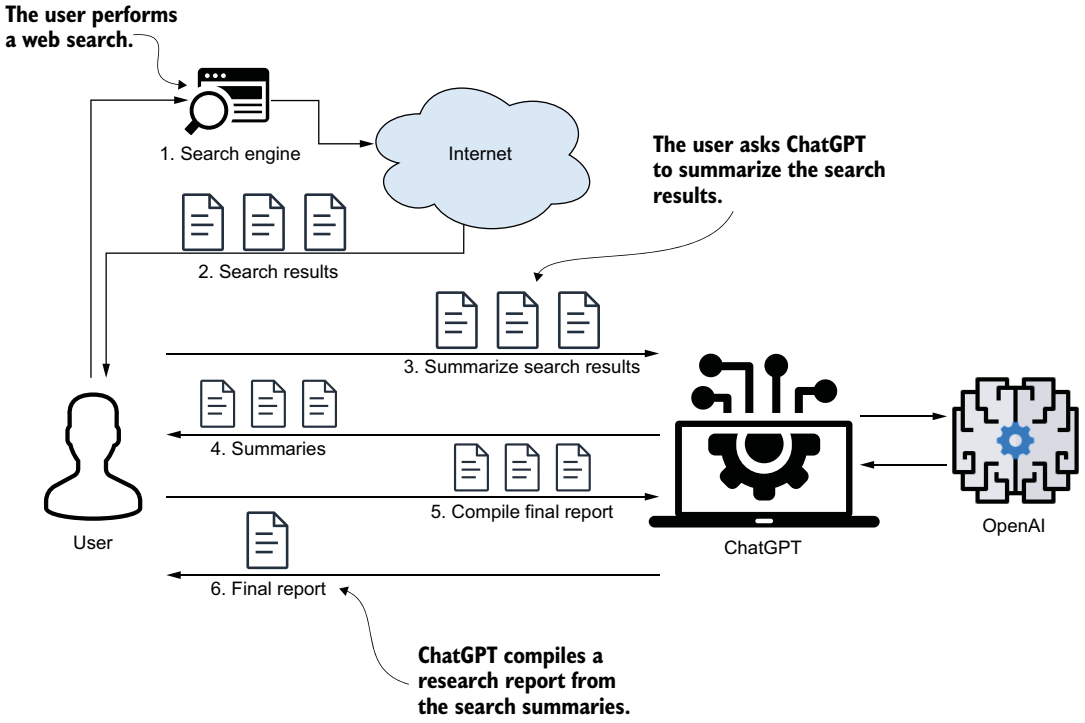


Figure 4.1 Semiautomated summarization with LLM. You prompt ChatGPT to summarize each web search result and compile them into a consolidated summary.

A more efficient method is to develop a fully automated research summarization engine. This engine can perform web searches, summarize the results, and compile a final report automatically (see figure 4.2). It's a valuable tool for handling any research query.

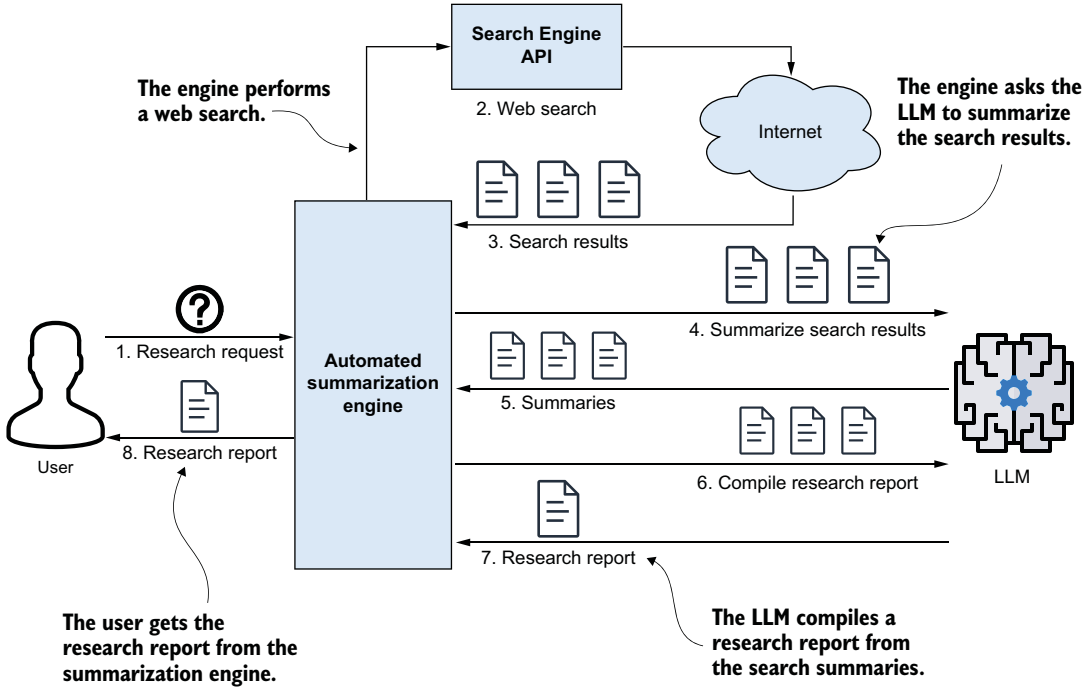


Figure 4.2 Automated research summarization engine. Ask a question, and the engine performs a web search, returns URLs, scrapes and summarizes web pages, and compiles a research report for you.

We'll build this engine using LangChain. First, we'll implement web searching and scraping, then set up the OpenAI LLM model for summarization, and finally integrate all components into a Python application. Initially, it will run as an executable and later, if you wish, you can expose it as a REST API.

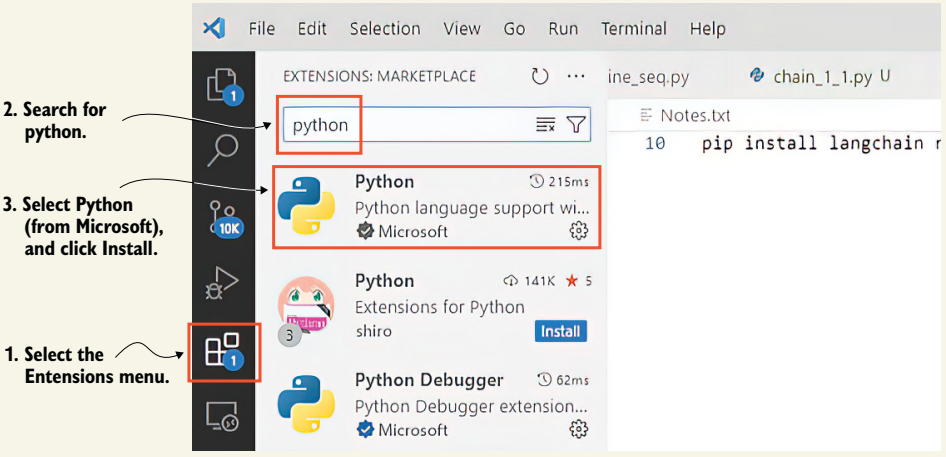
4.2 Setting up the project

I'm assuming you're using Visual Studio Code (VS Code) with the free Python extension and running on Windows. However, you also can opt for alternative Python IDEs such as PyCharm if you prefer.

Installing VS Code and the Python extension

Download and install the appropriate version of VS Code for your operating system from the official website (https://code.visualstudio.com/download). Once installed, open VS Code, and click the Extensions icon on the left-hand menu. Then, search for `Python`, select the Python extension (from Microsoft), and click Install, as shown in the following figure.

(continued)



The Extensions Marketplace panel

I'll briefly guide you through setting up a Python project in VS Code, creating a virtual environment, activating it, and installing necessary packages. Using your file explorer or shell, create an empty folder named `ch04` in your source code area, for example:

```
C:\Github\building-llm-applications\ch04
```

Open VS Code, choose File > Open Folder, navigate to the `ch04` folder, and click Select Folder. Next, open a terminal within VS Code by selecting Terminal > New Terminal. The terminal should display the path to the folder you just created. On Windows, you'll see something like this:

```
PS C:\Github\building-llm-applications\ch04>
```

If you've just installed VS Code or you're new to it, enable the terminal by running this command (you only need to do it once):

```
PS C:\Github\building-llm-applications\ch04> Set-ExecutionPolicy
➥-ExecutionPolicy AllSigned -Scope CurrentUser
```

Within this terminal, as usual, create a virtual environment, and activate it (I'm omitting the full path to `ch04` for convenience):

```
... ch04> python -m venv env_ch04
... ch04> .\env_ch04\Scripts\activate
```

If you activate the virtual environment in PowerShell using `activate.ps1`, you may see a prompt asking, "Do you want to run software from this untrusted publisher?" Respond by typing `A` (Always run) to proceed.

If you've cloned this repository from GitHub, you can install the required packages (`langchain`, `langchain_openai`, `langchain_community`, `requests`, `bs4`, `duckduckgo-search`, `ddgs`) with

```
(env_ch04) ... ch04> pip install -r requirements.txt
```

Once the required packages are installed, I recommend creating a custom run configuration to ensure you're running and debugging your code within the `env_ch04` virtual environment you just set up. To create the configuration, follow these steps:

1. In the top menu, go to Run > Open Configurations. This will open the `launch.json` file in the editor.
2. Replace or add the following configuration:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python Debugger: Current File",
            "type": "debugpy",
            "request": "launch",
            "program": "${file}",
            "console": "integratedTerminal",
            "python": "${workspaceFolder}/env_ch04/Scripts/python.exe"
        }
    ]
}
```

NOTE If you're using macOS or Linux, replace the path in the `"python"` field with `"${workspaceFolder}/env_ch04/bin/python"`.

This custom Run configuration ensures your code runs with the correct interpreter and environment. You'll use it later for debugging. With everything in place, you're now ready to start coding!

4.3 Implementing the core functionality

Looking again at figure 4.2, it's clear that our research summarization engine depends on two key capabilities we must provide: one for conducting web searches and another for extracting text from related web pages. Additionally, you'll need a utility function to initialize an instance of the LLM client you'd like to use.

4.3.1 Implementing web searching

We'll use the LangChain wrapper for the DuckDuckGo search engine to perform web searches. Its results method returns a list of objects, each containing the result URL in a property called `"link"`. Add a new empty file named `web_searching.py` to the project, and fill it with the following code:

```
from langchain_community.utilities import DuckDuckGoSearchAPIWrapper
from typing import List
```

```
def web_search(
    web_query: str,
    num_results: int) -> List[str]:
    return [r["link"]
        for r in DuckDuckGoSearchAPIWrapper().results(
            web_query, num_results)]
```

Create a separate Python file, such as `web_searching_try.py`, to test the search function:

```
from web_searching import web_search

result = web_search(
    web_query = "How many titles did Michael Jordan win?",
    num_results=5)
print(result)
```

> **NOTE** If you're unfamiliar with VS Code, you can execute the code by pressing F5 and then selecting Python Debugger > Python File. You can also set breakpoints and run the code step-by-step by pressing F10 on each line.

In the terminal, you'll get a list of URLs like the following, representing the results of your search:

```
['https://en.wikipedia.org/wiki/
List_of_career_achievements_by_Michael_Jordan', 'https://sportsbrief.com/nba/
40447-michael-jordans-achievements-awards-a-list-mjs-accomplishments/',
'https://www.rookieroad.com/basketball/how-many-championships-does-michael-
jordan-have-5263621/', 'https://www.hoopsaddict.com/michael-jordan-awards/',
'https://www.wsn.com/nba/michael-jordan-championship-rings/']
```

> **NOTE** Other web search engine wrappers provided by LangChain are `TavilySearchResults` and `GoogleSearchAPIWrapper`. Both require an API key, so I chose `DuckDuckGoSearchAPIWrapper` because it doesn't.

4.3.2 Implementing web scraping

We'll scrape the web pages from the result list using Beautiful Soup, which is a web scraper library. Place the code shown in the following listing in a file named `web_scraping.py`.

Listing 4.1 Code for the `web_scraping.py` file

```
import requests
from bs4 import BeautifulSoup

def web_scrape(url: str) -> str:
    try:
        headers = {                          # Add request headers; otherwise, Wikipedia will block the call.
            "User-Agent": (
                "Mozilla/5.0 (Windows NT 10.0; Win64; x64) "
                "AppleWebKit/537.36 (KHTML, like Gecko) "
```

```
                "Chrome/124.0.0.0 Safari/537.36"
            ),
            "Accept-Language": "en-US,en;q=0.9",
        }
        response = requests.get(url, headers=headers, timeout=15)

        if response.status_code == 200:
            soup = BeautifulSoup(response.text, "html.parser")
            page_text = soup.get_text(separator=" ", strip=True)

            return page_text
        else:
            return f"Could not retrieve the webpage: Status code
            ➥{response.status_code}"
    except Exception as e:
        print(e)
        return f"Could not retrieve the webpage: {e}"
```

Let's try out the function by placing the following code in a file named `web_scraping_try.py`:

```
from web_scraping import web_scrape
result = web_scrape('https://en.wikipedia.org/wiki
➥/List_of_career_achievements_by_Michael_Jordan')
print(result)
```

After you run this script, the output will be similar to the following excerpt:

```
List of career achievements by Michael Jordan - Wikipedia Jump to content
Main menu Main menu move to sidebar hide Navigation Main page Contents
Current events Random article About Wikipedia Contact us Donate Contribute
Help Learn to edit Community portal Recent changes Upload file Languages
Language links are at the top of the page across from the title. Search web
scraping Create account Log in Personal tools [SHORTENED…]
```

As you can see, much of the scraped content isn't relevant, but we don't need to worry for now. The LLM will extract the relevant bits we're interested in.

4.3.3 Instantiating the LLM client

For this use case, we'll use the OpenAI GPT-5 nano model. First, create an `.env` file in the root of the `ch04` folder with the following content, replacing `<YOUR_OPENAI_KEY>` with your actual OpenAI API key:

```
OPENAI_API_KEY=<YOUR_OPENAI_KEY>
```

Alternatively, if you cloned the GitHub repository or downloaded the code zip file from the Manning website, rename the `.env_example` file to `.env`, and then replace `<YOUR_OPENAI_KEY>` with your actual OpenAI API key.

Create a function to instantiate the OpenAI client as follows, and place it in a file named `llm_models.py` (replace `YOUR_OPENAI_API_KEY` with your OpenAI key):

```
from langchain_openai import ChatOpenAI
from dotenv import load_dotenv
```

```
import os

load_dotenv()
openai_api_key = os.getenv("OPENAI_API_KEY")

def get_llm():
  return ChatOpenAI(openai_api_key=openai_api_key,
       model_name="gpt-5-nano")
```

Loads the environment variables from the .env file

Gets the OpenAI API key from the environment variables

Instantiates and returns the ChatOpenAI model

4.3.4 JSON to Python object converter

Let's make a utility function that converts JSON text from the LLM into a Python object, which is usually a dictionary or sometimes a list. If the JSON is malformed, it will return an empty dictionary. Add the following code to a file called `utilities.py`:

```
import json

def to_obj(s):
    try:
        return json.loads(s)
    except Exception:
        return {}
```

To recap, we've set up a VS Code project and implemented the core capabilities we need. Before building the engine to orchestrate web searching, web scraping, and summarization requests to the LLM, let's step back and take a second look at the big picture.

4.4 Enhancing the architecture with query rewriting

Let's revisit the diagram in figure 4.2, which I've included again for convenience as figure 4.3 here. If you look closely at this architecture diagram, you might notice a potential improvement: instead of sending the original research request directly to the web search engine, you can use the LLM to create multiple search queries. This technique, called *query rewriting* or *multiple query generation*, is similar to a method used to enhance Retrieval-Augmented Generation (RAG) searches, which I'll cover in later chapters. Rewriting the original question into multiple queries offers several advantages. It can clarify ambiguous or unclear queries, fix grammar or syntax errors that could confuse the search engine, and add context to short queries to improve search results.

For complex queries, breaking them into simpler queries provides useful context for better answers. This is the key reason for rewriting a single query into multiple targeted searches. For instance, rather than querying the search engine with "How many titles did Michael Jordan win?" you can prompt ChatGPT as follows:

> Generate three web search queries to research the following topic, aiming for a comprehensive report: "How many titles did Michael Jordan win?"

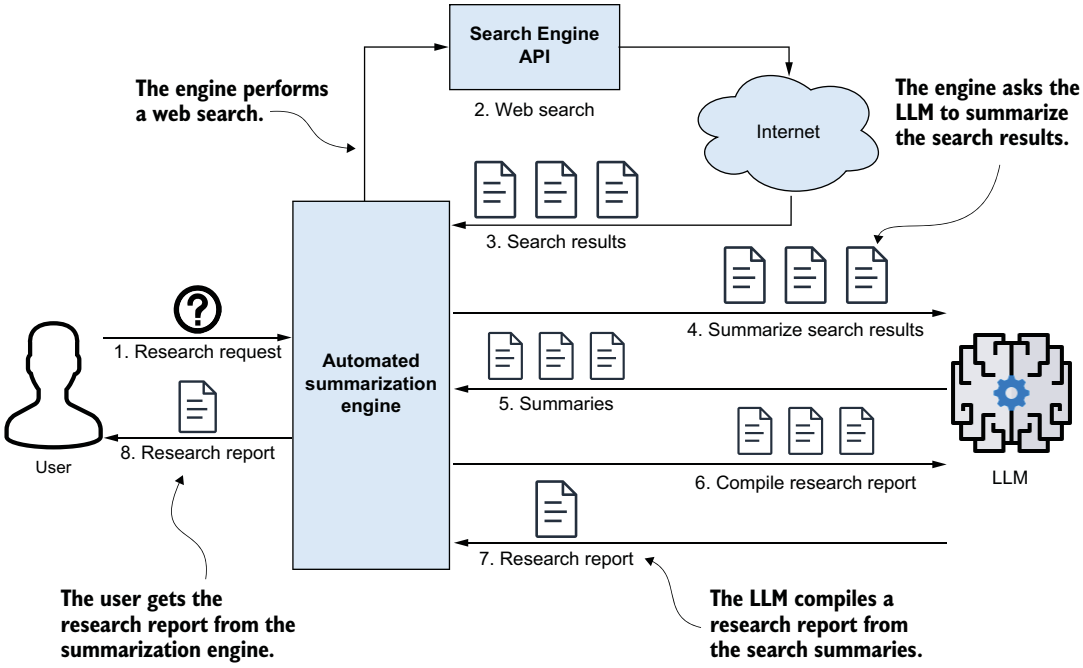


Figure 4.3 Automated research summarization engine. Ask a question, and the engine performs a web search, returns URLs, scrapes and summarizes web pages, and compiles a research report for you.

You'll receive queries like these:

```
1 "Michael Jordan NBA championships total"
2 "List of NBA titles won by Michael Jordan"
3 "Michael Jordan basketball career championships count"
```

Consequently, the engine will conduct research by performing these three web searches and gathering results from each. This updated workflow is depicted in the diagram in figure 4.4.

Technically, the updated architecture is more complex, which could raise concerns about performance due to the additional web searches. If processed sequentially, the response time would increase linearly; for instance, running four web searches would take four times as long. However, you can speed up processing through parallelization, which I'll cover in later sections. For now, we'll start with a sequential processing approach and explore parallelization later.

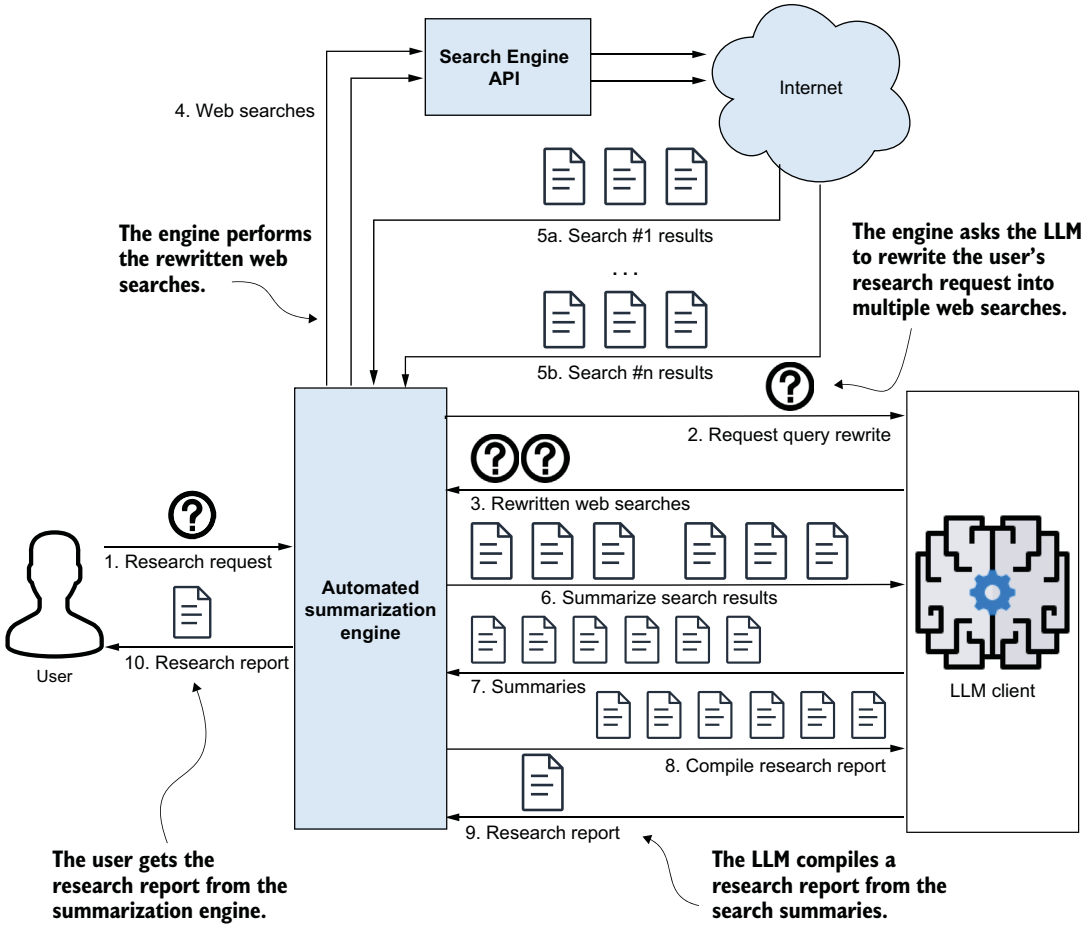


Figure 4.4 Revised system architecture diagram, incorporating query rewriting. The process begins by tasking the LLM to generate a specified number of queries based on the user's research question. These queries are then submitted to the search engine. The subsequent processing remains consistent with the illustration in figure 4.2.

4.5 Prompt engineering

Before we assemble the building blocks from section 4.3, we need to focus on some prompt engineering. We'll create prompts for generating web search queries, summarizing individual web pages, and producing the final report. Let's tackle each of these tasks step-by-step.

4.5.1 Crafting web search prompts

Imagine a user asks a finance-related question such as, "Should I invest in Apple stocks?" When guiding the LLM to generate web search queries based on this question, it's helpful to specify the persona that should frame these queries. For example,

you might begin the prompt with this: "You are an experienced finance analyst AI assistant. Your objective is to create detailed, insightful, unbiased, and well-structured financial reports based on the provided data and trends."

To generate these instructions dynamically based on the research question, you'll use a dedicated prompt. Start by designing a prompt that selects a suitable research assistant and provides instructions tailored to the user's query. Create a file named `prompts.py`, and begin with the following code.

Listing 4.2 `prompts.py`: Generating assistant instructions

```
from langchain_core.prompts import PromptTemplate

ASSISTANT_SELECTION_INSTRUCTIONS = """
You are skilled at assigning a research question to the correct
research assistant.
There are various research assistants available, each specialized
in an area of expertise.
Each assistant is identified by a specific type. Each assistant
has specific instructions to undertake
the research.

How to select the correct assistant: you must select the
relevant assistant depending on the topic of the question,
which should match the area of expertise of the assistant.

------
Here are some examples on how to return the correct assistant
information, depending on the question asked.

Examples:
Question: "Should I invest in Apple stocks?"
Response:
{{
    "assistant_type": "Financial analyst assistant",
    "assistant_instructions": "You are a seasoned finance
analyst AI assistant. Your primary goal is to compose
comprehensive, astute, impartial, and methodically arranged
financial reports based on provided data and trends.",
    "user_question": {user_question}
}}
Question: "what are the most interesting sites in Tel Aviv?"
Response:
{{
    "assistant_type": "Tour guide assistant",
    "assistant_instructions": "You are a world-travelled
AI tour guide assistant. Your main purpose is to draft
engaging, insightful, unbiased, and well-structured
travel reports on given locations, including history,
attractions,
and cultural insights.",
    "user_question": "{user_question}"
}}
```

Assistant selection instructions prompt

```
Question: "Is Messi a good soccer player?"
Response:
{{
    "assistant_type": "Sport expert assistant",
    "assistant_instructions": "You are an experienced AI
sport assistant. Your main purpose is to draft engaging,
insightful, unbiased, and well-structured sport reports on
given sport personalities, or sport events, including
factual details, statistics and insights.",
    "user_question": "{user_question}"
}}

------
Now that you have understood all the above, select the
correct research assistant for the following question.
Question: {user_question}
Response:

"""

ASSISTANT_SELECTION_PROMPT_TEMPLATE = PromptTemplate.from_template(
    template=ASSISTANT_SELECTION_INSTRUCTIONS
)
```

This prompt template has a couple of key features. First, it's a few-shot prompt, as demonstrated by the three examples included, which help the LLM grasp our specific requirements. Second, it directs the LLM to return results in JSON format, making it straightforward to convert the output into a Python dictionary for further processing.

With this prompt in place to select the appropriate assistant type and provide instructions, you can now proceed to create the prompt for generating web searches based on the user's question, as shown in the following listing.

Listing 4.3 `prompts.py`: Prompt for rewriting the user query

```
WEB_SEARCH_INSTRUCTIONS = """
{assistant_instructions}

Write {num_search_queries} web search queries to gather
as much information as possible
on the following question: {user_question}. Your objective is
to write a report based on the information you find.
You must respond with a list of queries such as
query1, query2, query3 in the following format:
[
    {{"search_query": "query1", "user_question": "{user_question}" }},
    {{"search_query": "query2", "user_question": "{user_question}" }},
    {{"search_query": "query3", "user_question": "{user_question}" }}
]
"""

WEB_SEARCH_PROMPT_TEMPLATE = PromptTemplate.from_template(
    template=WEB_SEARCH_INSTRUCTIONS
)
```

The {assistant_instructions} placeholder will be filled with the "assistant_instructions" output from the previous assistant selection prompt you saw. This prompt also returns results in JSON format, making it easy to convert the output into a list of Python dictionaries for further processing. Including the user question in the output ensures continuity throughout the process, allowing us to use it as needed, especially in the chain-based code that follows. With the web search prompt completed, let's move on to creating the summarization prompts.

4.5.2 Crafting summarization prompts

The prompt for summarizing result pages closely resembles prompts from the previous chapter on summarization. The code is shown in the following listing.

Listing 4.4 prompts.py: Prompt for summarizing result pages

```
SUMMARY_INSTRUCTIONS = """
Read the following text:
Text: {search_result_text}

-----------

Using the above text, answer in short the following question.
Question: {search_query}

If you cannot answer the question above using the text provided
above, then just summarize the text.
Include all factual information, numbers, stats etc if available.
"""

SUMMARY_PROMPT_TEMPLATE = PromptTemplate.from_template(
    template=SUMMARY_INSTRUCTIONS
)
```

4.5.3 Research report prompt

Similarly, the prompt for generating the research report is straightforward. In the following listing, note the clear and emphatic instructions designed to ensure the LLM produces the desired output.

Listing 4.5 prompts.py: Prompt for generating the research report

```
# Research Report prompts adapted from
# https://github.com/assafelovic/gpt-researcher/
# blob/master/gpt_researcher/master/prompts.py
RESEARCH_REPORT_INSTRUCTIONS = """
You are an AI critical thinker research assistant. Your sole
purpose is to write well written, critically acclaimed,
objective and structured reports on given text.

Information:
--------
{research_summary}
--------
```

```
Using the above information, answer the following question
or topic: "{user_question}" in a detailed report -- \
The report should focus on the answer to the question,
should be well structured, informative, \
in depth, with facts and numbers if available and a minimum of 1,200 words.

You should strive to write the report as long as you can using
all relevant and necessary information provided.
You must write the report with markdown syntax.
You MUST determine your own concrete and valid opinion based
on the given information. Do NOT infer general and meaningless
conclusions.
Write all used source urls at the end of the report, and make sure
to not add duplicated sources, but only one reference for each.
You must write the report in apa format.
Please do your best, this is very important to my career."""

RESEARCH_REPORT_PROMPT_TEMPLATE = PromptTemplate.from_template(
    template=RESEARCH_REPORT_INSTRUCTIONS
)
```

4.6 Initial implementation

In this section, I'll walk you through the initial implementation of our research summarization engine, following the steps laid out in the architectural diagram in figure 4.4. This first version is intentionally designed to be easy to follow, allowing you to see each part of the process in isolation before I introduce more advanced optimizations. As you work through the code, you'll notice how the different components—from prompt templates to web scraping utilities—interact to form a complete research workflow. The goal here is not only to implement the engine but also to build an intuitive understanding of how the system pieces fit together. By the end of this section, you'll have a fully functioning pipeline capable of performing automated web research and generating a structured report. This foundation will prepare you for the improvements we'll explore in the next section, where we shift from a sequential design to a more efficient LCEL-based approach.

4.6.1 Importing functions and prompt templates

To kick things off, create a Python file named `research_engine_seq.py`. Then, begin importing the functions and prompt templates you've crafted in the preceding sections:

```
from web_searching import web_search
from web_scraping import web_scrape
from llm_models import get_llm
from utilities import to_obj
from prompts import (
    ASSISTANT_SELECTION_PROMPT_TEMPLATE,
    WEB_SEARCH_PROMPT_TEMPLATE,
    SUMMARY_PROMPT_TEMPLATE,
    RESEARCH_REPORT_PROMPT_TEMPLATE
)
```

4.6.2 Setting constants and input variables

Now let's establish some constants to configure the application. The accuracy and diversity of the report will depend on the number of web searches and results per search you set. However, it's important to consider your budget when setting these values. For instance, configuring four web searches with five results per search would entail summarizing 20 web pages, which means roughly 2,000 tokens per page, for a total of 20 × 2,000 = 40,000 tokens. With the OpenAI GPT-5 mini model costing around $0.00005 per 1,000 output tokens, this would amount to approximately $0.002 per research request. I'll set lower configuration numbers, but feel free to adjust them as needed:

```
NUM_SEARCH_QUERIES = 2
NUM_SEARCH_RESULTS_PER_QUERY = 3
RESULT_TEXT_MAX_CHARACTERS = 10000
```

The sole input variable in the application is the one that captures the research question from the user:

```
question = 'What can I see and do in the Spanish town of Astorga?'
```

4.6.3 Instantiating the LLM client

Instantiating the LLM client is straightforward. You can do it simply as follows:

```
llm = get_llm()
```

4.6.4 Generating the web searches and collecting the results

In the process of generating web searches and collecting results, the first step is to execute the LLM prompt to determine the correct research assistant and related instructions based on the user's research question:

```
assistant_selection_prompt = ASSISTANT_SELECTION_PROMPT_TEMPLATE
➥.format(user_question=question)
assistant_instructions = llm.invoke(assistant_selection_prompt)
```

To execute this code, place a breakpoint on the line `llm = get_llm()`. If you're new to VS Code, you can set a breakpoint by clicking to the left of the line number where you want the debugger to pause.

Next, start the debugger by clicking the Run & Debug icon on the left sidebar (or use the shortcut Ctrl-Shift-D on Windows). From the dropdown at the top, select the Run configuration named Python Debugger: Current File, which you created at the end of section 4.2. Then, click the Play button next to it to start debugging.

Once the code runs and hits the breakpoint, inspect the value of the `assistant_instructions` variable in the Variables panel at the top left of the screen. Alternatively, you can print the value manually in the Debug Console panel at the bottom of the VS Code screen:

```
print(assistant_instructions)
```

You'll observe the following output:

```
content='{\n    "assistant_type": "Tour guide assistant",\n
"assistant_instructions": "You are a world-travelled AI tour guide assistant.
Your main purpose is to draft engaging, insightful, unbiased, and well-
structured travel reports on given locations, including history, attractions,
and cultural insights.",\n    "user_question": "What can I see and do in the
Spanish town of Astorga"\n}' response_metadata={'token_usage':
{'completion_tokens': 82, 'prompt_tokens': 432, 'total_tokens': 514},
'model_name': 'gpt-4o-mini-2024-07-18', 'system_fingerprint':
'fp_3bc1b5746c', 'finish_reason': 'stop', 'logprobs': None}
```

The relevant information is in the `content` property. To convert it into a Python object, you can proceed as follows:

```
assistant_instructions_dict = to_obj(assistant_instructions.content)
```

Printing the `assistant_instructions_dict` variable will yield the following output:

```
{'assistant_type': 'Tour guide assistant', 'assistant_instructions': 'You are
a world-travelled AI tour guide assistant. Your main purpose is to draft
engaging, insightful, unbiased, and well-structured travel reports on given
locations, including history, attractions, and cultural insights.',
'user_question': 'What can I see and do in the Spanish town of Astorga?'}
```

Now you can execute the prompt to generate web searches based on the original user research question:

```
web_search_prompt = WEB_SEARCH_PROMPT_TEMPLATE.format(
  assistant_instructions=assistant_instructions_dict[
        'assistant_instructions'],
  num_search_queries=NUM_SEARCH_QUERIES,
  user_question=assistant_instructions_dict[
        'user_question'])
web_search_queries = llm.invoke(web_search_prompt)
web_search_queries_list = to_obj(
    web_search_queries.content.replace('\n', ''))
```

The primary input for this prompt is the `assistant_instructions` output from the previous step. If you were to execute the code you've just written, upon printing `web_search_queries_list`, you would get a list of search queries like this:

```
[{'search_query': 'Astorga attractions', 'user_question': 'What can I see and
do in the Spanish town of Astorga?'}, {'search_query': 'Astorga history',
'user_question': 'What can I
see and do in the Spanish town of Astorga?'}]
```

Here's how you can fetch web searches using the `web_search()` function:

```
searches_and_result_urls = [{
    'result_urls': web_search(
   web_query=wq['search_query'],
   num_results=NUM_SEARCH_RESULTS_PER_QUERY),
    'search_query': wq['search_query']}
   for wq in web_search_queries_list]
```

Executing up to this point, the `searches_and_result_urls` variable would hold a list of Python dictionaries like this:

```
[{'result_urls': ['https://igotospain.com/one-day-in-astorga-on-the-camino-
de-santiago/', 'https://worldfreetours.com/blog/astorga/explore-astorga-for-
free/', 'https://budtravelagency.com/things-to-do-in-astorga-spain/'],
'search_query': 'things to do in Astorga'}, {'result_urls': ['https://
igotospain.com/one-day-in-astorga-on-the-camino-de-santiago/', 'https://
worldfreetours.com/blog/astorga/explore-astorga-for-free/', 'https://
www.thingstodoguru.com/things-to-do-in-astorga-es'], 'search_query': 'top
attractions in Astorga'}, {'result_urls': ['https://www.worldatlas.com/
cities/astorga-spain.html', 'https://citiesandattractions.com/spain/astorga-
spain-uncovering-the-jewels-of-a-hidden-spanish-gem/', 'https://ewtn.co.uk/
article-spanish-civil-war-martyrs-of-astorga-didnt-let-themselves-be-
overcome-by-fear/'], 'search_query': 'history of Astorga Spain'}]
```

Each dictionary shows a search query and its corresponding result URLs (three for each query here). The next step is to flatten the results so each dictionary contains a search query and just one result URL:

```
search_query_and_result_url_list = []
for qr in searches_and_result_urls:
    search_query_and_result_url_list.extend([{
        'search_query': qr['search_query'],
        'result_url': r}
           for r in qr['result_urls']])
```

Now `search_query_and_result_url_list` has six dictionaries, just as expected:

```
[{'search_query': 'Astorga attractions', 'result_url': 'https://
www.worldatlas.com/cities/astorga-spain.html'}, {'search_query': 'Astorga
attractions', 'result_url': 'https://citiesandattractions.com/spain/astorga-
spain-uncovering-the-jewels-of-a-hidden-spanish-gem/'}, {'search_query':
'Astorga attractions', 'result_url': 'https://www.caminoadventures.com/blog/
two-weeks-on-the-camino-de-santiago/'}, {'search_query': 'Astorga cultural
sites', 'result_url': 'https://www.worldatlas.com/cities/astorga-spain.html'},
{'search_query': 'Astorga cultural sites', 'result_url': 'https://interrailero
.com/que-ver-en-astorga/'}, {'search_query': 'Astorga cultural sites',
'result_url': 'https://www.atlasobscura.com/places/palacio-episcopal'}]
```

With the web search queries and all related result URLs ready, the next move is to start scraping the web pages linked to these URLs.

4.6.5 Scraping the web results

Now you'll use the `web_scrape()` function to pull text from web pages linked through your search results. Here's the code to do so:

```
result_text_list = [{
    'result_text': web_scrape(
   url=re['result_url'])[:RESULT_TEXT_MAX_CHARACTERS],
    'result_url': re['result_url'],
    'search_query': re['search_query']}
   for re in search_query_and_result_url_list]
```

This code populates `result_text_list` with six dictionaries, each carrying text from a web page:

```
[{'result_text': 'Astorga, Spain - WorldAtlas Astorga, Spain Astorga is a
municipality of just over 11,000 residents (as of the most recent 2018
estimates) in Northwestern Spain . This former Roman settlement (still
partially encircled by the ancient, albeit reconstructed walls) is the
capital of the traditional county, Maragatería, … Science Social Science
Society Economics Politics About Us Contact Us Privacy Copyright Search
WorldAtlas', 'result_url': 'https://www.worldatlas.com/cities/astorga-
spain.html', 'search_query': 'Astorga attractions'},
{'result_text': "Astorga, Spain: Uncovering the Jewels of a Hidden Spanish
Gem Skip to content Cities and Attractions Roaming Around: Your Guide to
Exploring the World Search for: Cities and Attractions Close menu Cities and
Attractions … Uncovering the Jewels of a Hidden Spanish Gem March 13, 2023
May 28, 2023 Spain Discover the charms of Astorga, Spain - from stunning
cathedrals and museums to hidden gems like the Chocolate Factory
Museum and Gaudi's Palace… ', 'result_url': 'https://
citiesandattractions.com/spain/astorga-spain-uncovering-the-jewels-of-a-
hidden-spanish-gem/'}, … ]
```

The next move is to summarize the information collected from each web page.

4.6.6 Summarizing the web results

You'll ask the language model to summarize the text from each web page using the `SUMMARY_PROMPT_TEMPLATE` you set up earlier. This summary will also keep the original search queries and URLs, which you'll need later:

```
result_text_summary_list = []
for rt in result_text_list:
    summary_prompt = SUMMARY_PROMPT_TEMPLATE.format(
    search_result_text=rt['result_text'],
    search_query=rt['search_query'])

    text_summary = llm.invoke(summary_prompt)

    result_text_summary_list.append({
        'text_summary': text_summary,
        'result_url': rt['result_url'],
        'search_query': rt['search_query']})
```

This process results in `result_text_summary_list`, a list of nine dictionaries. Each contains a summary of the scraped text, the URL where it was found, and the search query used to find it:

```
[{'text_summary': '\nAstorga is a municipality in Northwestern Spain with a
population of over 11,000 people. It is the capital of the Maragatería county
in the province of León, within the autonomous community of Castilla y León….
pilgrims on the Camino de Santiago. ', 'result_url': 'https://
www.worldatlas.com/cities/astorga-spain.html', 'search_query': 'Astorga
attractions'}, {'text_summary': '\nSome of the main attractions in Astorga,
Spain include … culture, and natural beauty. ', 'result_url': 'https://
```

```
citiesandattractions.com/spain/astorga-spain-uncovering-the-jewels-of-a-
hidden-spanish-gem/', 'search_query': 'Astorga attractions'}, …]
```

With these summaries, you've compiled all the information needed for the final research report.

4.6.7 Generating the research report

Let's review the prompt used to create the final report. At this stage, all the earlier components of the pipeline come together, and the model is finally given the full collection of summarized information it needs to generate a coherent, structured response:

```
RESEARCH_REPORT_INSTRUCTIONS = """
You are an AI critical thinker research assistant.
Your sole purpose is to write well written,
critically acclaimed, objective and structured
reports on given text.

Information:
--------
{research_summary}
--------

Using the above information, answer the …
…
"""
```

Let's prepare the final report by combining summaries and URLs into a format the prompt template expects. First, transform each dictionary into a string with the summary and its source URL:

```
stringified_summary_list = [
    f'Source URL: {sr["result_url"]}\nSummary: {sr["text_summary"]}'
        for sr in result_text_summary_list]
```

Inspecting `stringified_summary_list`, you'll find entries like these:

```
['Source URL: https://www.worldatlas.com/cities/astorga-spain.html\nSummary:
\nAstorga is a municipality in Northwestern Spain with a population of over
11,000 people. It is the capital of the Maragatería … pilgrims on the Camino
de Santiago. ', 'Source URL: https://citiesandattractions.com/spain/astorga-
spain-uncovering-the-jewels-of-a-hidden-spanish-gem/\nSummary: \nSome of the
main attractions in Astorga, Spain include the Episcopal Palace, the
Cathedral… and natural beauty. ', …]
```

Next, combine all the summary strings into one:

```
appended_result_summaries = '\n'.join(stringified_summary_list)
```

This gives you a single text block with all summaries and URLs:

```
Source URL: https://www.worldatlas.com/cities/astorga-spain.html
```

```
Summary:
Astorga is a municipality in Northwestern Spain with a population of over
11,000 people. It is the capital of the Maragatería … Astorga has a rich
history, including being a former Roman settlement, and has undergone periods
of decline and resurgence due to various conflicts. It is a popular stop for
tourists and pilgrims on the Camino de Santiago.
Source URL: https://citiesandattractions.com/spain/astorga-spain-uncovering-
the-jewels-of-a-hidden-spanish-gem/
Summary:
Some of the main attractions in Astorga, Spain include the Episcopal Palace,
the Cathedral of Santa Maria de Astorga, the Roman Walls and Museum, the
Chocolate Factory Museum, Gaudi's Palace, …
```

Now use this content with the `RESEARCH_REPORT_INSTRUCTIONS` prompt template to get the final research report:

```
research_report_prompt = RESEARCH_REPORT_PROMPT_TEMPLATE.format(
    research_summary=appended_result_summaries,
    user_question=question
)
research_report = llm.invoke(research_report_prompt)

print(f'strigified_summary_list={stringified_summary_list}')
print(f'merged_result_summaries={appended_result_summaries}')
print(f'research_report={research_report}')
```

If you run the entire `research_engine_seq.py` script, after about a minute, you'll receive a complete research report like the following, based on web summaries:

```
# Introduction
Astorga is a charming town located in the northwestern region of Spain, with
a population of over 11,000 residents. It is the capital of the Maragatería
county in the province of León, within the autonomous community of Castilla y
León. Astorga is a town known for its rich history, cultural significance,
and natural beauty. It is a popular destination for tourists and pilgrims on
the Camino de Santiago, with two Camino routes converging in the city. In
this report, we will explore the main attractions and activities that can be
seen and done in the Spanish town of Astorga.
# Historical and Cultural Significance
Astorga is a town with a long and rich history, dating back to the Roman
Empire. It was a former Roman settlement and has undergone periods of decline
and resurgence due to various conflicts. Today, visitors can still see the
remnants of the Roman walls and ruins, which are one of the main attractions
in Astorga. The city is also home to the 15th-century Cathedral of Astorga, a
mix of Gothic, Renaissance, and Baroque styles, and the neo-Gothic Episcopal
Palace designed by the famous Catalan architect Antoni Gaudí.
[… SHORTENED]
```

Great job on completing your first automated web research! I recommend going through the code for this initial implementation again. If you didn't have time to type

and run it, be sure to check it out in the GitHub repository. You'll likely understand it right away, but a review will help ensure everything is clear.

This initial version works well and generates the expected research report. However, it's a bit slow because it processes tasks sequentially. If we had set it up to handle 10 web searches with 10 results each, the time required would have significantly increased.

Can we make it faster? Absolutely. In the next section, we'll explore how to do that via LCEL, which was introduced in chapter 3.

4.7 Reimplementing the research summary engine in LCEL

The LangChain Expression Language (LCEL) offers a structured approach to organizing the core components of your LLM applications—such as web search, page scraping, and summarization—into an efficient chain or pipeline. This framework not only simplifies the creation of complex workflows from simple elements but also enhances them with advanced features such as streaming, parallel execution, and logging.

LangChain Expression Language (LCEL)

For those developing LLM applications, using LCEL is highly recommended. It allows you to interact with LLMs and chat models efficiently by creating and executing chains, providing several benefits:

- *Fallback*—Enables adding a fallback action for error handling
- *Parallel execution*—Executes independent chain components simultaneously to boost performance
- *Execution modes*—Supports developing in synchronous mode and then switching to streaming, batch, or asynchronous execution modes as needed
- *LangSmith tracing*—Automatically logs execution steps when upgrading to LangSmith, facilitating debugging and monitoring

A chain follows the `Runnable` protocol, meaning it requires the implementation of specific methods such as `invoke()`, `stream()`, and `batch()`, including their asynchronous versions. LangChain's framework ensures that its components, such as `PromptTemplate` and `JsonOutputFunctionsParser`, adhere to these standards.

LCEL streamlines complex chain creation by offering a unified interface (the `Runnable` protocol), composition tools, and the ability to easily parallelize processes. Though mastering LCEL may require some practice, the effort is rewarding, as it significantly enhances application performance and scalability.

My chain implementation strategy, shown in figure 4.5, involves constructing a mini-chain for each processing step shown earlier and integrating these into a master Web Research chain.

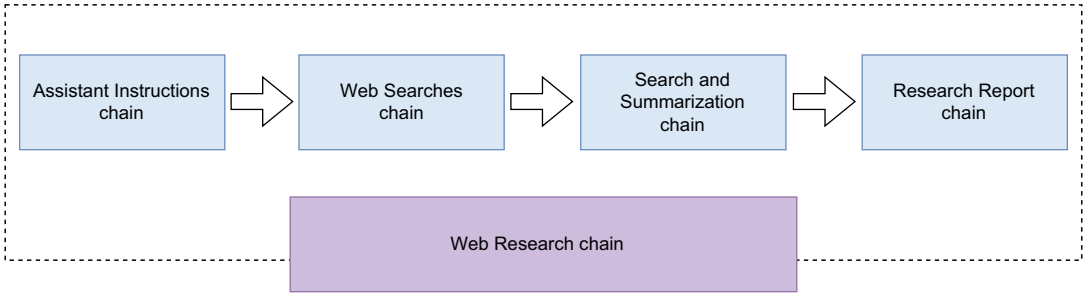


Figure 4.5 Architecture of the chain-based research summarization engine. Each step of the process is reimplemented as a mini-chain; all mini-chains are assembled into a master Web Research chain.

This master Web Research chain handles the entire process, as shown in figure 4.5, which illustrates the architecture of the chain-based research summarization engine. Each processing step is implemented as a mini-chain, all integrated into the master Web Research chain:

- *Assistant Instructions chain*—This chain selects the best research assistant from available options to answer the user's question. It also creates the system prompt that defines the assistant's skills and purpose.
- *Web Searches chain*—This chain generates multiple web searches based on the user's question. It provides context from different perspectives or breaks down complex queries into simpler ones.
- *Search and Summarization chain*—This chain performs web searches, retrieves URLs from search results, scrapes the relevant web pages, and summarizes the content of each page.
- *Research Report chain*—The final chain synthesizes the answer using the original question and the summaries generated from the search results.

The Search and Summarization chain coordinates three underlying chains within a single cohesive workflow. This is shown in figure 4.6.

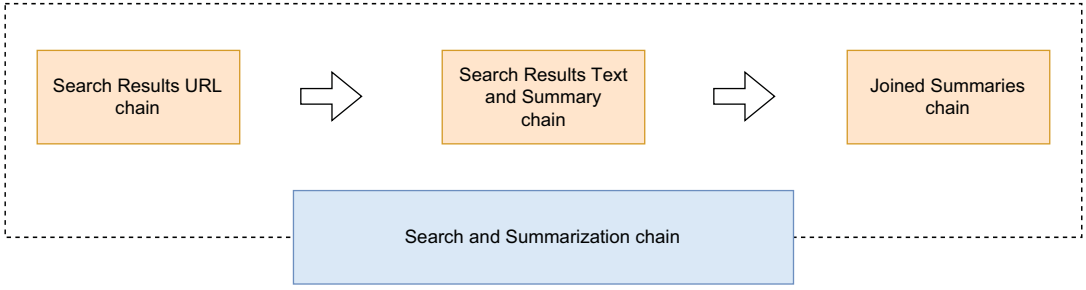


Figure 4.6 The Search and Summarization chain

The diagram in figure 4.6 shows how these three chains interact: a search chain that produces queries, a scraping-and-summarization chain that processes the resulting web pages, and a consolidation chain that merges all summaries into one final text block. Reimplementing this process into chains, as shown in figure 4.7, allows for parallel execution, significantly improving efficiency compared to the sequential approach in figure 4.5.

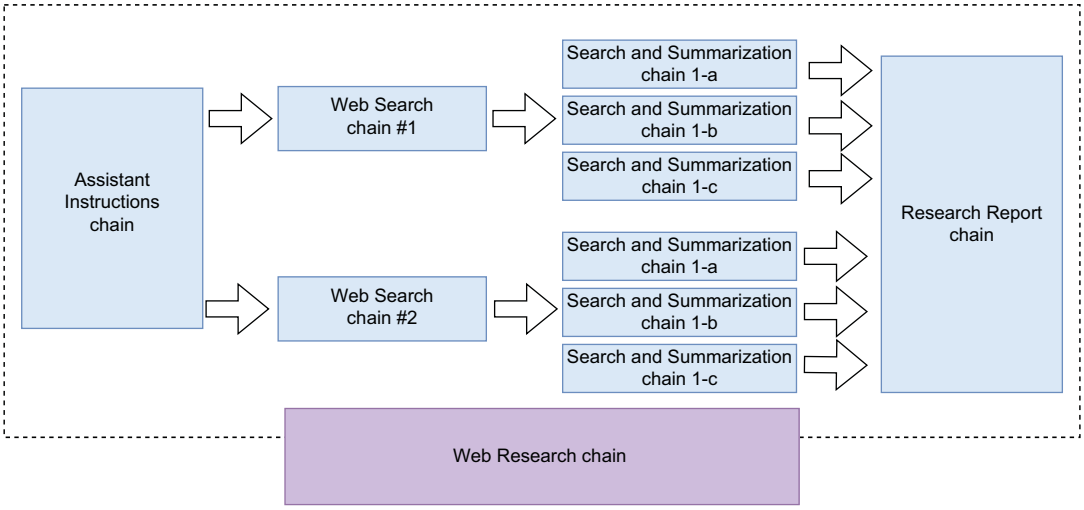


Figure 4.7 Chain architectural diagram showing how parallelization is applied in the research summarization engine, highlighting the execution of separate chain instances in parallel for efficiency

Figure 4.7 highlights parallelization at two key stages:

- A separate instance of the Search and Summarization chain is created and executed simultaneously for each web search initiated by the Web Searches chain.
- For each search result generated by the Search Result URLs chain, an individual Search Result Text and Summary chain instance is launched to run in parallel.

With this overview of the chain-based approach, we'll now explore the individual mini chains. Let's start with the Assistant Instructions chain.

4.7.1 Assistant Instructions chain

To begin processing a research question, the first task is to determine the most relevant research assistant and its prompt instructions. This is handled by the `assistant_instructions_chain`, as shown here. Place this code in a file named `chain_1_1.py`:

```
from llm_models import get_llm
from prompts import (
```

```
    ASSISTANT_SELECTION_PROMPT_TEMPLATE,
)
from langchain_core.output_parsers import StrOutputParser

assistant_instructions_chain = (
    ASSISTANT_SELECTION_PROMPT_TEMPLATE | get_llm()
)
```

For those new to LCEL syntax, the flow here is straightforward: the selection prompt feeds into the LLM, which then selects a research assistant based on the user's question. To test this setup, use the following script, saved as `chain_try_1_1.py`:

```
from chain_1_1 import assistant_instructions_chain

question = 'What can I see and do in the Spanish town of Astorga?'

assistant_instructions = assistant_instructions_chain.invoke(question)
print(assistant_instructions)
```

Running this code will produce output similar to what was shown in section 4.6.4, detailing the assistant type, instructions, and the user question (the output will come after a few seconds, and the metadata might look slightly different from what I've reported here):

```
content='{\n    "assistant_type": "Tour guide assistant",\n
"assistant_instructions": "You are a world-travelled AI tour guide assistant.
Your main purpose is to draft engaging, insightful, unbiased, and well-
structured travel reports on given locations, including history, attractions,
and cultural insights.",\n    "user_question": "What can I see and do in the
Spanish town of Astorga?"\n}' response_metadata={'token_usage':
{'completion_tokens': 82, 'prompt_tokens': 432, 'total_tokens': 514},
'model_name': 'gpt-4o-mini-2024-07-18', 'system_fingerprint':
'fp_3bc1b5746c', 'finish_reason': 'stop', 'logprobs': None}
```

While you could manually extract the needed output from the content property, using LCEL offers a more efficient approach. By adding a `StrOutputParser()` block to your chain, you can automatically extract the LLM's response directly from the content property. Update the chain accordingly, and save it as `chain_1_2.py`:

```
from llm_models import get_llm
from utilities import to_obj
from prompts import (
    ASSISTANT_SELECTION_PROMPT_TEMPLATE,
)
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser

assistant_instructions_chain = (
    {'user_question': RunnablePassthrough()}
    | ASSISTANT_SELECTION_PROMPT_TEMPLATE
    | get_llm() | StrOutputParser() | to_obj
)
```

This updated version introduces several enhancements:

- While the `ASSISTANT_SELECTION_PROMPT_TEMPLATE` automatically matches the input text question to the `{user_question}` field (refer to section 4.5.1 for a refresh on `{user_question}`), it's safer to explicitly map the input question to a `user_question` property in a Python dictionary using the `RunnablePassthrough()` function, which binds the external input to the `user_question` variable through an unnamed parameter. This is also beneficial because this piece of information is important for subsequent steps.
- The `StrOutputParser()` block extracts text from the LLM's content property, simplifying output handling.
- The response is converted to a Python dictionary by `to_obj()`, making it easier to work with in the chain.

Test the revised chain with this script, saved as `chain_try_1_2.py`:

```
from chain_1_2 import assistant_instructions_chain
question = 'What can I see and do in the Spanish town of Astorga?'

assistant_instructions_dict
➥= assistant_instructions_chain.invoke(question)
print(assistant_instructions_dict)
```

Test chain invocation

Running this should produce the expected output:

```
{'assistant_type': 'Tour guide assistant', 'assistant_instructions': 'You are
a world-travelled AI tour guide assistant. Your main purpose is to draft
engaging, insightful, unbiased, and well-structured travel reports on given
locations, including history, attractions, and cultural insights.',
'user_question': 'What can I see and do in the Spanish town of Astorga?'}
```

4.7.2 Web Searches chain

After the LLM selects the appropriate research assistant and details its role, you can prompt the LLM to generate web searches related to the user's query. Use the following code, saved as `chain_2_1.py`, for this step.

Listing 4.6 Chain for rewriting the user query into web searches

```
from llm_models import get_llm
from utilities import to_obj
from prompts import (
    WEB_SEARCH_PROMPT_TEMPLATE
)
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnableLambda

NUM_SEARCH_QUERIES = 2

web_searches_chain = (
    RunnableLambda(lambda x:
```

```
        {
            'assistant_instructions': x['assistant_instructions'],
            'num_search_queries': NUM_SEARCH_QUERIES,
            'user_question': x['user_question']
        }
    )
    | WEB_SEARCH_PROMPT_TEMPLATE
    | get_llm() | StrOutputParser() | to_obj
)
```

This implementation uses a `RunnableLambda` block to process input from the previous chain, transforming it into the format needed by `WEB_SEARCH_PROMPT_TEMPLATE`. The chain then continues similarly to previous examples. To test this chain, you can use the following script, saved as `chain_try_2_1.py`.

Listing 4.7 Script to test the Web Searches chain

```
from utilities import to_obj
from chain_2_1 import web_searches_chain

# test chain invocation
assistant_instruction_str = '{"assistant_type":
➥"Tour guide assistant",
➥"assistant_instructions": "You are a world-travelled
➥AI tour guide assistant. Your main purpose is to draft
➥engaging, insightful, unbiased, and well-structured travel
➥reports on given locations, including history, attractions,
➥and cultural insights.",
➥"user_question": "What can I see and do in the Spanish
➥town of Astorga?"}'
assistant_instruction_dict = to_obj(assistant_instruction_str)
web_searches_list = web_searches_chain.invoke(assistant_instruction_dict)
print(web_searches_list)
```

This script uses a mock input in `assistant_instruction_str` to mimic the output from the Assistant Instructions chain, testing the chain's response to this input. The expected output follows:

```
[{'search_query': 'Things to do in Astorga Spain', 'user_question': 'What can
I see and do in the Spanish town Astorga'}, {'search_query': 'Attractions in
Astorga Spain', 'user_question': 'What can I see and do in the Spanish town
Astorga'}, {'search_query': 'Historical sites in Astorga Spain',
'user_question': 'What can I see and do in the Spanish town Astorga'}]
```

The web searches are now generated. Let's proceed to the Search and Summarization chain next.

4.7.3 Search and Summarization chain

The Search and Summarization chain is designed to perform a web search based on a query from the previous chain, retrieve URLs from the search results, scrape the

corresponding web pages, and then summarize each page. As shown earlier in figure 4.6, this process is broken down into smaller sub-chains:

- Search Result URLs chain
- Search Result Text and Summary chain
- Joined Summary chain

We'll begin by building these sub-chains, starting with the Search Result URLs chain.

SEARCH RESULT URLS CHAIN

This sub-chain carries out a web search, retrieving a specific number of URLs from the search results. Save the following code as `chain_3_1.py`.

Listing 4.8 Search and Summarization chain

```
from web_searching import web_search
from langchain_core.runnables import RunnableLambda

NUM_SEARCH_RESULTS_PER_QUERY = 3

search_result_urls_chain = (
    RunnableLambda(lambda x:
        [
            {
                'result_url': url,
                'search_query': x['search_query'],
                'user_question': x['user_question']
            }
            for url in web_search(
                web_query=x['search_query'],
                num_results=NUM_SEARCH_RESULTS_PER_QUERY)
        ]
    )
)
```

Here are a couple of key points to note:

- A lambda function is used to pass the `web_query` parameter to the `web_search()` function. This function also formats the output as a list of dictionaries, each containing the resulting URL along with data from the previous chain, such as the search query and the original user question. These elements will be useful in subsequent stages.
- The lambda function receives its input from the output of the previous chain, specifically the Web Searches chain.

To test this sub-chain, we'll use simulated input that mirrors what would come from the Web Searches chain. Save the testing code shown in the following listing in a file named `chain_try_3_1.py`.

Listing 4.9 Script to test the Search and Summarization chain

```
from utilities import to_obj
from chain_3_1 import search_result_urls_chain

# test chain invocation
web_search_str = '{"search_query":
➥"Astorga Spain attractions",
➥"user_question": "What can I see and do in the
➥Spanish town of Astorga?"}'
web_search_dict = to_obj(web_search_str)
result_urls_list = search_result_urls_chain.invoke(web_search_dict)
print(result_urls_list)
```

After executing it, you should see output similar to this:

```
[{'result_url': 'https://loveatfirstadventure.com/astorga-spain/',
'search_query': 'Astorga Spain attractions', 'user_question': 'What can I see
and do in the Spanish town Astorga?'}, {'result_url': 'https://
igotospain.com/one-day-in-astorga-on-the-camino-de-santiago/',
'search_query': 'Astorga Spain attractions', 'user_question': 'What can I see
and do in the Spanish town Astorga'}, {'result_url': 'https://
citiesandattractions.com/spain/astorga-spain-uncovering-the-jewels-of-a-
hidden-spanish-gem/', 'search_query': 'Astorga Spain attractions',
'user_question': 'What can I see and do in the Spanish town Astorga'}]
```

SEARCH RESULT TEXT AND SUMMARY CHAIN

The Search Result Text and Summary sub-chain handles a URL from the previous sub-chain by doing the following:

- Scraping the web page's text using the provided URL
- Generating a summary of the scraped text
- Incorporating the source URL into the summary

These steps are effectively implemented in the following code, which should be saved as `chain_4_1.py`.

Listing 4.10 Search Result Text and Summary chain

```
from llm_models import get_llm
from web_scraping import web_scrape
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnableLambda, RunnableParallel
from prompts import (
    SUMMARY_PROMPT_TEMPLATE
)

RESULT_TEXT_MAX_CHARACTERS = 10000

search_result_text_and_summary_chain = (
    RunnableLambda(lambda x:
        {
```

```
            'search_result_text':
                web_scrape(url=x['result_url'])[
                    :RESULT_TEXT_MAX_CHARACTERS],
            'result_url': x['result_url'],
            'search_query': x['search_query'],
            'user_question': x['user_question']
        }
    )
    | RunnableParallel (
        {
            'text_summary': SUMMARY_PROMPT_TEMPLATE
               | get_llm() | StrOutputParser(),
            'result_url': lambda x: x['result_url'],
            'user_question': lambda x: x['user_question']
        }
    )
    | RunnableLambda(lambda x:
        {
            'summary':
               f"Source Url: {x['result_url']}\nSummary:
               ➥{x['text_summary']}",
            'user_question': x['user_question']
        }
    )
)
```

The code follows the conventions established in previous chains, and you might recall the purpose of `RunnableParallel` from chapter 3. The `RunnableParallel` block allows for the simultaneous execution of an inner chain (next to `text_summary`) along with other operations (in this case, two lambda functions) that depend on the same input, which comes from the initial `RunnableLambda` block. To execute this chain, save the following script as `chain_try_4_1.py`.

Listing 4.11 Testing the Search Result Text and Summary chain

```
from utilities import to_obj
from chain_4_1 import search_result_text_and_summary_chain

# test chain invocation
result_url_str = '{"result_url":
➥"https://citiesandattractions.com/spain/astorga-spain
➥-uncovering-the-jewels-of-a-hidden-spanish-gem/",
➥"search_query":"Astorga Spain attractions",
➥"user_question": "What can I see and do in the Spanish
➥town of Astorga?"}'
result_url_dict = to_obj(result_url_str)

search_text_summary =
➥search_result_text_and_summary_chain.invoke(result_url_dict)
print(search_text_summary)
```

You'll get output similar to this:

```
{'summary': 'Source Url: https://citiesandattractions.com/spain/astorga-
spain-uncovering-the-jewels-of-a-hidden-spanish-gem/\nSummary: \nAstorga,
Spain has several attractions including the Episcopal Palace, Cathedral of
Santa Maria de Astorga, Roman Walls and Museum, Chocolate Factory Museum,
Palace of Gaudi, Sierra de los Ancares, and wineries. The town is known for
its history, architecture, cuisine, and natural beauty. It offers unique
culinary experiences, such as the traditional dish "Cocido Maragato," and is
home to various hidden gems waiting to be discovered. ', 'user_question':
'What can I see and do in the Spanish town Astorga?'}
```

ASSEMBLING THE SEARCH AND SUMMARIZATION CHAIN

We've now assembled all the key components to build the Search and Summarization chain. Before diving into the LCEL implementation, take a moment to review figure 4.8. This diagram provides a clearer understanding of the process compared to the earlier figure 4.6.

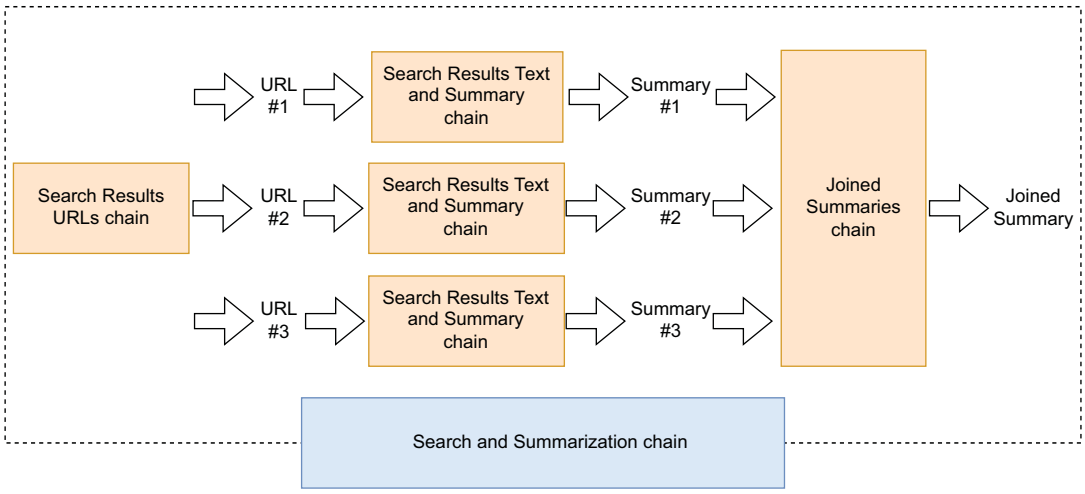


Figure 4.8 Enhanced Search and Summarization chain diagram

As illustrated in figure 4.8, the Search Result URLs chain generates multiple URLs, each of which triggers an instance of the Result Text and Summary chain. These instances run in parallel, each producing a summary for its respective web page. The Joined Summary chain then consolidates these summaries into a single text block. To implement this process, place the necessary code in a file named `chain_5_1.py`.

Listing 4.12 Search and Summarization chain

```
from llm_models import get_llm
from prompts import (
    RESEARCH_REPORT_PROMPT_TEMPLATE
)
```

```
from chain_1_2 import assistant_instructions_chain
from chain_2_1 import web_searches_chain
from chain_3_1 import search_result_urls_chain
from chain_4_1 import search_result_text_and_summary_chain

from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnableLambda

search_and_summarization_chain = (
    search_result_urls_chain
    | search_result_text_and_summary_chain.map() # parallelize for each url
    | RunnableLambda(lambda x:
        {
            'summary': '\n'.join([i['summary'] for i in x]),
            'user_question': x[0]['user_question'] if len(x) > 0 else ''
        })
)
```

The `map()` operator triggers multiple instances of the Result Text and Summary chain, one for each dictionary from the Search Result URLs chain containing a URL. This allows each instance to run simultaneously.

Additionally, the Joined Summaries sub-chain is integrated directly within the larger Search and Summarization chain rather than as a separate entity. This sub-chain merges summaries from each instance of the Result Text and Summary chain, functioning as a core part of the overall process. With these components in place, you're ready to complete the Web Research chain.

4.7.4 Web Research chain

Before diving into the implementation, let's take a look at figure 4.9 for a detailed overview of the Web Research chain. This builds on what was introduced in figure 4.7, similar to our approach with the Search and Summarization chain.

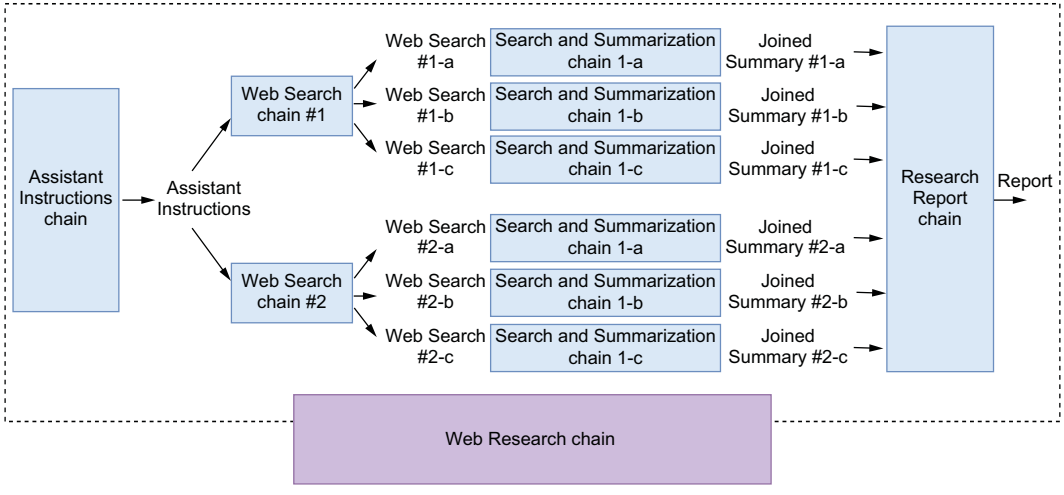


Figure 4.9 Overview of the Web Research chain

As shown in figure 4.9, web searches from the Web Searches chain trigger multiple instances of the Search and Summarization chains to run in parallel. Each chain generates a summary for a specific web search, and these summaries are then combined by the Research Report chain to create the final research report. The LCEL implementation of this process is outlined in the following listing. Add this code to the `chain_5_1.py` file.

Listing 4.13 Web Research chain

```
web_research_chain = (
    assistant_instructions_chain
    | web_searches_chain
    | search_and_summarization_chain.map() # parallelize for each web search
    | RunnableLambda(lambda x:
       {
           'research_summary': '\n\n'.join([i['summary'] for i in x]),
           'user_question': x[0]['user_question'] if len(x) > 0 else ''
        })
    | RESEARCH_REPORT_PROMPT_TEMPLATE | get_llm() | StrOutputParser()
)
```

This setup follows the same logic as the Search and Summarization chain. The `map()` operator is used here to initiate multiple instances of the Search and Summarization sub-chain, allowing them to run concurrently. The Research Report chain is then integrated as part of the overall Web Research chain. To test the Web Research chain, use the following script, and save it as `chain_try_5_1.py`.

Listing 4.14 Script to test the Web Research chain

```
from chain_5_1 import web_research_chain

question = 'What can I see and do in the Spanish town of Astorga?'

web_research_report = web_research_chain.invoke(question)
print(web_research_report)
```

After running this, you'll get a research report formatted in Markdown, as specified by the prompt. Note that the example report is shortened here for brevity:

```
# Introduction
Astorga, a small town in northwestern Spain, may not be on everyone's travel
radar, but it is a hidden gem waiting to be discovered. With a rich history
dating back to ancient Roman times, stunning architecture, delicious local
cuisine, and natural beauty, there is plenty to see and do in Astorga. In
this report, we will explore the top attractions and activities in Astorga,
along with practical information for planning a trip to this charming Spanish
town.
```

```
# History of Astorga
Astorga's history dates back to the ancient Roman settlement of Asturica
Augusta, founded in 14 BC. The town was an important mining center, and its
strategic location on the Pilgrim's Road, part of the Camino de Santiago,
made it a significant stop for pilgrims. Over the centuries, Astorga was
conquered and ruled by various civilizations, including the Visigoths, Moors,
and Christians. It has also been the site of many violent campaigns,
including the Spanish Civil War. Despite its tumultuous past, Astorga has
persevered and is now a peaceful and charming town with a rich cultural
heritage.
# Top Attractions in Astorga
## Episcopal Palace
One of the most iconic …
```

Congratulations! You've built an LCEL-based chain that integrates sub-chains, inline chains, lambda functions, and parallelization. This hands-on experience will make it easier for you to create your own LCEL chains. I recommend taking some time to experiment with the different chains in this application. Adjust the prompts, change the number of queries generated, or tailor the application to a specific type of web research you have in mind.

> **NOTE** The research summarization engine you've built draws inspiration from an open source project named GPT Researcher. Exploring its GitHub repository (https://github.com/assafelovic/gpt-researcher) will show you a robust platform supporting various search engines and LLMs, with features such as memory and compression. While it doesn't use LCEL, adapting its core functionalities into an LCEL-based framework was intended to clarify several technical aspects for you. I recommend exploring the GPT Researcher's codebase and running the project on your computer if possible. It's an excellent way to learn about building an LLM application, including aspects such as web UI integration, beyond just the engine itself.

Summary

- Research summarization engines generate comprehensive reports by querying multiple web sources and consolidating findings. They automate multisource research tasks that would otherwise require manual browsing and note-taking.
- A typical research summarization workflow consists of the following:
  - Accepting a research question from the user
  - Converting the question into multiple targeted web search queries using LLM reasoning
  - Extracting and summarizing content from each retrieved web page
  - Combining individual summaries into a unified final report
- The engine integrates three core operations: web search APIs (query submission), HTML parsing libraries (content extraction), and LLM calls (summarization and synthesis). Each operation handles errors independently.

- Prompt engineering shapes three critical stages:
  - *Search query generation*—Converting broad questions into specific, targeted search terms
  - *Content summarization*—Extracting key facts and arguments from raw web pages
  - *Report generation*—Combining partial summaries into coherent findings with citations
- LangChain Expression Language (LCEL) chains operations using the pipe operator (`|`) to compose sequential transformations. Each component's output becomes the next component's input automatically.
- `RunnableParallel` enables simultaneous execution of multiple operations on the same input. Wrap operations in `RunnableParallel({"key1": operation1, "key2": operation2})` to run them concurrently.
- The `.map()` operator triggers multiple chain instances, one for each item in a list. Use `chain.map()` to process URLs or search results in parallel, with each instance running simultaneously.
- `RunnableLambda` wraps Python functions for use in LCEL chains. Import with `from langchain_core.runnables import RunnableLambda` and use `RunnableLambda(lambda x: your_function(x))`.
- Chains implement the `Runnable` protocol with the following methods: `.invoke()` (single execution), `.stream()` (streaming output), and `.batch()` (batch processing), each with asynchronous versions for concurrent operations.
- Build chains modularly by creating separate chains for search query generation, URL fetching, content extraction, and summarization. Combine them with LCEL for a clean, maintainable architecture.
- `RunnablePassthrough` preserves input while allowing additional operations. Use it to pass the original question through the chain while also generating search queries from it.

5 Agentic workflows with LangGraph

This chapter covers

- An overview of agentic workflows and agents
- LangGraph fundamentals and state management
- Transitioning from LangChain chains to an agentic workflow

Large language models (LLMs) are driving a new generation of applications that require more than simple prompt–response exchanges. As applications become more complex, agentic workflows have become essential—a pattern where the LLM orchestrates a structured, multistep process using predefined components and explicit state management. Agentic workflows follow a well-defined and consistent sequence of steps. Instead of adapting their behavior dynamically during execution, they emphasize reliability, transparency, and control. In this approach, the LLM makes decisions within clearly defined boundaries, ensuring that each stage of the process remains structured and reproducible. Later in the book, we'll explore agent architectures that build on these principles to achieve greater autonomy and adaptability.

5.1 Understanding agentic workflows and agents

LLM-powered agent-based systems typically follow one of two core design patterns: agentic workflows and agents. Each pattern shapes how the application operates, as illustrated in figure 5.1. Because these terms are often used interchangeably—but have important differences—it's essential to understand exactly what is meant by agentic workflow and agent before diving deeper:

- *Agentic workflow* (or simply *workflow*)—Guides an application through a fixed sequence of predetermined steps. The LLM is used to select among predefined options, helping the system complete tasks and manage the overall flow.
- *Agent*—Uses language models for more than just task execution: they reason, make decisions, and dynamically determine the next steps based on available tools and evolving context. Here, a *tool* typically refers to a function that returns or processes data.

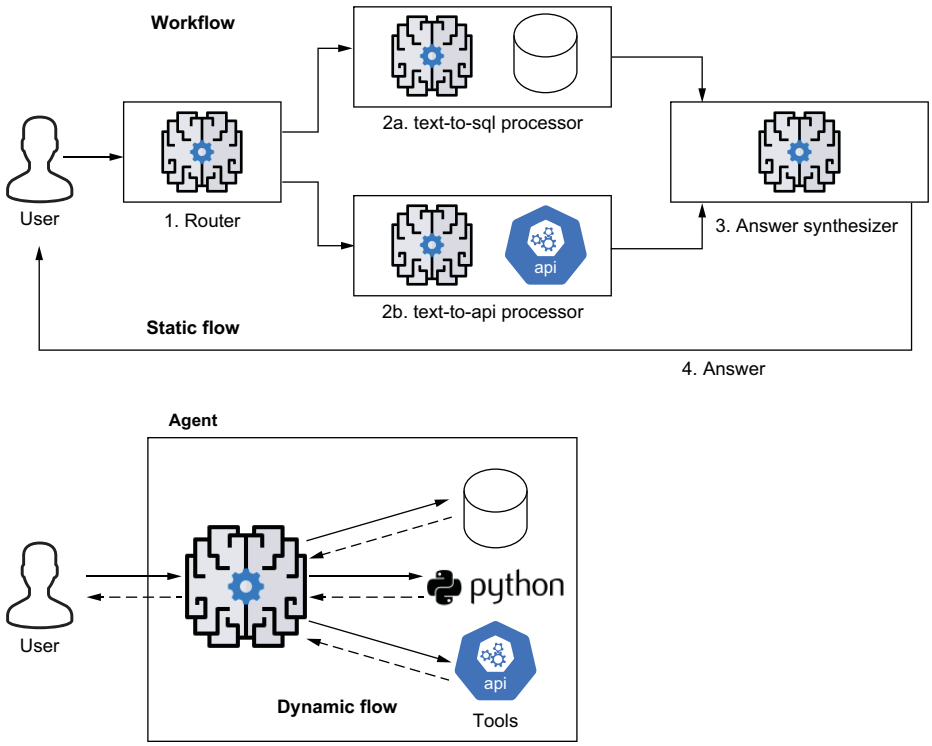


Figure 5.1 Workflows and agents. Workflows use the LLM to choose the next step from a fixed set of options, such as routing a request to a SQL database or a REST API, and synthesizing the answer with the related results. Agents, however, dynamically select and combine tools to achieve their objectives.

While both patterns rely on the LLM to drive application behavior, workflows maintain a structured and predictable path, whereas agents can adapt in real time based on new information and shifting goals. Let's discuss more about workflows next.

5.1.1 Workflows

Workflows use the LLM to pick the next step from a limited set of choices. They typically implement patterns such as Controller-Worker or Router, as illustrated in figure 5.2.

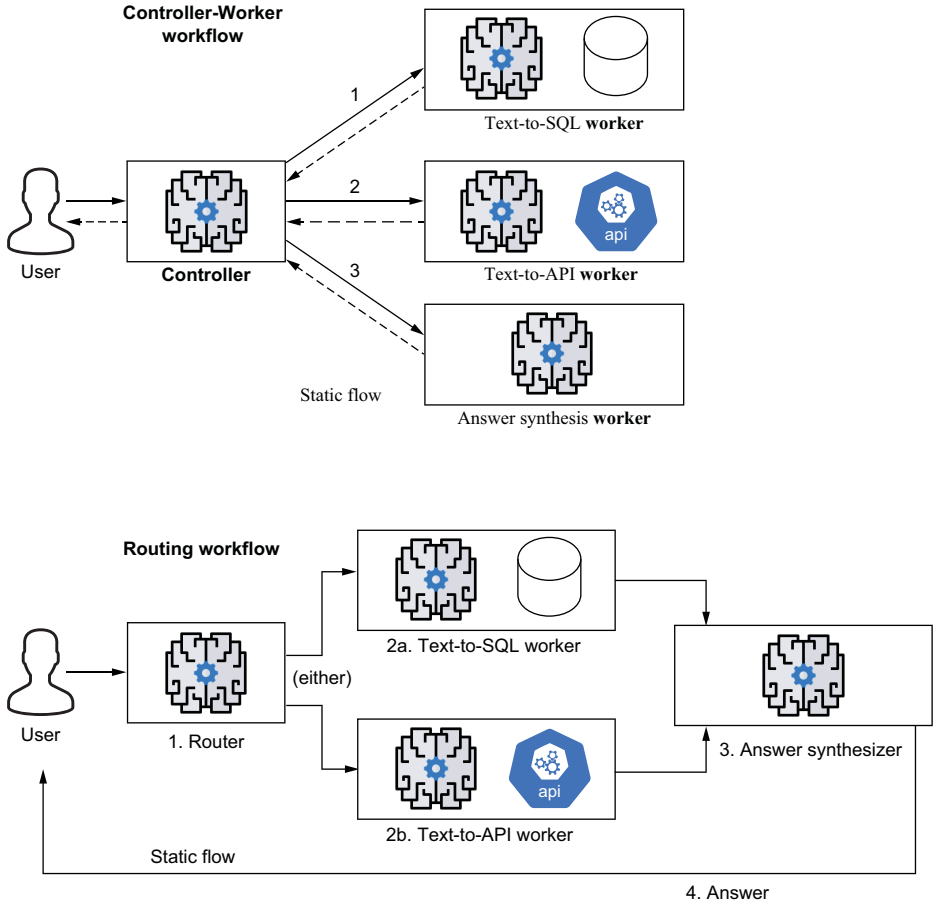


Figure 5.2 Common workflow patterns. The Controller-Worker pattern uses the LLM in the controller and orchestrates the flow by assigning various tasks to workers following a certain sequence. In the Router pattern, the LLM simply directs the task to the appropriate worker based on the context.

In the Controller-Worker pattern, the controller spawns tasks for workers following a certain sequence. In the Router pattern, the LLM simply directs the task to the proper processor (or worker). This chapter will focus on workflows, and we'll convert the web

research application we built earlier with LangChain into an agentic workflow built with LangGraph.

5.1.2 Agents

LLM-based agents use language models to perceive data, reason about it, decide on actions, and achieve goals. Advanced agents retain memory of past interactions, build dynamic workflows, and even learn from feedback. Unlike fixed prompt–response systems, agents generate new flows based on real-time data and available tools. We'll cover multi-tool agents as well as more complex multi-agent systems in part 5 of this book, between chapters 11 and 14.

5.1.3 When to use agent-based architectures

The concepts of LLM-based workflows and agents are closely related and often overlap, with no sharp dividing line between them. Both approaches are most valuable when your application needs to break complex tasks into smaller steps, make decisions based on previous results, access external tools or data, or maintain context throughout extended interactions. It's best to adopt agentic workflows or agents when your use case genuinely benefits from explicit state management and dynamic control—balancing the added power against the increased complexity.

> **TIP** I highly recommend Anthropic's article "Building Effective Agents" (https://mng.bz/lZ0o) for a deeper understanding of workflows, agents, and when to use them.

5.1.4 Agent development frameworks

A variety of frameworks are available for building agent-based systems, each with its own focus and tradeoffs:

- *AutoGPT*—Emphasizes fully autonomous, goal-driven agents with minimal supervision, but can face challenges with task consistency.
- *CrewAI*—Enables the creation of collaborative, multi-agent systems for specialized teams by using templates that can be created and shared across the community.
- *LangGraph*—Enables the creation of stateful, persistent agentic workflows using graph-based execution. LangGraph is powerful for complex applications, although it may require more technical expertise.
- *LlamaIndex*—Stands out in knowledge retrieval, though its scope is narrower than broader agent frameworks.
- *Microsoft Autogen*—Supports highly customizable multi-agent conversations, but comes with a steeper learning curve.
- *Microsoft Semantic Kernel*—Prioritizes memory and planning, and it integrates well with Azure services.
- *n8n and LangFlow*—Provide a visual interface and extensive integrations, making it accessible to nondevelopers, though advanced reasoning may require additional components.

- *OpenAI Agent SDK and Google Agent Development Kit (ADK)*—Provide streamlined APIs for developing multi-agent systems and agentic workflows efficiently.

5.2 LangGraph basics

LangGraph builds on LangChain to manage more complex agentic workflows with branching paths, stateful processing, and clear transitions between steps. It's a framework for building stateful, multi-step AI applications using a graph-based structure. In LangGraph, nodes represent individual tasks, such as generating text, calling an API, or analyzing data. Edges define the paths that connect these tasks. The state is information that moves between nodes and updates at each step. This setup is better than traditional chains when you need to make decisions, manage state, or handle complex agentic workflows.

> **NOTE** LangGraph isn't a replacement for LangChain but an extension. Think of LangChain as providing the building blocks and LangGraph as offering a blueprint to connect those parts into a complex system. LangChain gives you components such as LLMs, embeddings, and retrievers, while LangGraph helps you organize those components into a structured, stateful workflow.

To use LangGraph effectively, it's important to understand a few key concepts, including the core components of a graph (nodes and edges), how state flows through the graph, and how conditional edges control its behavior. These are the building blocks we'll explore in the next section.

5.3 Moving from LangChain chains to LangGraph

LangChain's simple, linear chains work for straightforward tasks but have limits when your applications get more complex. A typical LangChain setup often looks like this:

```
chain = (
    prompt_template
    | llm
    | output_parser
)
```

This setup struggles when tasks need to split into different paths, when you need to repeat steps based on new information, or when you want to manage state across multiple steps. It also falls short when multiple processes need to happen in parallel.

LangGraph solves these problems by offering better state management, conditional branching, and support for cyclical workflows. With explicit state management, you can define and track data consistently across the workflow, which is essential for memory and reasoning. Conditional branching allows agents to take different paths based on previous results, making decision-making smoother. Cyclical workflows let agents repeat tasks until they meet specific conditions, which helps refine results.

LangGraph also makes it easier to understand and debug complex workflows. Its graph-based structure offers a clearer view of how data flows through the system, which helps when you need to trace or fix problems.

This graph-based approach works well for a range of use cases, such as multi-step reasoning, task planning, managing context in long conversations, coordinating research tasks, and automating business processes. As your applications get more complex, the benefits of using LangGraph's agent-based architecture become clearer. It gives you the control and flexibility needed to build smart, multi-step systems that can adapt and make decisions on their own.

5.4 LangGraph core components

LangGraph provides a robust framework for building stateful, multi-step AI applications. Figure 5.3 presents the core components that form the LangGraph application's foundation.

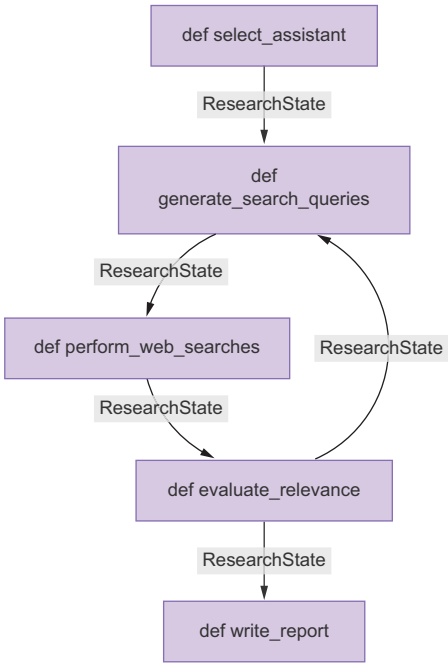


Figure 5.3 LangGraph core components. A strongly typed state (in this example, modeled with `ResearchState`) flows through the workflow. Nodes, usually Python functions (e.g., `def select_assistant`), perform tasks, and edges create directed data flows between nodes, in some cases, with conditional paths.

At the heart of every LangGraph application is a state object—in our example, `ResearchState`—which defines a clear and strongly typed state for the entire workflow. This state is typically defined as a Python `TypedDict`, ensuring that data passed between components is well-structured and type-checked.

In a LangGraph, each *node* functions as a processing unit. Nodes can handle tasks such as generating search queries, calling external APIs, summarizing results, and transforming data. These nodes are usually implemented as Python functions. The

edges between nodes determine the directed flow of data, defining how information moves through the graph.

One of LangGraph's powerful features is its *conditional edges*, which allow you to define dynamic execution paths based on the runtime state. Combined with entry points and end conditions, this gives you full control over where the graph begins, how it progresses, and when it completes. The following sections walk through how to define and connect these components, enabling you to build systems that can handle complex workflows and adaptive decision-making.

5.4.1 StateGraph structure

A core utility in LangGraph is the `StateGraph` class, which you use to define the graph that models your application's workflow. For example:

```
from langgraph.graph import StateGraph
from typing import TypedDict

class ResearchState(TypedDict):
    input_query: str
    intermediate_result: str
    final_output: str

graph = StateGraph(ResearchState)
```

Defines a state structure

Creates a graph

5.4.2 State management and typing

State management is central to LangGraph applications. Unlike chain-based methods that rely on implicit or loosely typed state, LangGraph enforces explicit, strongly typed state, making workflows more robust and predictable. Here's an extended version of the `ResearchState` that adds more detail:

```
from typing import TypedDict, Optional, List

class ResearchState(TypedDict):
    user_question: str
    assistant_info: Optional[dict]
    search_queries: Optional[List[dict]]
    search_results: Optional[List[dict]]
    research_summary: Optional[str]
    final_report: Optional[str]
```

Each node receives the current state and returns updates that merge into the overall state:

```
def process_node(state: dict) -> dict:

    result = do_something(state["input_data"])

    return {"output_data": result}
```

Processes the state

Returns the state update

5.4.3 Node functions and edge definitions

Nodes represent processing steps. Each node is a function that takes the current state and returns updates. For instance:

```
def generate_search_queries(state: dict) -> dict:
    """Generate search queries based on user question."""
    question = state["user_question"]
    queries = llm_generate_queries(question)
    return {"search_queries": queries}

graph.add_node("generate_queries", generate_
➥search_queries)
```

Adds a node to the graph

Edges define valid transitions between nodes. A simple linear edge looks like this:

```
graph.add_edge("generate_queries", "perform_searches")
```

A conditional edge uses a function to choose the next node based on the state:

```
def should_refine_queries(state: dict) -> str:
    if len(state["search_results"]) < 2:
        return "refine_queries"
    else:
        return "summarize_results"

graph.add_conditional_edge("perform_searches", should_refine_queries)
```

5.4.4 Entry points and end conditions

Every graph needs a starting point and clear end conditions. For example:

```
graph.set_entry_point("parse_question")

from langgraph.graph import END
graph.add_edge("write_final_report", END)
```

Sets the entry point

Defines the graph end

We'll explore a practical application of LangGraph next. This is where the concepts introduced in this section—typed state, node functions, edges, entry points, and end conditions—come together in a realistic workflow.

5.5 Turning the web research assistant into an AI agent

To demonstrate how LangGraph works, I'll show you how to transform the web research assistant from chapter 4—originally built with LangChain—into an agent-based system. This upgrade allows the application to assess the relevance of summaries from web page results and, if less than 50% of them are relevant, redirect the flow back to generating new search queries. If enough summaries are relevant, the application can proceed to write the final report as usual. Achieving this level of dynamic control would be very complex with plain LangChain, which justifies the move to an agent-based approach with LangGraph. This case study guides you through each step, highlighting the benefits of explicit state management and modular design.

5.5.1 Original LangChain implementation overview

Our original web research assistant used LangChain's sequential chains. The process followed these steps:

1. Choose the appropriate research assistant based on the user's question.
2. Generate search queries.
3. Perform web searches, and collect URLs.
4. Scrape and summarize each search result.
5. Compile a final research report.

Each step fed its output into the next step. You can see this in the following extract from the original implementation.

Listing 5.1 Original implementation of the web research assistant

```
assistant_instructions_chain = (
    {'user_question': RunnablePassthrough()}
    | ASSISTANT_SELECTION_PROMPT_TEMPLATE
    | get_llm()
    | StrOutputParser()
    | to_obj
)

web_searches_chain = (
    # ...input processing...
    | WEB_SEARCH_PROMPT_TEMPLATE
    | get_llm()
    | StrOutputParser()
    | to_obj
)

web_research_chain = (                    Final research chain
    assistant_instructions_chain
    | web_searches_chain
    | search_and_summarization_chain.map()
    | RunnableLambda(lambda x: # ...process results...)
    | RESEARCH_REPORT_PROMPT_TEMPLATE
    | get_llm()
    | StrOutputParser()
)
```

This approach works but has clear limitations:

- The flow is rigid and linear, making it difficult to adapt dynamically based on intermediate results. For instance, a conditional flow that redirects the application to generate new search queries if less than 50% of the summaries are relevant would be cumbersome to implement.
- Error handling is challenging, as the lack of explicit state makes it hard to track and manage failures effectively.

- State isn't explicitly managed, which complicates maintaining context across multiple steps.
- Debugging becomes difficult when issues arise, especially with complex flows, because it's unclear which part of the chain failed or why.

5.5.2 Identifying components for conversion

To convert the web research assistant to LangGraph, let's first identify the key components that will serve as nodes. Each node handles a specific part of the process:

- *Assistant Selector*—Determines which type of research assistant to use based on the user's question
- *Query Generator*—Creates search queries derived from the user's input
- *Web Searcher*—Conducts searches and gathers URLs based on the generated queries
- *Content Summarizer*—Scrapes and summarizes the content of web pages
- *Relevance Evaluator*—Assesses if the summaries are relevant enough to proceed or if new search queries are needed
- *Report Writer*—Compiles the final research report using the relevant summaries

Unlike a simple linear flow, this setup introduces a conditional element. After evaluating the relevance of the summaries, the flow can either proceed to the Report Writer if enough content is relevant or redirect back to the Query Generator to create new search queries. This decision is based on a defined threshold (e.g., if less than 50% of summaries are relevant) and can repeat up to a maximum of three iterations to avoid infinite loops.

The flow control is managed by a conditional routing function, `route_based_on_relevance`, which checks the relevance of the search results and the current iteration count. If the relevance is insufficient and the maximum number of iterations hasn't been reached, the application generates new queries and repeats the search and evaluation steps. If the maximum iteration count is reached, the application proceeds to compile a report using the available results, regardless of their relevance.

For each component, we define the following:

- *Input state*—The data each node requires to function
- *Processing*—The tasks each node performs
- *State updates*—The information each node returns to update the overall state

This modular and conditional approach makes the system flexible and adaptive, which would be cumbersome to achieve with plain LangChain's linear chains. Next, let's start the transformation process.

5.5.3 Step-by-step transformation process

Now I'll guide you through the process of converting a LangChain application to LangGraph. The following is a simplified version of the actual code, which you can find in the GitHub repository.

Step 1: Define the state

The first step is to design the state structure that will flow through the graph. A well-defined state helps you keep track of data across all nodes. In our case, we'll model a composite state using inner types, as shown in listing 5.2. This state structure clearly defines the data available at each stage, reducing ambiguity and simplifying debugging.

Listing 5.2 State type of the LangGraph-based research assistant

```
from typing import List, Dict, Any, TypedDict, Optional

class AssistantInfo(TypedDict):
    assistant_type: str
    assistant_instructions: str
    user_question: str

class SearchQuery(TypedDict):
    search_query: str
    user_question: str

class SearchResult(TypedDict):
    result_url: str
    search_query: str
    user_question: str
    is_fallback: Optional[bool]

class SearchSummary(TypedDict):
    summary: str
    result_url: str
    user_question: str
    is_fallback: Optional[bool]

class ResearchReport(TypedDict):
    report: str

class ResearchState(TypedDict):
    user_question: str
    assistant_info: Optional[AssistantInfo]
    search_queries: Optional[List[SearchQuery]]
    search_results: Optional[List[SearchResult]]
    search_summaries: Optional[List[SearchSummary]]
    research_summary: Optional[str]
    final_report: Optional[str]
    used_fallback_search: Optional[bool]
    relevance_evaluation: Optional[Dict[str, Any]]
    should_regenerate_queries: Optional[bool]
    iteration_count: Optional[int]
```

Typed dictionaries for state handling

Graph state

Step 2: Convert components to node functions

Next, we'll convert each component into a node function. Each function takes the current state, processes it, and returns updated state information, as you can see in the next listing, which is a simplified version of what you'll find in the GitHub repository.

Listing 5.3 Node functions

```
def select_assistant(state: dict) -> dict:
    """Select the appropriate research assistant."""
    user_question = state["user_question"]

    # Use the LLM to select an assistant
    prompt = ASSISTANT_SELECTION_PROMPT_TEMPLATE.format(
        user_question=user_question
    )
    response = get_llm().invoke(prompt)

    assistant_info = parse_assistant_info(
        response.content)

    return {"assistant_info": assistant_info}

def generate_search_queries(state: dict) -> dict:
    """Generate search queries based on the question."""
    assistant_info = state["assistant_info"]
    user_question = state["user_question"]

    prompt = WEB_SEARCH_PROMPT_TEMPLATE.format(
        assistant_instructions=assistant_info["assistant_instructions"],
        user_question=user_question,
        num_search_queries=3
    )
    response = get_llm().invoke(prompt)

    search_queries = parse_search_queries(
        response.content)

    return {"search_queries": search_queries}
```

Parses the response to extract assistant information

Returns the state update

Uses the LLM to create queries

Parses the response to obtain search queries

Returns the state update

Additional node functions follow the same pattern, each handling its own task. Next, I'll define the graph structure.

STEP 3: DEFINE THE GRAPH STRUCTURE

With node functions in place, we'll create the graph and define how the nodes connect, establishing the execution order and data flow, as shown in listing 5.4. Unlike a simple linear chain, this version of the graph introduces a new node for relevance evaluation and a conditional edge that dynamically alters the flow based on the relevance of the search results.

Listing 5.4 Graph structure

```
from langgraph.graph import StateGraph, END

graph = StateGraph(ResearchState)
```

Creates the graph

```
graph.add_node("select_assistant",
➥select_assistant)
graph.add_node("generate_search_queries",
➥generate_search_queries)
graph.add_node("perform_web_searches",
➥perform_web_searches)
graph.add_node("summarize_search_results",
➥summarize_search_results)
graph.add_node("evaluate_search_relevance",
➥evaluate_search_relevance)
graph.add_node("write_research_report",
➥write_research_report)
```

Adds nodes

Defines the conditional relevance evaluation

```
def route_based_on_relevance(state):
    iteration_count = state.get("iteration_count", 0) + 1
    state["iteration_count"] = iteration_count

    if iteration_count >= 3:
        return "write_research_report"
    if state.get("should_regenerate_queries", False):
        return "generate_search_queries"
    return "write_research_report"
```

```
graph.add_edge("select_assistant",
➥"generate_search_queries")
graph.add_edge("generate_search_queries",
➥"perform_web_searches")
graph.add_edge("perform_web_searches",
➥"summarize_search_results")
graph.add_edge("summarize_search_results",
➥"evaluate_search_relevance")
graph.add_edge("write_research_report", END)
```

Defines the edges between nodes

Defines the conditional edge

```
graph.add_conditional_edges(
    "evaluate_search_relevance",
    route_based_on_relevance,
    {
        "generate_search_queries": "generate_search_queries",
        "write_research_report": "write_research_report"
    }
)
```

Sets the entry point

```
graph.set_entry_point("select_assistant")
```

The new Relevance Evaluator node checks to see if enough of the summarized results are relevant. If less than 50% of the results meet the criteria, the graph redirects the flow back to the Query Generator to refine the search. If the summaries are sufficient or if the maximum of three iterations is reached, it moves forward to compile the final report. This conditional flow is a significant enhancement over the rigid linear chains of LangChain, allowing the system to adapt dynamically based on intermediate results.

STEP 4: COMPILE AND RUN THE GRAPH

After defining the graph, we'll compile and run it using an initial state, as shown in listing 5.5. This step involves setting up an initial state with all required fields, including additional parameters for controlling the conditional flow, such as `should_regenerate_queries` and `iteration_count`.

Listing 5.5 Running the graph

```
app = graph.compile()                          ← Compiles the graph

initial_state = {                              ← Creates the initial state
    "user_question": " What can you tell me about Astorga's roman spas?",
    "assistant_info": None,
    "search_queries": None,
    "search_results": None,
    "search_summaries": None,
    "research_summary": None,
    "final_report": None,
    "used_fallback_search": False,
    "relevance_evaluation": None,
    "should_regenerate_queries": None,
    "iteration_count": 0
}

result = app.invoke(initial_state)             ← Runs the graph

final_report = result["final_report"]          ← Extracts the final report
```

By introducing conditional edges and relevance evaluation, this step-by-step process transforms a rigid, linear chain into a flexible, stateful, and adaptive agent-based workflow. The system can now evaluate its own results, adapt by refining search queries if needed, and ensure that the final report is based on sufficiently relevant information. This adaptability would be cumbersome to implement in plain LangChain, justifying the shift to LangGraph for complex LLM applications.

5.5.4 Code comparison and benefits realized

This case study demonstrates how LangGraph enhances flexibility, control, and adaptability in complex, multi-step AI applications compared to traditional LangChain chains. The ability to implement conditional flows based on runtime evaluations makes LangGraph a powerful choice for building smart, context-aware agent-based systems. Specifically, the LangGraph approach offers the following significant benefits:

- *Explicit state management*—The state is clearly defined and passed through each node, making data handling transparent and reliable.
- *Modular components*—Each node handles a single task, simplifying testing, debugging, and maintenance.
- *Clear flow control*—The graph structure visually represents the execution order and data flow, making it easier to trace and understand complex processes.

- *Easier debugging*—With well-defined nodes and edges, it's straightforward to identify where errors occur and what data caused them.
- *Enhanced error handling*—Each node can implement specific error-handling strategies without affecting the rest of the system.
- *Conditional flow control*—The introduction of conditional edges based on relevance evaluation allows the application to dynamically alter its path—either refining search queries if results are insufficient or proceeding to report writing. This adaptability ensures that the application can respond intelligently to intermediate results, which would be cumbersome to implement in plain LangChain.
- *Future extensibility*—Adding or modifying nodes requires minimal changes to the overall system, allowing for smooth upgrades and new capabilities.

Summary

- Agentic workflows execute predefined steps in sequence. Agents dynamically select tools and adjust paths based on intermediate results or errors.
- LangGraph builds workflows as directed graphs. Nodes represent processing functions, edges define transitions, and conditional edges route execution based on the runtime state. Research assistants demonstrate this by deciding whether to search more sources or compile results based on content quality from previous searches.
- State management tracks data across workflow steps using typed state objects that nodes read from and write to. State is immutable per node but accumulates across the graph.
- Conditional edges route execution based on runtime conditions. A research workflow might loop back to search if retrieved content is insufficient, or it might proceed to synthesis if enough sources are found.
- The `StateGraph` class defines the workflow structure. Node functions perform discrete tasks (search, parse, summarize), and edges connect them based on the logic you define.
- Converting linear chains to LangGraph graphs separates concerns into discrete nodes. This simplifies debugging, testing, and extending workflows with new capabilities.
- LangGraph workflows preserve execution history and intermediate states. You can inspect reasoning paths, replay workflows from checkpoints, or branch from previous decisions. These capabilities are difficult to implement in simple LangChain chains.
- Define state using Python `TypedDict` for strong typing: `class ResearchState(TypedDict): question: str; search_queries: list[str]; results: list[dict]`. This ensures that data flowing between nodes is type-checked.
- Create a graph with `graph = StateGraph(ResearchState)` where `ResearchState` is your typed state definition. This enforces that all nodes receive and return compatible state structures.

- Add nodes with `graph.add_node("node_name", node_function)` where `node_function` is a Python function that takes state and returns state updates. Node functions should be pure functions when possible.
- Connect nodes with `graph.add_edge("source_node", "destination_node")` to create directed data flow. This establishes the execution order between nodes.
- Define the entry point with `graph.set_entry_point("first_node")` or use the `START` constant to mark where execution begins. The first node receives the initial state.
- Mark endpoints with `graph.add_edge("final_node", END)` to indicate workflow termination. Execution stops when reaching `END`, returning the final state.
- Compile the graph with `app = graph.compile()` before execution. This validates the graph structure and creates an executable application.
- Node functions receive current state as input and return partial state updates (not full state replacement). Return only the fields you want to update: `return {"search_queries": queries}`.
- Add conditional edges with `graph.add_conditional_edges("source_node", router_function, {"option1": "node1", "option2": "node2"})`. The router function returns a string matching one of the options.
- Router functions for conditional edges must return string values matching the next node names. Use `if` logic to determine routing: `return "search_more" if len(results) < 3 else "write_report"`.
- LangGraph extends LangChain, not replaces it. Use LangChain components (LLMs, retrievers, embeddings) as building blocks within LangGraph nodes for complex workflows.

Part 3

Q&A chatbots

This part marks a shift from summarizing information to answering meaningful questions. Here, you'll dive into Retrieval-Augmented Generation (RAG)—the core technique that enables LLMs to perform question answering over large or private knowledge bases. You'll learn how RAG combines three critical components—vector stores, retrievers, and language models—to locate semantically relevant information and synthesize grounded, coherent answers. Rather than relying solely on the model's internal knowledge, you'll see how to anchor responses in external data, dramatically improving accuracy and trustworthiness.

You'll begin by building the essential pieces of a RAG system from scratch to fully understand how they work together. This hands-on approach will give you an intuitive sense of how documents are ingested, embedded, and retrieved using semantic similarity rather than keyword matching. Once you've mastered the fundamentals, you'll take things further by introducing LangChain's modular RAG components, which streamline development and make complex pipelines easier to maintain. You'll also incorporate LangSmith to monitor, trace, and debug every step of your chatbot's reasoning process—ensuring visibility into how answers are formed and why certain results are chosen.

By the end of this part, you'll have built a fully functional, search-enabled chatbot capable of reasoning across multiple sources, handling follow-up questions, and maintaining conversational context. More importantly, you'll know when to keep your implementation minimal using direct APIs and when to use LangChain's abstractions for speed, modularity, and observability. This part lays the groundwork for building truly intelligent assistants that not only summarize knowledge but also actively help users find and understand the information they need.

6 RAG fundamentals with ChromaDB

This chapter covers

- Implementing semantic search using the RAG architecture
- Understanding vector stores and their functionality
- Implementing RAG with ChromaDB and OpenAI

In this chapter, we'll explore two key concepts: semantic search and Retrieval Augmented Generation (RAG). You'll see how large language models (LLMs) are used for semantic search through a chatbot, enabling you to query a system for information across multiple documents and retrieve the fragments that best match the meaning of your question, rather than just matching keywords. This approach is also known as Q&A over documents or querying a knowledge base.

In earlier chapters, you learned about summarization, a typical use case for LLMs. Now I'll walk you through the basics of building a Q&A chatbot that searches across multiple documents. You'll interact with the LLM to find the answers you're looking for.

This chapter focuses on RAG, the design pattern that powers semantic search systems, with a particular emphasis on the vector store—a key component of these systems. You'll learn the technical terminology related to Q&A and RAG systems and understand how terms like *semantic search* and *Q&A* are often used interchangeably.

By the end of this chapter, you'll have implemented a basic RAG-based architecture using the APIs of an LLM (OpenAI) and a vector store (ChromaDB). In chapter 7, you'll build on this foundation to create Q&A chatbots using the RAG architecture. There's a lot to cover, so let's get started.

6.1 Semantic search

Semantic search is a popular use case for LLMs, alongside summarization and code generation. It's one of the key applications driving the LLM and Generative AI boom.

> **DEFINITION** *Semantic search* means searching for information by focusing on its meaning. This involves understanding a query's meaning, retrieving relevant document fragments from a document store that closely match the query's meaning, and optionally generating a natural language answer.

Semantic search differs from traditional keyword-based searches, which fail to find information if exact words don't match. Semantic search produces relevant results even if the query and result don't share a single word.

Before diving into the code, you need to have a clear understanding of a semantic search chatbot's architecture. I'll start with a simple example to ease you in, but by the end of this section, you'll grasp the architecture of a real-world Q&A chatbot.

6.1.1 A basic Q&A chatbot over a single document

Let's start with a simple scenario that will help you understand how a Q&A chatbot works and get you familiar with its components. The first chatbot example answers questions about a single document, as shown in figure 6.1.

The main elements of this basic setup are listed here:

- *Document*—Contains the text for semantic search or information extraction
- *Prompt*—Encapsulates the user's question (semantic search) and the context (the document) with the information needed for the answer
- *LLM-based chatbot*—Sends the prompt to the LLM, which understands the question and context, selects relevant information, and synthesizes an answer for the user

Let's break down these concepts further.

> **DEFINITION** *Context* is the text or information in the prompt, along with the user's question, used to formulate an answer.

> **DEFINITION** *Synthesize* means to generate an answer from the question and context provided.

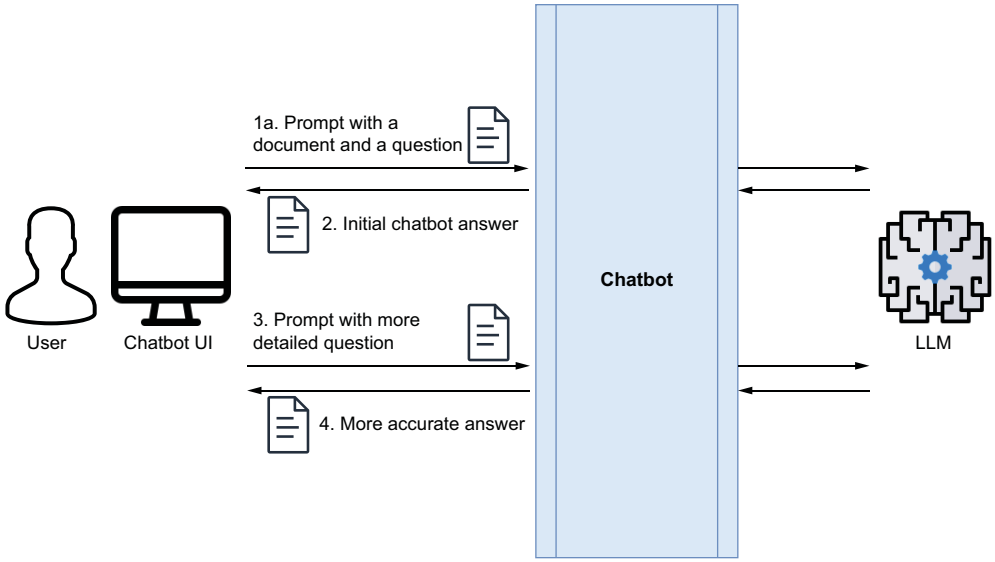


Figure 6.1 The simple Q&A chatbot process involves the following: (1) the user sends a prompt containing a document (context) and a question to the chatbot, (2) the chatbot returns an initial answer, (3) the user follows up with a more detailed question, and (4) the chatbot provides a more accurate answer.

You don't need to write any code to implement this initial setup. Just log in to ChatGPT or an alternative LLM-based chatbot such as Gemini or Claude, and you're ready to go. Let's try a simple Q&A interaction using text about Paestum from Britannica (www.britannica.com). Submit this prompt to ChatGPT (you can find this prompt in a text file of the GitHub repository):

> Read the following text and let me know how many temples are in Paestum, who built them, and what architectural style they are:
>
> —
>
> Paestum, Greek Poseidonia, ancient city in southern Italy near the west coast, 22 miles (35 km) southeast of modern Salerno and 5 miles (8 km) south of the Sele (ancient Silarus) River. Paestum is noted for its splendidly preserved Greek temples.
>
> Visit the ruins of the ancient Greek colony of Paestum and discover its history, culture, and society. See all videos for this article
>
> Poseidonia was probably founded about 600 BC by Greek colonists from Sybaris, along the Gulf of Taranto, and it had become a flourishing town by 540, judging from its temples. After many years' resistance the city came under the domination of the Lucanians (an indigenous Italic people) sometime before 400 BC, after which its name was changed to Paestum. Alexander, the king of Epirus, defeated the Lucanians at Paestum about 332 BC, but the city remained Lucanian until 273, when it came under Roman rule and a Latin colony was founded there. The city supported

> Rome during the Second Punic War. The locality was still prosperous during the early years of the Roman Empire, but the gradual silting up of the mouth of the Silarus River eventually created a malarial swamp, and Paestum was finally deserted after being sacked by Muslim raiders in AD 871. The abandoned site's remains were rediscovered in the 18th century.
>
> Temple ruins of columns and statues at Karnak, Egypt (Egyptian architecture; Egyptian archeology; Egyptian history)
>
> The ancient Greek part of Paestum consists of two sacred areas containing three Doric temples in a remarkable state of preservation. During the ensuing Roman period a typical forum and town layout grew up between the two ancient Greek sanctuaries. Of the three temples, the Temple of Athena (the so-called Temple of Ceres) and the Temple of Hera I (the so-called Basilica) date from the 6th century BC, while the Temple of Hera II (the so-called Temple of Neptune) was probably built about 460 BC and is the best preserved of the three. The Temple of Peace in the forum is a Corinthian-Doric building begun perhaps in the 2nd century BC. Traces of a Roman amphitheatre and other buildings, as well as intersecting main streets, have also been found. The circuit of the town walls, which are built of travertine blocks and are 15–20 feet (5–6 m) thick, is about 3 miles (5 km) in circumference. In July 1969 a farmer uncovered an ancient Lucanian tomb that contained Greek frescoes painted in the early classical style. Paestum's archaeological museum contains these and other treasures from the site.

Here's the answer I got (keep in mind, you might get a slightly different result because of the stochastic nature of LLMs, but it's still likely to be incomplete):

> "There are three temples in Paestum, and they are constructed in the Doric architectural style. These temples are:
>
> - The Temple of Athena (also known as the Temple of Ceres), from the 6th century BC.
> - The Temple of Hera I (also known as the Basilica), also from the 6th century BC.
> - The Temple of Hera II (also known as the Temple of Neptune), built around 460 BC and the best-preserved of the three.
>
> These temples are remarkable for their well-preserved Doric architecture and are important historical landmarks.

This is almost what I wanted, but it's missing who built the temples. If this had been a Q&A engine instead of a chatbot, the client system wouldn't have received any answer about who built the temples.

NOTE A Q&A engine, like a summarization engine, is called by a client system and returns one answer. A chatbot, used by a person, allows for multiple questions so you can refine the answer through interaction.

Next, I send this prompt:

> This answer is almost what I expected, but you forgot to answer something: who built the temples?

> I apologize for missing that part. The temples in Paestum were built by Greek colonists from Sybaris who founded the city of Poseidonia (later Paestum) around 600 BC. So, the Greek colonists constructed these temples in the Doric architectural style.

This shows the advantage of a chatbot over an engine: after a couple of rounds of questions, ChatGPT answered all my questions about the temples in Paestum. Importantly, I didn't have to resubmit the full text; ChatGPT remembered the initial text.

NOTE A chatbot like ChatGPT creates a stateful session that remembers the exchange by passing the history of the conversation as an additional context, allowing the user to refine the answer without resending the initial text.

Let's try another prompt:

Is the archaeological site in Paestum recognized by UNESCO?

> Yes, the archaeological site of Paestum is recognized by UNESCO. In 1998, it was designated a UNESCO World Heritage Site. This designation acknowledges the historical and architectural significance of the well-preserved Greek temples and ancient ruins in Paestum.

Hold on! The original text didn't include this information. ChatGPT used its own knowledge, likely from Wikipedia or other sources, to answer correctly. This eagerness to answer questions using unprovided content can be helpful, as shown in the example, but it can also be dangerous, as you'll see shortly. Let's try another question:

How many columns do the three temples have in total?

> The three temples in Paestum have a total of 94 columns:
>
> - The Temple of Athena (Temple of Ceres) has 6 columns on its shorter sides and 13 columns on its longer sides, for a total of 32 columns.
> - The Temple of Hera I (Basilica) has 6 columns on its shorter sides and 9 columns on its longer sides, for a total of 30 columns.
> - The Temple of Hera II (Temple of Neptune) has 6 columns on its shorter sides and 14 columns on its longer sides, for a total of 32 columns.

ChatGPT tried to answer, but the answer is wrong. The Temple of Hera I has 50 columns, not 30, as you can check yourself on the Wikipedia.

WARNING When a chatbot can't find the information in the provided text, it might rely on its own knowledge, which can lead to incorrect or made-up information, known as *hallucination*.

In one instance, ChatGPT provided accurate information about UNESCO, yet in another, it gave an incorrect answer about the number of columns in a structure. This raises an important question: How can you tell when ChatGPT is providing reliable information and when it isn't? As a general rule, you should be cautious about trusting ChatGPT for facts or details you haven't independently verified or supplied yourself.

TIP To avoid this issue, instruct ChatGPT to use only the information you provided and indicate if it doesn't know the answer.

You can attempt to address the previous issue with this prompt:

> How many columns do the three temples have in total? Use only the information I gave you in the text. If you don't know the answer, just say so.

You'll get a more accurate response, as follows:

> " I apologize for the error in my previous response. The text you provided doesn't mention the total number of columns in the three temples in Paestum.

TIP Designing safe prompts for Q&A chatbots reduces the chance of hallucinations. You'll learn more about this in the rest of the book.

Now let's move on to a more complex use case. This time, the chatbot must pull specific from a much larger amount of text.

6.1.2 A more complex Q&A chatbot over a knowledge base

The following summarizes the design of the basic LLM-based Q&A chatbot, such as ChatGPT, which processes a single piece of text:

1. You send a prompt to the chatbot with the text you want to search for and the question you want to ask.
2. The prompt should instruct the chatbot to formulate an answer using only the provided text.
3. The chatbot should create a session, retaining the conversation history to refine answers.

The approach described so far works well for searching within a single text. But what if your chatbot needs to answer questions about company information scattered across multiple sources—such as intranet pages, shared folders, and documents in various formats (e.g., PDF, DOCX, TXT, or PPT)? That's the challenge we'll tackle in the next chapters.

When designing an enterprise Q&A chatbot, one of the main obstacles is that you can't include all the company's content in the prompt along with the user's question. Especially when dealing with large documents, this would quickly exceed the model's context window. In practice, narrowing down the context to what's most relevant improves speed, cost-efficiency, and accuracy. In fact, providing less—but more focused—context often yields better results than overloading the model with loosely related information.

To achieve this, the chatbot must be able to access the company's knowledge base and retrieve only the specific content needed to answer a given question. Ideally, it should "understand" the knowledge base well enough that you can ask a question naturally—without manually supplying background context each time—as illustrated in figure 6.2.

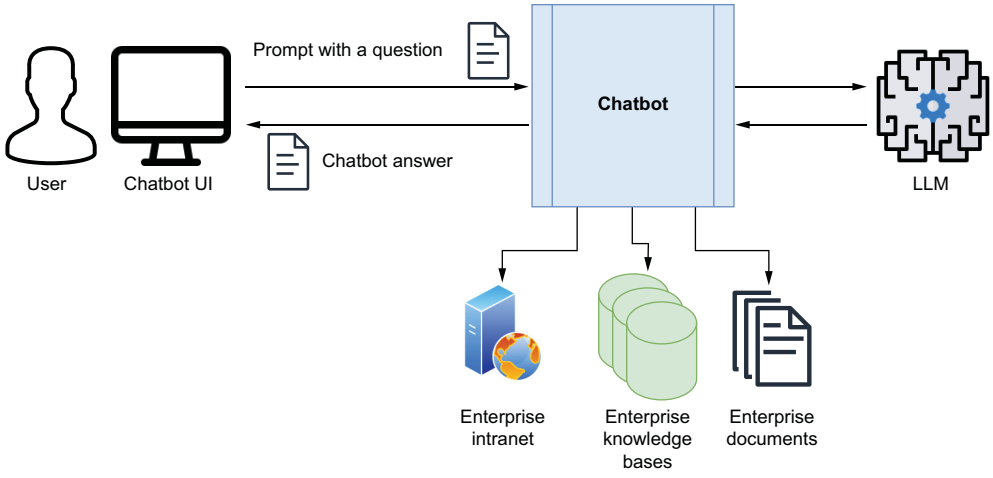


Figure 6.2 Hypothetical design for an enterprise Q&A chatbot. The knowledge of the chatbot is expanded with various data sources: the enterprise's intranet, knowledge bases, and documents.

Now you might wonder how you can connect the ChatGPT chatbot (or Gemini chat, Claude chat, etc.) to the company's intranet, knowledge bases, and documents, as shown in figure 6.2, so you can send just the question in the prompt. Unfortunately, you can't connect ChatGPT to your local data directly. The preceding solution doesn't work with standard ChatGPT. The closest alternative is to use ChatGPT Plus and configure a custom version through OpenAI's My GPTs offering, allowing you to upload documents for lookup by later queries. This is convenient for simple use cases. However, for more control over how the chatbot interacts with text sources and the LLM, you need a different approach. Enter the RAG design pattern.

6.1.3 The RAG design pattern

The Retrieval Augmented Generation (RAG) design pattern is a classic solution for building a Q&A chatbot. Let's break down what RAG stands for:

- *Retrieval*—This step involves retrieving context from a pre-prepared data source, typically a vector store optimized for semantic search. Retrieval is a key part of the RAG architecture.
- *Augmented*—This means the answer is improved or enhanced by the context provided during the retrieval step.
- *Generation*—This refers to generating the answer to your question. Because this book focuses on LLMs and generative AI, answer generation is performed by an LLM, requiring the chatbot to interact with it.

You might wonder where this information is retrieved from and how. The key is preparing the information so your custom chatbot can easily access and use it to augment

the LLM's generated answer. The RAG design pattern has the following two stages, which I'll break down further in the next subsections:

1. *Content ingestion stage (indexing)*—All the content users will query is stored in a special database and indexed in a special format for efficient retrieval (I'll clarify what *special* means in this context shortly).
2. *Question-answering stage (Q&A; retrieval and generation)*—The chatbot takes a user's question, retrieves relevant information from the special database, and feeds it along with the user's question to the LLM. The LLM generates and returns the augmented answer.

CONTENT INGESTION STAGE (INDEXING)

Before users can query the Q&A chatbot, you need to store relevant content, such as enterprise documents from various sources and formats, into a vector store, a special database optimized for quick search and retrieval, as shown in figure 6.3.

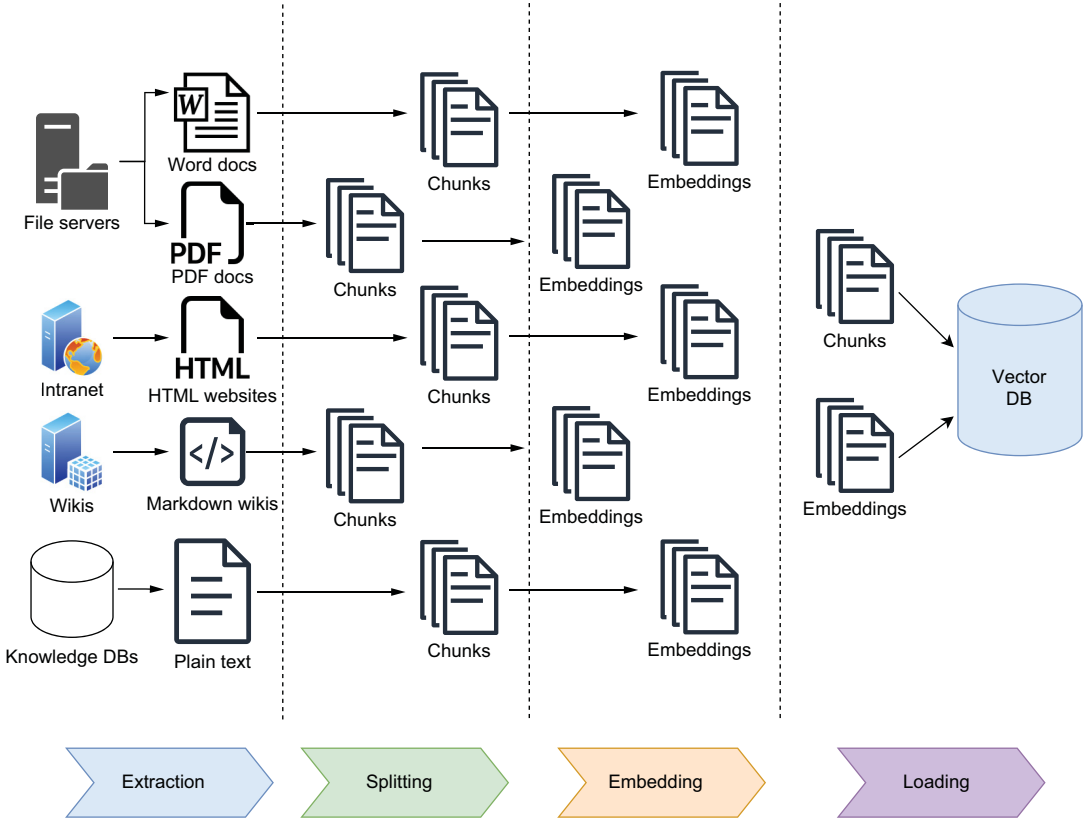


Figure 6.3 RAG content ingestion stage. Documents are extracted from sources, split into chunks, and converted into embeddings while being stored in a vector database, which stores a copy of the original chunks and their embeddings (vector form).

During the content ingestion stage, text is extracted from the sources and split into small chunks. Each chunk is then transformed into an *embedding*, a numerical vector representation of the text. Splitting content into chunks is crucial because embedding models work on finite sizes, and you want the search to target small, relevant content pieces instead of large, mixed-relevance sections. You can create embeddings using the vector store's proprietary model (if available), an LLM provider's model (e.g., OpenAI), or a dedicated embedding library. The embeddings and content chunks are then stored in a vector database.

The purpose of the embeddings is to index the content for efficient lookup during the Q&A stage. This means the text in the user's question doesn't need to match the text in the results exactly to produce relevant answers. For example, querying the vector store for "feline animals" will return chunks mentioning cat, lion, and tiger, even if the word "feline" isn't in any document chunk.

Question-answering stage (retrieval and generation)

Once the information has been split into small chunks, transformed into embeddings, and stored in a vector store, users can query your Q&A chatbot. Let's walk through the Q&A stage workflow, as shown in figure 6.4:

1. The chatbot uses a retriever to transform the user question into embeddings.
2. The retriever uses the embedding to perform a similarity search in the vector store.
3. The vector store returns several relevant text chunks.
4. The retrieved content is fed into a prompt as a "context" along with the original user question.
5. The chatbot sends the prompt to the LLM, which synthesizes a response and returns it to the user.

When the chatbot receives a user question, a retriever converts the natural language query into a vector representation using the same embedding model employed during the content ingestion stage. It then queries the vector store by performing a similarity search between the question vector and the stored text chunk vectors. The store returns the most relevant document chunks—those whose embeddings are closest to the query—typically ranked by vector distance.

Once the relevant content chunks are retrieved, the chatbot sends the LLM a prompt that includes the initial question and a context incorporating the retrieved document chunks. The LLM then generates (or *synthesizes* in RAG terminology) the answer and returns it to the chatbot, which then delivers it to the user.

This final step is similar to the basic Q&A over a single document use case. You provide the LLM with the initial question and a context (previously, this was the entire input text in the simple scenario) that provides the information for the answer. Now the context is represented by chunks retrieved from the vector store. The main difference between Q&A over a single document and over a range of documents in a vector database is the additional components: the vector store provides the information for the

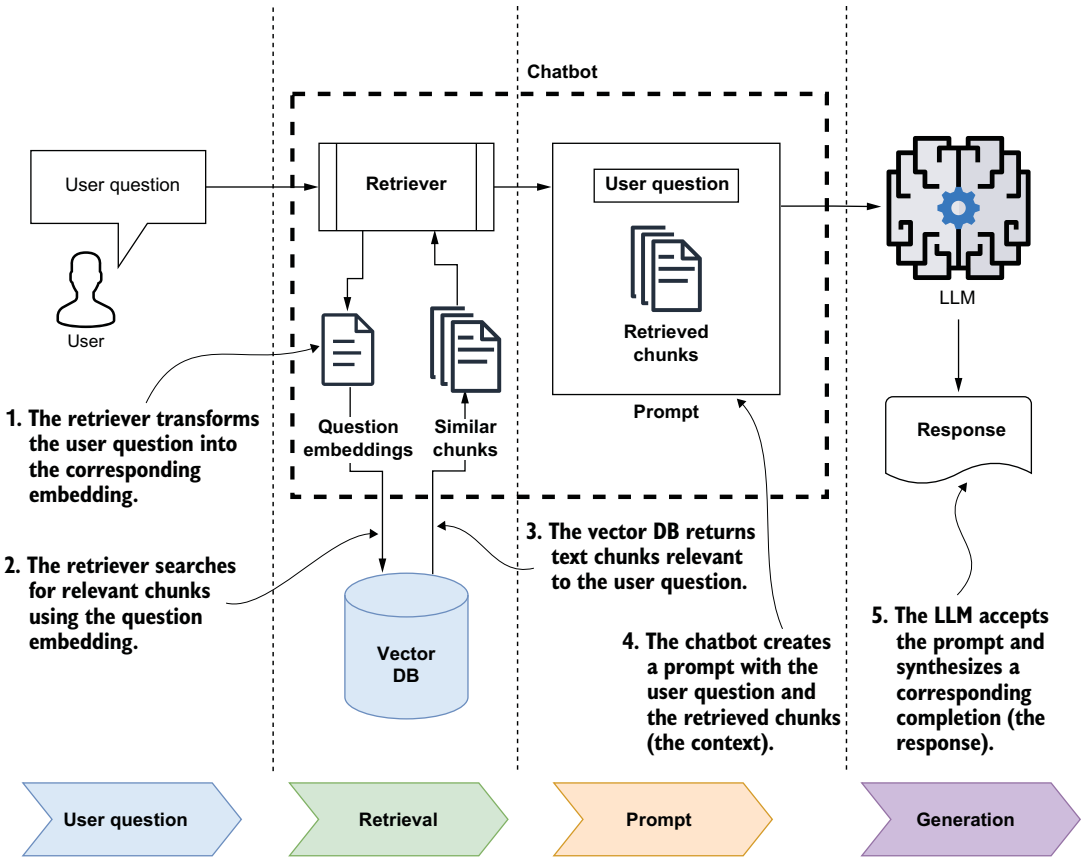


Figure 6.4 RAG Q&A stage: Retrieval and generation

answer. The role ChatGPT played in the basic chatbot use case is now split between an orchestrating chatbot, which accepts the query and retrieves the information, and an LLM, which synthesizes the answer.

Now that you understand the high-level architecture of the RAG design pattern, let's examine one of its key components: the vector store. After that, you'll be ready to attempt your first RAG implementation.

6.2 Vector stores

I've mentioned vector stores several times, but only briefly. In this section, I'll explain what they are, their purpose, and what they offer, as well as give a few examples.

6.2.1 What's a vector store?

A *vector store* is a storage system designed to efficiently store and query high-dimensional vectors. Vectors are key in AI because embeddings—numerical representations of text,

images, sounds, or videos—are built using them. In short, embeddings are vectors that capture the meaning of words in their dimensions.

The main use of vector stores in LLM and machine learning applications is to store embeddings that act as indexes for text chunks (or chunks of video, image, or audio). Searches in vector stores are *similarity searches*, which measure the distance between the embeddings of the query and those of the stored chunks. The result is either the closest vector or a list of the closest vectors. This semantic similarity reflects how close the meanings of the text chunks are.

6.2.2 How do vector stores work?

Vector distance calculations use common functions such as Euclidean distance, cosine distance, and Hamming distance. These are used in machine learning algorithms such as k-Nearest Neighbors (KNN) and the more scalable Approximate Nearest Neighbor (ANN) search, which is the standard algorithm for similarity searches.

> **TIP** I won't cover distance metrics or similarity search algorithms here. If you're interested, check out this academic paper by Yikun Han et al. (https://arxiv.org/pdf/2310.11703.pdf) or the informal article "Distance Metrics in Vector Search" by Erika Shorten (https://mng.bz/BzQ2).

The first vector stores (e.g., Milvus) appeared in 2019 to support dense vector search, mainly for image recognition. These stores efficiently stored and compared image embeddings. This is called *dense vector* search because most of the dimensions of the vectors, or embeddings, have nonzero values.

Earlier search techniques, such as Term Frequency-Inverse Document Frequency (TF-IDF), used *sparse vectors*, where most values are zero. These were used for *lexical search*, focusing on exact word matches and implemented in systems such as Lucene or BM25.

Milvus was initially built for image-based embeddings, where the vectors represented the meaning of an image. Later, vector stores expanded to tasks such as product recommendations, and with the rise of LLMs, new vector stores emerged specializing in text-based semantic similarity search.

6.2.3 Vector libraries vs. vector databases

The first vector stores (known as *vector libraries*), such as Facebook AI Similarity Search (FAISS; developed by Meta), offered minimal functionality to keep things simple. They stored embeddings in memory using immutable data structures and provided efficient similarity search capabilities. However, as LLM adoption grew, these libraries revealed several limitations:

- *Handling underlying text*—Vector libraries only stored embeddings, requiring you to store the original data, such as text or images, elsewhere. This meant creating a unique identifier for each piece of data to synchronize the original text and its embeddings, complicating implementation and maintenance.

- *No updates*—Vector libraries used immutable data structures, preventing updates. This made them unsuitable for use cases with frequently changing data, especially on an intraday basis.
- *Limited querying during data ingestion*—Vector libraries didn't support querying during data ingestion due to the risk of simultaneous read and write operations, which could affect performance and scalability.

To address these issues, vendors such as Pinecone developed *vector databases* offering more features:

- *Handling text and embeddings*—Vector databases store both the text and related embeddings, simplifying the client application's workflow. Many can even handle embedding creation. They also allow storing metadata associated with the text, such as provenance and lineage information.
- *Full CRUD (create, read, update, delete) capabilities*—Vector databases support updating data, making them suitable for use cases with frequent data changes.
- *Querying during import*—Vector databases allow similarity searches while importing new data, enhancing scalability and performance.

Vector databases soon introduced features such as caching, sharding, and partitioning, which improved scalability, performance, robustness, and durability, similar to traditional relational and NoSQL databases. Meanwhile, relational databases such as PostgreSQL and NoSQL databases such as MongoDB, which already offered these benefits, adapted by adding support for vector types. This allows embeddings to be stored alongside text in the same record or document, making it easy to link text with its corresponding embedding. For the rest of the book, I'll use *vector store*, *vector database*, and *vector library* interchangeably, as they have converged in meaning.

6.2.4 Most popular vector stores

Compiling a table summarizing the characteristics of the most popular vector stores is challenging due to their rapid evolution and convergence. However, table 6.1 gives a rough idea of what's available in the market at the time of publication, providing a starting point for your exploration.

Table 6.1 Most popular vector stores and related characteristics

Vector store	Type	Website
FAISS	Vector library	https://github.com/facebookresearch/faiss/wiki/
Milvus	Vector database	https://milvus.io
Qdrant	Vector database	https://qdrant.tech
Chroma	Vector database	www.trychroma.com
Weaviate	Vector database	https://weaviate.io
Pinecone	Vector database	www.pinecone.io

Table 6.1 Most popular vector stores and related characteristics *(continued)*

Vector store	Type	Website
Vald	Vector database	https://vald.vdaas.org
ScaNN	Vector library	https://mng.bz/dWaw
KDB	Time series database	https://kdb.ai
Elasticsearch	Search engine	www.elastic.co
OpenSearch	Fork of Elasticsearch	https://opensearch.org
PgVector	PostgresSQL extension	https://github.com/pgvector/pgvector
MongoDB Atlas	MongoDB extension	www.mongodb.com/

6.2.5 Storing text and performing a semantic search using Chroma

Before guiding you through building an enterprise Q&A chatbot, you first need to learn how to store text in a vector store and perform semantic searches. This will help you understand the fundamentals. We'll use Chroma, a vector database that's easy to set up and use. You just need to install the related Python package with pip. Let's get started!

SETTING UP CHROMADB

First, you need to set up a Chroma Jupyter Notebook environment. Follow these steps to do so:

1. Create the virtual environment for chapter 6's code. Open a terminal, create a `ch06` folder, navigate into it, and run the following:

   ```
   C:\Github\building-llm-applications\ch06>python -m venv env_ch06
   ```

2. Activate the virtual environment:

   ```
   C:\Github\building-llm-applications\ch06>.\env_ch06\Scripts\activate
   ```

   You should see

   ```
   (env_ch06) C:\Github\building-llm-applications\ch06>
   ```

3. Install the necessary packages (`notebook`, `chromadb`, and `openai`):

   ```
   pip install -r requirements.txt
   ```

4. If you cloned the GitHub repository, start the Jupyter Notebook with Jupyter Notebook `06-chromadb-ingestion-and-querying.ipynb`; otherwise, create it from scratch:

   ```
   jupyter notebook
   ```

5. Create a notebook by choosing File > New > Notebook, and save it as `06-chromadb-ingestion-and-querying.ipynb`.

In your notebook, import the `chromadb` module, and create an in-memory client for the vector database. Keep in mind that the in-memory client will lose all data when the notebook session ends:

```
import chromadb
chroma_client = chromadb.Client()
```

Next, create a collection to store the content on Paestum from the Britannica website (a collection is like a bucket where you store documents and their related embeddings):

```
tourism_collection = chroma_client.create_collection(
    name="tourism_collection")
```

INSERTING THE CONTENT

Add the Paestum content to the collection, splitting it manually into a list of smaller chunks (or *documents*). Chroma will then generate embeddings from the text you provide, using its default embeddings model unless you specify a different one. I've shortened the text for convenience, but you can find the full version in the `Paestum-Britannica.txt` file on my GitHub repository, or check my notebook (also on GitHub) if needed. When storing the text, it's useful to include metadata such as the source of each document and an associated ID, as shown in the following listing.

Listing 6.1 Creating and populating a ChromaDB collection

```
tourism_collection.add(
    documents=[
        "Paestum, Greek Poseidonia, …[shortened] … Greek temples.",
        "Poseidonia was probably …[shortened] … in the 18th century.",
        "The ancient Greek part of …[shortened] … from the site."
    ],
    metadatas=[
        {"source": "https://www.britannica.com/place/Paestum"},
        {"source": "https://www.britannica.com/place/Paestum"},
        {"source": "https://www.britannica.com/place/Paestum"}
    ],
    ids=["paestum-br-01", "paestum-br-02", "paestum-br-03"]
)
```

After running this code, you're ready to perform a search on the vector store. We'll do that next.

PERFORMING A SEMANTIC SEARCH

Let's perform a query similar to the one you executed against ChatGPT. Ask for the number of Doric temples in Paestum, specifying that you only want the closest result:

```
results = tourism_collection.query(
    query_texts=["How many Doric temples are in Paestum"],
    n_results=1
)
print(results)
```

Here's a shortened version of the result:

```
{'ids': [['paestum-br-03']], 'distances': [[0.7664762139320374]],
'metadatas': [[{'source': 'https://www.britannica.com/place/Paestum'}]],
'embeddings': None, 'documents': [['The ancient Greek part of Paestum
consists of two sacred areas containing three Doric temples in a remarkable
state of preservation. …[SHORTENED] … Paestum's archaeological museum
contains these and other treasures from the site.']]}
```

Chroma understands the query's meaning and returns the correct text chunk containing the answer, along with metadata about the source and the distance between the query and answer embeddings.

> **NOTE** Unlike querying ChatGPT, where you had to provide the question and the full text, querying Chroma only requires sending the question, as the content is already stored in ChromaDB.

Checking semantic proximity

To see how close the returned text chunk (`paestum-br-03`) is to the question compared to the other text chunks (`paestum-br-01` and `paestum-br-02`), request three results:

```
results = tourism_collection.query(
    query_texts=["How many Doric temples are in Paestum"],
    n_results=3
)
print(results)
```

You should see the following:

```
{'ids': [['paestum-br-03', 'paestum-br-01', 'paestum-br-02']], 'distances':
[[0.7664762139320374, 0.8946815729141235, 1.336229681968689]], 'metadatas':
[[{'source': 'https://www.britannica.com/place/Paestum'}, {'source': 'https:/
/www.britannica.com/place/Paestum'}, {'source': 'https://www.britannica.com/
place/Paestum'}]], 'embeddings': None, 'documents': [['The ancient Greek part
of … [SHORTENED]  … the 18th century.']]}
```

The embeddings for `paestum-br-03` are the closest to the question's embeddings, with a distance of 0.76. The chunk `paestum-br-02` is the farthest, with a distance of 1.33, proving that Chroma identified the most relevant chunk correctly.

> **NOTE** The vector database doesn't generate an answer like ChatGPT does. It returns the semantically closest text chunks to your query. For a properly formulated answer, you still need an LLM to process the original question and the retrieved text chunks. This approach saves costs because LLM vendors such as OpenAI charge based on the number of tokens processed.

This section has given you a glimpse of what you can do with Chroma. For more details, check out the official documentation at https://docs.trychroma.com/, especially the Client-Server Mode section, to learn how to run Chroma in client/server mode if you prefer it to run on a separate host from your LLM solution. Now that you know how to query the vector store, you can attempt to implement the full RAG pattern, including generating complete answers.

Instantiating ChromaDB in different ways

So far, you've worked with a local in-memory instance of ChromaDB. You can also create a client for a local on-disk instance like this:

```
client = chromadb.PersistentClient(path="./chroma_db")
collection = client.create_collection("my_persistent_collection")
```

Alternatively, you can set up an HTTP client on the same computer or a different one. To do this, open a command shell, and run the following command (assuming you've installed ChromaDB via pip):

```
chroma run --port 8010
```

Next, instantiate the HTTP client in your notebook or application like this:

```
client = chromadb.HttpClient(host="http://localhost", port=8010)
```

Once the client is set up, you can interact with it in the same way you do with the in-memory client.

6.3 Implementing RAG from scratch

Let's implement RAG by building a chatbot that uses the GPT-5-nano model and a vector database. We'll then ask it the same question about Paestum's temples that you asked ChatGPT in section 6.1.1. When using ChatGPT, you had to send a prompt with both the question and the full text on Paestum from Britannica. Once you build your own chatbot, you'll only need to ask the question, as shown in figure 6.5. As you can see in the architectural diagram in figure 6.5, the chatbot will query ChromaDB, retrieve the content, and feed it to GPT-5-nano with the original question to get the full answer.

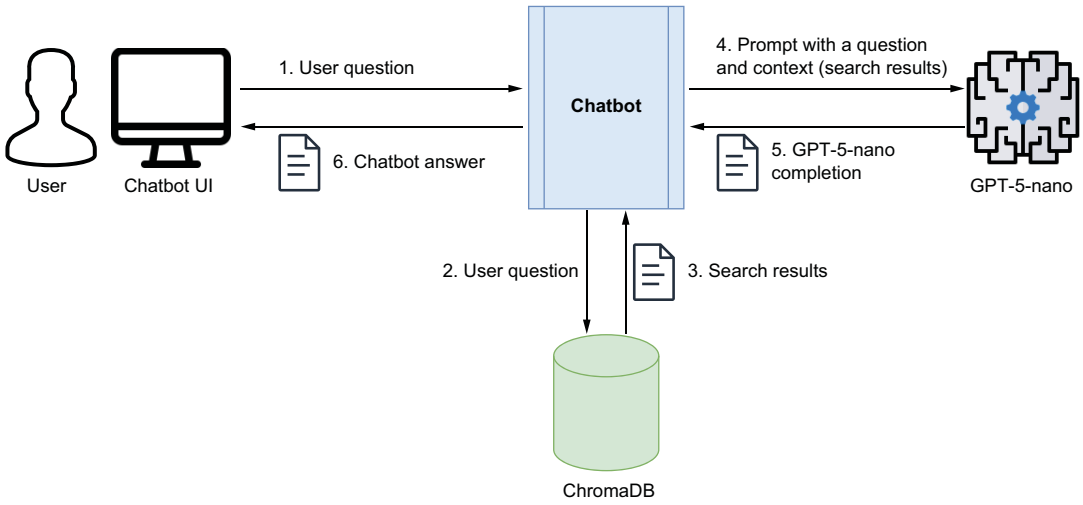


Figure 6.5 RAG architecture, including ChromaDB and the GPT-5-nano model

We'll build the chatbot step-by-step by implementing a few functions. First, import the OpenAI library, and set the OpenAI API key (assuming you're using the same notebook from the previous section):

```
from openai import OpenAI
import getpass

OPENAI_API_KEY = getpass.getpass('Enter your OPENAI_API_KEY')
```

Now instantiate the OpenAI client:

```
openai_client = OpenAI(api_key=OPENAI_API_KEY)
```

6.3.1 Retrieving content from the vector database

You already know how to perform a semantic search against the vector store. Here, let's wrap that code into a reusable function:

```
def query_vector_database(question):
    results = tourism_collection.query(
        query_texts=[question],
        n_results=1
    )
    results_text = results['documents'][0][0]
    return results_text
```

Let's try out this function against the same question we asked previously:

```
results_text = query_vector_database("How many Doric
➥temples are in Paestum")
print(results_text)
```

You'll see output like this:

```
The ancient Greek part of Paestum consists of two sacred areas containing
three Doric temples in a remarkable state of preservation. During the ensuing
Roman period a typical forum [SHORTENED] …
```

This is the result we expected: notice the retrieved chunk correctly contains the text "three Doric temples."

6.3.2 Invoking the LLM

We need to craft a prompt that combines the user's question with the context retrieved from the vector database and then submit it to the LLM. To get started, we'll use a simple prompt and encapsulate the code for calling the LLM in a new function, as shown in the following listing.

Listing 6.2 Functions to define and execute a prompt

```
def prompt_template(question, context):
    return f'Read the following text and answer this question:
    ➥{question}. \nContext: {context}'
```

```
def execute_llm_prompt(prompt_input):
    prompt_response = openai_client.chat.completions.create(
        model='gpt-5-nano',
        messages=[
            {"role": "system", "content": "You are an assistant
            ➥for question-answering tasks."},
            {"role": "user", "content": prompt_input}
        ])
    return prompt_response
```

USING A SIMPLE Q&A PROMPT

Let's test the functions with a simple Q&A prompt. For this, we'll use the question that made ChatGPT hallucinate earlier:

```
trick_question = "How many columns have the three temples got in total?"
tq_result_text = query_vector_database(trick_question)
tq_prompt = prompt_template(trick_question, tq_result_text)
tq_prompt_response = execute_llm_prompt(tq_prompt)
print(tq_prompt_response)
```

We get this as output:

```
ChatCompletion(id='chatcmpl-CGB90lzMBRKYQgWuHCbVLxFPhbMxU', choices=[Choice(
finish_reason='stop', index=0, logprobs=None, message=ChatCompletionMessage(
content='The text does not provide the number of columns for the three
temples.', refusal=None, role='assistant', annotations=[], audio=None,
function_call=None, tool_calls=None))], created=1757972122, model='gpt-5-
nano-2025-08-07', object='chat.completion', service_tier='default',
system_fingerprint=None, usage=CompletionUsage(completion_tokens=983,
prompt_tokens=290, total_tokens=1273, completion_tokens_details=
CompletionTokensDetails(accepted_prediction_tokens=0, audio_tokens=0,
reasoning_tokens=960, rejected_prediction_tokens=0), prompt_tokens_details=
PromptTokensDetails(audio_tokens=0, cached_tokens=0)))
```

The GPT-5-nano model didn't hallucinate. Instead, the model correctly recognized that it didn't have enough information to answer the question. A few months ago, when I ran the same code against the GPT-3.5-turbo model, it gave a wrong answer of 24 columns, with incorrect assumptions about the number of columns in each temple. Next, I'll show you how I fixed the issue.

USING A SAFER Q&A PROMPT

Hallucinations can be mitigated in general by using a well-known prompt for Q&A, available on the LangChain Hub web page (part of LangSmith).

> **TIP** The LangChain Hub (https://smith.langchain.com/hub) is a popular LLM resource, which is constantly updated with open source models, prompts, and advice on use cases. I highly recommend checking it out.

Here's the recommended hallucination-safe RAG prompt from https://smith.langchain.com/hub/rlm/rag-prompt:

Use the following pieces of retrieved context to answer the question. If you don't know the answer, just say that you don't know. Use three sentences maximum and keep the answer concise.

QUESTION {question}

CONTEXT {context}

ANSWER

Let's update the prompt template function accordingly:

```
def prompt_template(question, text):
    return f'Use the following pieces of retrieved context to
      ➥answer the question. Only use the retrieved context to
      ➥answer the question. If you don\'t know the answer, or
      ➥the answer is not contained in the retrieved context,
      ➥just say that you don\'t know. Use three sentences
      ➥maximum and keep the answer concise. \nQuestion:
      ➥{question}\nContext: {text}. Remember: if you do not
      ➥know, just say: I do not know. Do not make up an
      ➥answer. For example do not say the three temples have
      ➥got a total of three columns. \nAnswer:'
```

Now let's resubmit the trick question:

```
trick_question = "How many columns have the three temples got in total?"
tq_result_text = query_vector_database(trick_question)
tq_prompt = prompt_template(trick_question, tq_result_text)
tq_prompt_response = execute_llm_prompt(tq_prompt)
print(tq_prompt_response)
```

We get this as output:

```
ChatCompletion(id='chatcmpl-9nCco9P3xSdArsptotrmJEjtd2N5D', choices=[Choice(
finish_reason='stop', index=0, logprobs=None, message=ChatCompletionMessage(
content='I do not know.', role='assistant', function_call=None, tool_calls=
None))], created=1721513630, model='gpt-4o-mini-2024-07-18', object='chat
.completion', service_tier=None, system_fingerprint='fp_8b761cb050', usage=
CompletionUsage(completion_tokens=5, prompt_tokens=383, total_tokens=388))
```

Well done! You've prevented the LLM from hallucinating. Your chatbot will now only use the knowledge stored in the vector database or admit explicitly that it doesn't know the answer.

6.3.3 Building the chatbot

We can now implement the chatbot with a single function. We'll use the code covered in this section:

```
def my_chatbot(question):
    results_text = query_vector_database(question)
    prompt_input = prompt_template(question,
                                   results_text)
    prompt_output = execute_llm_prompt(
        prompt_input)

    return prompt_output
```

Retrieves content from the vector store

Creates the LLM prompt

Executes the LLM prompt

Let's test it with the original question:

```
question = """Let me know how many temples there
are in Paestum, who constructed them, and what
architectural style they are"""
result = my_chatbot(question)
print(result)
```

We get the following output:

```
ChatCompletion(id='chatcmpl-CGBGz5h3kD6006MccRdhPYB7HuwSR', choices=[Choice(
finish_reason='stop', index=0, logprobs=None, message=ChatCompletionMessage(
content='There are three Doric temples in the ancient Greek part of Paestum.
They are the Temple of Athena (Ceres), the Temple of Hera I (Basilica), and
the Temple of Hera II (Neptune); Athena and Hera I date from the 6th century
BC, while Hera II was probably built about 460 BC. I do not know who
constructed them.', refusal=None, role='assistant', annotations=[],
audio=None, function_call=None, tool_calls=None))], created=1757972617,
model='gpt-5-nano-2025-08-07', object='chat.completion', service_tier=
'default', system_fingerprint=None, usage=CompletionUsage(completion_tokens=
1495, prompt_tokens=398, total_tokens=1893, completion_tokens_details=
CompletionTokensDetails(accepted_prediction_tokens=0, audio_tokens=0,
reasoning_tokens=1408, rejected_prediction_tokens=0), prompt_tokens_details=
PromptTokensDetails(audio_tokens=0, cached_tokens=0)))
```

The synthesized response is comprehensive, as it answers all the questions we asked.

You should be proud of what you've achieved so far! You've implemented a basic chatbot that can answer questions based on text imported into the vector database and provide additional information if needed. It won't return information not in the vector database, so it won't hallucinate or make up answers.

The key takeaway is that you now understand the internals of a Q&A LLM-based system and the components and workflow of the RAG design pattern. This knowledge will help you when using frameworks such as LangChain, LlamaIndex, and Semantic Kernel, which might hide their implementation details. You'll be better equipped to troubleshoot problems and understand what's going on behind the scenes. Before re-implementing RAG with LangChain, let's recap the RAG terminology you've learned so far.

6.3.4 Recap of RAG terminology

Throughout this chapter, you've been learning and refining RAG terminology. Some terms may have similar meanings to ones you've seen earlier. Table 6.2 will help consolidate your understanding, especially for concepts that can be expressed with different terms.

Table 6.2 RAG glossary

Term	Definition	Alternative terms
Retrieval Augmented Generation (RAG)	Use case involving the generation of text (typically an answer) augmented with information retrieved from a content store optimized for semantic searches, typically a vector store	Q&A
Text chunk	A fragment of text from a document. Documents are split into chunks for more effective searching, especially when stored in specialized unstructured text stores such as vector stores.	Text fragment, chunk, text node, node
Embeddings	Numerical (vector) representation of a piece of text, used to index text chunks for semantic searches	Vector
RAG content ingestion stage	Phase in the RAG design where text is imported and indexed into a context store for efficient retrieval against a natural language question. In a vector store, text is broken into chunks and indexed through associated embeddings.	Text indexing, text vectorization, indexing stage
Vector store	In-memory store or specialized database holding text chunks and their related embeddings, which serve as their index	Vector database
Semantic similarity	Comparing pieces of text based on their meaning, typically by calculating the distance between the embeddings of the text pieces. This can be done using cosine distance or Euclidean distance.	Vector similarity, cosine similarity
Semantic search	Searching for information based on its meaning. This involves performing semantic similarity between the embeddings of the search question and the text chunks in a vector store.	Q&A, vector search
Context	Text (or information) provided in the prompt along with the user question, which is used to formulate an answer. This can be a full document or a list of text chunks retrieved from a vector store through semantic search.	–
Synthesize	Generating an answer, typically from a user question and a context that provides the necessary information	Generate
RAG question-answering stage	Phase in the RAG design where a user asks a search question, the application performs a semantic search against a content store (typically a vector store), and it feeds the LLM the original question along with the context retrieved from the store. The LLM then synthesizes and returns the answer to the application, which passes it on to the user.	RAG Q&A stage; retrieval and generation stage

You're now ready to reimplement RAG with LangChain. We'll tackle that in the next chapter.

Summary

- Basic Q&A chatbots pass a question and supporting document directly to an LLM in a single prompt. This works for simple use cases but doesn't scale to large knowledge bases.
- Retrieval-Augmented Generation (RAG) systems answer questions across large knowledge bases. They combine vector search to find relevant documents with LLM synthesis to generate coherent answers.
- RAG operates in two sequential stages: Ingestion converts documents to embeddings and stores them in vector databases; retrieval finds similar documents based on query embeddings and passes them to the LLM.
- Vector stores are databases optimized for similarity search using embeddings. They store text chunks alongside their vector representations and return the most semantically similar results for a given query.
- Platforms such as ChromaDB and Pinecone provide persistent storage and advanced indexing strategies, enabling efficient retrieval across millions of documents. Choose between them based on scale, latency requirements, and deployment constraints.
- RAG systems require three API integrations: an embedding model API (OpenAI, Cohere, Google Vertex AI), a vector store connection, and an LLM API for answer generation. Configure API keys and endpoints for each service.
- Embedding dimensions must match between ingestion and retrieval. OpenAI's `text-embedding-3-small` uses 1,536 dimensions; switching models requires re-embedding your entire corpus.
- The RAG pipeline chains as follows: question → embed query → search vector store → retrieve documents → insert into prompt → generate answer with LLM.
- RAG accuracy depends on chunk quality and retrieval relevance. Poor chunking (mid-sentence splits, orphaned context) degrades results even with perfect retrieval.
- Test different k values (typically 2–8) based on your content density and context window size. Start with k=4 as a baseline, then optimize based on answer quality.

7 Q&A chatbots with LangChain and LangSmith

This chapter covers

- Implementing RAG with LangChain
- Q&A across multiple documents
- Tracing RAG chain execution with LangSmith
- Alternative implementations using the LangChain Q&A specialized functionality

Now that you understand the Retrieval-Augmented Generation (RAG) design pattern, building a RAG-based chatbot with LangChain will feel much more approachable. In this chapter, I'll walk you through how to use the LangChain object model to manage interactions with source documents, the vector store, and the LLM.

We'll also explore how to use LangSmith's tracing tools to monitor and troubleshoot the chatbot workflow. On top of that, I'll demonstrate alternative implementations that use LangChain's specialized Q&A classes and functions.

By the end of this chapter, you'll have the skills to create a search-enabled chatbot that can seamlessly connect to private data sources. But before diving into the

implementation, let's take a moment to review the key LangChain classes that support the Q&A chatbot use case.

7.1 LangChain object model for Q&A chatbots

As discussed earlier, one of LangChain's biggest advantages for LLM-based applications is its ability to orchestrate communication between components such as data loaders, vector stores, and LLMs. Instead of integrating directly with each API, LangChain provides abstractions that let you swap out any component with a different provider—without disrupting the overall design of your application.

Beyond abstraction, LangChain also includes ready-to-use implementations for many common tasks in LLM development. These include splitting source text, retrieving relevant context from a vector store, generating prompts, managing the context window, and more.

Among LangChain's many use cases, Q&A is one of the most fundamental. The workflow for Q&A applications can be broken down into the following two main stages, which we'll discuss next, including the components involved:

- Content ingestion (indexing) stage
- Question-answering (Q&A; retrieval and generation) stage

7.1.1 Content ingestion (indexing) stage

Figure 7.1 summarizes the object model for the content ingestion stage of the Q&A use case. Here are the components shown in the figure:

- `Document`—Models the text content and related metadata.
- `BaseLoader`—Loads text from external sources into the document model.
- `TextSplitter`—Splits documents into smaller chunks for efficient processing. Consequently, a `TextSplitter` has the following signature: `Document -> list[Document]`.
- `VectorStore`—Stores text chunks and related embeddings for efficient retrieval.
- `Embeddings`—Converts text into embeddings (vector representations).

This is a static view of how LangChain class families connect. For a dynamic view, see the sequence diagram in figure 7.2, which shows the typical content ingestion process.

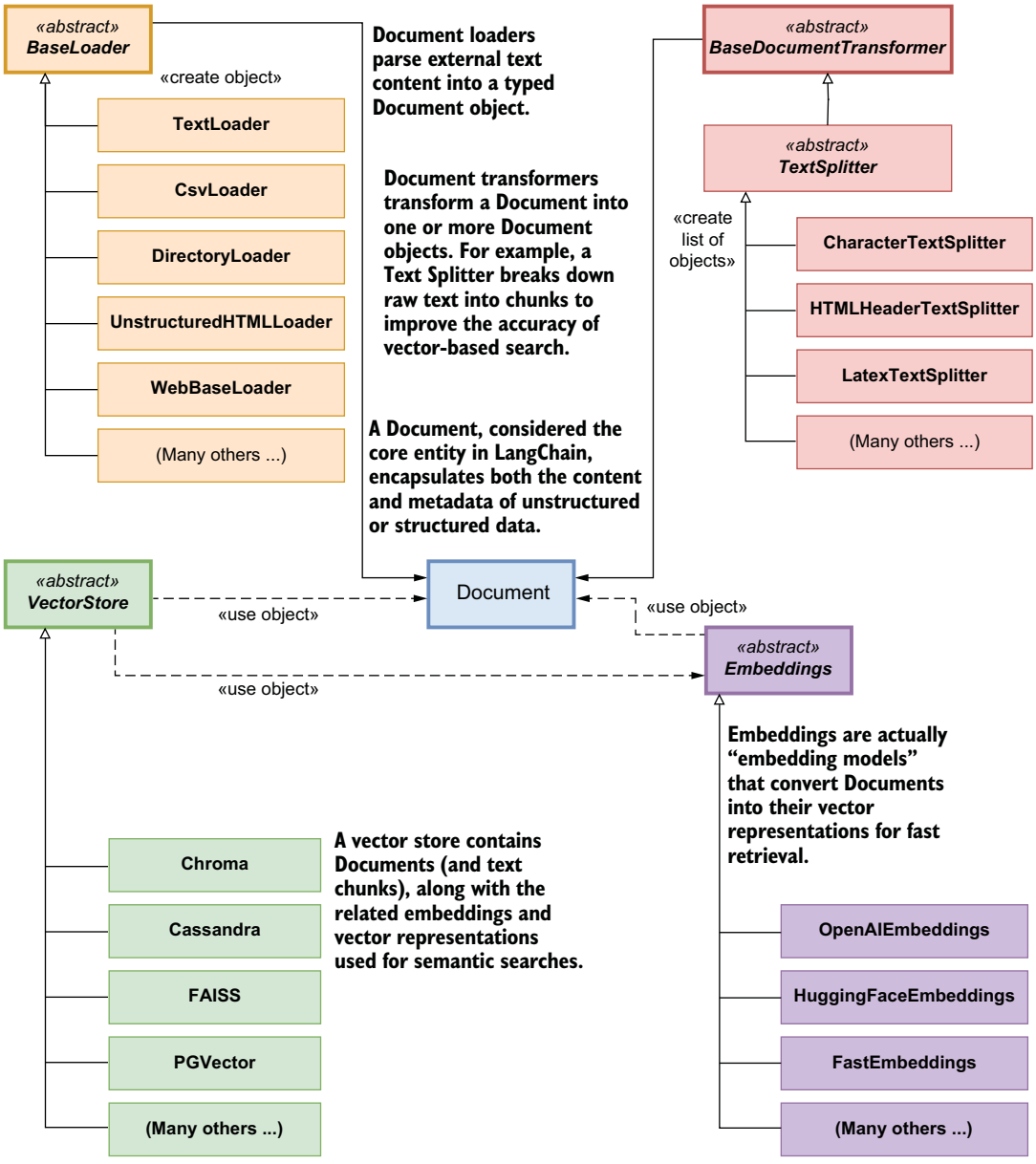


Figure 7.1 Object model associated with the content ingestion stage

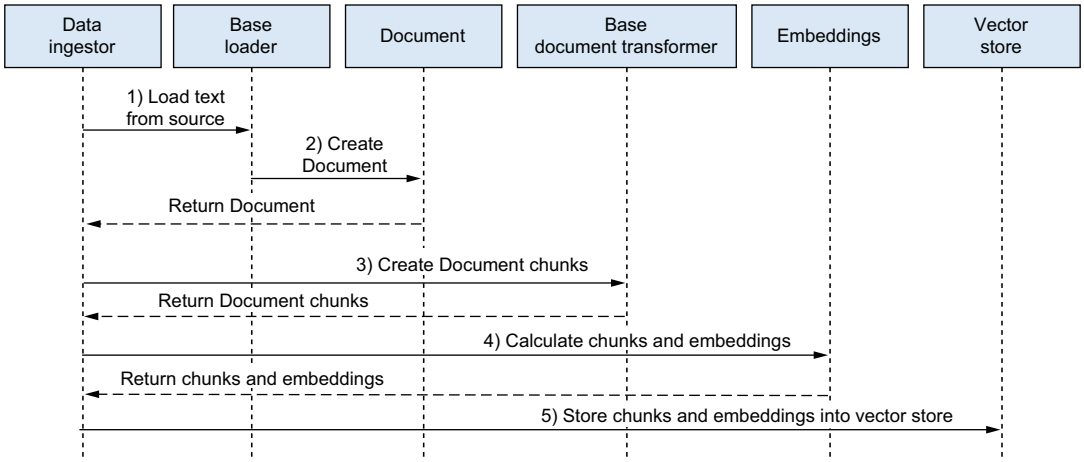


Figure 7.2 Content ingestion sequence diagram

Here's an example of how LangChain classes interact during content ingestion:

1. The data ingestion orchestrator initializes a specific `Loader` to import text from an external source, such as `CsvLoader` for CSV files.
2. The `Loader` parses the text and converts it into a `Document` object.
3. The `Document` is passed to a `DocumentTransformer`, usually a `TextSplitter`, to divide it into smaller chunks.
4. The chunks are processed by an `Embeddings` model to create embeddings.
5. Both the chunks and embeddings are sent to the `VectorStore` for storage.

NOTE Some vector stores can automatically compute embeddings if an embedding model is specified when the vector store client is instantiated. In such cases, you don't need to precompute the embeddings before adding the text chunks to the store—steps 4 and 5 can be combined into a single operation.

7.1.2 Q&A (retrieval and generation) stage

Figure 7.3 summarizes the object model for the retrieval and generation stage of the Q&A use case. The figure includes the following components:

- `VectorStore`—Stores and retrieves relevant text chunks
- `Retriever`—Retrieves relevant text chunks from the vector store based on the similarity between the query's embedding and the stored text embeddings
- `Embeddings` (embedding models)—Ensures consistent embeddings for queries and documents
- `Prompt/PromptTemplate`—Constructs the input for the language model, typically using the user question and a context made of retrieved text chunks
- `LanguageModel`—Generates answers using the provided context and query

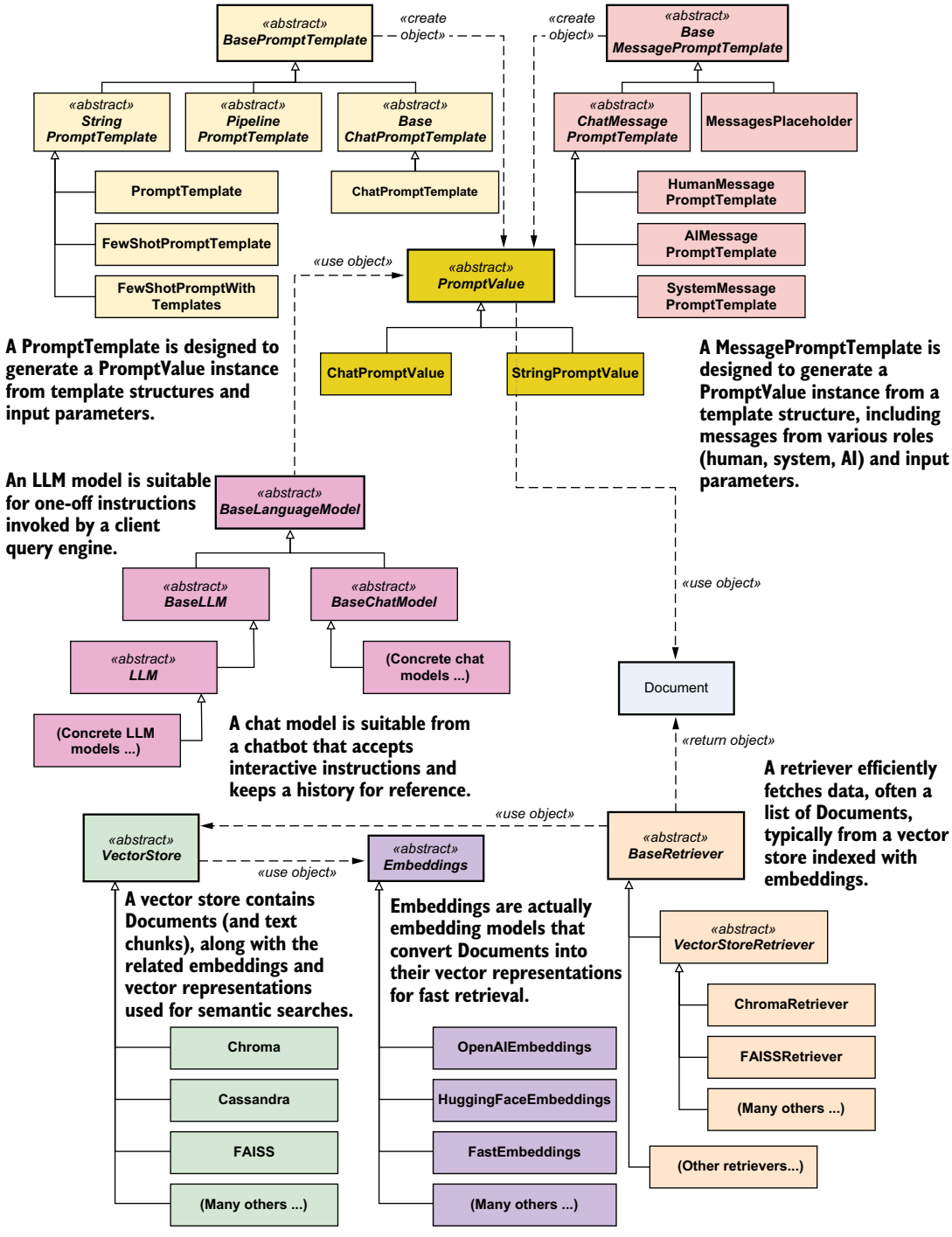


Figure 7.3 Object model associated with the retrieval and generation stage

For a dynamic view of the Q&A workflow, see the sequence diagram in figure 7.4. Here's how LangChain classes interact during the Q&A process:

1. The Q&A orchestrator sends the user's question to the vector store `Retriever`.
2. The `Retriever` generates the question's embedding using an `Embeddings` model.
3. The `Retriever` searches the vector store for documents with similar embeddings and returns them to the Q&A orchestrator.
4. The Q&A orchestrator combines the retrieved documents and the user's question into a prompt using a `PromptTemplate`.
5. The orchestrator sends the prompt to the `LanguageModel`, which returns the answer.

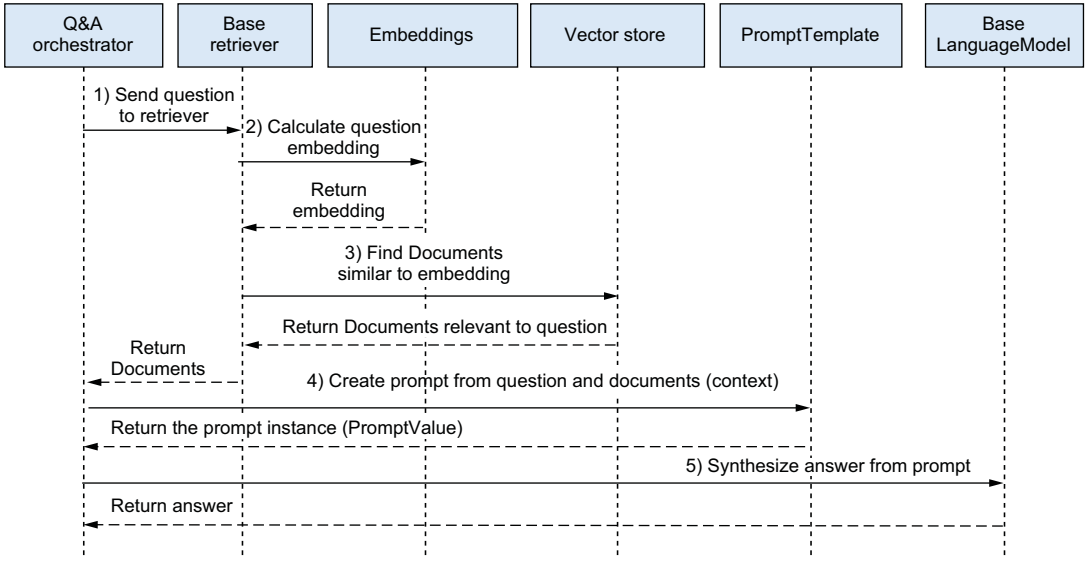


Figure 7.4 Q&A sequence diagram

Now that you understand the Q&A object model, let's get started with the implementation! The first step is storing your documents, as described next.

7.2 Vector store content ingestion

Before querying your documents, you must store them in a vector database. To make it more interesting, we'll import content about Paestum from various sources and in different formats, rather than just using the small Britannica article on Paestum as before.

First, install the required packages for loading, splitting documents, and creating embeddings. Open a new operating system shell, navigate to the folder for this chapter,

create and activate the virtual environment for chapter 7, and install the required packages:

```
C:\Github\building-llm-applications\ch07>python -m venv env_ch07
C:\Github\building-llm-applications\ch07>.\env_ch07\Scripts\activate

(env_ch07) C:\Github\building-llm-applications\ch07>
➥pip install -r requirements.txt
```

Once the installation is complete, start a Jupyter Notebook, as usual:

```
(env_ch07) C:\Github\building-llm-applications\ch07>jupyter notebook
```

Create a new notebook by choosing File > New > Notebook, give it a name (e.g., `ch07-QA_across_documents`), and save it. Then, import the necessary libraries on the notebook:

```
from langchain_community.document_loaders
➥import WikipediaLoader, Docx2txtLoader,
➥PyPDFLoader, TextLoader

from langchain_chroma import Chroma
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_openai import OpenAIEmbeddings
```

Finally, capture the OpenAI API key as usual:

```
import getpass
OPENAI_API_KEY = getpass.getpass('Enter your OPENAI_API_KEY')
```

7.2.1 Splitting and storing the documents

You can make the content more searchable. Follow these steps:

1. Split each document into chunks of about 500 characters. You can add overlap, usually around 10%, but for now, set the overlap to 0. The size of chunks and overlap considerations will be covered in chapter 8 on advanced indexing techniques.
2. Calculate the embeddings of the document chunks, and store them in the Chroma vector database.

Instantiate the splitter, embeddings model, and vector database client as follows:

```
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500, chunk_overlap=0)
embeddings_model = OpenAIEmbeddings(
    openai_api_key=OPENAI_API_KEY)
vector_db = Chroma("tourist_info", embeddings_model)
```

Process each document by loading it, splitting it into chunks, and storing the chunks with the related embeddings into the vector database:

```
wikipedia_loader = WikipediaLoader(query="Paestum")
wikipedia_chunks = text_splitter.split_documents(
    wikipedia_loader.load())
vector_db.add_documents(wikipedia_chunks)
```

Once the content has been split and stored in the vector database, you'll see an output similar to this:

```
['d3197373-7df1-4c24-8a0a-0145c176042c',
 '435add06-6b85-421a-8ad8-be88b7defe08',
 '7fbd7575-5f56-4fed-9642-34f587325699',
 '6a798aca-9dcb-433c-b0d3-196acfd83b0e',
…
]
```

> **NOTE** The `WikipediaLoader` also loads content from other Wikipedia hyperlinks referenced in the requested article. For example, the Paestum article references the National Archaeological Museum of Paestum, the Lucania region, Lucanians, and the temples of Hera and Athena. As a result, it will load these related articles, providing more content than you might expect.

For other document formats (don't run it yet, as I'll improve this code shortly), we'll use the following:

```
word_loader = Docx2txtLoader("Paestum/Paestum-Britannica.docx")
word_chunks = text_splitter.split_documents(
    word_loader.load())
vector_db.add_documents(word_chunks)

pdf_loader = PyPDFLoader("Paestum/PaestumRevisited.pdf")
pdf_chunks = text_splitter.split_documents(
    pdf_loader.load())
vector_db.add_documents(pdf_chunks)

txt_loader = TextLoader("Paestum/Paestum-Encyclopedia.txt")
txt_chunks = text_splitter.split_documents(
    txt_loader.load())
vector_db.add_documents(txt_chunks)
```

> **NOTE** If you prefer, you can process the full `PaestumRevisited-Stockholms-Universitet.pdf` document instead of its shorter version, `Paestum-Revisited.pdf`. This will take considerably longer, but it will provide information for a wider range of questions.

7.2.2 Removing duplication

You might have noticed that the preceding code has a lot of duplication. Let's extract some common functionality into a function:

```
def split_and_import(loader):
     chunks = text_splitter.split_documents(loader.load())
     vector_db.add_documents(chunks)
     print(f"Ingested chunks created by {loader}")
```

Now you can call this function after instantiating each loader, as shown in the next listing.

Listing 7.1 Refactored ingestion of different file types

```
wikipedia_loader = WikipediaLoader(query="Paestum")
split_and_import(wikipedia_loader)

word_loader = Docx2txtLoader("Paestum/Paestum-Britannica.docx")
split_and_import(word_loader)

pdf_loader = PyPDFLoader("Paestum/PaestumRevisited.pdf")
split_and_import(pdf_loader)

txt_loader = TextLoader("Paestum/Paestum-Encyclopedia.txt")
split_and_import(txt_loader)
```

After running this code, you'll see output like this:

```
Ingested chunks created by <langchain_community.document_loaders
     .wikipedia.WikipediaLoader object at 0x000001EABD096510>
Ingested chunks created by [… SHORTENED]
```

7.2.3 Ingesting multiple documents from a folder

I've just shown you how to ingest different types of documents (web, DOCX, PDF, TXT) using a specialized document loader for each type. This approach works fine for a few documents, but if you need to load many documents, a more efficient method is needed. After placing all the documents in a folder (e.g., `/CilentoTourist-Info`), you can achieve this in the following ways, which we'll explore next:

- Iterating over the files in the folder and calling the relevant loader
- Using the purpose-built `DirectoryLoader`

Iterating over all files in a folder

You can ingest all the files located in a folder into a vector store by iterating over the files, identifying the file type, and using the relevant loader as described in the previous section. Before iterating over the files, create a loader factory as shown here.

Listing 7.2 Loader factory: Instantiating the relevant loader

```
loader_classes = {
    'docx': WordLoader,
    'pdf': PdfLoader,
    'txt': TxtLoader
}

import os

def get_loader(filename):
    _, file_extension = os.path.splitext(filename)
    file_extension = file_extension.lstrip('.')

    loader_class = loader_classes.get(
        file_extension)
```

Extracts the file extension

Removes the leading dot from the extension

Gets the loader class from the dictionary

```
if loader_class:
    return loader_class(filename)
else:
    raise ValueError(f"No loader available for file
    ➥extension '{file_extension}'")
```

Instantiates and returns the correct loader

Exercise

Iterate over the files in `/CilentoTouristInfo`, and load them into the vector store using `get_loader()` and `split_and_import()`. If you're unsure, check the "Ingesting Multiple Documents from a Folder" section in my Python notebook `ch07-QA_across_documents.ipynb` on GitHub.

INGESTING ALL FILES WITH DIRECTORYLOADER

An alternative method for loading all files in a folder into the vector store is by using the `DirectoryLoader`. This loader, part of a third-party package by Unstructured (a company that offers a platform and tools for ingesting and processing unstructured documents for RAG and fine-tuning), takes a folder path and a *glob* pattern (a string with wildcard characters to specify a set of filenames).

First, install the `unstructured` package or the `langchain-unstructured` wrapper along with its dependencies. Follow the instructions for your operating system in the LangChain documentation (https://python.langchain.com/), and then choose Integrations > Providers table > Unstructured or the Unstructured documentation (https://docs.unstructured.io/).

In your Jupyter Notebook, import the `DirectoryLoader`:

```
from langchain_community.document_loaders import DirectoryLoader
```

You can now load and ingest the files into the vector store with the following code:

```
folder_path = "CilentoTouristInfo"
pattern = "**/*.{docx,pdf,txt}"

directory_loader = DirectoryLoader(folder_path,
➥pattern)
split_and_import(directory_loader)
```

Pattern to match .docx, .pdf, and .txt files

Initializes the DirectoryLoader with the folder path and pattern

Note that while the preceding code is included in the GitHub repository, its execution depends on successfully installing the `unstructured` or `langchain-unstructured` package. Because setup can vary across operating systems, the code may not run consistently in all environments. I've added comments in the code to clarify this.

7.3 Q&A across stored documents

Now that you've stored all the content about Paestum, let's explore how to use the vector store to retrieve information across multiple documents. This step is important because it shows whether your ingestion and chunking strategy is working as

expected—specifically, whether the system can surface the most relevant pieces of text when a user asks a question that spans different sources.

7.3.1 Querying the vector store directly

Let's query the vector store to see what documents are retrieved for a question that requires information from different sources:

```
query = "Where was Poseidonia and who renamed it to Paestum?"
results = vector_db.similarity_search(query, 4)
print(results)
```

Retrieves the four closest results

Here's an excerpt from the results, showing that the most relevant chunks returned by the vector store come from different sources, as you can verify in the related metadata:

```
[Document(metadata={'source': 'Paestum/Paestum-Britannica.docx'},
page_content='Paestum, Greek\xa0Poseidonia, ancient city in
southern\xa0Italy\xa0near the west coast, 22 miles (35 km) [SHORTENED..]'),
Document(metadata={'source': 'Paestum/Paestum-Britannica.docx'},
page_content='Paestum, Greek\xa0Poseidonia, ancient city in
southern\xa0Italy\xa0near the west coast, 22 miles (35 km) southeast of
modern\xa0Salerno\xa0and 5 [SHORTENED..]), Document(metadata={'source':
'https://en.wikipedia.org/wiki/Paestum', 'summary': 'Paestum ( PEST-əm, US
also  PEE-stəm ) was a major ancient Greek city on the coast of the
Tyrrhenian Sea, in Magna Graecia. The ruins of Paestum are famous for
their three ancient Greek temples in the Doric order dating from about
550 to 450 BC that are in an excellent state of preservation.
[SHORTENED..]'), Document(metadata={'source': 'Paestum/Paestum-
Britannica.docx'}, page_content='Poseidonia was probably founded about
600\xa0BC\xa0by Greek colonists from\xa0Sybaris, along the\xa0Gulf of
Taranto, and it had become a flourishing town by 540, judging from its
temples. [SHORTENED..]')]
```

Although you now know which documents the vector store returns for your query, the next step is to get a well-formulated answer. We'll get that answer via the LangChain chain.

7.3.2 Asking a question through a LangChain chain

Unlike when you implemented RAG using the OpenAI GPT-5-nano model directly in chapter 5, with LangChain, you don't need to manually craft a prompt and configure it with the original question and the context from the vector store. You can set up a RAG chain, which will automatically instantiate and execute a prompt template, as shown in figure 7.5.

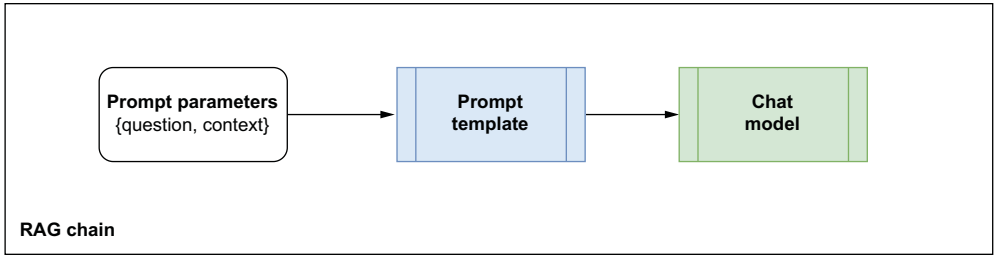


Figure 7.5 LangChain RAG chain packaging the prompt parameters into a prompt instance, which is then sent to the chat or LLM model

Set up the prompt using a template from the LangChain Hub (https://smith.langchain.com/hub). For clarity, I've simply copied this prompt from the LangChain Hub and set it explicitly here:

```
from langchain_core.prompts import PromptTemplate

rag_prompt_template = """Use the following pieces of context
to answer the question at the end.
If you don't know the answer, just say that you don't know,
don't try to make up an answer.
Use three sentences maximum and keep the
answer as concise as possible.
{context}
Question: {question}
Helpful Answer:"""

rag_prompt = PromptTemplate.from_template(rag_prompt_template)
```

Alternatively, you can pull the prompt instance directly from the LangChain Hub:

```
from langchain import hub
rag_prompt = hub.pull("rlm/rag-prompt")
```

7.3.3 Completing the RAG chain setup

To complete the setup of the RAG chain, we need a few more components, as shown in figure 7.6:

- Retriever—This component retrieves relevant text content from the vector store and injects it into the context parameter of the prompt.
- Question feeder—Implemented as a simple pass-through component, it feeds the user's question through the `Runnable` interface (an abstract class on which every LangChain component is based).
- Chat model—This component processes the prompt to generate the answer.

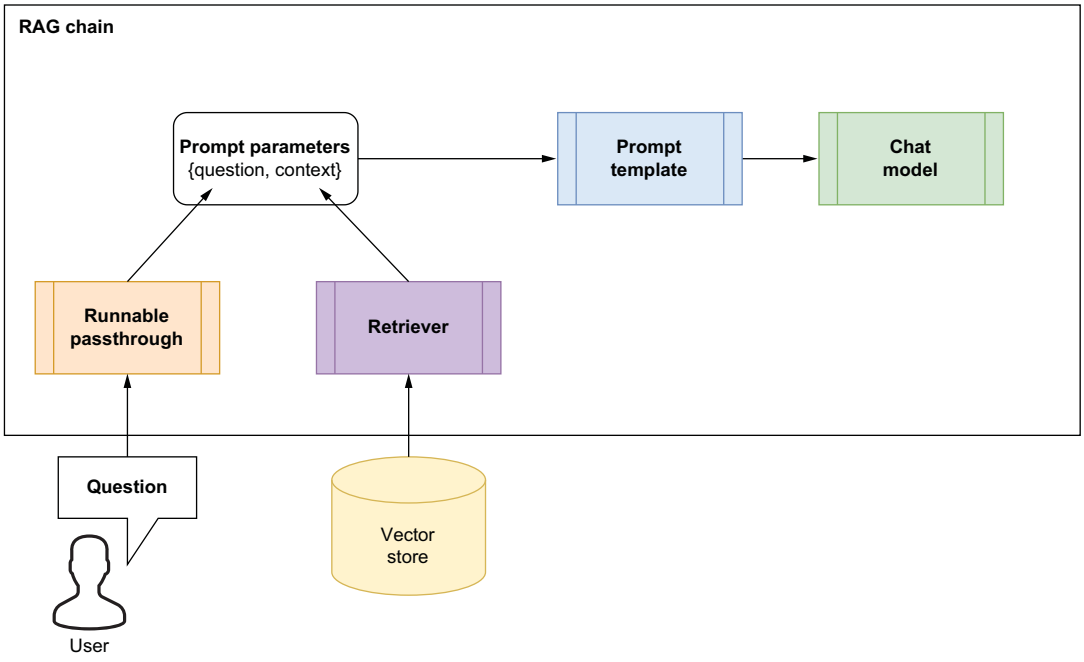


Figure 7.6 Amended RAG chain, including runnable pass-through and retriever

Instantiate the retriever, question feeder, and chat model:

```
retriever = vector_db.as_retriever()

from langchain_core.runnables import RunnablePassthrough
question_feeder = RunnablePassthrough()

from langchain_openai import ChatOpenAI

chatbot = ChatOpenAI(openai_api_key=OPENAI_API_KEY,
                     model_name="gpt-5-nano")
```

Set up the RAG chain. As mentioned earlier, each block in a LangChain chain implements the `Runnable` interface and accepts a dictionary as input. This is why the first block in the chain is a dictionary:

```
rag_chain = {"context": retriever,
             "question": question_feeder}|rag_prompt|chatbot
```

Create a utility function to execute the chain:

```
def execute_chain(chain, question):
    answer = chain.invoke(question)
    return answer
```

Ask a question:

```
question = """Where was Poseidonia and who renamed
it to Paestum. Also tell me the source."""
answer = execute_chain(rag_chain, question)
print(answer.content)
```

The answer you'll get should be similar to this:

```
- Poseidonia was a Greek settlement on the Tyrrhenian coast of southern
  Italy, at the Gulf of Taranto (Magna Graecia).
- It was renamed Paestum by the Romans after they took control (273 BCE).
- Source: Britannica, Paestum entry (Paestum-Britannica.docx).
```

Inspect the full answer object if needed:

```
print(answer)
```

You should see something like this:

```
content='- Poseidonia was a Greek settlement on the Tyrrhenian coast of
southern Italy, at the Gulf of Taranto (Magna Graecia).\n- It was renamed
Paestum by the Romans after they took control (273 BCE).\n- Source:
Britannica, Paestum entry (Paestum-Britannica.docx).'
additional_kwargs={'refusal': None} response_metadata={'token_usage':
{'completion_tokens': 1999, 'prompt_tokens': 1889, 'total_tokens': 3888,
'completion_tokens_details': {'accepted_prediction_tokens': 0,
'audio_tokens': 0, 'reasoning_tokens': 1920, 'rejected_prediction_tokens':
0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0}},
'model_name': 'gpt-5-nano-2025-08-07', 'system_fingerprint': None, 'id':
'chatcmpl-CGCV6EBbH29VRQ0siMkomrR0brmPt', 'service_tier': 'default',
'finish_reason': 'stop', 'logprobs': None} id='run--78fad7ef-2b09-4b87-b206-
f1b5e94216e8-0' usage_metadata={'input_tokens': 1889, 'output_tokens': 1999,
'total_tokens': 3888, 'input_token_details': {'audio': 0, 'cache_read': 0},
'output_token_details': {'audio': 0, 'reasoning': 1920}}
```

7.3.4 Follow-up question

Let's find out if the chatbot can sustain the conversation by asking a follow-up question. I'll make it deliberately vague to see if the chatbot can understand what I'm after:

```
question = """And then, what they do?
Tell me only if you know.
Also tell me the source"""
answer = execute_chain(rag_chain, question)
print(answer.content)
```

The chatbot returns the following:

```
Sirens are female humanlike beings with alluring voices, who appear in the
Odyssey in a scene where Odysseus saves his crew's lives. Source: Siren
(mythology), Wikipedia.
```

It seems the chatbot is a bit lost and doesn't understand that "they" refers to the Romans. This is because the chatbot has no memory of previous questions and answers. Currently, it's stateless and simply passes questions from the user to the LLM and back, without retaining any memory of the conversation flow. Let's see how we can fix this.

7.4 Chatbot memory of message history

One of the most useful features of an LLM-based chatbot, compared to an LLM-based engine, is its ability to remember previous questions and responses, allowing you to continue querying until you get a satisfactory answer. This capability is crucial for providing context to each new prompt. Now let's see how to incorporate message history into the RAG setup we finalized in the previous section, which I've summarized in the following listing for convenience.

Listing 7.3 Initial RAG setup before incorporating message history

```
from langchain_core.runnables import RunnablePassthrough
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI

rag_prompt_template = """Use the following pieces of context
to answer the question at the end.
If you don't know the answer, just say that you don't know,
don't try to make up an answer.
Use three sentences maximum and keep the
answer as concise as possible.
{context}
Question: {question}
Helpful Answer:"""

rag_prompt = PromptTemplate.from_template(rag_prompt_template)

retriever = vector_db.as_retriever()

question_feeder = RunnablePassthrough()

chatbot = ChatOpenAI(openai_api_key=OPENAI_API_KEY,
                     model_name="gpt-5-nano")

rag_chain = {"context": retriever,
             "question": question_feeder}|rag_prompt|chatbot

def execute_chain(chain, question):
    answer = rag_chain.invoke(question)
    return answer
```

First, we should amend the prompt to include message history. We'll do that next.

7.4.1 Amending the prompt

The original RAG prompt doesn't account for message history, so we need to modify it. Because message history is a core feature of the memory-enabled RAG design, we should use a different prompt helper than `PromptTemplate.from_template`. This helper was based on a template parameterized by a user question (`{question}`) and the context (`{context}`) pulled by the retriever from the vector store. LangChain provides a more suitable prompt helper, `ChatPromptTemplate.from_messages`, which creates a prompt from a list of chat messages.

CHAT MESSAGES

Most chat-oriented LLMs use a standard format for messages and roles associated with chat messages. A chat message is typically a key–value pair like

```
("role", "message text")
```

where

- `"role"` can be `"system"`, `"human"`, or `"ai"` (see table 7.1 for descriptions).
- `"message text"` stands for the text of the exchanged message.

Table 7.1 Chat message roles

Role	Description	Sample message
`"system"`	This role represents the chatbot application. You typically create one system message at startup to instruct the chatbot on its persona.	You are a world-class expert in Roman and Greek history, especially in towns located in southern Italy. Provide interesting insights on local history and recommend places to visit with knowledgeable and engaging answers.
`"human"`	This message comes from a user, typically a question.	Can you please recommend some attractions around Paestum and give me some information on them?
`"ai"`	This is the synthesized response from the LLM.	The best attractions in Paestum are the three Greek temples built around 500 BC.
Custom	You can use other nonstandard roles to incorporate messages containing useful information. For example, `"Context"` for text retrieved from the vector store, or `"Chat History"` for the entire chat history.	–

A chat history is a list of such messages:

```
chat_history = [
    ("system", """You are a world-class expert in Roman and
    Greek history, especially in towns located in southern Italy.
    Provide interesting insights on local history and recommend
    places to visit with knowledgeable and engaging answers."""),
```

```
    ("human", """Can you please recommend some attractions
    around Paestum and give me some information on them?"""),
    ("context", """Paestum was a major ancient Greek city on the
    coast of the Tyrrhenian Sea, in Magna Graecia. The ruins of
    Paestum are famous for their three ancient Greek temples in
    the Doric order dating … [SHORTENED]"""),
    ("ai", """The best attractions in Paestum are the three Greek
    temples built around 500 BC""")
]
```

CHAT-BASED PROMPT

Now that you understand how to model chat messages, you can create a message-based prompt:

```
from langchain_core.prompts import ChatPromptTemplate
rag_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", """You are a helpful assistant, world-class
        expert in Roman and Greek history, especially in towns
        located in southern Italy. Provide interesting insights
        on local history and recommend places to visit with
        knowledgeable and engaging answers. Answer all questions
        to the best of your ability, but only use what has been
        provided in the context. If you don't know, just say you
        don't know. Use three sentences maximum and keep
        the answer as concise as possible."""),
        ("placeholder", "{chat_history_messages}"),
        ("assistant", "{retrieved_context}"),
        ("human", "{question}"),
    ]
)
```

You'll re-instantiate this prompt at each interaction with the user. Specifically, after the initial interaction, which starts with an empty `chat_history`, you'll feed the new `{question}`, the newly `{retrieved_context}`, and the accumulated `{chat_history_memory}`. You already know how to feed a new question and a newly retrieved context: it's the same as before, but the prompt is now message-based. In the next section, I'll show you how to update the message history at each user interaction.

7.4.2 Updating the chat message history

LangChain provides the `ChatMessageHistory` class to model chat message history. We can instantiate the `chat_history_memory` variable as a `ChatMessageHistory` type to hold the chat history:

```
from langchain_community.chat_message_histories import ChatMessageHistory
chat_history_memory = ChatMessageHistory()
```

You can add messages for `"human"` (user question) and `"ai"` (LLM response) to the chat history using the `ChatMessageHistory` convenience methods listed in table 7.2.

Table 7.2 `ChatMessageHistory` convenience methods for standard roles

Role	Convenience method
Human	`add_user_message(user_question)`
AI	`add_ai_message(llm_response)`

You don't need to add messages associated with the `"system"` and `"chat_history"` roles to the chat message history, as they are already part of the prompt and won't provide the LLM with additional information. The only messages that need tracking are those encapsulating user questions and LLM responses.

Update the chat history in the `execute_chain()` function as follows:

```
def execute_chain_with_memory(chain, question):
    chat_history_memory.add_user_message(question)
    answer = chain.invoke(question)
    chat_history_memory.add_ai_message(answer)
    print(f'Full chat message history: {chat_history_memory.messages}\n\n')
    return answer
```

When the `chat_history_memory` object is updated with the latest Human or AI messages, you can retrieve the entire message history using the `messages` property of the `ChatMessageHistory` class:

```
full_message_history = chat_history_memory.messages
```

You'll inject this into the prompt each time the user asks a new question.

7.4.3 Feeding the chat history to the RAG chain

After updating your code to include message history in the prompt and updating it after each user question and LLM response, you need to feed the updated message history to the RAG chain. You can do this by redefining the chain with LangChain Expression Language (LCEL):

```
rag_chain = {
    "retrieved_context": retriever,
    "question": question_feeder,
    "chat_history_messages": chat_history_memory.messages
} | rag_prompt | chatbot
```

In this setup, the `chat_history_messages` prompt parameter is fed through the corresponding `chat_history_messages` property.

7.4.4 Putting everything together

To clarify the changes made for integrating chatbot memory into the RAG chain, the complete code is given in listing 7,4. Seeing all the components together helps illustrate how the chat history is retrieved, inserted into the prompt, and updated after

each interaction. This consolidated view also shows how LCEL ties the retriever, prompt, memory, and model into a single coherent workflow.

Listing 7.4 RAG chain with chatbot memory

```
from langchain_core.runnables import RunnablePassthrough
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_community.chat_message_histories import ChatMessageHistory
from langchain_core.runnables import RunnableLambda

rag_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", """You are a helpful assistant, world-class
            expert in Roman and Greek history, especially in towns
            located in southern Italy. Provide interesting insights
            on local history and recommend places to visit with
            knowledgeable and engaging answers. Answer all questions
            to the best of your ability, but only use what has been
            provided in the context. If you don't know, just say
            you don't know. Use three sentences maximum and keep
            the answer as concise as possible."""),
        ("placeholder", "{chat_history_messages}"),
        ("assistant", "{retrieved_context}"),
        ("human", "{question}"),
    ])

retriever = vector_db.as_retriever()
question_feeder = RunnablePassthrough()
chatbot = ChatOpenAI(openai_api_key=OPENAI_API_KEY,
                     model_name="gpt-5-nano")
chat_history_memory = ChatMessageHistory()

def get_messages(x):
    return chat_history_memory.messages

rag_chain = {
    "retrieved_context": retriever,
    "question": question_feeder,
    "chat_history_messages": RunnableLambda(get_messages)
} | rag_prompt | chatbot

def execute_chain_with_memory(chain, question):
    chat_history_memory.add_user_message(question)
    answer = chain.invoke(question)
    chat_history_memory.add_ai_message(answer)
    print(f'Full chat message history: {chat_history_memory.messages}\n\n')
    return answer
```

TESTING THE AMENDED CHAIN

Now let's test the updated chain. We'll use the same question we asked previously:

```
question = """Where was Poseidonia and who renamed
it to Paestum? Also tell me the source."""
```

```
answer = execute_chain_with_memory(rag_chain, question)
print(answer.content)
```

Here is the chat message history accumulated so far, followed by the answer:

```
Full chat message history: [HumanMessage(content='Where was Poseidonia and
who renamed \nit to Paestum? Also tell me the source.', additional_kwargs={},
response_metadata={}), AIMessage(content='Poseidonia was a Greek city on the
Tyrrhenian coast of southern Italy (Magna Graecia), founded around 600 BCE by
settlers from Sybaris. It was renamed Paestum by the Romans after they took
over (273 BCE). Source: Paestum article, Wikipedia (https://en.wikipedia.org/
wiki/Paestum).', additional_kwargs={'refusal': None},
response_metadata={'token_usage': {'completion_tokens': 1811,
'prompt_tokens': 1951, 'total_tokens': 3762, 'completion_tokens_details':
{'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens':
1728, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details':
{'audio_tokens': 0, 'cached_tokens': 0}}, 'model_name': 'gpt-5-nano-2025-08-
07', 'system_fingerprint': None, 'id': 'chatcmpl-
CGCq8gylIKzPqi8dJGz73VXr8Mkcd', 'service_tier': 'default', 'finish_reason':
'stop', 'logprobs': None}, id='run--0c943760-b79c-4eae-aa6b-fbc2858aed7b-0',
usage_metadata={'input_tokens': 1951, 'output_tokens': 1811, 'total_tokens':
3762, 'input_token_details': {'audio': 0, 'cache_read': 0},
'output_token_details': {'audio': 0, 'reasoning': 1728}})]
```

> “ Poseidonia was an ancient Greek city located on the coast of the Tyrrhenian Sea in what is now southern Italy. It was renamed Paestum by the Romans after they took control of the city in 273 BC, following its conquest by the Lucanians. The source of this information is from Wikipedia.

The answer is similar to what we got with the memoryless chatbot. Let’s see what happens if we now ask the same follow up-question we asked previously:

```
question = """And then what did they do?
Also tell me the source"""
answer = execute_chain_with_memory(rag_chain, question)
print(answer.content)
```

Now we get the following response:

```
Full chat message history: [HumanMessage(content='Where was Poseidonia and
who renamed \nit to Paestum? Also tell me the source.', additional_kwargs={},
response_metadata={}), AIMessage(content='Poseidonia was a Greek city on the
Tyrrhenian coast of southern Italy (Magna Graecia), founded around 600 BCE by
settlers from Sybaris. It was renamed Paestum by the Romans after they took
over (273 BCE). Source: Paestum article, Wikipedia (https://en.wikipedia.org/
wiki/Paestum).', additional_kwargs={'refusal': None},
response_metadata={'token_usage': {'completion_tokens': 1811,
'prompt_tokens': 1951, 'total_tokens': 3762, 'completion_tokens_details':
{'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens':
1728, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details':
{'audio_tokens': 0, 'cached_tokens': 0}}, 'model_name': 'gpt-5-nano-2025-08-
07', 'system_fingerprint': None, 'id': 'chatcmpl-
CGCq8gylIKzPqi8dJGz73VXr8Mkcd', 'service_tier': 'default', 'finish_reason':
'stop', 'logprobs': None}, id='run--0c943760-b79c-4eae-aa6b-fbc2858aed7b-0',
```

```
usage_metadata={'input_tokens': 1951, 'output_tokens': 1811, 'total_tokens':
3762, 'input_token_details': {'audio': 0, 'cache_read': 0},
'output_token_details': {'audio': 0, 'reasoning': 1728}}),
HumanMessage(content='And then what did they do? \nAlso tell me the source',
additional_kwargs={}, response_metadata={}), AIMessage(content='The available
source states that the Romans took over Poseidonia around 273 BCE and renamed
it Paestum; it does not detail any further actions. Source: Paestum article,
Wikipedia (https://en.wikipedia.org/wiki/Paestum).',
additional_kwargs={'refusal': None}, response_metadata={'token_usage':
{'completion_tokens': 1084, 'prompt_tokens': 1868, 'total_tokens': 2952,
'completion_tokens_details': {'accepted_prediction_tokens': 0,
'audio_tokens': 0, 'reasoning_tokens': 1024, 'rejected_prediction_tokens':
0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0}},
'model_name': 'gpt-5-nano-2025-08-07', 'system_fingerprint': None, 'id':
'chatcmpl-CGCrgyIn28RiUGKQuPOQQOTH2ZRgP', 'service_tier': 'default',
'finish_reason': 'stop', 'logprobs': None}, id='run--c52135d6-bbba-42e5-9e79-
fb9a413e38e1-0', usage_metadata={'input_tokens': 1868, 'output_tokens': 1084,
'total_tokens': 2952, 'input_token_details': {'audio': 0, 'cache_read': 0},
'output_token_details': {'audio': 0, 'reasoning': 1024}})]
```

> “ After the Romans renamed Poseidonia to Paestum, they developed the city further, enhancing its infrastructure and economy, particularly through agriculture and trade. They also constructed significant buildings, including temples dedicated to Greek gods, which demonstrate the city’s cultural continuity. The source of this information is from historical texts on Roman history and archaeology.

As you can see, the chat history now contains all the exchanged questions and answers. Most importantly, the chatbot understands that “they” refers to the Romans and provides a coherent response. Congratulations! You’ve now completed a fully functional chatbot that remembers previous messages and can sustain a proper conversation. Before considering the chatbot complete, however, I’d like to cover another important topic: tracing its chain execution with LangSmith.

7.5 Tracing execution with LangSmith

LangSmith is a comprehensive developer platform for every stage of the LLM-based application life cycle, whether you’re using LangChain or not. It helps you debug, collaborate on, test, and monitor your LLM applications. LLM applications can be unpredictable due to their natural language inputs, which can create edge cases that are hard to reproduce and debug with traditional tools. LangSmith addresses these challenges throughout the LLM application development life cycle:

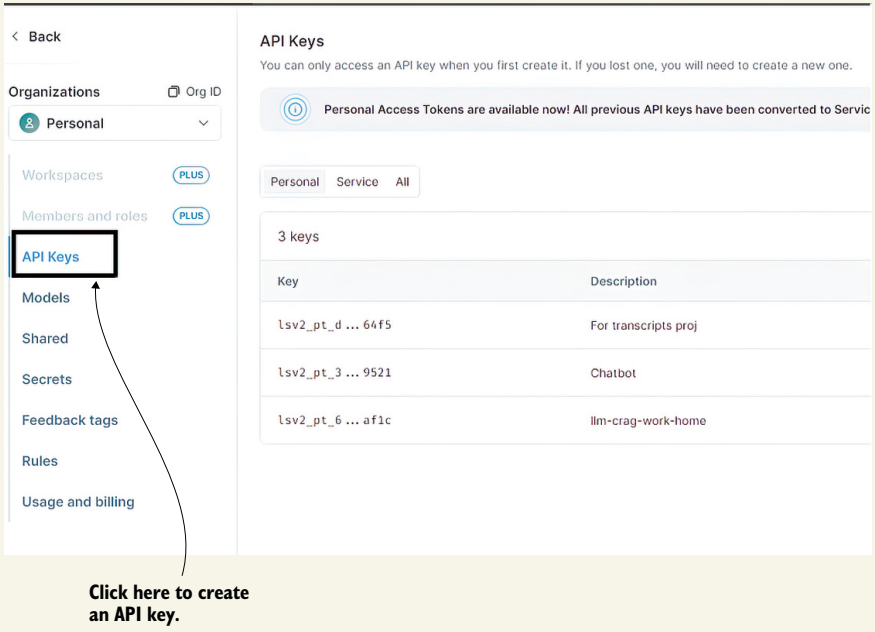
- *Development and debugging*—LangSmith’s tracing feature ensures that your LLM application workflow executes as expected and helps troubleshoot deviations. Its Hub provides standard, battle-tested prompts for common use cases, speeding up implementation.
- *Evaluation and testing*—LangSmith supports testing with built-in evaluators for relevance, correctness, and sensitivity of LLM completions. It also provides tools to build datasets from various sources, including production data, for continuous and regression testing.

- *Monitoring*—LangSmith's tracing capabilities allow you to monitor the real-time status of your LLM production application. It supports feedback through human labels and annotations, enabling investigation and correction when the application deviates from the expected path.

Setting up LangSmith's API key

To enable LangSmith's tracing, you just need to set up the LangSmith API key and some tracing configurations at the top of your application. You can set up a LangSmith API key as follows:

1. Register at LangSmith (https://smith.langchain.com/). Click the Sign-Up button to create an account using GitHub, Google, Discord, or an email address.
2. Log in to the LangSmith website, click the Settings cogwheel on the left-hand menu, and then click the API Keys menu item to access the API Keys page, as shown in the following figure.



API Keys screen

3. Click the Create API Key button in the top-right corner (not shown in the figure). Provide a description for the key (e.g., `AI Agents and applications examples`), and select Personal Access Token as the key type. Then, click Create API Key. Be sure to copy the key and store it in a secure location, as you won't be able to view it again on the LangSmith site—only the first and last few characters will remain visible.

With the LangSmith API key created, you can now use it in your projects.

The easiest way to enable tracing through LangSmith is by setting a few environment variables before launching your Jupyter Notebook. If your notebook is already running, close it first. Then, follow these steps (assuming you're using a Windows cmd shell):

1. Navigate to the notebook folder, and activate your virtual environment:

```
C:\Github\building-llm-applications\ch07> env_ch07\Scripts\activate
```

2. Set the following environment variables, which activate and configure tracing through LangSmith (I assume you're in `C:\Github\building-llm-applications\ch07`, so I've shortened the folder name):

```
(env_ch07) C:\...\ch07>set LANGSMITH_TRACING=true
(env_ch07) C:\...\ch07>set
➥ LANGSMITH_ENDPOINT=https://api.smith.langchain.com
(env_ch07) C:\...\ch07>set LANGSMITH_PROJECT=Q & A chatbot
(env_ch07) C:\...\ch07>set LANGSMITH_API_KEY=<YOUR_LANGSMITH_API_KEY>
```

3. Restart the Jupyter Notebook:

```
(env_ch07) C:\...\ch07>jupyter notebook 07-QA_across_documents.ipynb
```

4. Rerun the notebook cells up to the end of section 7.3 or optionally through section 7.4.

Once you've completed these steps, LangSmith will have captured tracing information for your notebook execution. You can view and analyze this trace data directly in the LangSmith dashboard.

To inspect the traces, go to the LangSmith website (https://smith.langchain.com), and choose Projects > Q&A Chatbot. Click the latest trace, as shown in figure 7.7.

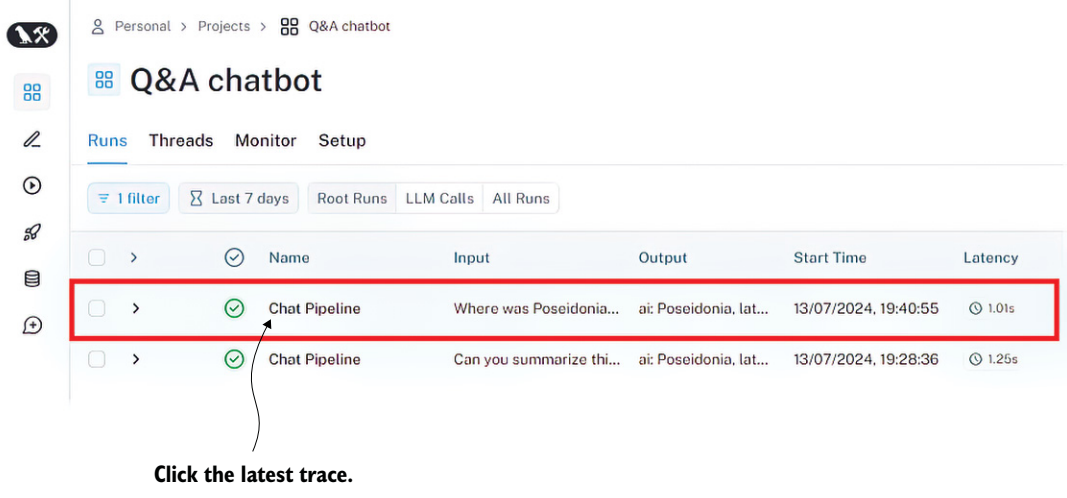


Figure 7.7 LangSmith high-level trace. Click the latest trace to get high-level details of the chain execution.

The current view shows all the traces associated with your Q&A chatbot project (a *project*, in LangSmith's terminology, is the collection of all the traces associated with an application or a part of an application), ordered by execution time, with the most recent at the top. When you click any trace, such as the latest one at the top, you'll get more details. I've split the three panels into two figures for clarity. Figure 7.8 includes the left panel showing all traces of the Q&A chatbot project and the middle panel showing the runs of the selected trace.

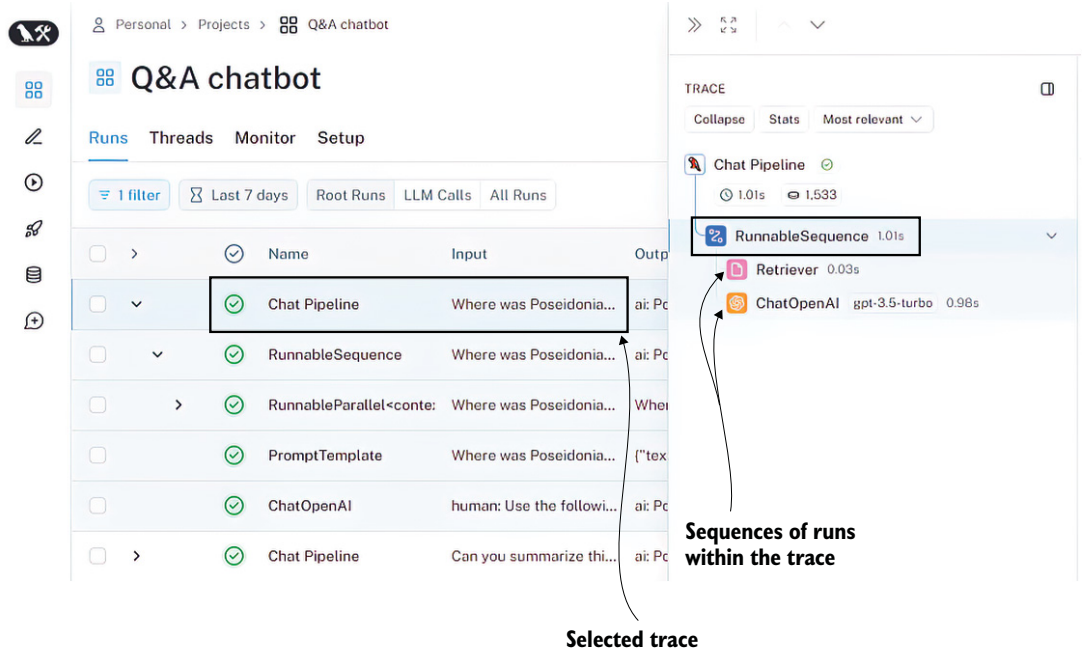


Figure 7.8 LangSmith trace details obtained by clicking one of the traces, for example, the latest one at the top

Figure 7.9 shows the middle panel and the right-hand panel. The middle panel shows the runs of the selected trace, as shown previously, and the right panel shows the input to the selected trace and its output.

Let's go through the three panels shown in the preceding figures that you see when clicking the latest trace on the Q&A chatbot project web page:

- *Left panel*—Shows the list of all traces associated with your project (Q&A chatbot), ordered from the latest at the top to the oldest at the bottom. Each trace can be expanded into its inner steps. Click the latest trace, which is the top one.
- *Middle panel*—Graphically displays the chain execution runs for the selected trace, including the vector-store retriever and the ChatOpenAI LLM client.

Each run represents the execution details of a chain component within the trace. These runs are ordered by execution time and include their duration down to the centisecond.

- *Right panel*—Shows the trace input (the user question) and its output (the synthesized response).

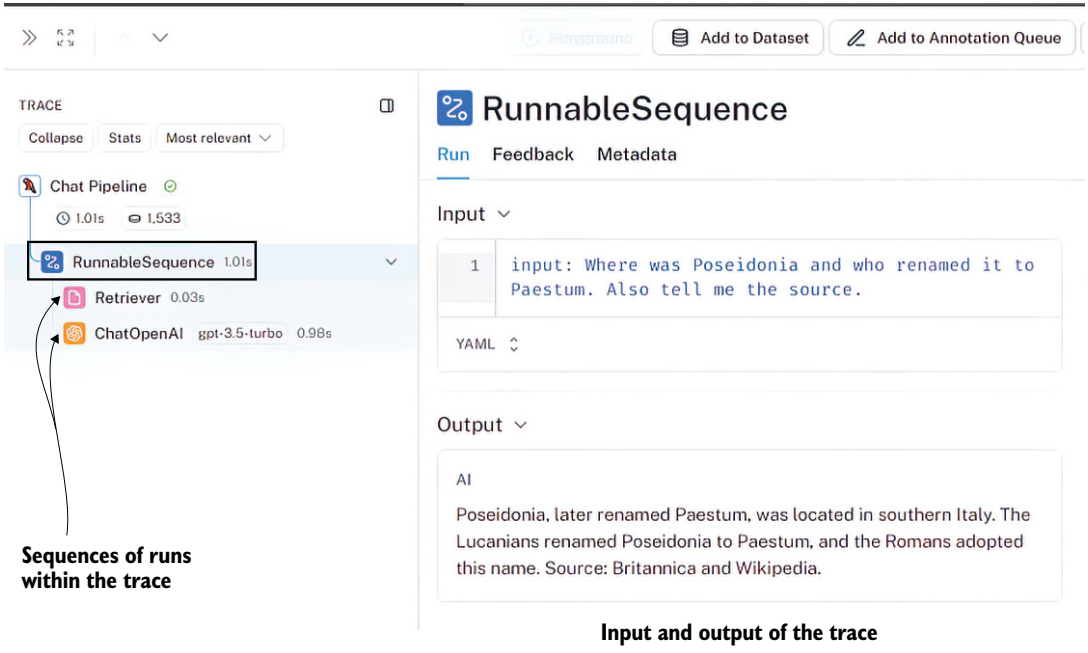


Figure 7.9 Middle and right-hand panels you get when clicking the latest trace

You can get further details by clicking one of the runs in the middle panel, as shown in figure 7.10. For instance, clicking the Retriever run in the middle panel shows its details in the right-hand panel. You'll see the original query in the Input section and the documents retrieved from the vector store in the Output section.

This is just a glimpse of LangSmith's tracing capabilities using a simple chain. I encourage you to examine each chain step in detail. Start by clicking a trace substep in the left panel, and then inspect its associated runs by selecting the one you want to examine in the middle panel. If you're eager to experiment, create a new LangSmith project to trace the execution of the research summarization engine we built in chapter 4. This will help you appreciate a more complex trace that captures a wider range of chain components.

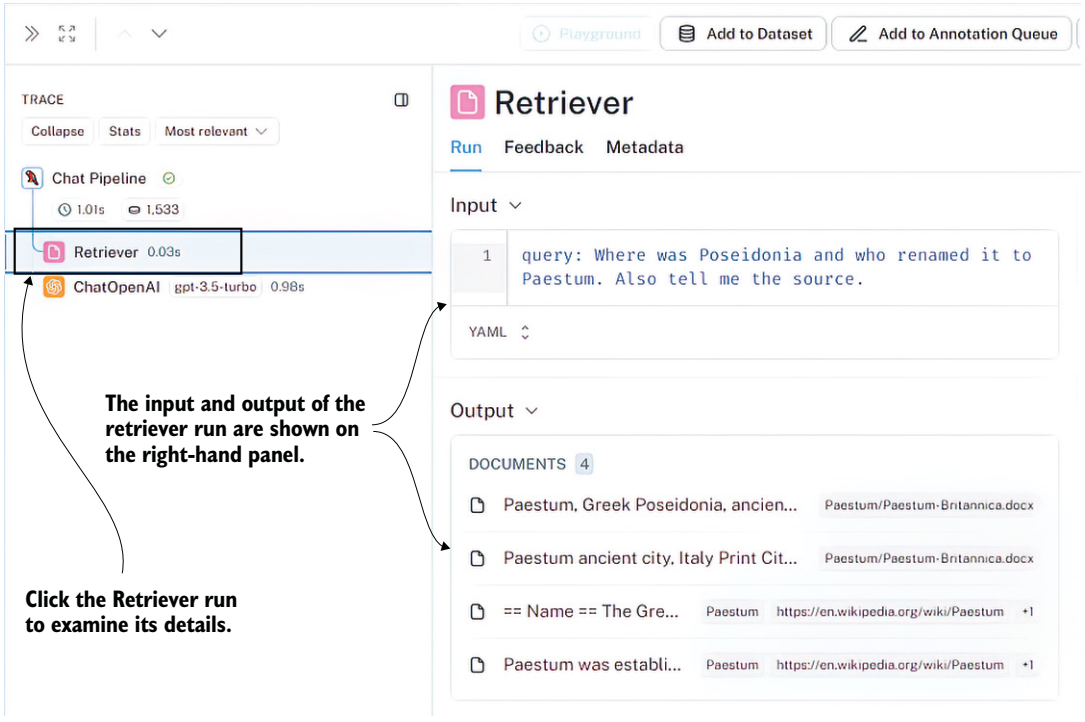


Figure 7.10 Run details

Summary

- LangChain provides abstraction classes for Retrieval-Augmented Generation (RAG) components. This enables swapping of vector stores (ChromaDB to Pinecone), embedding models (OpenAI to Cohere), or LLMs (GPT to Claude) with minimal code changes.
- LangChain's content ingestion phase uses these classes:
  - `BaseLoader`—Imports text sources (PDFs, web pages, databases) into `Document` objects with content and metadata.
  - `BaseDocumentTransformer`—Modifies `Document` objects through operations such as splitting large texts into chunks or extracting entities. Splitters such as `RecursiveCharacterTextSplitter` handle semantic boundaries.
  - `VectorStore`—Stores `Document` object chunks with their embeddings and provides similarity search methods. Implementations include ChromaDB, Pinecone, Facebook AI Similarity Search (FAISS), and Weaviate.
  - `Embeddings`—Abstraction layer for embedding models that converts text into vector representations for similarity matching. Supports OpenAI, Cohere, Hugging Face, and Google models.

- LangChain's Q&A retrieval phase uses these abstractions:
  - `BasePromptTemplate`—Defines prompt structure with placeholders that get filled with retrieved documents and user questions at runtime.
  - `BaseRetriever`—Handles document retrieval from vector stores using similarity scoring and filtering. Supports custom retrieval logic such as metadata filtering or hybrid search.
  - `BaseLanguageModel`—Provides a unified interface to different LLM providers (OpenAI, Anthropic, local models). Switch providers by changing a single configuration parameter.
- RAG chains connect components using LCEL. These include a pass-through for the question, a retriever to fetch documents, a prompt template to combine them, and an LLM to generate the final answer.
- Conversational RAG maintains chat history across turns. This allows follow-up questions such as "What about cost?" to reference previous context without repeating the entire conversation.
- `ChatPromptTemplate.from_messages` structures conversation prompts with system instructions, chat history arrays, and current user messages. History is passed as a list of `HumanMessage` and `AIMessage` objects.
- LangSmith traces RAG execution by recording inputs and outputs at each chain step (retrieval, prompt formatting, LLM generation). This reveals which documents were retrieved and how the LLM used them.
- Store chat history as alternating `HumanMessage` and `AIMessage` objects: `from langchain.schema import HumanMessage, AIMessage`. Append after each turn to maintain conversation context.
- Implement context compression for long conversations by using a sliding window (keeping the last N turns) or summarization (periodically condensing history) to stay within token limits.

Part 4

Advanced RAG

This part takes you past the basics of Retrieval-Augmented Generation (RAG) and into the techniques that make RAG robust enough for real-world use. You'll go beyond simple chunking and keyword retrieval to tackle the deeper issues that cause weak or inconsistent answers—vague queries, poor indexing, and shallow retrieval logic. Instead, you'll learn how to design smarter indexing pipelines, rewrite and route questions more effectively, and refine results with post-retrieval reranking. The goal is to build retrieval systems that consistently surface the right evidence, maintain meaningful context, and empower the model to generate clear, reliable answers.

You'll explore advanced embedding and indexing strategies that account for structure and hierarchy in your data—whether it's text, HTML, Markdown, tables, or even multimodal sources. You'll experiment with methods such as multi-vector retrieval and context expansion to make sure that the model doesn't lose coherence across chunks. On the query side, you'll refine your system's understanding with rewriting, step-back reasoning, Hypothetical Document Embeddings (HyDE), and question decomposition. You'll also learn how to route questions intelligently between different data stores—vector databases, SQL systems, document stores, or knowledge graphs—and how to merge their outputs using techniques such as Reciprocal Rank Fusion (RRF).

By the end of this part, you'll have a deep understanding of what separates a proof-of-concept RAG pipeline from a production-grade knowledge system. You'll walk away with a toolkit of strategies for indexing, retrieval, and reranking that keeps your AI applications accurate, grounded, and scalable—even as your data, users, and complexity grow.

8 Advanced indexing

This chapter covers

- Using advanced RAG techniques
- Selecting optimal chunk splitting strategies
- Using multiple embeddings to enhance coarse chunk retrieval
- Expanding granular chunks to add context during retrieval
- Indexing strategies for semi-structured and multimodal content

In chapter 7, you explored the fundamentals of the Retrieval-Augmented Generation (RAG) architecture—a core pattern for building LLM-powered applications. To keep things simple, we worked with a stripped-down version. That minimal setup is useful for learning, but it often leads to disappointing results in practice: inaccurate answers, overlooked data, or weak use of context, even when the vector store contains exactly what you need. These issues usually stem from vague queries, suboptimal indexing, or failing to use metadata effectively.

This chapter focuses on how to overcome those challenges. Building robust LLM applications with LangChain is less about wiring components together and

more about refining the design—iterating on retrieval strategies, experimenting with prompts, and applying advanced RAG techniques. True proficiency comes from mastering these refinements.

We'll begin with advanced indexing strategies, such as creating multiple embeddings for larger text chunks in the vector database. This approach improves retrieval precision and ensures richer, more accurate context for response generation.

8.1 Improving RAG accuracy

To boost the accuracy of RAG, it's important to examine each step in both the content ingestion and the question-answering (Q&A) workflows. Every stage can introduce challenges—but each also presents opportunities for improvement. Let's begin with the content ingestion stage.

8.1.1 Content ingestion stage

Retrieval accuracy can be improved through an optimized content ingestion process that aligns with the specific features of each content store. Relying only on basic indexing can reduce indexing depth and weaken retrieval performance. Figure 8.1 shows two key areas for improvement in the ingestion stage: refining embedding calculations and optimizing how embeddings are linked to related text chunks in the vector store.

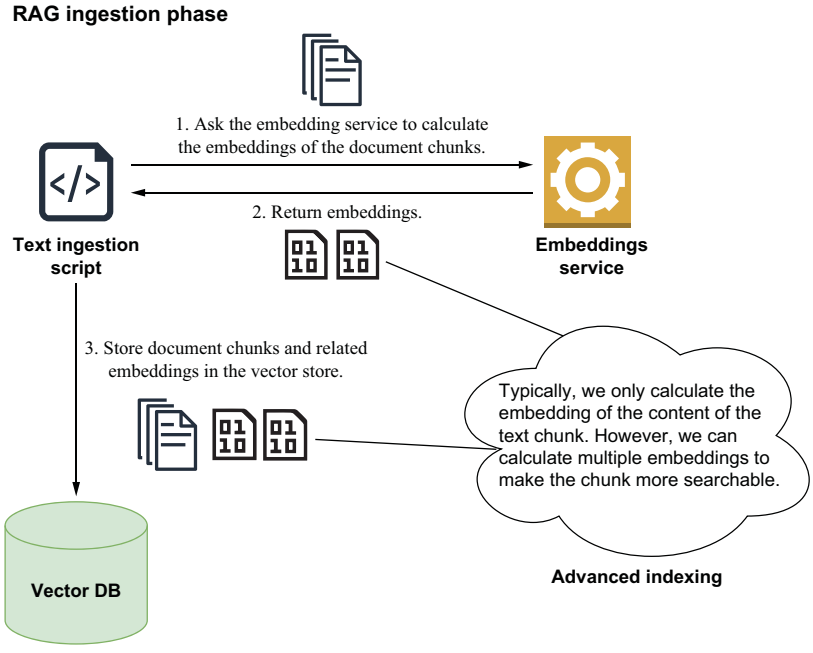


Figure 8.1 Common accuracy issues in the ingestion stage of a simple RAG architecture are often due to inadequate indexing that only uses basic embeddings for each text chunk. Advanced indexing techniques involve generating multiple embeddings for each chunk, enhancing searchability.

Even if a question is clear, retrieval can fail with overly simple indexing strategies. In vector indexing, chunk size and overlap length are crucial: smaller chunks may work well for precise questions but fail with broader queries, while larger chunks may lack detail for specific questions. To address this, you can add additional embeddings based on distinct chunk features, as illustrated in figure 8.1. This multifaceted approach helps make each text chunk more adaptable and searchable. We'll explore many advanced indexing techniques in this chapter.

8.1.2 Question-answering stage

The effectiveness of the question-answering stage depends largely on how accurately the system interprets and processes user queries. Many potential issues, highlighted in figure 8.2, can disrupt this process.

Walking through the workflow of the Q&A stage of the RAG architecture, shown in the previous figure, you'll encounter several pitfalls that can lead to suboptimal answers. Each issue has a targeted solution, as described here and summarized in table 8.1 that follows:

- *Poor question formulation*—If a user's question is unclear, the vector store may return weak context, leading to poor results. The LLM struggles when working with unclear queries and subpar context. A fix is to rephrase the question into a clearer, more detailed form before passing it to the retrieval system.
- *Ineffective question for retrieval*—Using the original question for both retrieval and generation can fail, especially when the query is broad or abstract. Broad questions may not pinpoint relevant content, resulting in less accurate context. You can address this by breaking down broad questions into specific subquestions to retrieve more precise information.
- *Limited data relevance in the content store*—Most RAG systems rely only on vector stores, but adding structured data sources, such as relational databases, tables, or graph databases, can improve results. Route queries to the appropriate content store based on the type of data needed to enhance answer accuracy.
- *Limited querying capabilities against structured data*—While vector stores and LLMs excel with natural language, relational and graph databases don't process it directly, creating a barrier to using structured content. Use an LLM to generate structured queries (e.g., SQL) tailored to each data source to overcome this.
- *Irrelevant search results fed to the LLM*—Even with clearer questions and better indexing, irrelevant data can sometimes slip through, adding noise to the answer. Reduce this by applying filtering or postprocessing steps to keep only the most relevant results.
- *Insufficient improvement in answer accuracy*—Sometimes fixing individual issues doesn't yield expected improvements. In these cases, boost accuracy by combining multiple techniques—such as advanced indexing, question transformations, and multi-store routing—into an ensemble strategy that maximizes precision.

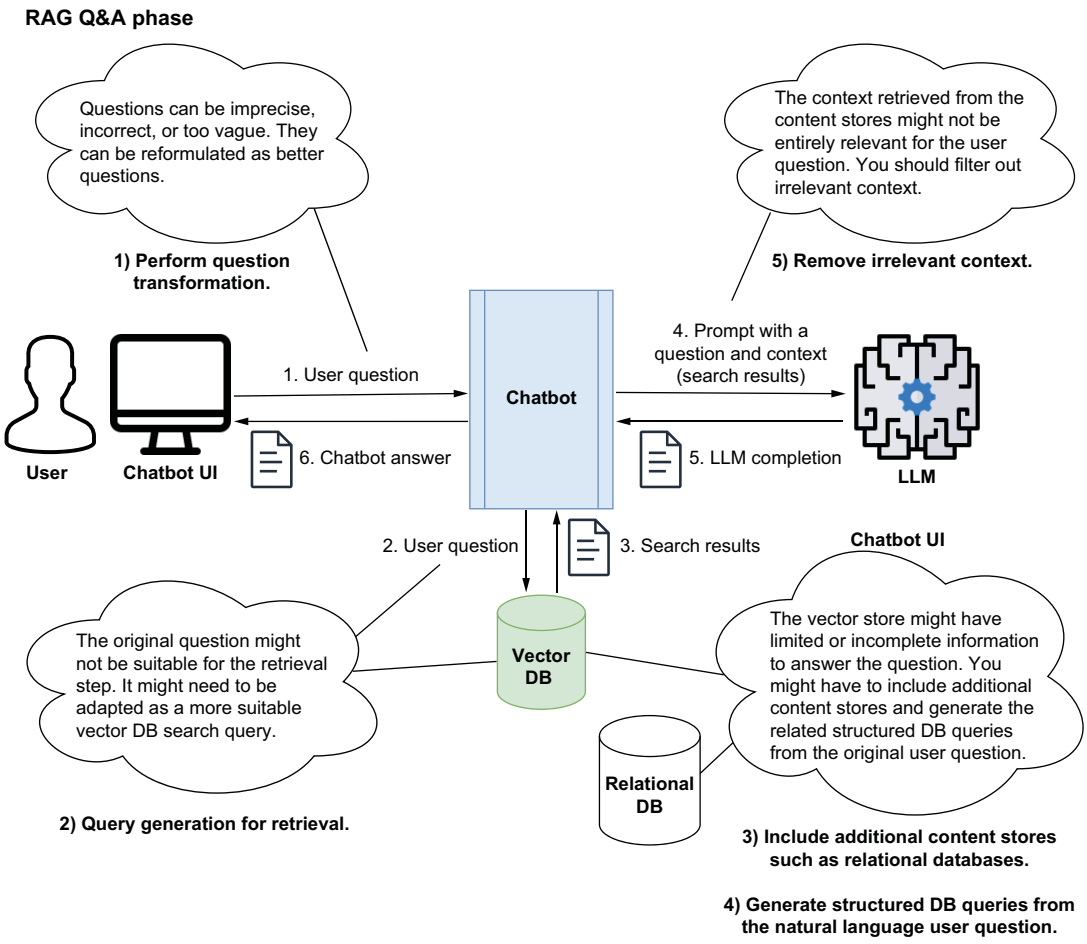


Figure 8.2 Common issues in the Q&A stage of the naive RAG architecture and their solutions: (1) Handle poorly formulated questions with question transformation, (2) enhance retrieval accuracy by transforming the original user questions into more suitable vector DB search queries, (3) include relevant data sources by adding structured data content stores, such as relational DBs, (4) generate DB queries for structured data content stores, (5) and filter out irrelevant context retrieved from the content stores.

Table 8.1 Common issues in naive RAG architecture and recommended solutions

Issue	Solution
Retrieval returning the wrong content chunks	Advanced document indexing techniques
Poor question formulation	Question transformations
Ineffective question for retrieval	Question transformations
Limited data relevance in content store	Routing to multiple content stores
Limited querying capabilities for structured data	Content store query generation
Irrelevant retrieved results fed to LLM	Retrieval postprocessing

This chapter, along with the next two, will dive into each problem and its solution in detail. Next, let's turn to advanced document indexing.

8.2 Advanced document indexing

For an LLM to generate high-quality answers, the relevance and accuracy of the text chunks retrieved from the vector store (or any document store) are critical. The quality of these chunks depends on several key factors:

- *Splitting strategy*—The granularity of document chunks impacts retrieval accuracy. Smaller chunks yield more precise results for specific queries but lack broader context. Larger chunks offer richer context but may miss fine details. Choosing the right split size is essential and can be optimized using various techniques. Additional factors—such as chunk overlap and document hierarchy—also play a crucial role in balancing context and relevance.
- *Embedding strategy*—How you index each chunk is equally important. You can use embeddings, metadata, or a combination. Advanced strategies use multiple indexes—such as child document embeddings, summaries, or hypothetical questions associated with a chunk—to capture both fine-grained details and broader context.
- *Sentence expansion*—One method to deliver larger chunks without losing detail is to expand smaller chunks by including surrounding sentences during retrieval. This approach provides additional context without sacrificing specificity.
- *Indexing structured and semi-structured data*—Retrieving structured data (e.g., database tables or multimedia content) using unstructured queries requires specialized techniques. This can include generating embeddings for database rows, images, or even audio files.

This chapter will cover each of these methods in detail. We'll get started with strategies for optimal document splitting.

8.3 Splitting strategy

During the RAG ingestion phase, documents are split into chunks before being stored in a vector database or document store. Each chunk is indexed using embeddings and sometimes metadata (e.g., by tagging it with relevant keywords). Vector similarity searches rely on the embeddings index, while metadata searches use the keyword index.

The easiest way to improve the relevance of document chunk retrieval is to choose the right document splitting strategy for your use case. Ideally, the document store should return all relevant chunks that provide the LLM with enough context to generate accurate answers. The size of these chunks plays a critical role in retrieval performance, as shown in figure 8.3.

Smaller, more granular chunks are better suited for answering detailed questions because they focus on specific topics. However, they contain less surrounding context, which can reduce effectiveness for broader queries. In contrast, larger chunks are more effective for general questions because they provide more context but lose focus on fine-grained details.

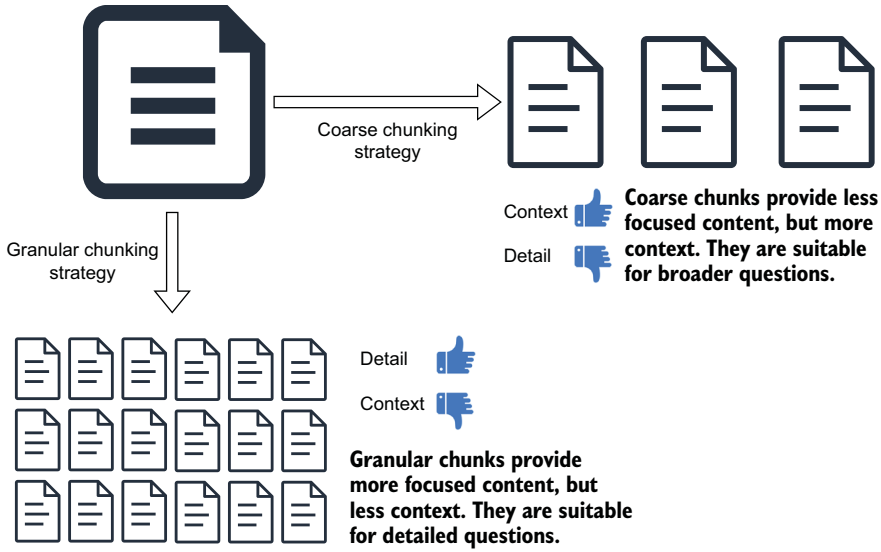


Figure 8.3 Impact of chunk size on answer accuracy. Coarse chunks provide more context but less focus, and they are suitable for broader questions. Granular chunks provide less context but more focus, and they are suitable for more detailed questions.

The challenge is to balance chunk size based on expected question types. Small chunks work well when queries are precise, as the vector representation of the question will match closely with those of the relevant chunks. But for broader questions, small chunks might miss context, making retrieval less accurate. Larger chunks help cover broader topics but at the cost of losing detailed semantic information. The added context in these larger chunks, however, can be valuable during answer generation, as it provides the LLM with more background data to work with.

Finding the right balance between granular and coarse chunks depends on your use case. Ask yourself this: Are the questions likely to be detailed or broad? The chunk size should match the expected query type.

8.3.1 Splitting strategies

The size of the chunks isn't the only factor. You also need to decide *how* to split the document. There are two main approaches:

- *Splitting by document hierarchy*—This approach respects the natural structure of the document (e.g., chapters, sections, paragraphs). It works well when the document is organized by topics, as chunks represent coherent subtopics. Tools such as `HTMLHeaderTextSplitter` and `MarkdownHeaderTextSplitter` in LangChain target specific document types and maintain semantic accuracy. However, chunk sizes can vary greatly.
- *Splitting by absolute size*—You can define chunk size by characters, tokens, sentences, or words. This results in more consistent chunk sizes, but context might

be lost if chunks split mid-sentence. `CharacterTextSplitter` and its variants support different granularity levels, but you'll need to test for optimal size. Evaluating a range of fixed sizes is necessary to find what works best for your use case.

8.3.2 Factors to consider

For each splitting strategy, keep the following in mind:

- *Document type*—If you're working with mixed content (e.g., text, tables, images), maintaining related content within the same chunk is crucial. In this case, splitting by document hierarchy is more effective than a fixed-size approach.
- *Search type*—If you're planning to use metadata search, keywords can be refined depending on the chunk granularity. You can also attach a mix of broad and detailed tags to each chunk to increase retrieval flexibility.

8.3.3 Choosing the right strategy

Selecting the best strategy often requires a mix of trial and error, but experience will eventually guide you to the right balance between document hierarchy and absolute size. Table 8.2 summarizes the pros and cons of each strategy and provides relevant LangChain classes to implement them.

Table 8.2 Splitting strategies, pros and cons, and related LangChain classes

Splitting strategy	Pros	Cons	LangChain classes
Document hierarchy	More accurate semantic meaning	Chunk size varies significantly.	`HTMLHeaderTextSplitter`, `MarkdownHeaderTextSplitter`
By size: number of tokens	Consistent chunk size	Incomplete sentences appear at boundaries.	`TokenTextSplitter`
By size: number of characters	Consistent chunk size	Incomplete sentences may reduce semantic value.	`CharacterTextSplitter`
By sentence, paragraph, or word	Retains semantic meaning in most cases	Small chunks may lack enough context.	`RecursiveCharacterTextSplitter`

Each method has its use case, and the choice depends on your specific needs and document structure. In the following sections, I'll cover these strategies in detail, starting with document hierarchy splitting.

8.3.4 Splitting by HTML header

In this section, I'll show you how to split documents using the `HTMLHeaderTextSplitter` class on various online documents from Wikivoyage (www.wikivoyage.org), a travel-focused site from the same group as Wikipedia. We'll explore how different levels of splitting granularity affect the accuracy of responses.

First, set up your environment by creating a new folder and virtual environment, and then install the required packages:

```
C:\Github\building-llm-applications\ch08> python -m venv env_ch08
C:\Github\building-llm-applications\ch08> env_ch08\Scripts\activate
(env_ch08) C:\Github\building-llm-applications\ch08>
➥pip install -r requirements.txt
```

Next, start a new Jupyter Notebook, or open the existing one in the cloned repository:

```
(env_ch08) C:\Github\building-llm-applications\ch08> jupyter notebook
```

Save the notebook as `08-advanced_indexing.ipynb`, and import the required libraries:

```
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings
import getpass

OPENAI_API_KEY = getpass.getpass('Enter your OPENAI_API_KEY')
```

SETTING UP CHROMADB COLLECTIONS

Now create a ChromaDB collection to store the more granular chunks:

```
cornwall_granular_collection = Chroma(
    collection_name="cornwall_granular",
    embedding_function=OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY),
)

cornwall_granular_collection.reset_collection()
```

Creates a ChromaDB collection

Resets the collection if it already exists

This will initialize a new Chroma collection called `cornwall_granular_collection`. If the collection already exists, it will be reset to start fresh. Next, set up a second collection for coarser chunks:

```
cornwall_coarse_collection = Chroma(
    collection_name="cornwall_coarse",
    embedding_function=OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY),
)

cornwall_coarse_collection.reset_collection()
```

Creates a ChromaDB collection

Resets the collection in case it already exists

LOADING HTML CONTENT WITH ASYNCHTMLLOADER

The final step is to ingest some content about Cornwall, a region in the UK known for its stunning seaside resorts, using an HTML loader:

```
from langchain_community.document_loaders import AsyncHtmlLoader
destination_url = "https://en.wikivoyage.org/wiki/Cornwall"
html_loader = AsyncHtmlLoader(destination_url)
docs = html_loader.load()
```

This snippet fetches the Cornwall page content, which we'll use to create both granular and coarse chunks. In the following steps, I'll show how to split the content based on HTML headers and analyze the impact on retrieval accuracy.

Splitting content into granular chunks using HTMLSectionSplitter

Before storing the content from the `docs` object into the vector database, decide on a splitting strategy. Because this content is from an HTML page, you can split it by `H1` and `H2` tags, which separate the content into sections. This will generate more granular chunks, as shown in the following listing.

Listing 8.1 Splitting content with the `HTMLSectionSplitter`

```
from langchain_text_splitters import HTMLSectionSplitter

headers_to_split_on = [("h1", "Header 1"), ("h2", "Header 2")]
html_section_splitter = HTMLSectionSplitter(
    headers_to_split_on=headers_to_split_on)

def split_docs_into_granular_chunks(docs):
    all_chunks = []
    for doc in docs:
        html_string = doc.page_content
        temp_chunks = html_section_splitter.split_text(
            html_string)
        all_chunks.extend(temp_chunks)

    return all_chunks
```

Extracts the HTML text from the document

Splits by H1 and H2 sections

You can now generate the granular chunks:

```
granular_chunks = split_docs_into_granular_chunks(docs)
```

Now insert the granular chunks into the Chroma collection:

```
cornwall_granular_collection.add_documents(documents=granular_chunks)
```

Searching granular chunks

You can now run a search for specific content within the granular chunks:

```
results = corwnall_granular_collection.similarity_search(
    query="Events or festivals in Cornwall",k=3)
for doc in results:
    print(doc)
```

Splitting content into coarse chunks using RecursiveCharacterTextSplitter

For larger, coarser chunks, use the `RecursiveCharacterTextSplitter` class. Start by creating the necessary objects:

```
from langchain_community.document_transformers import Html2TextTransformer
from langchain_text_splitters import RecursiveCharacterTextSplitter

html2text_transformer = Html2TextTransformer()
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=3000, chunk_overlap=300
)
```

Next, define a function to split the content into coarse chunks:

```
def split_docs_into_coarse_chunks(docs):
    text_docs = html2text_transformer.transform_documents(
        docs)
    coarse_chunks = text_splitter.split_documents(
        text_docs)

    return coarse_chunks
```

Converts HTML to plain text

Splits text into larger chunks

Then, generate the coarse chunks:

```
coarse_chunks = split_docs_into_coarse_chunks(docs)
```

Insert these chunks into the corresponding Chroma collection:

```
cornwall_coarse_collection.add_documents(documents=coarse_chunks)
```

SEARCHING COARSE CHUNKS

You can now search for more general content within the coarse chunks:

```
results = corwnall_coarse_collection.similarity_search(
    query="Events or festivals in Cornwall",k=3)
for doc in results:
    print(doc)
```

INGESTING CONTENT FROM MULTIPLE URLS

To make the searches more comprehensive, load additional content into the collections. Listing 8.2 shows how to set up new granular and coarse collections for various UK destinations and ingest the related content chunks. If you'd like to minimize processing costs, consider reducing the size of the `uk_destinations` list.

Listing 8.2 Creating collections for multiple UK destinations

```
uk_granular_collection = Chroma(
    collection_name="uk_granular",
    embedding_function=OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY),
)
uk_granular_collection.reset_collection()

uk_coarse_collection = Chroma(
    collection_name="uk_coarse",
    embedding_function=OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY),
)
uk_coarse_collection.reset_collection()

uk_destinations = [
    "Cornwall", "North_Cornwall", "South_Cornwall", "West_Cornwall",
    "Tintagel", "Bodmin", "Wadebridge", "Penzance", "Newquay",
    "St_Ives", "Port_Isaac", "Looe", "Polperro", "Porthleven",
    "East_Sussex", "Brighton", "Battle", "Hastings_(England)",
    "Rye_(England)", "Seaford", "Ashdown_Forest"
]
```

Creates a ChromaDB collection

Resets the collection if it already exists

Resets the collection if it already exists

```
wikivoyage_root_url = "https://en.wikivoyage.org/wiki"

uk_destination_urls = [f'{wikivoyage_root_url}/{d}
➥' for d in uk_destinations]

for destination_url in uk_destination_urls:
    html_loader = AsyncHtmlLoader(destination_url)
    docs = html_loader.load()

    granular_chunks = split_docs_into_granular_chunks(docs)
    uk_granular_collection.add_documents(documents=granular_chunks)

    coarse_chunks = split_docs_into_coarse_chunks(docs)
    uk_coarse_collection.add_documents(documents=coarse_chunks)
```

Loader for one destination

Documents of one destination

You can now perform both granular and coarse searches:

```
granular_results = uk_granular_collection.similarity_search(
    query="Events or festivals in East Sussex",k=4)
for doc in granular_results:
    print(doc)

coarse_results = uk_coarse_collection.similarity_search(
    query="Events or festivals in East Sussex",k=4)
for doc in coarse_results:
    print(doc)
```

Try experimenting with different queries, like `"Beaches in Cornwall"`, to see how the results vary between granular and coarse chunks. This approach will help you fine-tune the balance between detailed and general content retrieval.

8.4 Embedding strategy

Previously, I discussed how keyword searches are more flexible than vector searches because you can tag a document with multiple keywords. The same idea can be applied to embeddings—you can store multiple vectors per document, which increases the flexibility and accuracy of vector searches. I'll walk through various multi-vector strategies in the following sections, mainly focusing on how to use LangChain's `MultiVectorRetriever`.

The key to these strategies is a two-layer chunk structure. The top layer includes *synthesis chunks*—the chunks fed into the LLM to generate answers. The lower layer consists of *retrieval chunks*, smaller segments that create precise embeddings for retrieving the synthesis chunks. I suggest testing all multi-vector retriever techniques for your use case, as they usually yield similar performance improvements. However, because the structure of your text may favor one technique over another, experimenting with each will help you identify the best fit.

8.4.1 Embedding child chunks with ParentDocumentRetriever

A common challenge with chunk size is balancing between context and detail. Large chunks work for broad questions but struggle with detailed queries. Small chunks, while supporting detailed queries, often lack the context needed for generating comprehensive answers. This creates a tradeoff—if chunks are too small, the response might be incomplete, but if they're too large, the retrieval may be less precise.

To solve this problem, split the document into larger *parent* chunks, and create smaller *child* chunks within each parent. Use the child chunks solely for generating more granular embeddings, which are then stored against the parent chunk. This hybrid approach allows each document to have embeddings for both broad and detailed queries, as illustrated in figure 8.4.

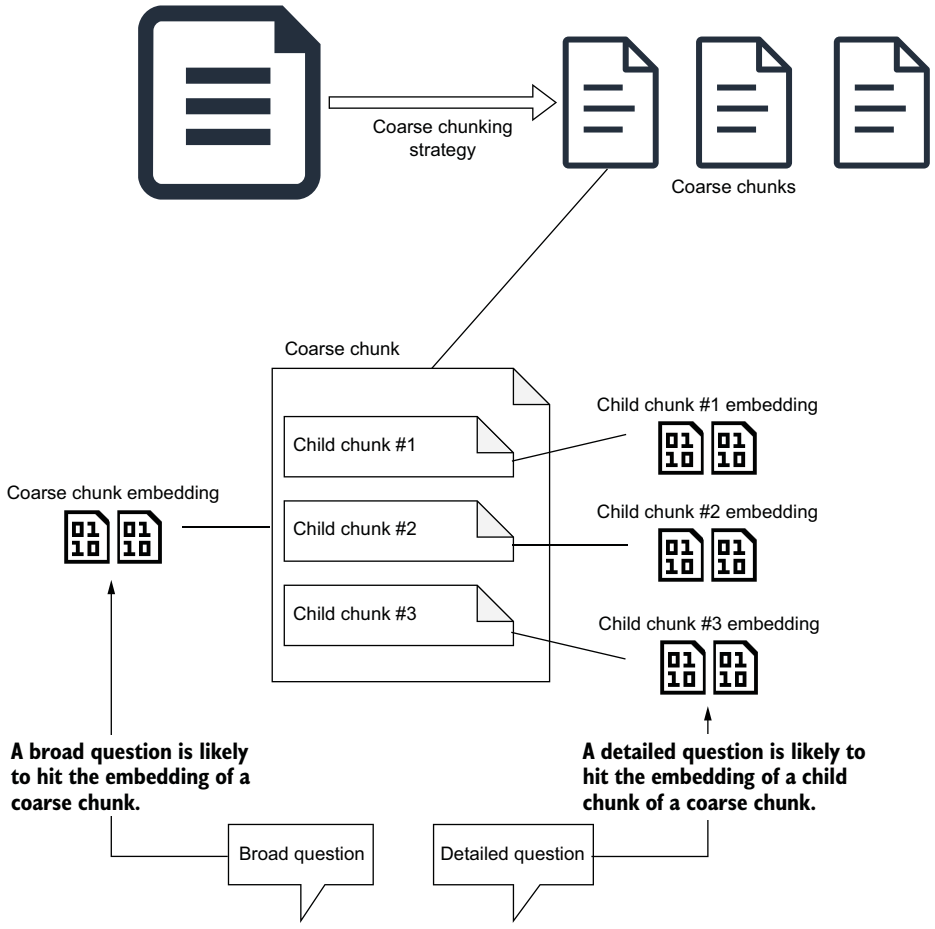


Figure 8.4 Child chunk embeddings. A coarse document chunk is indexed with its own embedding and the embeddings generated from its smaller child chunks, allowing it to match effectively with both broad and detailed queries.

The advantage of this approach is that when a broad query is made, the parent chunk is retrieved, providing rich context. For more detailed queries, the child embeddings ensure precise matching, which still leads to the retrieval of the context-rich parent chunk. This structure helps the LLM generate more accurate and contextually relevant answers.

I'll show you how to implement the technique using `ParentDocumentRetriever`. Start by importing the necessary libraries:

```
from langchain_classic.retrievers import ParentDocumentRetriever
from langchain_classic.storage import InMemoryStore
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings
from langchain_community.document_loaders import AsyncHtmlLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
```

SETTING UP THE PARENTDOCUMENTRETRIEVER

This approach uses two types of stores: a *document store*, which holds the complete parent documents, and a *vector store*, which contains the smaller chunks and their corresponding embeddings. Each chunk maintains a reference to its parent document. The approach begins by splitting content into large, coarse chunks for synthesis and then further dividing each into smaller child chunks for retrieval. Listing 8.3 demonstrates how to configure the splitters and set up the retriever. As shown, documents are stored in an `InMemoryStore`—a general-purpose, in-memory key–value store designed to hold serializable Python objects such as strings, lists, and dictionaries. It's particularly useful for caching and intermediate data storage.

Listing 8.3 Parent and child splitters for coarse and granular chunks

```
parent_splitter = RecursiveCharacterTextSplitter(
    chunk_size=3000)
child_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500)

child_chunks_collection = Chroma(
    collection_name="uk_child_chunks",
    embedding_function=OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY),
)

child_chunks_collection.reset_collection()

doc_store = InMemoryStore()

parent_doc_retriever = ParentDocumentRetriever(
    vectorstore=child_chunks_collection,
    docstore=doc_store,
    child_splitter=child_splitter,
    parent_splitter=parent_splitter
)
```

Splitter to generate parent coarse chunks from original documents (parsed from web pages)

Splitter to generate child granular chunks from parent

Vector store collection to host child granular chunks

Makes sure the collection is empty

Document store to host parent coarse chunks

Retriever to link parent coarse chunks to child granular chunks

INGESTING CONTENT INTO DOCUMENT AND VECTOR STORES

Now generate both coarse and granular chunks using the configured splitters, and add them to the respective stores. This happens in each `parent_doc_retriever.add_documents()` call, for the corresponding destination URL:

```
for destination_url in uk_destination_urls:
    html_loader = AsyncHtmlLoader(destination_url)
    html_docs =  html_loader.load()
    text_docs = html2text_transformer.transform_documents(
       html_docs)

    print(f'Ingesting {destination_url}')
    parent_doc_retriever.add_documents(
       text_docs, ids=None)
```

Loader for the destination web page

HTML documents of one destination

Transforms HTML documents into clean text documents

Ingests coarse chunks into the document store and granular chunks into the vector store

VERIFYING THE IN-MEMORY DOCUMENT STORE

You can check if the coarse chunks have been correctly stored via the following:

```
list(doc_store.yield_keys())
```

PERFORMING A SEARCH ON GRANULAR INFORMATION

Now perform a search on the child chunks using the `ParentDocumentRetriever`:

```
retrieved_docs = parent_doc_retriever.invoke("Cornwall Ranger")
```

The first document retrieved (`retrieved_docs[0]`) will contain rich contextual information:

```
Document (metadata={'source': 'https://en.wikivoyage.org/wiki/
South_Cornwall', 'title': 'South Cornwall - Travel guide at Wikivoyage',
'language': 'en'}, page_content="Trains from London take about 3 hr 20 min to
Plymouth.\n\n### By car\n\n[edit]\n\nCornwall can be accessed by road via the
A30 which runs from the end of the M5\nat Exeter, all the way through the
heart of Devon and Cornwall down to Land's\nEnd. It is a grade-separated
expressway as far as Carland Cross near Truro\n(the expressway is expected to
be open as far as Camborne (between Redruth and\nHayle) by March 2024). You
can also get to Cornwall via the A38, crossing the\nRiver Tamar at Plymouth
via the Tamar Bridge, which levies a toll on eastbound\nvehicles. On summer
Saturdays and during bank holiday weekends roads to\nCornwall are usually
busy.\n\n## Get around\n\n[edit]\n\n### By bus\n\n[edit]\n\nThanks to
Transport for Cornwall, all bus tickets are interchangeable across\nthe
different companies. The **Cornwall All Day ticket** allows unlimited\ntravel
for a calendar day. As of 2023, fares are £5 for adults and £4 for\nunder-
19s. Payment … [SHORTENED] …
```

COMPARING WITH DIRECT SEMANTIC SEARCH ON CHILD CHUNKS

Now compare the results by directly searching only the child chunks:

```
child_docs_only = child_chunks_collection
➥.similarity_search("Cornwall Ranger")
```

The first result is much shorter and lacks context:

```
Document(metadata={'doc_id': '34645d23-ed05-4a53-b3af-c8ab21e3f513',
'language': 'en', 'source': 'https://en.wikivoyage.org/wiki/South_Cornwall',
'title': 'South Cornwall - Travel guide at Wikivoyage'},
page_content='The **Cornwall Ranger** ticket allows unlimited train travel in
Cornwall and\nPlymouth for a calendar day. As of 2023, this costs £14 for
adults and £7 for\nunder-16s.\n\n## See\n\n[edit]\n\nThe **Eden Project** ,
near St Austell...')
```

The result we've obtained with the `ParentDocumentRetriever` is particularly useful when used as context for LLM synthesis, as it provides broader details around the specific information. Next, I'll introduce another technique that uses embedding strategies to further optimize RAG retrieval accuracy.

8.4.2 Embedding child chunks with MultiVectorRetriever

An alternative method for embedding child chunks and linking them to the larger parent chunks used in synthesis is to use the `MultiVectorRetriever`. Begin by importing the necessary libraries, including `InMemoryByteStore`, which is specifically designed for storing binary data. In this store, keys are strings and values are bytes, making it ideal for use cases involving embeddings, models, or files where raw byte storage is preferred or required:

```
from langchain_classic.retrievers.multi_vector import MultiVectorRetriever
from langchain_classic.storage import InMemoryByteStore
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings
from langchain_community.document_loaders import AsyncHtmlLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
import uuid
```

SETTING UP THE MULTIVECTORRETRIEVER

You can use a similar approach as with the `ParentDocumentRetriever` by defining parent and child splitters and then injecting them into `MultiVectorRetriever`, as shown in the following listing.

Listing 8.4 Parent and child splitters for `MultiVectorRetriever`

```
parent_splitter = RecursiveCharacterTextSplitter(
    chunk_size=3000)
child_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500)

child_chunks_collection = Chroma(
    collection_name="uk_child_chunks",
    embedding_function=OpenAIEmbeddings(
        openai_api_key=OPENAI_API_KEY),
)

child_chunks_collection.reset_collection()
```

Splitter to generate parent coarse chunks from original documents (parsed from web pages)

Splitter to generate child granular chunks from parent coarse chunks

Vector store collection to host child granular chunks

Makes sure the collection is empty

```
doc_byte_store = InMemoryByteStore()
doc_key = "doc_id"

multi_vector_retriever = MultiVectorRetriever(
    vectorstore=child_chunks_collection,
    byte_store=doc_byte_store
)
```

Document store to host parent coarse chunks

Retriever to link parent coarse chunks to child granular chunks

INGESTING CONTENT INTO DOCUMENT AND VECTOR STORES

The next step is to load and ingest content into the `MultiVectorRetriever`, as shown in listing 8.5. While ingestion may feel slower, this is expected due to the added complexity of managing multiple vector representations.

Listing 8.5 Ingesting content into document and vector stores

```
for destination_url in uk_destination_urls:
    html_loader = AsyncHtmlLoader(destination_url)
    html_docs =  html_loader.load()
    text_docs = html2text_transformer.transform_documents(
        html_docs)

    coarse_chunks = parent_splitter.split_documents(
        text_docs)

    coarse_chunks_ids = [str(uuid.uuid4()) for _ in coarse_chunks]
    all_granular_chunks = []
    for i, coarse_chunk in enumerate(
        coarse_chunks):

        coarse_chunk_id = coarse_chunks_ids[i]

        granular_chunks = child_splitter.split_documents(
            [coarse_chunk])

        for granular_chunk in granular_chunks:
            granular_chunk.metadata[doc_key] = \
               coarse_chunk_id

        all_granular_chunks.extend(granular_chunks)

    print(f'Ingesting {destination_url}')
    multi_vector_retriever.vectorstore.add_documents(
        all_granular_chunks)
    multi_vector_retriever.docstore.mset(
        list(zip(coarse_chunks_ids, coarse_chunks)))
```

Loader for one destination

Documents of one destination

Transforms HTML documents into clean text documents

Splits the destination content into parent coarse chunks

Iterates over the parent coarse chunks

Creates child granular chunks from each parent coarse chunk

Links each child granular chunk to its parent coarse chunk

Ingests the child granular chunks into the vector store

Ingests the parent coarse chunks into the document store

PERFORMING A SEARCH ON GRANULAR INFORMATION

Now perform a search using `MultiVectorRetriever`, just like you did with the `ParentDocumentRetriever`:

```
retrieved_docs = multi_vector_retriever.invoke(
    "Cornwall Ranger")
```

Printing the first result shows that the retrieved document contains rich, detailed information:

```
Document(metadata={'source': 'https://en.wikivoyage.org/wiki/South_Cornwall',
'title': 'South Cornwall - Travel guide at Wikivoyage', 'language': 'en'},
page_content="Trains from London take about 3 hr 20 min to Plymouth.\n\n###
By car\n\nCornwall can be accessed by road via the A30 which runs from the
end of the M5\nat Exeter, all the way through the heart of Devon and
Cornwall...")
```

COMPARING WITH DIRECT SEMANTIC SEARCH ON CHILD CHUNKS

For comparison, run the same search directly on the child chunk collection:

```
child_docs_only =  child_chunks_collection.similarity_search(
    "Cornwall Ranger")
```

The first document retrieved from the child collection (`child_docs_only[0]`) is more concise and lacks the broader context:

```
Document(metadata={'doc_id': '04c7f88e-e090-4057-af5b-ea584e777b3f',
'language': 'en', 'source': 'https://en.wikivoyage.org/wiki/South_Cornwall',
'title': 'South Cornwall - Travel guide at Wikivoyage'},
page_content='The **Cornwall Ranger** ticket allows unlimited train travel in
Cornwall and\nPlymouth for a calendar day. As of 2023, this costs £14 for
adults and £7 for\nunder-16s.\n\n## See\n\nThe **Eden Project** , near St
Austell...')
```

This result is similar to what you observed with the `ParentDocumentRetriever`. The broader parent chunks provide more useful context for synthesis, making them a better fit for complex or detailed queries.

In the next section, I'll cover additional strategies for improving RAG accuracy using advanced embedding techniques. We'll start with embedding summaries.

8.4.3 Embedding document summaries

Embeddings from coarse chunks are often ineffective because they capture too much irrelevant content. A large chunk may include filler text or minor details that dilute the semantic value of the embeddings, making them less focused and less useful.

To address this, you can create a summary of the coarse chunk and generate embeddings from it. These summary embeddings are then stored alongside the original chunk embeddings, as shown in figure 8.5. Because the summary is more concise and relevant, the resulting embeddings are denser and more effective for retrieval, reducing noise and improving search precision.

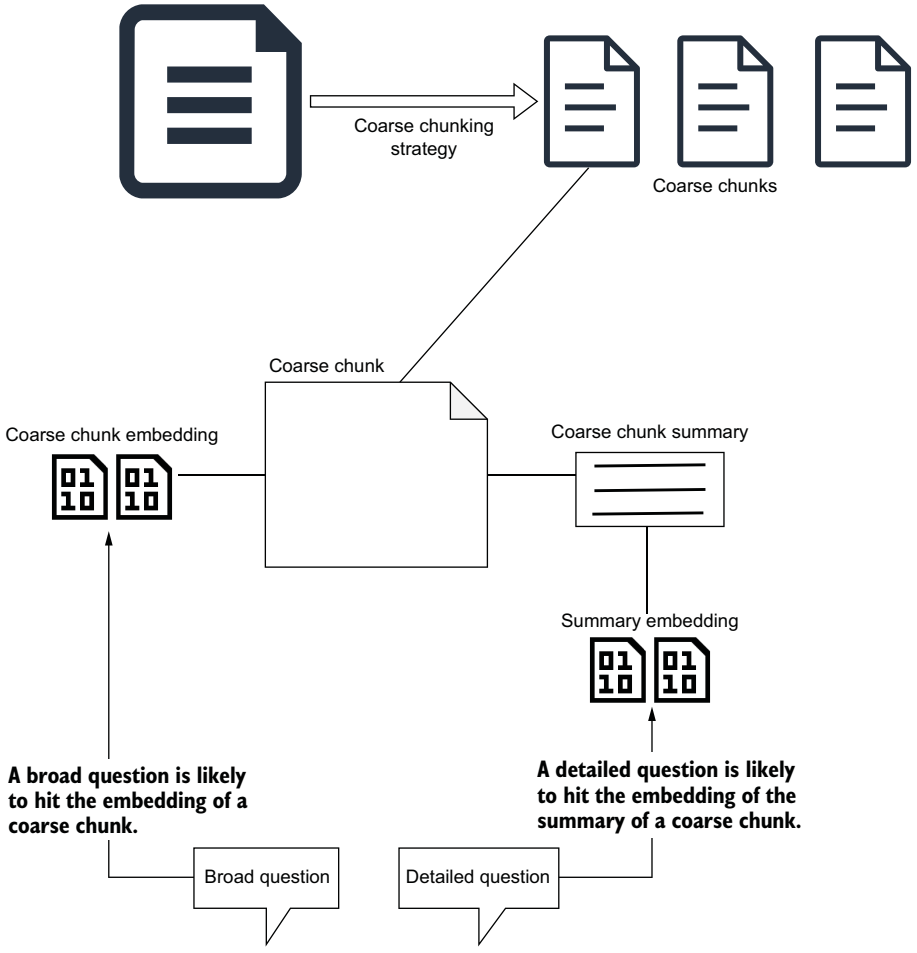


Figure 8.5 Chunk summary embedding. A coarse chunk is indexed with its own embedding and an additional embedding from its summary, allowing for more accurate retrieval when answering detailed questions.

To start, import the required libraries for building the summarization and retrieval setup:

```
from langchain_classic.retrievers.multi_vector import MultiVectorRetriever
from langchain_classic.storage import InMemoryByteStore
from langchain_chroma import Chroma
from langchain_openai import ChatOpenAI
from langchain_openai import OpenAIEmbeddings
from langchain_community.document_loaders import AsyncHtmlLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_core.documents import Document
from langchain_core.output_parsers import StrOutputParser
```

```
from langchain_core.prompts import ChatPromptTemplate
import uuid
```

Setting up the MultiVectorRetriever

First, create a collection to store the summaries and set up the document store (`InMemoryByteStore`). Then, configure the `MultiVectorRetriever` to use these components, as shown in the following listing.

Listing 8.6 Setting up the `MultiVectorRetriever`

```
parent_splitter = RecursiveCharacterTextSplitter(
    chunk_size=3000)

summaries_collection = Chroma(
    collection_name="uk_summaries",
    embedding_function=OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY),
)

summaries_collection.reset_collection()

doc_byte_store = InMemoryByteStore()
doc_key = "doc_id"

multi_vector_retriever = MultiVectorRetriever(
    vectorstore=summaries_collection,
    byte_store=doc_byte_store
)
```

Splitter to generate parent coarse chunks from original documents (parsed from web pages)

Vector store collection to host child granular chunks

Makes sure the collection is empty

Document store to host parent coarse chunks

Retriever to link parent coarse chunks to child granular chunks

Setting Up the summarization chain

Use an LLM to generate summaries of the coarse chunks. Define a summarization chain that extracts the content, prompts the LLM, and parses the response into a usable format:

```
llm = ChatOpenAI(model="gpt-5-nano", openai_api_key=OPENAI_API_KEY)

summarization_chain = (
    {"document": lambda x: x.page_content}
    | ChatPromptTemplate.from_template("Summarize the
    ➥following document:\n\n{document}")
    | llm
    | StrOutputParser())
```

Grabs the text content from the document

Instantiates a prompt asking to generate summary of the provided text

Sends the LLM the instantiated prompt

Extracts the summary text from the response

Ingesting coarse chunks and summaries into stores

Next, load the content, split it into coarse chunks, and generate summaries for those chunks. Then, store the summaries in the document store while storing the corresponding coarse chunks in the vector store, as shown in the following listing.

Listing 8.7 Ingesting coarse chunks and their summaries

```
for destination_url in uk_destination_urls:
    html_loader = AsyncHtmlLoader(destination_url)
    html_docs =  html_loader.load()
    text_docs = html2text_transformer.transform_documents(
        html_docs)

    coarse_chunks = parent_splitter.split_documents(
        text_docs)

    coarse_chunks_ids = [str(uuid.uuid4()) for _ in coarse_chunks]
    all_summaries = []
    for i, coarse_chunk in enumerate(
        coarse_chunks):

        coarse_chunk_id = coarse_chunks_ids[i]

        summary_text =  summarization_chain.invoke(
            coarse_chunk)
        summary_doc = Document(page_content=summary_text,
                               metadata={doc_key: coarse_chunk_id})

        all_summaries.append(summary_doc)

    print(f'Ingesting {destination_url}')
    multi_vector_retriever.vectorstore.add_documents(
        all_summaries)
    multi_vector_retriever.docstore.mset(
        list(zip(coarse_chunks_ids, coarse_chunks)))
```

Loader for one destination

Documents of one destination

Transforms HTML documents into clean text documents

Splits the destination content into coarse chunks

Iterates over the coarse chunks

Generates a summary for the coarse chunk through the summarization chain

Links each summary to its related coarse chunk

Ingests the summaries into the vector store

Ingests the coarse chunks into the document store

When running the code in listing 8.6, you may notice that processing is slower compared to using child embeddings. This slowdown is due to the time required to submit each coarse chunk for summarization to the LLM, which is a more computationally intensive step.

PERFORMING A SEARCH USING THE MULTIVECTORRETRIEVER

Once the ingestion is complete, you can perform a search using the `MultiVectorRetriever`, which now uses the summaries for each travel destination:

```
retrieved_docs = multi_vector_retriever.invoke("Cornwall travel")
```

If you print the first result (`retrieved_docs_only[0]`), you'll see a large chunk similar to those retrieved when using child embeddings. These larger chunks provide more context, making them effective when passed as input to the LLM.

COMPARING WITH DIRECT SEMANTIC SEARCH ON SUMMARIES

For comparison, perform a direct search on the summaries alone:

```
summary_docs_only =  summaries_collection.similarity_search(
    "Cornwall Travel")

print(summary_docs_only[0])
```

The first result from the summary search is concise and lacks broader context:

```
Document(metadata={'doc_id': 'ee55d250-bc53-46ce-9204-8fd2c1a05662'},
page_content="Cornwall is a county located in the southwest of the United
Kingdom, known for its distinctive character, warm climate, and beautiful
coastline. It is popular among holidaymakers due to its rich Celtic heritage,
cultural tourism, and historical connections to arts and mining, which is
recognized by UNESCO. Over 30% of Cornwall is designated as an Area of
Outstanding Natural Beauty (AONB). ...")
```

This result confirms a pattern observed earlier: directly searching against summaries or child chunks retrieves focused but context-limited information, while using a multi-vector approach retrieves broader context that is more useful for synthesis. Next, let's explore one more advanced multi-vector embedding technique in the following section.

8.4.4 Embedding hypothetical questions

When querying a vector store, your natural language question is converted into a vector, and the system calculates its similarity (e.g., cosine distance) to the stored vectors. The documents linked to the closest vectors are then returned. This approach works well if the question is semantically similar to the ideal answer. But often, the wording of the question and the phrasing of the ideal answer may not match closely enough, causing the search to miss relevant documents.

To address this, you can generate hypothetical questions that each chunk is likely to answer and then store the chunk using embeddings derived from these questions, as shown in figure 8.6. This method increases the chances that the stored vectors will align more closely with a user's query, making it more likely to retrieve relevant information, even if the original document's embeddings aren't a perfect match for the question.

To begin, import all the required libraries to set up the `MultiVectorRetriever` with hypothetical question embeddings:

```
from langchain_classic.retrievers.multi_vector import MultiVectorRetriever
from langchain_classic.storage import InMemoryByteStore
from langchain_chroma import Chroma
from langchain_openai import ChatOpenAI
from langchain_openai import OpenAIEmbeddings
from langchain_community.document_loaders import AsyncHtmlLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_core.documents import Document
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
import uuid
from typing import List
from pydantic import BaseModel, Field
```

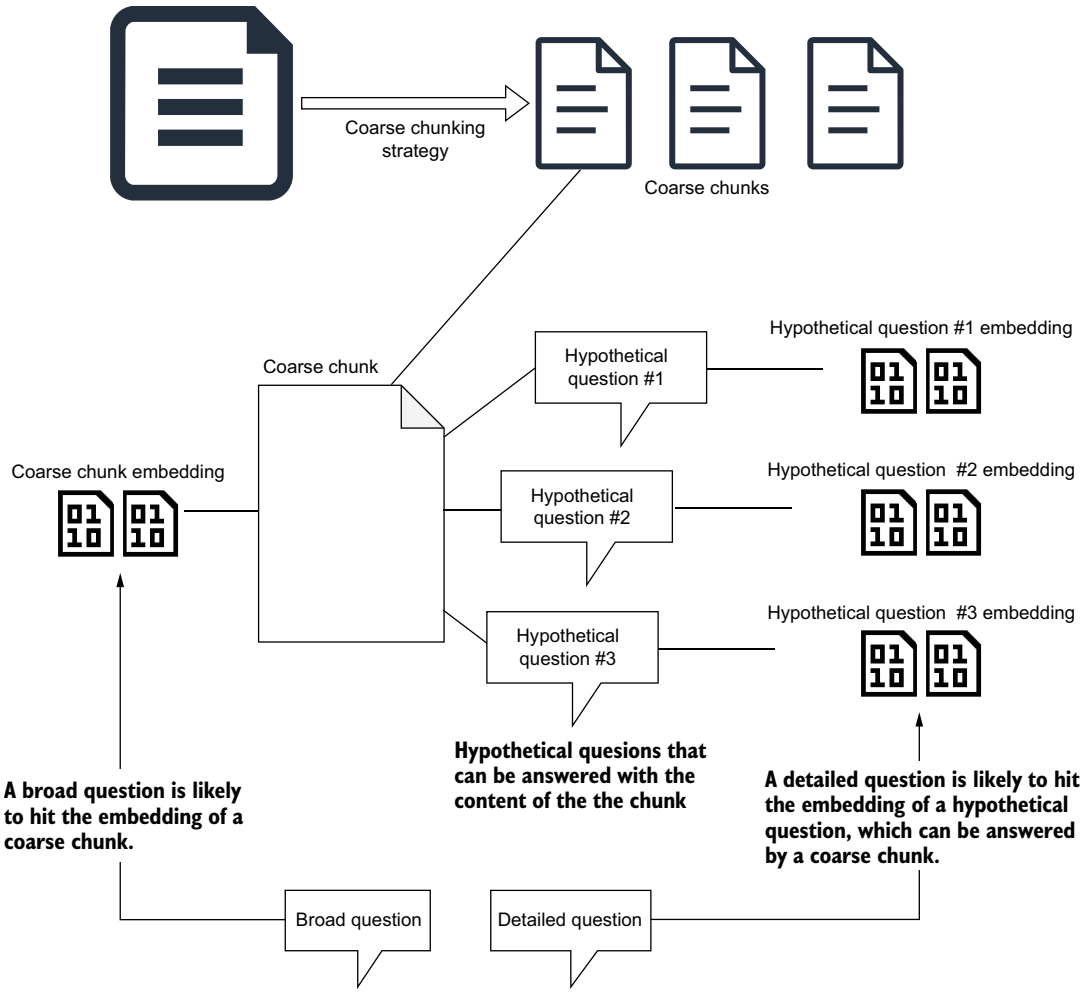


Figure 8.6 Hypothetical question embeddings. A document chunk is indexed with its own embedding and additional embeddings generated from hypothetical questions that it can answer. This allows for more accurate matching with user queries.

SETTING UP THE MULTIVECTORRETRIEVER

Set up the `MultiVectorRetriever` similarly to how it was configured for summary embeddings, but this time, use a vector store specifically for storing hypothetical questions, as shown in the following listing.

Listing 8.8 `MultiVectorRetriever` with hypothetical questions

```
parent_splitter = RecursiveCharacterTextSplitter(
➥chunk_size=3000)
```

Splitter to generate parent coarse chunks from original documents (parsed from web pages)

```
hypothetical_questions_collection = Chroma(
    collection_name="uk_hypothetical_questions",
    embedding_function=OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY),
)

hypothetical_questions_collection
➥.reset_collection()

doc_byte_store = InMemoryByteStore()
doc_key = "doc_id"

multi_vector_retriever = MultiVectorRetriever(
    vectorstore=hypothetical_questions_collection,
    byte_store=doc_byte_store
)
```

Vector store collection to host child granular chunks

Makes sure the collection is empty

Document store to host parent coarse chunks

Retriever to link parent coarse chunks to child granular

SETTING UP THE HYPOTHETICAL QUESTION GENERATION CHAIN

Create a chain to generate hypothetical questions for each document chunk. Use structured output from the LLM to ensure the generated questions are returned as a list of strings:

```
class HypotheticalQuestions(BaseModel):
    """A list of hypotetical questions for given text."""

    questions: List[str] = Field(..., description
    ➥="List of hypothetical questions for given text")

llm_with_structured_output = ChatOpenAI(
    model="gpt-5-nano",
    openai_api_key=OPENAI_API_KEY).with_structured_output(
        HypotheticalQuestions
)
```

You can see the full question generation chain in the following listing.

Listing 8.9 Chain for generating hypothetical questions from text

```
hypothetical_questions_chain = (
    {"document_text": lambda x: x.page_content}
    | ChatPromptTemplate.from_template(
        "Generate a list of exactly 4 hypothetical questions
        ➥that the below text could be used to answer:
        ➥\n\n{document_text}"
    )
    | llm_with_structured_output
    | (lambda x: x.questions)
)
```

Grabs the text content from the document

Instantiates a prompt asking to generate four hypothetical questions on the provided text

Invokes the LLM configured to return an object containing the questions as a typed list of strings

Grabs the list of questions from the response

INGESTING COARSE CHUNKS AND RELATED HYPOTHETICAL QUESTIONS

Now generate coarse chunks, create the hypothetical questions for each, and store them in the respective collections, as shown in the following listing.

Listing 8.10 Ingesting coarse chunks and hypothetical questions

```
for destination_url in uk_destination_urls:
    html_loader = AsyncHtmlLoader(destination_url)
    html_docs =  html_loader.load()
    text_docs = html2text_transformer.transform_documents(
        html_docs)

    coarse_chunks = parent_splitter.split_documents(
        text_docs)

    coarse_chunks_ids = [str(uuid.uuid4()) for _ in coarse_chunks]
    all_hypothetical_questions = []
    for i, coarse_chunk in enumerate(
        coarse_chunks):

        coarse_chunk_id = coarse_chunks_ids[i]

        hypothetical_questions = hypothetical_questions_chain.invoke(
            coarse_chunk)
        hypothetical_questions_docs = [Document(
            page_content=question, metadata={doc_key: coarse_chunk_id})
                    for question
                    in hypothetical_questions]

        all_hypothetical_questions.extend(hypothetical_questions_docs)

    print(f'Ingesting {destination_url}')
    multi_vector_retriever.vectorstore.add_documents(
        all_hypothetical_questions)
    multi_vector_retriever.docstore.mset(
        list(zip(coarse_chunks_ids, coarse_chunks)))
```

Loader for one destination

Documents of one destination

Transforms HTML documents into clean text documents

Splits the destination content into coarse chunks

Iterates over the coarse chunks

Generates a list of hypothetical questions for the coarse chunk through the question generation chain

Links each hypothetical question to its related coarse chunk

Ingests the hypothetical questions into the vector store

Ingests the coarse chunks into the document store

PERFORMING A SEARCH USING THE MULTIVECTORRETRIEVER

After ingestion, perform a search using the `MultiVectorRetriever`, which now uses the stored hypothetical question embeddings:

```
retrieved_docs = multi_vector_retriever.invoke(
    "How can you go to Brighton from London?")
```

The first retrieved document (`retrieved_docs[0]`) is a detailed response with rich contextual information:

```
[Document(metadata={'source': 'https://en.wikivoyage.org/wiki/Brighton',
'title': 'Brighton - Travel guide at Wikivoyage', 'language': 'en'},
page_content='### By plane\n\n[edit]\n\nThe city\'s proximity to London means
Brighton is well served by airports.\nBrighton can be reached from Gatwick by
train in as little as 25 minutes\n£9.80-£11.90, Jan 2023).\n\n  * 50.8332-
0.2923 Shoreham Airport (Brighton City Airport), Cecil Pashley Way, Shoreham-
by-Sea, BN43 5FF (Probably best to get a train from Brighton to Shoreham \\-
about 15 minutes, then a taxi from there to the airport), ☏ +44 1273 467373,
```

```
reception@flybrighton.com. This airport (**ESH** IATA) is 5 miles (8 km) to
the west of Brighton. It is the nearest airport for light aircraft and also
offers sightseeing flights. However, there are no scheduled flights from
here. This is the oldest licensed airport in the UK. (updated Sep 2017)\n\n##
Get around\n\n[edit]\n\n50°50′14″N 0°8′56″W\n\nMap of Brighton\n\nBrightonians
often give directions relative to a prominent landmark, the\n**Clock Tower**
, which stands due south of the rail station where Queen\'s\nRoad meets Dyke
Road (oh yes it does), West Street, North Street and Western\nRoad.\n\nThe
oldest part of the city is the **Lanes** , which is bounded by North\nStreet,
West Street and East Street, through which runs Middle Street, and\nShip
Street. Beware the spelling of the similar-named **North Laine**
(meaning\n"north fields") which is a boutique and alternative shopping
nirvana, to the\nnorth side of North Street.\n\nWestern Road, a major
shopping street runs east-west from the Clock Tower,\nwhilst Eastern Road
runs up a hill towards the main hospital from the area\nknown as the **Old
Steine** (rhymes with clean) which has Brighton Pier at the\nseafront
here.\n\n… [SHORTENED…].
```

COMPARING WITH DIRECT SEARCH ON HYPOTHETICAL QUESTIONS

Now run a search directly on the hypothetical questions collection for comparison:

```
hypothetical_question_docs_only =
➥hypothetical_questions_collection.similarity_search(
    "How can you go to Brighton from London?")
```

The results show hypothetical questions closely matching the query:

```
[Document(metadata={'doc_id': 'af848894-8591-4c28-8295-f3b833ffaa43'},
page_content='What if someone wanted to travel from Brighton to London
quickly?'),
Document(metadata={'doc_id': '7fa14e56-270c-4461-88ab-9b546afb07b1'},
page_content='What if someone wants to attend a performance at Glyndebourne
Opera House, how can they arrange their visit from Brighton?'),
Document(metadata={'doc_id': '7fa14e56-270c-4461-88ab-9b546afb07b1'},
page_content='If a traveler is looking for a day trip from Brighton to
France, what options do they have?'),
Document(metadata={'doc_id': '7fa14e56-270c-4461-88ab-9b546afb07b1'},
page_content='How might a visitor plan their itinerary to include both Lewes
and Worthing in one day from Brighton?')]
```

This pattern is consistent with earlier techniques: using hypothetical question embeddings helps the search engine focus on the specific intent behind the query, making it easier to retrieve relevant documents even when the exact wording varies. Next, I'll explore another technique to further improve document retrieval accuracy.

8.5 Granular chunk expansion

As previously discussed, the main drawback of splitting a document into small granular chunks is that while these chunks are effective for detailed questions, they often lack the context needed to generate complete answers. One way to address this is through *chunk expansion*, as shown in figure 8.7.

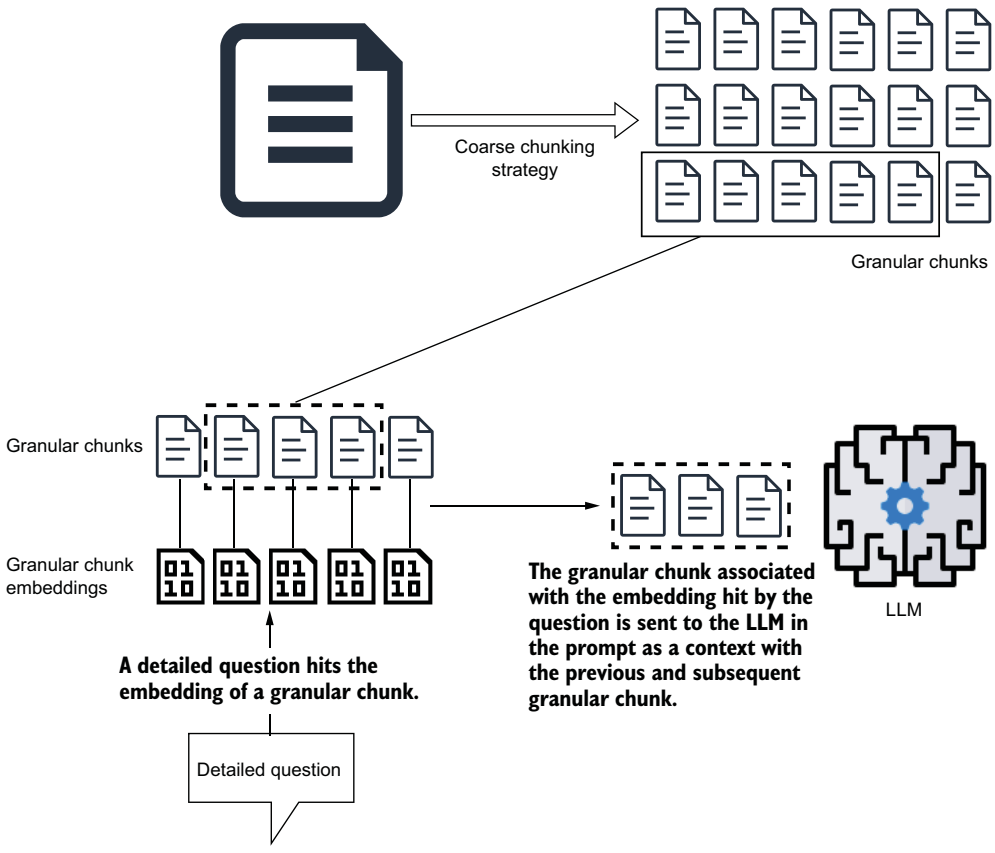


Figure 8.7 Sentence expansion. A granular chunk can be enhanced by including the content from its preceding and following chunks to provide additional context.

The idea is to store an expanded version of each chunk that includes content from the chunks immediately before and after it. This expanded version is stored in a separate document store. So when the vector store retrieves a relevant granular chunk, the linked expanded chunk is returned instead, offering a richer context for the LLM to produce a more complete answer.

This technique can be easily implemented using the `MultiVectorRetriever`. Let's look at how to set this up next.

SETTING UP THE MULTIVECTORRETRIEVER FOR CHUNK EXPANSION

First, configure the `MultiVectorRetriever` by creating a collection to hold granular chunks and an in-memory document store for the expanded chunks. This code is shown in the following listing.

Listing 8.11 `MultiVectorRetriever` for granular chunk expansion

```
from langchain_classic.retrievers.multi_vector import MultiVectorRetriever
from langchain_classic.storage import InMemoryByteStore
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings
from langchain_community.document_loaders import AsyncHtmlLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
import uuid

granular_chunk_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500)

granular_chunks_collection = Chroma(
    collection_name="uk_granular_chunks",
    embedding_function=OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY),
)

granular_chunks_collection.reset_collection()

expanded_chunk_store = InMemoryByteStore()
doc_key = "doc_id"

multi_vector_retriever = MultiVectorRetriever(
    vectorstore=granular_chunks_collection,
    byte_store=expanded_chunk_store
)
```

Splitter to generate granular chunks from original documents (parsed from web pages)

Vector store collection to host child granular chunks

Makes sure the collection is empty

Document store to host expanded chunks

Retriever to link parent coarse chunks to child granular chunks

INGESTING GRANULAR AND EXPANDED CHUNKS

Now generate expanded chunks by including the content from adjacent chunks. The necessary code is shown in the following listing.

Listing 8.12 Generating and storing expanded chunks

```
for destination_url in uk_destination_urls:
    html_loader = AsyncHtmlLoader(destination_url)
    html_docs =  html_loader.load()
    text_docs = html2text_transformer.transform_documents(
        html_docs)

    granular_chunks = granular_chunk_splitter.split_documents(
        text_docs)

    expanded_chunk_store_items = []
    for i, granular_chunk in enumerate(
        granular_chunks):

        this_chunk_num = i
        previous_chunk_num = i-1
        next_chunk_num = i+1
```

Loader for one destination

Documents of one destination

Transforms HTML documents into clean text documents

Splits the destination content into granular chunks

Iterates over the granular chunks

Determines the index of the current chunk and its previous and next chunks

```
        if i==0:
            previous_chunk_num = None
        elif i==(len(granular_chunks)-1):
            next_chunk_num = None

        expanded_chunk_text = ""
        if previous_chunk_num:
            expanded_chunk_text += granular_chunks[
                previous_chunk_num].page_content
            expanded_chunk_text += "\n"

        expanded_chunk_text += granular_chunks[
            this_chunk_num].page_content
        expanded_chunk_text += "\n"

        if next_chunk_num:
            expanded_chunk_text += granular_chunks[
                next_chunk_num].page_content
            expanded_chunk_text += "\n"

        expanded_chunk_id = str(uuid.uuid4())
        expanded_chunk_doc = Document(
            page_content=expanded_chunk_text)

        expanded_chunk_store_item = (expanded_chunk_id,
                                     expanded_chunk_doc)
        expanded_chunk_store_items.append(
            expanded_chunk_store_item)

        granular_chunk.metadata[
            doc_key] = expanded_chunk_id

print(f'Ingesting {destination_url}')
multi_vector_retriever.vectorstore.add_documents(
    granular_chunks)
multi_vector_retriever.docstore.mset(
    expanded_chunk_store_items)
```

Determines the index of the current chunk and its previous and next chunks

Assembles the text of the expanded chunk by including the previous and next chunk

Generates the ID of the expanded chunk

Creates the expanded chunk document

Links each granular chunk to its related expanded chunk

Ingests the granular chunks into the vector store

Ingests the expanded chunks into the document store

PERFORMING A SEARCH USING THE MULTIVECTORRETRIEVER

After the ingestion step, run a search using the `MultiVectorRetriever`, which now uses expanded chunks for a more complete context:

```
retrieved_docs = multi_vector_retriever.invoke("Cornwall Ranger")
```

The first document retrieved will include content from surrounding chunks, giving the LLM more context to generate a richer response:

```
Document(page_content="Buses only serve designated stops when in towns;
otherwise, you can flag them\ndown anywhere that's safe for them to
stop.\n\n### By train\n\n[edit]\n\n**CrossCountry Trains** and **Great
Western Railway** operate regular train\nservices between the main centres of
population, the latter company also\nserving a number of other towns on
```

```
branch lines. For train times and fares\nvisit National Rail Enquiries.\nThe
**Cornwall Ranger** ticket allows unlimited train travel in Cornwall
and\nPlymouth for a calendar day. As of 2023, this costs £14 for adults and
£7 for\nunder-16s.\n\n## See\n\n[edit]\n\nThe **Eden Project** , near St
Austell, a fabulous collection of flora from\nall over the planet housed in
two space age transparent domes, and a massive\nzip line.\n##
See\n\n[edit]\n\ nThe **Lost Gardens of Heligan** , near Mevagissey, 80 acres
(32 hectares) of\nstunning landscaped scenery with a huge complex of walled
flower and vegetable\ngardens.\n\nThe **National Maritime Museum** Falmouth
is the Home of the National Maritime\nMuseum's small boat collection and
other exhibits.\n")
```

COMPARING WITH DIRECT SEMANTIC SEARCH ON GRANULAR CHUNKS

For comparison, run a search directly on the granular chunks without expansion:

```
child_docs_only = granular_chunks_collection
➥.similarity_search("Cornwall Ranger")
```

The result will likely be more concise and lack the broader context:

```
Document(metadata={'doc_id': '04c7f88e-e090-4057-af5b-ea584e777b3f',
'language': 'en', 'source': 'https://en.wikivoyage.org/wiki/South_Cornwall',
'title': 'South Cornwall - Travel guide at Wikivoyage'}, page_content='The
**Cornwall Ranger** ticket allows unlimited train travel in Cornwall
and\nPlymouth for a calendar day. As of 2023, this costs £14 for adults and
£7 for\nunder-16s.\n\n## See\n\n[edit]\n\nThe **Eden Project** , near St
Austell, a fabulous collection of flora from\nall over the planet housed in
two space age transparent domes, and a massive\nzip line.')
```

This smaller chunk lacks the surrounding details and may not provide enough context to synthesize a complete answer. Chunk expansion offers a way to improve the effectiveness of granular embeddings by attaching broader context. The next section will cover how to efficiently handle mixed structured and unstructured content.

8.6 Semi-structured content

When dealing with documents that mix unstructured text and structured data (e.g., tables), it's essential to handle each type separately. You should extract structured content, such as tables, and generate embeddings for their summaries—just as you would for text chunks, as discussed in section 1.4.3.

Store the coarse text chunks and the full tables in a document store and place the embeddings for the summaries of both text and tables in the vector store using a `MultiVectorRetriever`, as shown in figure 8.8. This setup allows seamless retrieval of both structured and unstructured content.

When a search hits the embedding of a table's summary, the entire table (stored in the document store) is returned to the LLM for synthesis, providing the necessary context to generate a complete response.

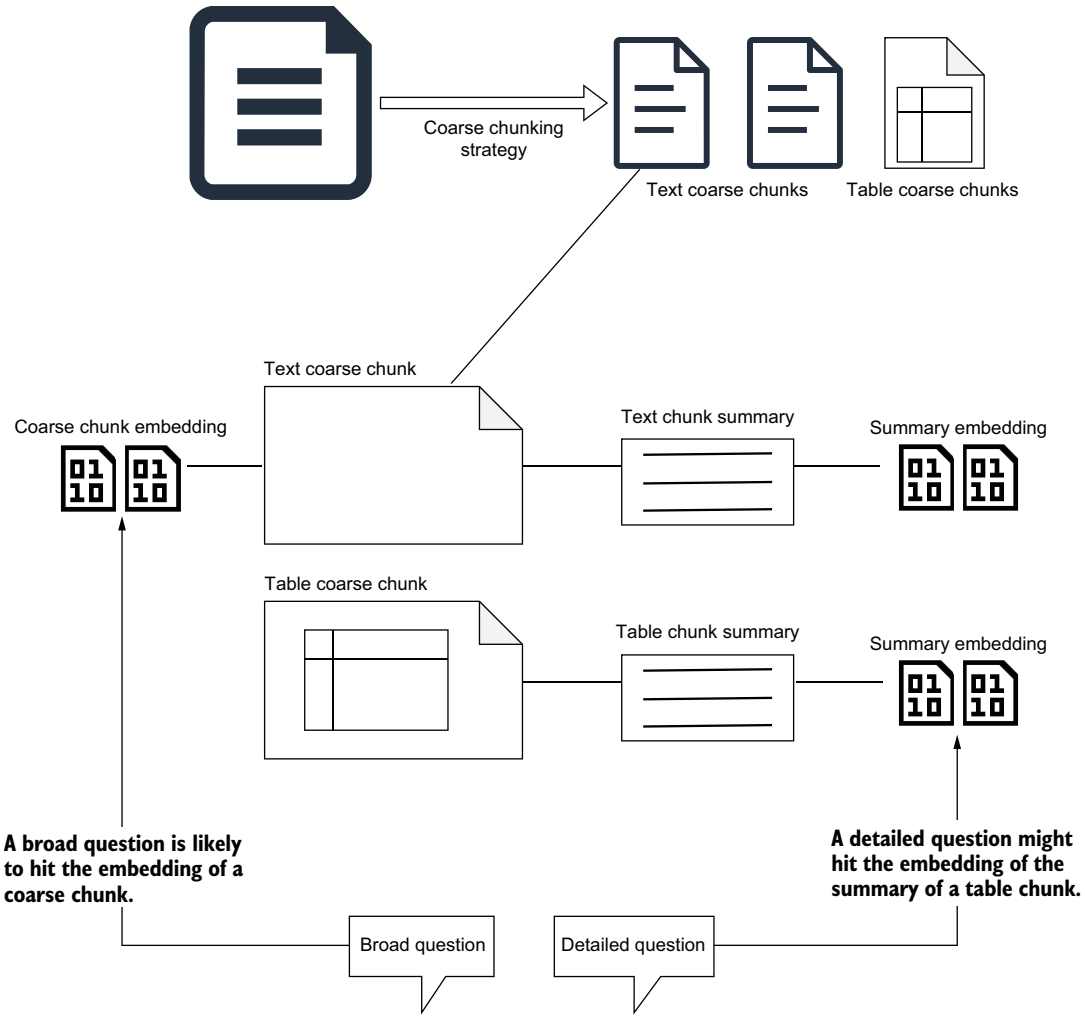


Figure 8.8 Embedding structured and unstructured content. Structured data, such as tables, should be summarized and embedded just like unstructured text chunks. This ensures that the embeddings match detailed questions as effectively as text embeddings. Use the `MultiVectorRetriever` to manage both types of content.

8.7 Multimodal RAG

You've likely come across the term *multimodal LLMs*. Models such as GPT-4V extend traditional LLMs to handle not only text but also images and audio. This opens the door for extending the RAG architecture to support multimodal data.

The approach is similar to handling semi-structured content. During the data preparation stage, you can use a multimodal LLM to generate a summary of an image, just as you would for a table. Then, create embeddings for the image summary, and link these embeddings to the raw image stored in a document store, as shown in figure 8.9.

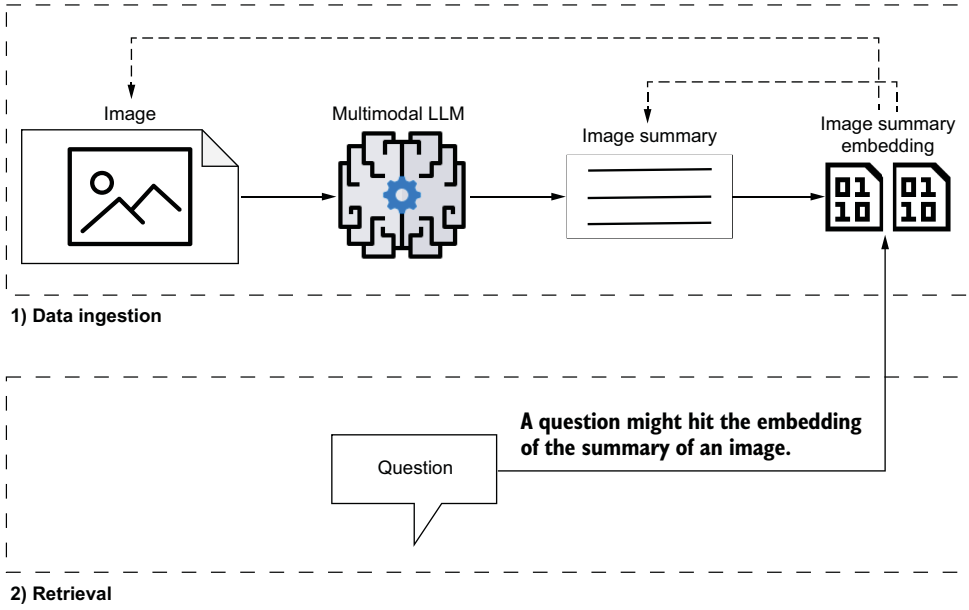


Figure 8.9 Multimodal RAG workflow. (1) Data ingestion: use a multimodal LLM to generate an image summary, store the summary embeddings in a vector store, and keep the raw image in a document store. (2) Retrieval: if the summary embeddings match a query, the raw image is returned by the `MultiVectorRetriever` and fed into the LLM for synthesis.

During retrieval, if the summary's embeddings match a user query, the `MultiVectorRetriever` returns the raw image—just as it would return a table for semi-structured text. The image is then passed to the multimodal LLM along with its summary, providing a rich context for generating a response.

> **NOTE** This book doesn't cover multimodal RAG in detail, as it's an advanced topic that would require in-depth explanations beyond the intended scope. However, with what you've learned so far, you have the foundation to explore it on your own. For more information, I recommend the article "Understanding Multimodal LMMs" by Sebastian Raschka (https://mng.bz/Jw0a).

Summary

- Basic Retrieval-Augmented Generation (RAG) implementations often return low-relevance documents (semantically similar but contextually wrong) or fail to capture multi-hop reasoning across chunks. Optimization techniques address these gaps.
- RAG optimization techniques include advanced document chunking strategies, multi-vector indexing (multiple embeddings per document), query rewriting, and hybrid search that combines dense and sparse retrieval.
- Advanced indexing improves retrieval through refined chunking (splitting by semantic boundaries not fixed character counts), metadata filtering (date

ranges, document types), and parent-child relationships (small chunks for search, large chunks for context).

- Document splitting strategies depend on the content structure:
  - *Size-based splitting*—By character count (500–1,000 chars), sentences (2–5 sentences), or paragraphs. Use when content lacks explicit structure.
  - *Structure-based splitting*—By headers in Markdown, chapters in books, sections in technical docs. Use when documents have clear hierarchical organization.
- HTML and Markdown documents maintain semantic coherence when split by their native structure (h1/h2 tags, section breaks). This preserves context better than arbitrary character limits.
- Parent-child indexing stores small chunks for precise searching but retrieves their larger parent chunks for final context. A 200-character child chunk points to its 2,000-character parent document.
- `ParentDocumentRetriever` embeds child chunks while storing references to parent documents for retrieval. Configure child splitter for search granularity and parent splitter for context size.
- Multi-vector retrieval performance varies by content type and query patterns. Test `ParentDocumentRetriever` against standard vector search on your dataset to validate improvements.
- Context expansion retrieves adjacent chunks surrounding each matched result at retrieval time. A query matching chunk 5 also fetches chunks 4 and 6 for better continuity.
- Semi-structured documents (tables, charts, forms) require extraction libraries such as `Unstructured.io` or custom parsers. Embed table summaries or extracted data separately from flowing text.
- `MarkdownHeaderTextSplitter` preserves document hierarchy by including parent headers in metadata for each chunk. This enables hierarchical filtering and better context understanding.
- Create a parent-child retriever using `ParentDocumentRetriever` with `child_splitter` for small chunks and `parent_splitter` for large chunks. Store both in separate vector stores.
- Parent-child retrieval adds storage overhead (both child and parent chunks stored) and complexity. Be sure to benchmark quality improvement against storage and computational costs before deploying.
- `SemanticChunker` uses embedding similarity to detect natural breakpoints: `from langchain_experimental.text_splitter import SemanticChunker`. This requires additional embedding API calls during ingestion.
- Metadata filtering support varies by vector store. ChromaDB supports `where` filters, but implementation differs across platforms—check your vector store's documentation.
- Context window retrieval fetches *N* chunks before and after each match. Implement by retrieving matches and then fetching adjacent chunks by document ID and position metadata.

9 Question transformations

This chapter covers

- Rewriting user questions with Rewrite-Retrieve-Read for better embedding alignment
- Using step-back queries to retrieve higher-level context
- Generating hypothetical documents to align questions with embeddings
- Decomposing complex queries into single or multi-step sequences

In some cases, you might spend a lot of time preparing Retrieval-Augmented Generation (RAG) data—collecting documents, splitting them into chunks, and generating embeddings for synthesis and retrieval (as covered earlier in chapter 8). Yet, you may still see low-quality results from the vector store. This problem might not come from missing relevant content in the vector store, but instead from issues in the user's question itself. For instance, the question might be poorly phrased, unclear, or overly complex. Questions that aren't clearly and simply stated can confuse both the vector store and the LLM, leading to weaker retrieval results.

In this section, I'll show you techniques to refine the user's question, making it easier for the query engine and the LLM to understand. By improving the question, you'll likely see better retrieval performance, providing more relevant context for the LLM to deliver a solid answer. Let's begin with a straightforward method: using the LLM to help rephrase the question.

9.1 Rewrite-Retrieve-Read

To improve a poorly worded question, one effective method is to have an LLM rewrite it into a clearer form. This approach, covered in "Query Rewriting for Retrieval-Augmented Large Language Models" by Xinbei Ma et al., inspired the diagram in figure 9.1. In the standard Retrieve-and-Read workflow, the retriever processes the original question directly and sends results to the LLM for synthesis. By adding a Rewrite step up front, a rewriter (often an LLM) reformulates the question before passing it to the retriever. This improved workflow is known as Rewrite-Retrieve-Read.

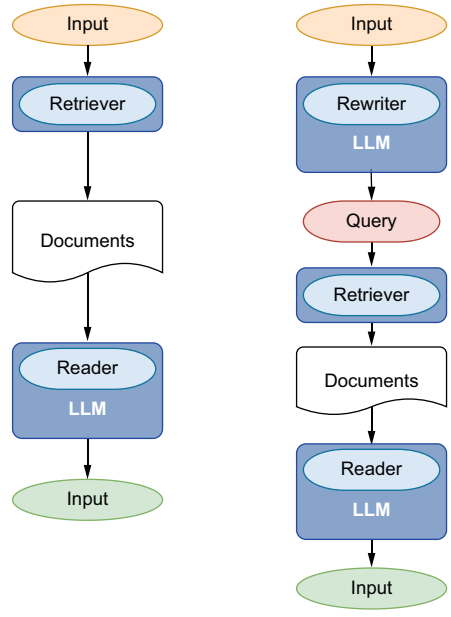


Figure 9.1 In the standard Retrieve-and-Read setup, the retriever processes the user question directly, delivering results to the LLM for synthesis. In the Rewrite-Retrieve-Read approach, an initial Rewrite step uses an LLM to rephrase the query before it reaches the retriever, enhancing the clarity of the retrieval process. (Source: https://arxiv.org/pdf/2305.14283.pdf).

I usually apply this technique to create a tailored search query for the vector store, keeping the original question in the answer-synthesis prompt. This approach allows the rewritten query to optimize retrieval, particularly for semantic searches in the vector database, while preserving the original question for the LLM to synthesize the answer. See the workflow in figure 9.2, which is an amended version of the RAG diagram you saw in figure 5.2 (chapter 5, section 5.1.1).

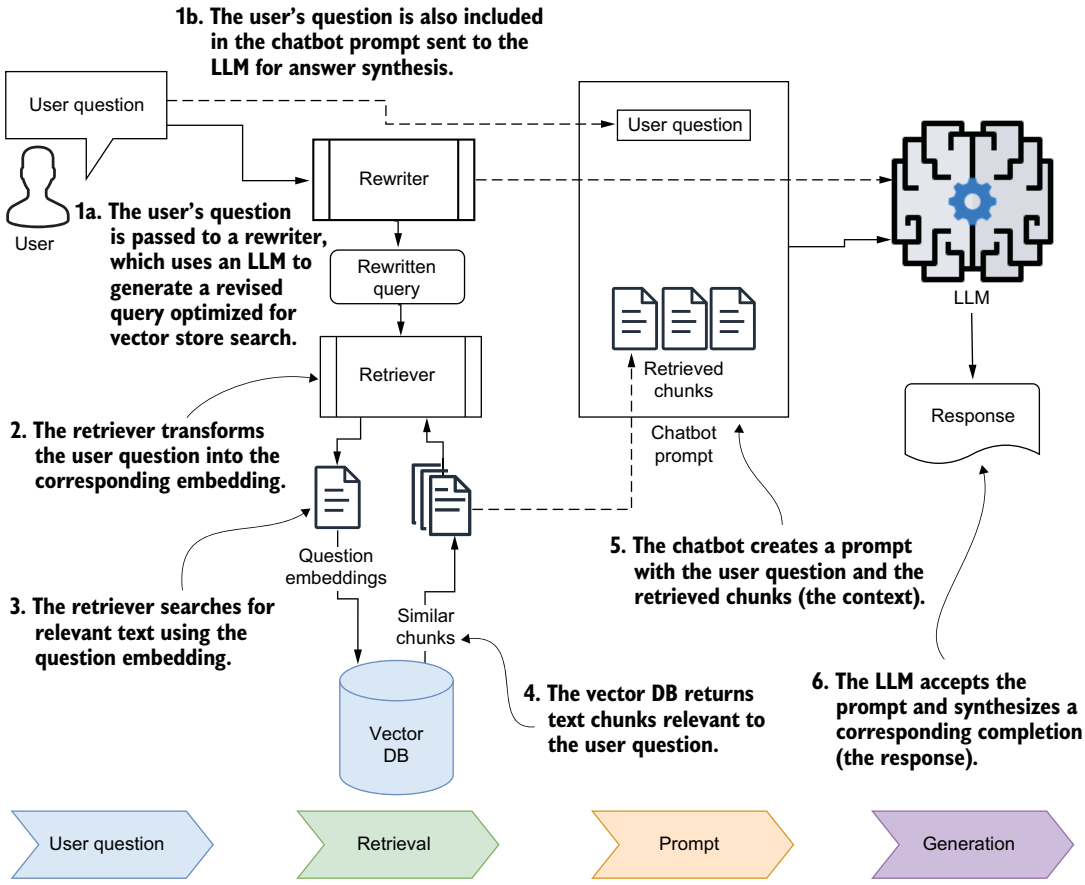


Figure 9.2 Using query rewriting to generate an optimized vector store query, while preserving the original question for answer synthesis via the chatbot prompt

A prompt for rewriting the query can be as simple as the following, adapted from a popular prompt in the LangChain Hub:

```
Revise the original question to make it more refined and precise for search
on ChromaDB, allowing for a more accurate and insightful response.
Original question: {user_question}
Revised ChromaDB query:
```

To apply the Rewrite-Retrieve-Read technique and use the preceding prompt to rewrite the original user question, open a new operating system shell, navigate to the chapter 9 code folder, and set up your environment:

```
C:\Github\building-llm-applications\ch09>
C:\Github\building-llm-applications\ch09>python -m venv env_ch09
C:\Github\building-llm-applications\ch09>env_ch09\Scripts\activate
```

```
(env_ch09) C:\Github\building-llm-applications\ch09>
➥pip install -r requirements.txt
(env_ch09) C:\Github\building-llm-applications\ch09>jupyter notebook
```

After launching Jupyter Notebook, create a new notebook named `09-question_transformations.ipynb`. Then, re-import the UK tourist destination content from Wikivoyage as done in the previous chapter. For convenience, I've included the code here.

Listing 9.1 Splitting and ingesting content from URLs

```
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings
from langchain_text_splitters import HTMLSectionSplitter
from langchain_community.document_loaders import AsyncHtmlLoader

import getpass

OPENAI_API_KEY = getpass.getpass('Enter your OPENAI_API_KEY')

uk_granular_collection = Chroma(
    collection_name="uk_granular",
    embedding_function=OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY),
)

uk_granular_collection.reset_collection()      <-- In case it exists

uk_destinations = [
    "Cornwall", "North_Cornwall", "South_Cornwall", "West_Cornwall",
    "Tintagel", "Bodmin", "Wadebridge", "Penzance", "Newquay",
    "St_Ives", "Port_Isaac", "Looe", "Polperro", "Porthleven",
    "East_Sussex", "Brighton", "Battle", "Hastings_(England)",
    "Rye_(England)", "Seaford", "Ashdown_Forest"
]

wikivoyage_root_url = "https://en.wikivoyage.org/wiki"

uk_destination_urls = [f'{wikivoyage_root_url}/{d}'
                       for d in uk_destinations]

headers_to_split_on = [("h1", "Header 1"),("h2", "Header 2")]
html_section_splitter = HTMLSectionSplitter(
    headers_to_split_on=headers_to_split_on)

def split_docs_into_granular_chunks(docs):
    all_chunks = []
    for doc in docs:
        html_string = doc.page_content                <-- Extracts the HTML text from the document
        temp_chunks = html_section_splitter.split_text(
            html_string)                              <-- Each chunk is an H1 or H2 HTML section.
        h2_temp_chunks = [chunk for chunk in
                          temp_chunks if "Header 2"
                          in chunk.metadata]          <-- Only keeps content associated with H2 sections
        all_chunks.extend(h2_temp_chunks)
```

```
    return all_chunks

for destination_url in uk_destination_urls:
    html_loader = AsyncHtmlLoader(
        destination_url)
    docs =  html_loader.load()

    for doc in docs:
        print(doc.metadata)
        granular_chunks = split_docs_into_granular_chunks(docs)
        uk_granular_collection.add_documents(
            documents=granular_chunks)
```

Loader for one destination

Documents of one destination

This setup prepares your data for effective query rewriting and retrieval using the Rewrite-Retrieve-Read workflow. The Rewrite step will help craft refined search queries, improving retrieval results and the overall quality of the responses generated.

9.1.1 Retrieving content using the original user question

Let's start by performing a search with the original user question:

```
user_question = "Tell me some fun things I can enjoy in Cornwall"
initial_results = uk_granular_collection.similarity_search(
    query=user_question,k=4)
for doc in initial_results:
    print(doc)
```

The output will look something like this (I've reduced it in various places):

```
page_content='Do
 [ edit ]

 Cornwall, in particular Newquay, is the UK's  surfing  capital, with
 equipment hire and surf schools present on many of the county's beaches,
 and events like the UK championships or Boardmasters festival.
 The  South West Coast Path  runs [REDUCED…]
 The  Camel Trail  is an  18-mile (29   km)  [REDUCED…]
 The  Cornish Film Festival  is held annually [REDUCED…]
 The  Royal Cornwall Show  is an agricultural show [REDUCED…]
Camel Creek Adventure Park , Tredinnick,  Wadebridge  offers great family
days out at Cornwall's top theme park.

 Festivals
 [ edit ]
 St Piran's Day  (Cornish:  Gool Peran ) is the national day of Cornwall,
 [REDUCED…].' metadata={'Header 2': 'Do'}
page_content='Do
 [ edit ]

 The  South West Coast Path  runs [REDUCED…]
 The  Camel Trail  is an  18-mile (29   km)  off-road cycle-track [REDUCED…]
 The  Cornish Film Festival  is held annually [REDUCED…]
Cornwall, in particular Newquay, is the UK's  surfing  capital, [REDUCED…]
```

```
 Cricket:   Cornwall CCC  play in the National Counties [REDUCED…] '
 metadata={'Header 2': 'Do'}
page_content='Buy
 [ edit ]

 In the village centre you will find the usual [REDUCED…] '
 metadata={'Header 2': 'Buy'}
page_content='Do
 [ edit ]

 The  South West Coast Path  runs along [REDUCED…]
 St Piran's Day  (Cornish:  Gool Peran ) is the national day of
 Cornwall[REDUCED…] ' metadata={'Header 2': 'Do'}
```

This output provides some information on fun activities, but it appears somehow limited. This might be due to the way the question has been worded.

9.1.2 Setting up the query rewriter chain

To refine the user question into a more effective query for ChromaDB, we'll use the LLM to rewrite it into a format better suited for semantic search. Setting up a query rewriter chain will help us automate this transformation. Start by importing the necessary libraries:

```
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
llm = ChatOpenAI(model="gpt-5-mini", openai_api_key=OPENAI_API_KEY)
```

Next, define the prompt that instructs the LLM to rewrite the user question:

```
rewriter_prompt_template = """
Generate search query for the ChromaDB vector store
from a user question, allowing for a more accurate
response through semantic search.
Just return the revised ChromaDB query, with quotes around it.

User question: {user_question}
Revised ChromaDB query:
"""

rewriter_prompt = ChatPromptTemplate.from_template(
    rewriter_prompt_template)
```

Now construct the chain to execute the rewriting process:

```
rewriter_chain = rewriter_prompt | llm | StrOutputParser()
```

This setup allows you to pass a user question to the rewriter chain, which generates a tailored query optimized for ChromaDB, enhancing retrieval accuracy.

9.1.3 Retrieving content with the rewritten query

Now let's use the rewriter chain to create a more targeted query and see if it returns more accurate results compared to the original question. First, generate the rewritten query:

```
user_question ="Tell me some fun things I can do in Cornwall"

search_query = rewriter_chain.invoke(
    {"user_question": user_question})
print(search_query)
```

If you print `search_query`, you should see something like this:

```
"fun activities to do in Cornwall"
```

Now use this refined query to perform the vector store search:

```
improved_results = uk_granular_collection.similarity_search(
    query=search_query,k=3)
```

Finally, print the results to review their relevance:

```
for doc in improved_results:
    print(doc)
```

You'll get the following output, which I've reduced to save space:

```
page_content='Do
 [ edit ]

 Cornwall, in particular Newquay, is the UK's  surfing  capital, [REDUCED…]
 The  South West Coast Path  runs along the coastline  [REDUCED…]
The Cornish section is supposed to be the most  [REDUCED…]
 The  Camel Trail  is an  18-mile (29   km)  off-road cycle-track  [REDUCED…]
 The  Cornish Film Festival  is held annually each November around  Newquay .
 The  Royal Cornwall Show  is an agricultural show  [REDUCED…]
Camel Creek Adventure Park , Tredinnick,  Wadebridge  offers  [REDUCED…]
 Festivals
 [ edit ]

' Obby 'Oss  is held annually on May Day (1 May), [REDUCED…]
 St Piran's Day  (Cornish:  Gool Peran ) is the national day  [REDUCED…]'
metadata={'Header 2': 'Do'}
page_content='Do
 [ edit ]

 The  South West Coast Path  runs  [REDUCED…]
 The  Camel Trail  is an  18-mile (29   km)  off-road cycle- [REDUCED…]
 The  Cornish Film Festival  is held annually each November around  Newquay .
 Cornwall, in particular Newquay, is the UK's  surfing  capital, [REDUCED…]
 Cricket:   Cornwall CCC  play in the National Counties Cricket  [REDUCED…]'
metadata={'Header 2': 'Do'}
```

```
page_content='Do
 [ edit ]

 Helford River  is an idyllic river estuary between Falmouth and Penzance.
 An ideal stop over for yachts heading for the Isles of Scilly, or further
 afield, with a selection of excellent pubs and other attractions. There is
 also a passenger ferry  [REDUCED…].
 The  South West Coast Path  runs along the coastline  [REDUCED…].
 [REDUCED…]
 Festivals
 [ edit ]

 Allantide  (Cornish:  Kalan Gwav  or  Nos Kalan Gwav ) ia a Cornish festival
 [REDUCED…] Chewidden Thursday  is a festival celebrated by the tin miners
 [REDUCED…].
 Furry Dance , also known as Flora Day, takes place in   [REDUCED…].
 Golowan , sometimes also  Goluan  or  Gol-Jowan , is the Cornish word for
 the Midsummer celebrations, widespread prior to the late 19th century and
 most popular in the Penwith area and in particular  Penzance  and  Newlyn .
 [REDUCED…].
 Guldize  is an ancient harvest festival in Autumn, [REDUCED…]
 Montol Festival  is an annual heritage, arts and community  [REDUCED…].
 Nickanan Night  is traditionally held on the Monday before Lent. [REDUCED…]
 St Piran's Day  (Cornish:  Gool Peran ) is the national day of Cornwall
 [REDUCED…].' metadata={'Header 2': 'Do'}
```

This approach should yield results that align more closely with your original intent, displaying a broader selection of content on activities available in Cornwall. Using the rewritten question, the vector store retriever provides a more diverse set of text chunks compared to the results from the original question.

9.1.4 Combining everything into a single RAG chain

Now you can build a complete workflow that transforms the initial user question into a search query for vector retrieval. The original question is retained in the prompt to generate the final answer. The following listing shows the full RAG chain, including the query rewriting step.

Listing 9.2 Combined RAG chain with query rewriting

```
from langchain_core.runnables import RunnablePassthrough
retriever = uk_granular_collection.as_retriever()

rag_prompt_template = """
Given a question and some context, answer the question.
If you do not know the answer, just say I do not know.

Context: {context}
Question: {question}
"""
```

```
rag_prompt = ChatPromptTemplate.from_template(
    rag_prompt_template)

rewrite_retrieve_read_rag_chain = (
    {
        "context": {"user_question": RunnablePassthrough()}
            | rewriter_chain | retriever,
        "question": RunnablePassthrough(),
    }
    | rag_prompt
    | llm
    | StrOutputParser()
)
```

The context is returned by the retriever after feeding to it the rewritten query.

The original user question

Now run the complete workflow:

```
user_question = "Tell me some fun things I can do in Cornwall"

answer = rewrite_retrieve_read_rag_chain.invoke(user_question)
print(answer)
```

When you print the `answer`, you should see a response like this:

```
In Cornwall, you can enjoy a variety of fun activities such as:

1. Surfing in Newquay, known as the UK's surfing capital, where you can find
equipment hire and surf schools on many beaches.
2. Walking along the scenic South West Coast Path, which offers beautiful
views and takes you to towns, cliffs, and beaches.
3. Cycling on the Camel Trail, an 18-mile off-road cycle track that follows
the picturesque river Camel.
4. Attending the Cornish Film Festival held annually in November around
Newquay.
5. Exploring the Helford River, where you can take a ferry ride, visit pubs,
and enjoy attractions like the Gweek Seal Sanctuary and Trebah Gardens.
6. Participating in local festivals such as St Piran's Day or the Furry Dance
in Helston.
7. Visiting Camel Creek Adventure Park for family-friendly entertainment and
activities.
8. Enjoying various agricultural shows, like the Royal Cornwall Show in June.
There are plenty of options for both adventure and relaxation in Cornwall!
```

This output demonstrates a satisfying answer based on the combined Rewrite-Retrieve-Read workflow. Next, I'll show you further ways to refine the Rewrite-Retrieve-Read process.

9.2 Generating multiple queries

The Rewrite-Retrieve-Read approach assumes the original user question was poorly phrased. But if the question is well-formed and contains multiple implicit questions, rewriting it as a single improved query may not be effective. In these cases, it's better to have the LLM break down the original question into multiple explicit questions.

Each question can be executed separately against the vector store, with the answers then synthesized into a comprehensive response. Figure 9.3 illustrates this workflow.

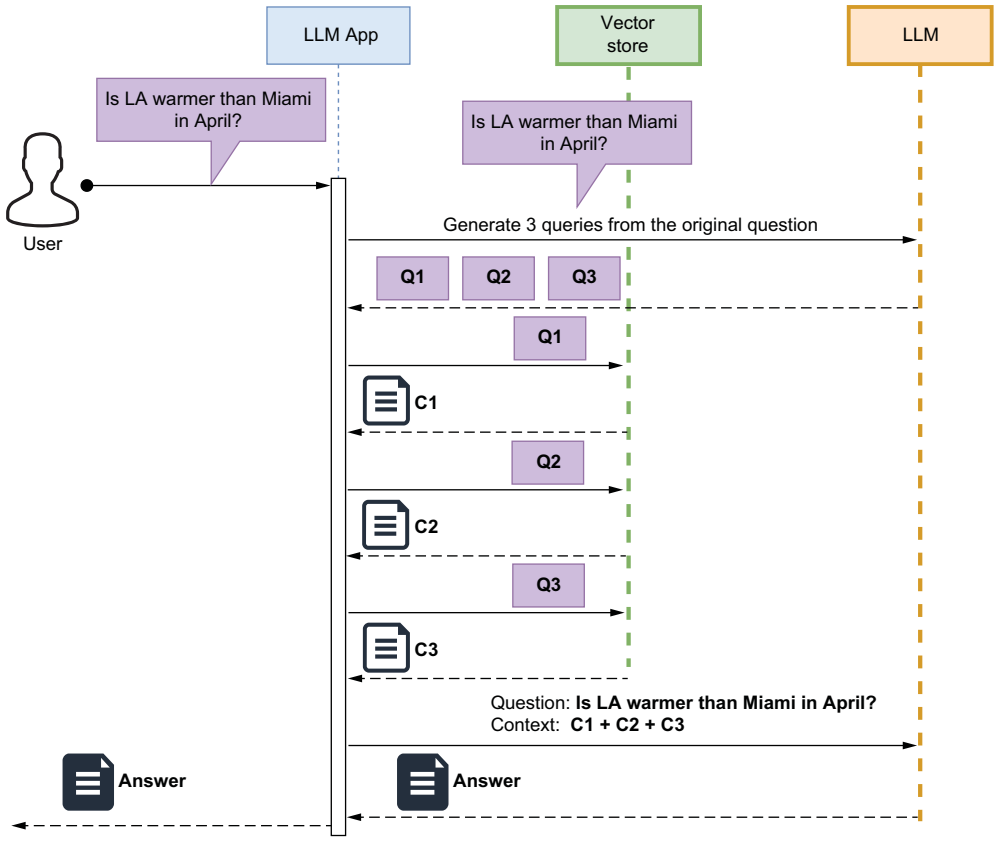


Figure 9.3 Workflow for multiple query generation

In figure 9.3, the LLM application reformulates the original question into multiple explicit questions, executes them against the vector store to gather context, and then synthesizes the answer using a prompt that includes both the original question and the retrieved context.

For example, if you prompt ChatGPT with the following

> Reformulate the following question into multiple explicit questions for my vector store so I can get more easily a correct answer:
>
> Is LA warmer than Miami in April?

you might receive the following questions, which can be executed individually against the vector store:

> “ What is the average temperature in Los Angeles during April?
>
> What is the average temperature in Miami during April?
>
> Can you compare the April temperatures in Los Angeles and Miami and determine which one tends to be warmer?
>
> Is there a notable difference in temperature between Los Angeles and Miami in April?
>
> Could you provide insights into how the temperatures in Los Angeles and Miami compare specifically in the month of April?

This approach is especially useful when designing generic LLM applications. You can automatically generate multiple queries for any user question using a prompt like this, adapted from a LangChain example:

```
QUERY_PROMPT = PromptTemplate(
    input_variables=["question"],
    template="""You are an AI language model assistant.
Your task is to generate five different versions of the given
user question to retrieve relevant documents from a vector
database. By generating multiple perspectives on the user
question, your goal is to help the user overcome some of the
limitations of the distance-based similarity search.
Provide these alternative questions separated by newlines.
Original question: {question}""",
)
```

LangChain’s `MultiQueryRetriever` class allows you to use a prompt, an LLM reference, and a retriever (e.g., derived from a vector database). When a `MultiQueryRetriever` instance processes a user query, it executes the entire multi-query workflow automatically, as shown earlier in figure 9.3. This approach combines multi-retrieval with answer synthesis. Next, I’ll show you how to implement a custom `MultiQueryRetriever`.

9.2.1 Setting up the chain for generating multiple queries

First, let’s import the necessary libraries. Use the following code:

```
from langchain_classic.retrievers.multi_query import MultiQueryRetriever
from langchain_core.prompts import ChatPromptTemplate

from typing import List
from langchain_core.output_parsers import BaseOutputParser
from pydantic import BaseModel, Field
```

Begin by setting up the prompt shown earlier to instruct the LLM to generate multiple variations of a single user question:

```
multi_query_gen_prompt_template = """
You are an AI language model assistant. Your task
```

```
is to generate five different versions of the given
user question to retrieve relevant documents from a vector
database. By generating multiple perspectives on the user
question, your goal is to help the user overcome some of
the limitations of the distance-based similarity search.
Provide these alternative questions separated by newlines.
Original question: {question}
"""

multi_query_gen_prompt = ChatPromptTemplate.from_template(
    multi_query_gen_prompt_template)
```

Because the LLM is generating five alternative questions, it's useful to format the output as a list of strings, with each string representing one question. This lets you process each question independently. To do this, implement a custom result parser instead of the standard `StrOutputParser`:

```
class LineListOutputParser(BaseOutputParser[List[str]]):
    """Parse out a question from each output line."""

    def parse(self, text: str) -> List[str]:
        lines = text.strip().split("\n")
        return list(filter(None, lines))

questions_parser = LineListOutputParser()
```

With these components, you can set up the chain to generate multiple queries:

```
llm = ChatOpenAI(model="gpt-5-nano", openai_api_key=OPENAI_API_KEY)
multi_query_gen_chain = multi_query_gen_prompt | llm | questions_parser
```

Now try running this chain:

```
user_question = "Tell me some fun things I can do in Cornwall."

multiple_queries = multi_query_gen_chain.invoke(user_question)
```

When you print `multiple_queries`, you should get a list of questions similar to this:

```
['What are some enjoyable activities to explore in Cornwall?  ',
 'Can you suggest interesting attractions or events in Cornwall for a fun
    experience?  ',
 'What are the top leisure activities to try out while visiting Cornwall?  ',
 'What fun experiences or adventures does Cornwall have to offer?  ',
 'Could you recommend some entertaining things to do while in Cornwall?']
```

One final step remains to complete the setup. Let's set up the multi-query retriever next.

9.2.2 Setting up a custom multi-query retriever

To configure a custom multi-query retriever, start by creating a standard retriever and then embedding it within the multi-query retriever:

```
basic_retriever = uk_granular_collection.as_retriever()

multi_query_retriever = MultiQueryRetriever(
    retriever=basic_retriever, llm_chain=multi_query_gen_chain,
    parser_key="lines"
)
```

The key for the parsed output

Now test it with an example query:

```
user_question = "Tell me some fun things I can do in Cornwall"

retrieved_docs = multi_query_retriever.invoke(user_question)
```

When you print `retrieved_docs`, you should see output similar to this:

```
[Document(metadata={'Header 2': 'Do'}, page_content='Do \n [ edit ] \n \n
Cornwall, in particular Newquay, is the UK\'s  surfing  capital, with
equipment hire and surf schools present on many of the county\'s beaches, and
events like the UK championships or Boardmasters festival. \n The  South West
Coast Path  runs along the coastline of Britain\'s south-west peninsula. The
Cornish section is supposed to be the most scenic (unless you talk to someone
in Devon, [... REDUCED ...] There is large parties widespread across the
whole of Cornwall, with people dressing in the black, white and silver
national colours.'),
 Document(metadata={'Header 2': 'Do'}, page_content='Do \n [ edit ] \n \n The
South West Coast Path  runs along the coastline of Britain's south-west
peninsula. The Cornish section is supposed to be the most scenic (unless you
talk to someone [... REDUCED ...] They are unbiased and won't express an
opinion on accommodations, more than giving its tourist board rating and
facilities."),
 Document(metadata={'Header 2': 'Contents'}, page_content='Contents \n \n \n
\n \n \n \n \n 1   Towns and villages [... REDUCED ...]  Cornwall . It
includes much of the Cornish coast along the Celtic Sea and some top surfing
areas.'),
 Document(metadata={'Header 2': 'Do'}, page_content="Do \n [ edit ] \n \n The
South West Coast Path  runs along the coastline of Britain's south-west
peninsula. The Cornish section is supposed to be the most scenic (unless you
talk to someone in Devon, in which [... REDUCED ...], with people dressing in
the black, white and silver national colours."),
 Document(metadata={'Header 2': 'Festivals'}, page_content='Festivals \n [
edit ] \n \n These festivals tend to not be public holidays and not all are
celebrated fully across [... REDUCED ...] is large parties widespread across
the whole of Cornwall, with people dressing in the black, white and silver
national colours.  \n Tom Bawcock\'s Eve : on 23rd December, stargazey pies
are traditionally consumed. In mythology, pies were seen bizarrely as the
reason the devil stayed out of Cornwall.'),
 Document(metadata={'Header 2': 'Contents'}, page_content='Contents \n \n \n
\n \n \n \n \n 1   Towns and villages  [... REDUCED ...] South Cornwall  is
in  Cornwall . It includes much of the stunning Cornish coast along the
English Channel of the Atlantic Ocean.'),
 Document(metadata={'Header 2': 'See'}, page_content="See \n [ edit ] \n \n
The Cheesewring at Minions, Bodmin Moor. \n St. Michael's Mount lies offshore
close to Penzance. \n Cornwall boasts many attractions  [... REDUCED ...]
Madron   is a sheltered garden bursting with exotic trees and shrubs. \n \n
Kynance Cove offers great views towards the Lizard.")]
```

9.2.3 Using a standard MultiQueryRetriever instance

For straightforward use cases, you can set up multi-query generation with a standard `MultiQueryRetriever` instance. First, instantiate the multi-query retriever:

```
std_multi_query_retriever = MultiQueryRetriever.from_llm(
    retriever=basic_retriever, llm=llm
)
```

Now test it with the same question:

```
user_question = " Tell me some fun things I can do in Cornwall"

retrieved_docs = std_multi_query_retriever.invoke(user_question)
```

The output in `retrieved_docs` will be similar to the results you saw previously. In some cases, rewriting the original question or breaking it into multiple explicit questions may not yield the desired results. Continue reading to explore a technique that can improve accuracy in these situations.

9.3 Step-back question

When you send a highly detailed question directly to the vector store—assuming your documents are split into small, specific chunks—you might retrieve information that's too focused and misses the broader context. This can limit the LLM's ability to generate a comprehensive answer.

As discussed in section 7.4, one solution is to create two sets of document chunks: coarse chunks for synthesis and fine-grained chunks for detailed retrieval. Another solution is to adjust the user question rather than the document chunks, using an approach called a *step-back question*.

In this approach, you start with the user's detailed question but then create a broader question to retrieve a more generalized context. This *step-back context* provides a higher-level view than the original context derived from the specific question. You then provide the LLM with both the detailed context and the broader context to enable a fuller response, as illustrated in figure 9.4 where the LLM application first sends the detailed question (Q_D) to the vector store to retrieve a detailed context (C_D). It then prompts the LLM to generate a more abstract question (Q_A) based on Q_D, which is also executed in the vector store to obtain an abstract context (C_A). Finally, the LLM application combines Q_D, C_D, and C_A into a single prompt, enabling the LLM to synthesize a comprehensive answer.

The LLM application sends the original detailed question to the vector store to retrieve detailed context and then generates and executes a broader question to obtain abstract context. It combines both contexts with the original question to enable the LLM to create a comprehensive answer. Developed by Huaixiu Steven Zheng et al., this technique is explained further in "Take a Step Back: Evoking Reasoning via Abstraction in Large Language Models" (https://mng.bz/rZEy).

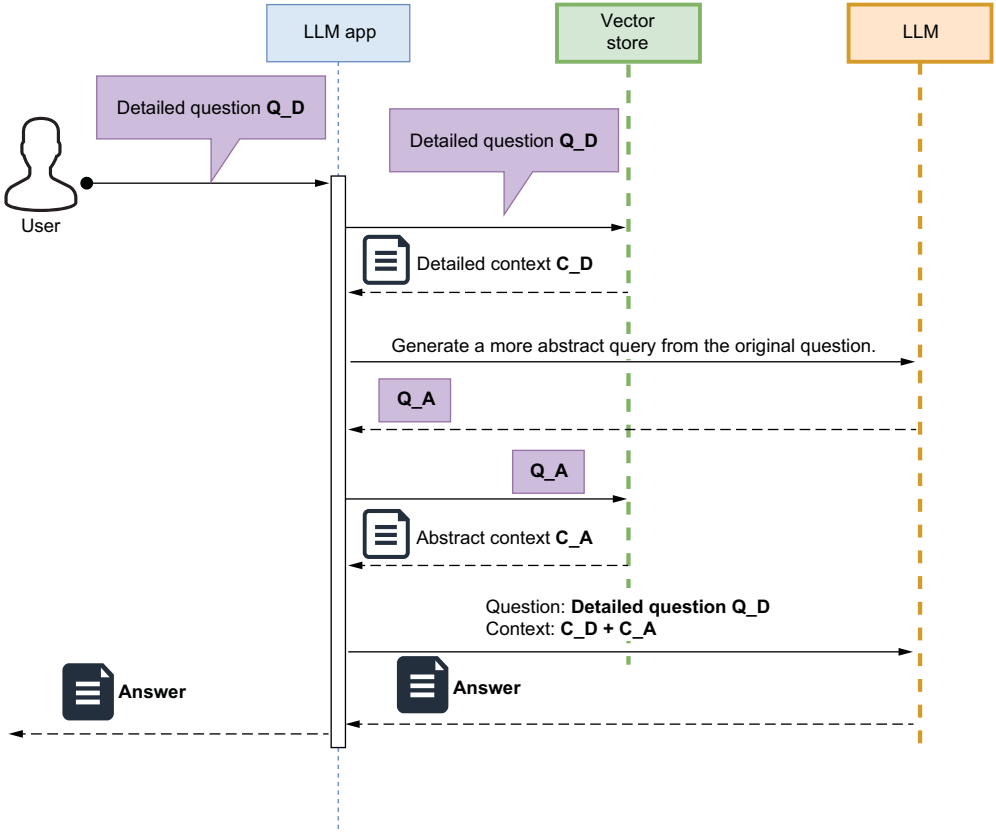


Figure 9.4 Step-back question workflow

To implement this, use a prompt like the following to generate the step-back question:

```
Generate a less specific question (aka Step-back question)
for the following detailed question, so that a wider context
can be retrieved.
Detailed question: {detailed_question}
Step-back question:
```

For example, if you input the prompt with the detailed question, "Can you give me some tips for a trip to Brighton?" the more abstract (step-back) question might look like this:

```
Step-back question: "What should I know before visiting a
popular coastal town?"
```

This broader question helps retrieve more general information, which, combined with the detailed context, allows the LLM to produce a well-rounded answer.

9.3.1 Setting up the chain to generate a step-back question

Implementing the step-back question technique is straightforward: it involves crafting an effective prompt to generate a broader question and then following a standard RAG workflow. Here's a sample implementation, which closely resembles the pattern used for the Rewrite-Retrieve-Read technique. Start by setting up the prompt in your Jupyter Notebook:

```
llm = ChatOpenAI(model="gpt-5", openai_api_key=OPENAI_API_KEY)

step_back_prompt_template = """
Generate a less specific question (aka Step-back question)
for the following detailed question, so that a wider context
can be retrieved.
Detailed question: {detailed_question}
Step-back question:
"""

step_back_prompt = ChatPromptTemplate.from_template(
    step_back_prompt_template)
```

> **NOTE** I've chosen to use GPT-5 over GPT-5-nano and GPT-5-mini because it tends to generate more abstract yet contextually relevant queries, along with more coherent and well-synthesized final answers. That said, I encourage you to experiment with different models to see how their outputs vary.

Now create the chain:

```
step_back_question_gen_chain = step_back_prompt | llm | StrOutputParser()
```

Try out the chain with a sample question:

```
user_question = "Can you give me some tips for a trip to Brighton?"

step_back_question = step_back_question_gen_chain.invoke(user_question)
```

When you print `step_back_question`, you should get a response like this:

```
'What are some general tips for planning a successful trip to a coastal city?'
```

This generated step-back question can then be used within a RAG architecture to retrieve broader context from the vector store, which can subsequently be provided to the LLM to help synthesize a more complete answer.

9.3.2 Incorporating step-back question generation into the RAG chain

You can integrate the step-back question generation chain into a RAG workflow. The following listing shows you how.

Listing 9.3 Integrating step-back question generation within RAG

```
retriever = uk_granular_collection.as_retriever()

rag_prompt_template = """
```

```
Given a question and some context, answer the question.
If you do not know the answer, just say I do not know.

Context: {context}
Question: {question}
"""

rag_prompt = ChatPromptTemplate.from_template(rag_prompt_template)

step_back_question_rag_chain = (
    {
        "context": {"detailed_question": RunnablePassthrough()}
           | step_back_question_gen_chain
           | retriever,
        "question": RunnablePassthrough(),
    }
    | rag_prompt
    | llm
    | StrOutputParser()
)
```

The context is returned by the retriever after feeding to it the step-back question.

The original user question

Now try running the chain:

```
user_question = "Can you give me some tips for a trip to Brighton?"

answer = step_back_question_rag_chain.invoke(user_question)
print(answer)
```

You should see a synthesized response like this:

```
Here are some tips for a trip to Brighton:

1. **Stay Safe**: While Brighton is generally safe, be cautious in busy
areas, especially West Street after midnight due to the nightlife crowd.
2. **Watch for Traffic**: Be mindful of traffic, especially in busy areas.
3. **Valuables**: Take standard precautions with your valuables to avoid
theft.
4. **Homelessness**: Be aware that there may be homeless individuals asking
for money, but most are harmless.
5. **Beaches**: Lifeguards patrol the beaches from late May to early
September. Pay attention to signposts about which areas are covered.
6. **Emergency Contacts**: In case of emergencies related to the sea, call
999 and ask for the Coastguard.
7. **Explore Local Venues**: Enjoy local venues favored by residents for a
civilized night out.
8. **Cultural Areas**: Visit areas like The Lanes and North Laine for a
vibrant cultural experience.
9. **Stay Informed**: Keep an eye on your surroundings, especially in crowded
places.

Enjoy your trip to Brighton!
```

This technique offers an alternative to embedding-focused methods, enhancing retrieval by broadening the question itself to improve context. Next, I'll introduce another technique that also optimizes retrieval through question transformation.

9.4 Hypothetical Document Embeddings (HyDE)

As discussed in chapter 8, embedding hypothetical questions can enhance RAG retrieval by indexing document chunks with additional embeddings that represent questions answerable by the content in each chunk. This approach makes embeddings of these hypothetical questions more semantically similar to the user's question than embeddings of the raw chunk text alone.

A similar effect can be achieved with Hypothetical Document Embeddings (HyDE), a technique that keeps the original chunk embeddings unchanged while generating hypothetical documents based on the user's question. The HyDE technique is shown in figure 9.5. In this approach, the LLM generates hypothetical documents that would answer the user's question. Rather than querying the document store with the user's original question, these generated documents are used. Because these hypothetical documents are semantically closer to the document chunk text, they improve the relevance of retrieved content.

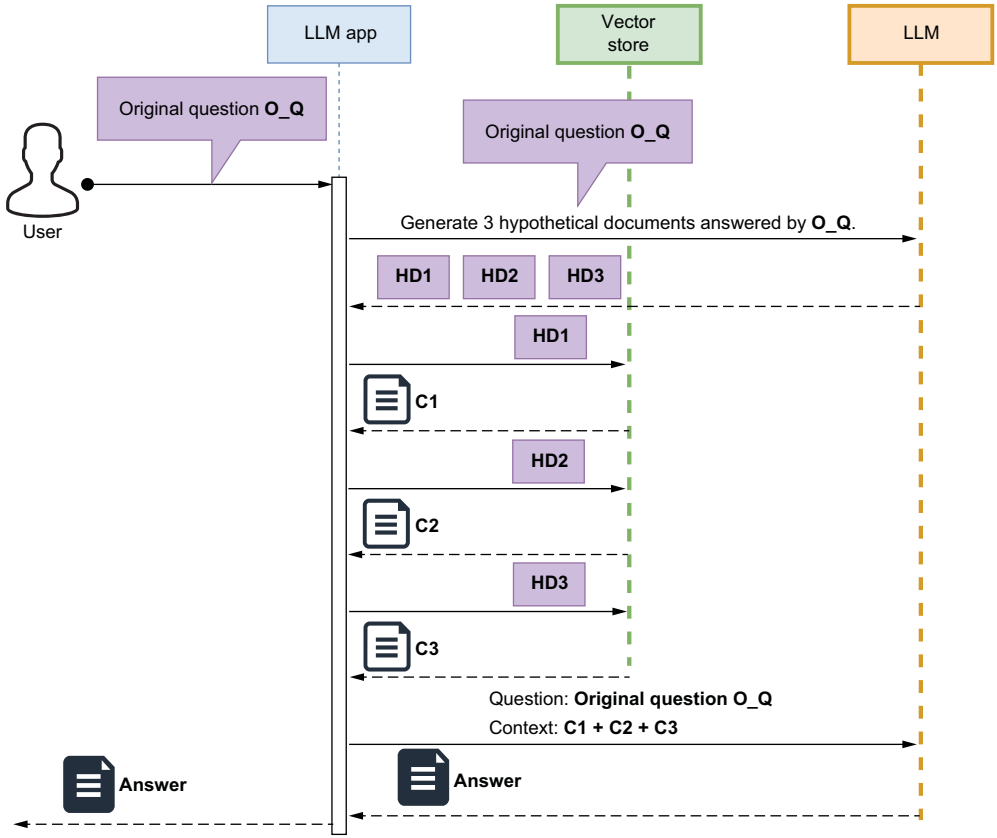


Figure 9.5 The Hypothetical Document Embeddings (HyDE) technique

As shown in the sequence diagram in figure 9.5, the generated hypothetical documents are used to retrieve relevant content, aiming to increase the semantic similarity between the user's question and the document chunk embeddings. This technique was introduced by Luyu Gao et al. in "Precise Zero-Shot Dense Retrieval Without Relevance Labels" (https://arxiv.org/pdf/2212.10496v1.pdf). Next, let's implement this workflow.

9.4.1 Generating a hypothetical document for the user question

Implementing HyDE follows a familiar pattern: you design a chain to generate a hypothetical document that could answer the user's question and then use this generated document as input for retrieval within a larger RAG workflow. First, set up the prompt:

```
llm = ChatOpenAI(model="gpt-5-nano", openai_api_key=OPENAI_API_KEY)

hyde_prompt_template = """
Write one sentence that could answer the provided question.
Do not add anything else.
Question: {question}
Sentence:
"""

hyde_prompt = ChatPromptTemplate.from_template(hyde_prompt_template)
```

Next build the HyDE chain:

```
hyde_chain = hyde_prompt | llm | StrOutputParser()
```

Test it with a sample question:

```
user_question = "What are the best beaches in Cornwall?"

hypotetical_document = hyde_chain.invoke(user_question)
```

When you print `hypothetical_document`, you should see output like this:

```
Some of the best beaches in Cornwall include Fistral Beach, Porthcurno Beach,
and St Ives Bay.
```

This generated hypothetical document can then be used in the RAG chain to retrieve relevant content, improving the alignment between the user's question and the document chunks in the vector store. Let's look at integrating this step into the broader RAG workflow next.

9.4.2 Integrating the HyDE chain into the RAG chain

You can incorporate the HyDE chain into a RAG workflow, as shown here.

Listing 9.4 Integrating a HyDE chain within a RAG workflow

```
retriever = uk_granular_collection.as_retriever()

rag_prompt_template = """
```

```
Given a question and some context, answer the question.
Only use the provided context to answer the question.
If you do not know the answer, just say I do not know.

Context: {context}
Question: {question}
"""

rag_prompt = ChatPromptTemplate.from_template(rag_prompt_template)

hyde_rag_chain = (
    {
        "context": {"question": RunnablePassthrough()}
           | hyde_chain | retriever,
        "question": RunnablePassthrough(),
    }
    | rag_prompt
    | llm
    | StrOutputParser()
)
```

The context is returned by the retriever after feeding to it the hypothetical document.

The original user question

Now try the complete RAG chain:

```
user_question = "What are the best beaches in Cornwall?"

answer = hyde_rag_chain.invoke(user_question)
print(answer)
```

You should see a synthesized answer similar to this:

```
The best beaches in Cornwall mentioned in the context include Bude, Polzeath,
Watergate Bay, Perranporth, Porthtowan, Fistral Beach, Newquay, St Agnes, St
Ives, Gyllyngvase beach in Falmouth, and Praa Sands. Additionally, in
Newquay, popular beaches are Crantock Beach, Fistral Beach, Great Western,
Harbour, Holywell Bay, Lusty Glaze Beach, Porth Joke, Porth, Tolcarne Beach,
Towan Beach, Whipsiderry, and Watergate Bay.
```

This concludes the integration of HyDE into the RAG chain, enhancing the retrieval process by using a hypothetical document. Before moving on, I'll briefly revisit the multi-query generation technique we discussed earlier to refine the retrieval focus.

9.5 Single-step and multi-step decomposition

In the multiple questions or subquestions method covered in section 9.2, the original user question contained several implicit questions. For that case, we instructed the LLM to generate a set of explicit, independent questions, each of which could be executed separately (or in parallel) on the vector store. This approach, called *single-step decomposition*, is effective when the questions are independent and the original complex question can be split into simple, single-step questions.

However, if the original question includes several interdependent questions, a different approach is required. For example, if your data store contains tourist information, consider using this question: "What is the average August temperature at the

most popular sandy beach in Cornwall?" This question requires a sequence of dependent queries, where each answer informs the next question.

In this case, you could instruct the LLM to generate a strategic plan for breaking down the question into a sequence of interdependent queries. Each query would contain a parameter filled with information from the previous answer. Once the LLM returns this sequence of parameterized questions, you can execute them step-by-step, storing each answer and feeding it into the subsequent query. This process continues until you reach the final answer, as shown in figure 9.6.

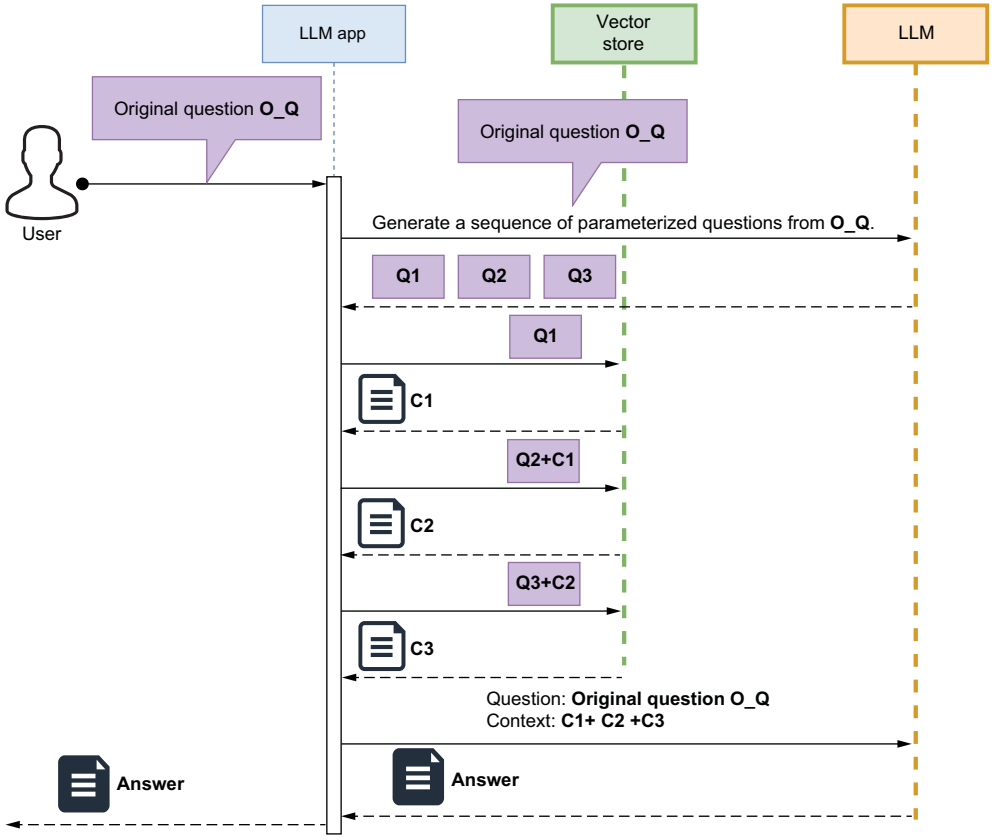


Figure 9.6 Multi-step decomposition workflow

In the workflow illustrated in the sequence diagram, the original complex question is sent to the LLM, which breaks it down into a series of parameterized questions. Each question is executed on the vector store, using previous answers as parameters. After all questions are answered, the LLM combines the collected information to generate a

final response. To prompt the LLM to generate this question sequence, use a template like this:

```
Break down the following question into multiple dependent steps
That I can execute on a vector store or other data sources (e.g.,
databases) to obtain the final answer. Assume you have granular
data in your data sources, but not aggregate data. Each
subsequent question can take a parameter from the result of the
previous step.
For each step, include the question and the corresponding
parameter that will be filled with the previous answer.
Provide the sequence of questions in JSON format as a list,
where each step includes:
- Question: The text of the question.
- Parameter: The parameter to be filled with the previous answer.
—---
Original question: {user_question}
Multiple dependent questions:
```

To illustrate what such a sequence of questions might look like, you can try running the preceding prompt in ChatGPT. Replacing the `user_question` with "What is the average August temperature at the most popular sandy beach in Cornwall?" will produce a response similar to this:

```
[
  {
    "Question": "What is the most popular sandy beach in Cornwall?",
    "Parameter": "None (initial query to the full data source)"
  },
  {
    "Question": "What are the recorded daily temperatures in August for
    [Beach Name]?",
    "Parameter": "Beach Name from the previous answer"
  },
  {
    "Question": "What is the average temperature from the following list of
    daily temperatures: [Daily Temperatures]?",
    "Parameter": "Daily Temperatures from the previous answer"
  }
]
```

Each question can then be executed against the vector store or SQL database, using the answer from the previous step as its parameter value. After processing the final question, you'll have the context needed to answer the original question with the LLM.

> **NOTE** If using a SQL database, you could simplify this with native SQL functions such as `AVG`. However, this example illustrates the concept rather than the optimal implementation.

While I won't provide a full implementation here, this method builds on advanced RAG techniques, such as the following:

- Routing a natural language question to the relevant content store

- Generating a query for specific content stores, such as SQL databases, from a natural language question

While LangChain doesn't offer a dedicated class for multi-step question decomposition, you may find inspiration in LlamaIndex's `MultiStepQueryEngine` class. Explore this class for further ideas.

This concludes our exploration of question transformation techniques for enhancing retrieval effectiveness. In the next chapter, you'll learn additional methods to improve RAG performance.

Summary

- Vague or overly complex questions produce poor retrieval results because vector stores match semantic similarity, not logical precision. Query optimization transforms unclear queries into effective retrieval inputs.
- Query rewriting (Rewrite-Retrieve-Read) uses an LLM to reformulate the user's question into clearer, more specific versions before searching. "Fix my code" becomes "How to resolve `ImportError: No module named pandas` in Python?"
- Multi-query retrieval generates three to five variations of the original question, retrieves documents for each variation, and merges results using Reciprocal Rank Fusion (RRF). This captures diverse phrasings of the same intent.
- The coarse-to-fine retrieval technique searches using high-level summaries (500-word chunks) to find relevant sections and then re-searches within those sections using fine-grained chunks (100-word chunks). This narrows the scope progressively.
- Step-back prompting generates a broader version of the question before retrieval to first retrieve foundational knowledge. "How do I configure SSL in Flask?" becomes "What is SSL, and how do web frameworks handle it?"
- Hypothetical Document Embeddings (HyDE) generates a fake answer to the user's question. It then searches for documents similar to that generated answer rather than the original question.
- Query decomposition breaks complex questions into independent subquestions. "What are the performance differences between PostgreSQL and MongoDB for time-series data?" splits into separate queries about each database's time-series capabilities.
- Reciprocal Rank Fusion (RRF) scores documents by summing 1 / (rank + k) across all query result lists, with k typically set to 60. Documents appearing in multiple lists score higher.
- Step-back prompting involves two retrieval steps: first retrieve docs for the broad question, then retrieve for the specific question, and combine both contexts in the final prompt.
- Test query optimization techniques on your specific dataset. `MultiQueryRetriever` works well for ambiguous questions; HyDE excels when queries are conceptually different from document phrasing.

10 Query generation, routing, and retrieval postprocessing

This chapter covers

- Generating metadata queries directly from user questions
- Converting user questions into database-specific queries (e.g., SQL, SPARQL)
- Routing questions to the appropriate handler based on intent
- Enhancing result relevance using Reciprocal Rank Fusion

In chapters 8 and 9, you improved Retrieval-Augmented Generation (RAG) answer accuracy using advanced indexing and query transformations. Optimizing indexing strengthens embedding effectiveness for broader chunks, adding richer context, while query transformations boost the precision of vector store retrieval.

Now we'll dive into three more advanced RAG techniques. First, you'll learn to generate queries specific to the type of content store in use. For instance, you'll see how to generate SQL from a user's natural language question to retrieve data from a relational database. Your setup might include several types of content stores—

such as vector stores, a relational database, or even a knowledge graph database. You'll use the large language model (LLM) to direct the user's question to the right content store. Finally, you'll refine the retrieved results to send only the most relevant content for synthesis, filtering out unnecessary data to maintain clarity and relevance.

10.1 Content database query generation

To give an LLM the best information possible for answering user questions, you often need to access other databases beyond your vector store. Many of these databases hold structured data and only accept structured queries. You might wonder if the difference between unstructured user questions and the structured queries required for these databases poses a problem. This gap can be bridged with an LLM's help. Let's look at common content stores in LLM applications and typical ways to retrieve data from them:

- *Vector store (or vector database)*—You're already familiar with vector stores, which hold document chunks along with their embeddings in a vector-based index to enable *semantic retrieval*. This approach, known as *dense retrieval*, uses embeddings—compact vectors with hundreds or thousands of dimensions that capture the semantic meaning of text. Similarity between a user query and stored chunks is computed based on the distance between these dense vectors. An alternative is *sparse retrieval* (also called *lexical retrieval*), which many vector databases support and which is based on word-level similarity. During the indexing phase, each document chunk is tokenized, and an inverted index is created to map every unique token to the list of chunks in which it appears. Note that this form of tokenization differs from the tokenization used by LLMs, as this tokenization is optimized for search and retrieval rather than language modeling. During querying, the user's question is tokenized in the same way, and each token is matched against the inverted index to retrieve relevant chunks. These chunks are then ranked using relevance scoring methods such as BM25 or Term Frequency-Inverse Document Frequency (TF–IDF), based on how well the term statistics of each chunk align with the query. Sparse search excels at precise, keyword-driven queries, supports Boolean logic (e.g., "must" and "must not"), and offers strong explainability by directly linking results to matching terms in the query.
- *Relational (SQL) database*—An LLM application can connect to a relational database, which stores structured facts in tables. Data is typically retrieved using SQL queries. For example, a database might contain seasonal temperatures at tourist resorts or lists of available hotels and car rentals by location. LLMs can assist by generating SQL queries from natural language questions—a technique known as text-to-SQL. This approach is gaining popularity as a way to make databases accessible to nontechnical users.

- *Document, key–value, or object databases*—These databases store data as documents or objects, typically in JSON or Binary JSON (BSON) format. Because many LLMs have been extensively trained on JSON structures, they can accurately convert user questions into JSON-based queries that align with the database schema. Notably, many document databases have recently been rebranded as vector databases after introducing support for vector fields—used to store embeddings—and adding vector search and similarity capabilities.
- *Knowledge graph databases*—Knowledge graphs represent data as a graph, where nodes correspond to entities and edges define their relationships. Originally popularized by companies such as Facebook and LinkedIn to infer connections in social networks, graph databases are now increasingly used in LLM applications. LLMs can transform unstructured text into structured knowledge graphs—a process often referred to as *knowledge graph–enhanced RAG*—resulting in a more compact and structured representation compared to vector stores. Once data is stored in a graph database, it can be queried using graph-specific languages such as SPARQL or Cypher, enabling complex reasoning and inference that go beyond simple similarity search. This forms the basis of *GraphRAG*. We'll explore how LLMs can assist in building these graph structures from raw text, generating SPARQL or Cypher queries, and finally converting query results back into natural language responses.

Now let's explore how a user's question is transformed into a structured query for different types of databases. We'll begin with the retrieval of document chunks from a vector store using metadata.

10.2 Self-querying (metadata query enrichment)

A vector store typically indexes document chunks by embedding for dense search, but it can also use keyword-based indexing. Here are a few ways this can be done:

- *Explicit metadata tags*—You can add metadata to each chunk, such as the timestamp, filename or URL, topic, and keywords. These keywords can come from user input or ones you assign manually.
- *Keyword extraction via algorithm*—You can use algorithms such as TF-IDF or its extension, BM25, to identify relevant keywords for each chunk based on word frequency and importance.
- *Keyword suggestions from the LLM*—You can ask the LLM to generate keywords for tagging each chunk during ingestion.

With keywords attached to chunks via any of these methods, you can perform a semantic search, focusing only on chunks filtered by a keyword-based (or *sparse*) search. If your chatbot's UI allows users to filter results directly—such as with dropdown options—your vector store query can explicitly include a metadata filter based on those selections. However, more commonly, you'll automate this filtering by inferring

relevant metadata from the user's question. This technique, known as *self-querying* or *self-metadata querying*, enables your application to automatically generate a query enriched with metadata filters based on the user's question.

In a self-querying flow, the user's original question is transformed into an enriched query with both a metadata filter and a semantic component, enabling a combined dense and sparse search, as illustrated in figure 10.1.

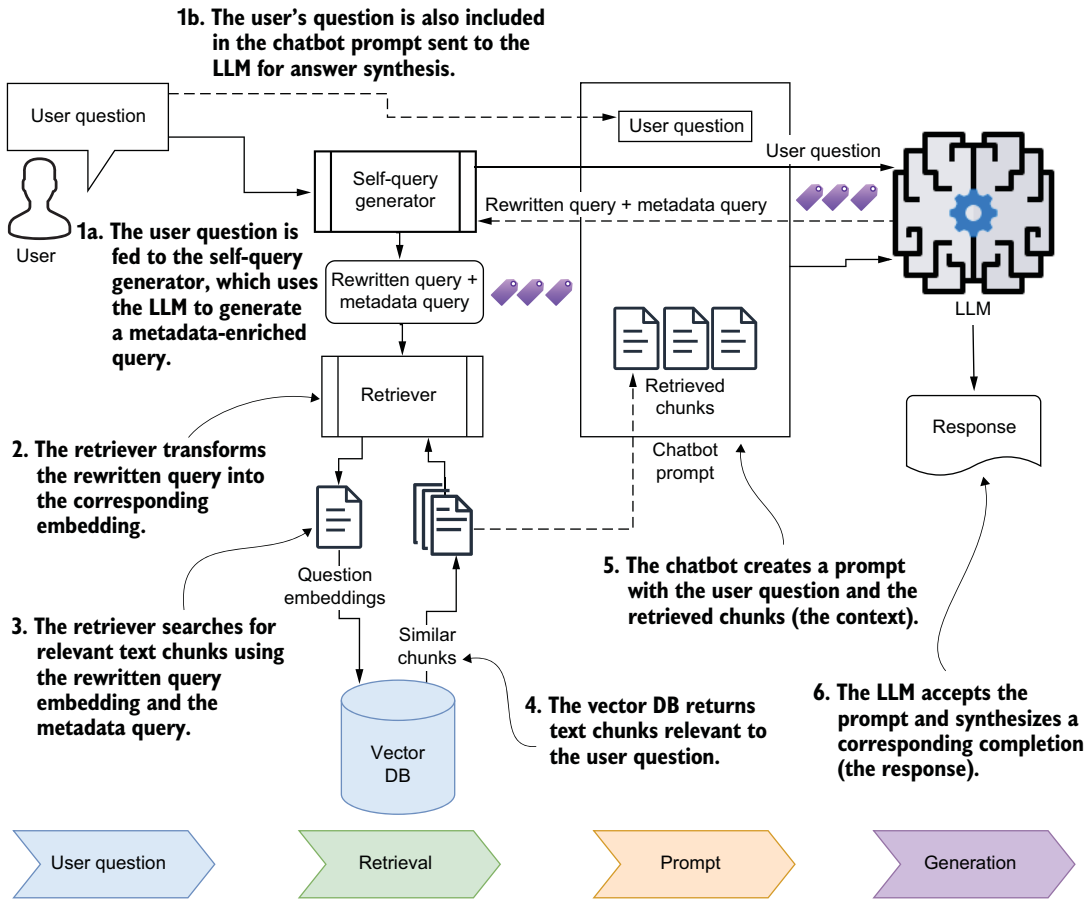


Figure 10.1 Self-querying workflow. The original question turns into a semantic query with an embedded metadata filter. When this enriched query runs on the vector store, it first selects chunks matching the metadata filter and then applies semantic search on this refined set.

Now that you understand the basics of self-metadata querying, let's go through the steps to implement it. We'll start with the ingestion phase, where you'll tag each chunk with relevant metadata keywords. Then, in the Q&A phase, I'll show you two

methods for generating self-metadata queries: one using the built-in `SelfQueryRetriever` and another that uses LLM function calling.

10.2.1 Ingestion: Metadata enrichment

To use metadata effectively, start by re-importing the UK tourist destination data into a new collection, this time storing metadata for each chunk. The following subsections show how to set up the environment.

INITIALIZING THE ENVIRONMENT

Open a new operating system shell, navigate to the chapter 10 code folder, activate the virtual environment, install the required packages, and create a new Jupyter Notebook:

```
C:\Github\building-llm-applications\ch10>
C:\Github\building-llm-applications\ch10>python -m venv env_ch10
c:\GitHub\building-llm-applications\ch10>env_ch10\Scripts\activate
(env_ch10) C:\Github\building-llm-applications\ch10>
➥pip install -r requirements.txt
(env_ch10) C:\Github\building-llm-applications\ch10>jupyter notebook
```

Then, in Jupyter Notebook, go to File > New > Notebook, and save the file as `10-query_generation.ipynb`.

DEFINING METADATA

Identify keywords to tag each chunk, such as the following:

- `source`—URL of the original content
- `destination`—The tourist destination referenced
- `region`—The UK region of the destination

Manually define mappings for destination and region, and then dynamically generate the source URL for each chunk.

SETTING UP THE CHROMADB COLLECTION

Configure the ChromaDB collection with the code shown in the following listing.

Listing 10.1 Setting up the ChromaDB collection

```
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings
import getpass

OPENAI_API_KEY = getpass.getpass('Enter your OPENAI_API_KEY')

uk_with_metadata_collection = Chroma(
    collection_name="uk_with_metadata_collection",
    embedding_function=OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY))

uk_with_metadata_collection.reset_collection()
```

In case it already exists

Defining the ingestion content and splitting strategy

Outline the content, and define a text splitting strategy to process the documents. The following listing shows how to set this up.

Listing 10.2 Defining ingestion content and splitting strategy

```
from langchain_community.document_loaders import AsyncHtmlLoader
from langchain_community.document_transformers import Html2TextTransformer
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_core.documents import Document

html2text_transformer = Html2TextTransformer()

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000, chunk_overlap=100
)

def split_docs_into_chunks(docs):
    text_docs = html2text_transformer.transform_documents(
        docs)
    chunks = text_splitter.split_documents(
        text_docs)

    return chunks

uk_destinations = [
    ("Cornwall", "Cornwall"), ("North_Cornwall", "Cornwall"),
    ("South_Cornwall", "Cornwall"), ("West_Cornwall", "Cornwall"),
    ("Tintagel", "Cornwall"), ("Bodmin", "Cornwall"),
    ("Wadebridge", "Cornwall"),
    ("Penzance", "Cornwall"), ("Newquay", "Cornwall"),
    ("St_Ives", "Cornwall"),
    ("Port_Isaac", "Cornwall"), ("Looe", "Cornwall"),
    ("Polperro", "Cornwall"),
    ("Porthleven", "Cornwall"),
    ("East_Sussex", "East_Sussex"), ("Brighton", "East_Sussex"),
    ("Battle", "East_Sussex"), ("Hastings_(England)", "East_Sussex"),
    ("Rye_(England)", "East_Sussex"), ("Seaford", "East_Sussex"),
    ("Ashdown_Forest", "East_Sussex")
]

wikivoyage_root_url = "https://en.wikivoyage.org/wiki"

uk_destination_url_with_metadata = [
    ( f'{wikivoyage_root_url}/{destination}', destination, region)
    for destination, region in uk_destinations]
```

Instantiates a relatively fine-chunk splitting strategy

Transforms HTML documents into clean text documents

Prepares metadata to be imported: URL, UK destination, and UK region

The next step is to ingest the content and the related metadata.

Ingesting content with metadata

Enrich the content with metadata by processing each document chunk, as shown in the following listing.

Listing 10.3 Enriching chunks with related metadata

```
for (url, destination, region) in uk_destination_url_with_metadata:
    html_loader = AsyncHtmlLoader(url)
    docs =  html_loader.load()

    docs_with_metadata = [
        Document(page_content=d.page_content,
        metadata = {
            'source': url,
            'destination': destination,
            'region': region})
        for d in docs]

    chunks = split_docs_into_chunks(docs_with_metadata)

    print(f'Importing: {destination}')
    uk_with_metadata_collection.add_documents(documents=chunks)
```

Loader for one destination

Documents (chunks) related to one destination

Now your collection is ready, with each document chunk enriched with metadata. You can query this content and apply metadata filters to refine search results based on keywords such as destination, region, or source.

10.2.2 Q&A on a metadata-enriched collection

There are three ways to query metadata-enriched content:

- *Explicit metadata filters*—Specify the metadata filter manually.
- `SelfQueryRetriever`—Automatically generate the metadata filter using the `SelfQueryRetriever`.
- *Structured LLM function call*—Infer the metadata filter with a structured call to an LLM function.

Let's explore these methods, starting with explicit filtering.

QUERYING WITH AN EXPLICIT METADATA FILTER

You can use the metadata attached to each chunk by explicitly adding a filter to the retriever. Here's an example:

```
question =  "Events or festivals"
metadata_retriever = uk_with_metadata_collection.as_retriever(
    search_kwargs={'k':2, 'filter':{'destination': 'Newquay'}})

result_docs = metadata_retriever.invoke(question)
```

When you print `result_docs`, you'll see that only chunks tagged with `'destination: Newquay'` are returned, confirming that the filter is working correctly:

```
[Document(metadata={'destination': 'Newquay', 'region': 'Cornwall', 'source':
'https://en.wikivoyage.org/wiki/Newquay'}, page_content="## Do\n\n[edit]\n\n
* Cornish Film Festival. Held annually for two weeks each November around
Newquay. (updated Jan 2024)\n  * 50.415741-5.0914781 Newquay Golf Club, Tower
```

```
Road, TR7 1LT, ☏ +44 1637 872091, info@newquaygolfclub.co.uk. 9AM-4PM. A
semi-private golf club established in 1890. Total yardage Championship: 6141,
Men: 5708, and Women: 5364. £31 for non-members. (updated Apr 2019)\n\n###
Beaches\n\n[edit]\n\nFistral Beach\n\nNewquay is well known as a surfer's
paradise. Therefore it offers plenty of\nbeaches:"),
 Document(metadata={'destination': 'Newquay', 'region': 'Cornwall', 'source':
'https://en.wikivoyage.org/wiki/Newquay'}, page_content="##
Eat\n\n[edit]\n\n### Budget\n\n[edit]\n\nThere are lots of cheap eats in the
town centre.\n\n  * 50.415513-5.0868851 Harbour Rest Cafe, 2 S Quay Hill.
(updated Feb 2023)\n  * 50.414042-5.0808662 Bunters, 15A East St. (updated
Feb 2023)\n  * 50.413988-5.0809823 Andy's Cafe, 15 East St. (updated Feb
2023)\n  * 50.413981-5.0802984 Oceans, 1bh, 34 East St. (updated Feb 2023)\n
* 50.41337-5.0862025 Loafers Sandwich Bar, 1A Gover Ln. (updated Feb 2023)\n
* 50.418831-5.0665726 Kao Hom Thai Food, Henver Rd. (updated Feb 2023)\n  *
50.41749-5.0641777 Oceans, 1bh, 34 East St. (updated Feb 2023)\n  *
50.417024-5.0644678 The Cornish Coffee Bean, 14, Chester Court, Chester Rd.
(updated Feb 2023)\n\n### Mid-range\n\n[edit]")]
```

To adjust the filter, instantiate a new retriever with the updated parameters.

AUTOMATICALLY GENERATING METADATA FILTERS WITH SELFQUERYRETRIEVER

You can also generate metadata filters automatically with `SelfQueryRetriever`. This tool interprets the user's question to infer the appropriate filter criteria. The underlying engine that performs this inference is, of course, the LLM—meaning this approach introduces additional cost and latency. First, import the necessary libraries, as shown in the following listing.

Listing 10.4 Setting up metadata field information

```
from langchain_classic.chains.query_constructor.base import AttributeInfo
from langchain_classic.retrievers.self_query.base
➥import SelfQueryRetriever
from langchain_openai import ChatOpenAI
```

Requires pip install lark

Next, define the metadata attributes to infer from the question:

```
metadata_field_info = [
    AttributeInfo(
        name="destination",
        description="The specific UK destination to be searched",
        type="string",
    ),
    AttributeInfo(
        name="region",
        description="The name of the UK region to be searched",
        type="string",
    )
]
```

Now set up the `SelfQueryRetriever` with the question, without specifying a manual filter:

```
question = "Tell me about events or festivals in the UK town of Newquay"

llm = ChatOpenAI(model="gpt-5-nano", openai_api_key=OPENAI_API_KEY)

self_query_retriever = SelfQueryRetriever.from_llm(
    llm, uk_with_metadata_collection, question,
    metadata_field_info, verbose=True
)
```

Invoke the retriever with the question:

```
result_docs = self_query_retriever.invoke(question)
```

Printing `result_docs` will confirm that only chunks related to Newquay are retrieved, matching the inferred filter:

```
[Document(metadata={'destination': 'Newquay', 'region': 'Cornwall', 'source':
'https://en.wikivoyage.org/wiki/Newquay'}, page_content="## Do\n\n[edit]\n\n
* Cornish Film Festival. Held annually for two weeks each November around
Newquay. [... REDUCED …] on the cliff above Towan Beach. Attached surf
school and backpackers bar. £10.50 with breakfast included.\n\n###
Mid-range\n\n[edit]"),
 Document(metadata={'destination': 'Newquay', 'region': 'Cornwall', 'source':
'https://en.wikivoyage.org/wiki/Newquay'}, page_content="###
Mid-range\n\n[edit]\n\n  * 50.41326-5.0855229 Concho Lounge,  [... REDUCED …]
town centre is home to a large number of pubs and bars."),
 Document(metadata={'destination': 'Newquay', 'region': 'Cornwall', 'source':
'https://en.wikivoyage.org/wiki/Newquay'}, page_content='* Newquay Tourist
Information Centre, ☏ +44 1637 854020.\n\n## [... REDUCED …] . Leave this
road near\nIndian Queens and continue on the A39 and then A392 which takes
you directly\ninto the town.\n\n### By train\n\n[edit]')]
```

GENERATING METADATA FILTERS WITH AN LLM FUNCTION CALL

You can also infer metadata filters by having the LLM map the question to a predefined metadata template with attributes you stored during ingestion. This approach offers greater flexibility than the `SelfQueryRetriever` but requires more setup. First, import the libraries necessary to create a structured query with specific filters, as shown in the following listing.

Listing 10.5 Importing libraries

```
import datetime
from typing import Literal, Optional, Tuple, List

from pydantic import BaseModel, Field
from langchain_classic.chains.query_constructor.ir import (
    Comparator,
    Comparison,
    Operation,
    Operator,
    StructuredQuery,
)
from langchain_classic.retrievers.self_query.chroma import ChromaTranslator
```

The `DestinationSearch` class translates the user question into a structured object with a `content_search` field containing the question (minus filtering details) and fields for inferred search filters. The following listing shows this setup.

Listing 10.6 Strongly typed structured question

```
class DestinationSearch(BaseModel):
    """Search over a vector database of tourist destinations."""

    content_search: str = Field(
        "",
        description="""Similarity search query applied
        to tourist destinations.""",
    )
    destination: str = Field(
        ...,
        description="The specific UK destination to be searched.",
    )
    region: str = Field(
        ...,
        description="The name of the UK region to be searched.",
    )

    def pretty_print(self) -> None:
        for field in self.__fields__:
            if getattr(self, field) is not None and getattr(
                self, field) != getattr(
                self.__fields__[field], "default", None
            ):
                print(f"{field}: {getattr(self, field)}")
```

Building a ChromaDB filter statement from the structured query

Next, create a function to convert a `DestinationSearch` object into a filter compatible with ChromaDB, as shown in the following listing.

Listing 10.7 Building a ChromaDB-specific filter statement

```
def build_filter(destination_search: DestinationSearch):
    comparisons = []

    destination = destination_search.destination
    region = destination_search.region

    if destination and destination != '':
        comparisons.append(
            Comparison(
                comparator=Comparator.EQ,
                attribute="destination",
                value=destination,
            )
        )
```

Gets destination and region from the structured query

If the destination exists, creates an "equality" operation

```
    if region and region != '':
        comparisons.append(
            Comparison(
                comparator=Comparator.EQ,
                attribute="region",
                value=region,
            )
        )

    search_filter = Operation(operator=Operator.AND,
                              arguments=comparisons)

    chroma_filter = ChromaTranslator().visit_operation(
        search_filter)

    return chroma_filter
```

If the region exists, creates an "equality" operation

Creates a combined search filter

Transforms the filter into Chroma format

BUILDING A QUERY CHAIN TO CONVERT THE QUESTION INTO A STRUCTURED QUERY

Now define the query generator chain to convert the user question into a structured query with metadata filters. The following listing shows how.

Listing 10.8 Query generator chain

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

system_message = """You are an expert at converting user
questions into vector database queries.
You have access to a database of tourist destinations.
Given a question, return a database query optimized
to retrieve the most relevant results.

If there are acronyms or words you are not familiar with,
do not try to rephrase them."""
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system_message),
        ("human", "{question}"),
    ]
)
llm = ChatOpenAI(model="gpt-5-nano", openai_api_key=OPENAI_API_KEY)
structured_llm = llm.with_structured_output(
    DestinationSearch, method="function_calling")

query_generator = prompt | structured_llm
```

Let's try out the chain with the same question used earlier:

```
question = "Tell me about events or festivals in the UK town of Newquay"

structured_query =query_generator.invoke(question)
```

Printing `structured_query` shows the question converted into a structured object:

```
DestinationSearch(content_search='events festivals',
➥destination='Newquay', region='Cornwall')
```

With the structured query created, generate a ChromaDB-compatible search filter:

```
search_filter = build_filter(structured_query)
```

The `search_filter` result will look like this:

```
{'$and': [{'destination': {'$eq': 'Newquay'}},
  {'region': {'$eq': 'Cornwall'}}]}
```

Perform the vector search using the generated structured query and ChromaDB filter:

```
search_query = structured_query.content_search

metadata_retriever = uk_with_metadata_collection.as_retriever(
    search_kwargs={'k':3, 'filter': search_filter})

answer = metadata_retriever.invoke(search_query)
```

The `answer` should closely match the output from the `SelfQueryRetriever`, returning chunks associated with Newquay:

```
[Document(metadata={'destination': 'Newquay', 'region': 'Cornwall', 'source':
'https://en.wikivoyage.org/wiki/Newquay'}, page_content="## Do\n\n[edit]\n\n
* Cornish Film Festival. Held annually for two weeks each   [... REDUCED …]
Therefore it offers plenty of\nbeaches:"), Document(metadata={'destination':
'Newquay', 'region': 'Cornwall', 'source': 'https://en.wikivoyage.org/wiki/
Newquay'}, page_content='## See\n\n[edit]\n\n  * 50.414578-5.0848411 Blue
Reef Aquarium, Towan Promenade, TR7 1DU (right next to Towan beach), ☏ +44
1637 878134. Although it is small, it is well worth checking out. It has an
octopus along with   [... REDUCED …]  November, 10:00-18:00. Small, but nice
Japanese garden. (updated Jul 2024)\n\n## Do\n\n[edit]'),
Document(metadata={'destination': 'Newquay', 'region': 'Cornwall', 'source':
'https://en.wikivoyage.org/wiki/Newquay'}, page_content="##
Eat\n\n[edit]\n\n### Budget\n\n[edit]\n\nThere are lots of   [... REDUCED …]
Chester Court, Chester Rd. (updated Feb 2023)\n\n### Mid-range\n\n[edit]")]
```

So far, you've focused on generating metadata-enriched queries for vector stores. In the next section, you'll learn how to generate SQL queries from natural language questions, enabling retrieval of structured data from relational databases.

10.3 Generating a structured SQL query

Many LLMs can transform user questions into SQL queries, enabling access to relational databases directly from LLM applications. While LLMs are continually improving in generating accurate SQL, challenges remain, especially when working with complex schemas or specific database structures. LangChain enhances these

capabilities with evolving text-to-SQL features, but there are some common issues you should consider.

A helpful reference here is the "Evaluating the Text-to-SQL Capabilities of Large Language Models" paper by Nitarshan Rajkumar et al. (https://arxiv.org/pdf/2204.00498.pdf). Though dated, this paper offers practical insights into common pitfalls and solutions. The main finding was that *hallucinations*—incorrect table and column names—can often be reduced by using few-shot prompts that include the schema and sample records for the target table. An example schema from the paper looks like this:

```
CREATE TABLE "state" (
    "state_name" TEXT,
    "population" INT DEFAULT NULL,
    "area" DOUBLE DEFAULT NULL,
    "country_name" VARCHAR(3) NOT NULL DEFAULT '',
    "capital" TEXT,
    "density" DOUBLE DEFAULT NULL
);
/* example rows
state_name     population     area      country_name
➥     capital       density
alabama        3894000        51700.0   usa
➥              montgomery    75.319149
alaska         401800         591000.0  usa
➥              juneau       0.679865
arizona        2718000        114000.0  usa
➥              phoenix      23.842105
*/
-- Answer the following question using the above table schema:
-- {user_question}
```

Using the `CREATE TABLE` command along with sample data helps the LLM better understand the structure and constraints, minimizing incorrect column and table references.

10.3.1 Installing SQLite

SQLite doesn't require full installation. Unzip the package, place it in a folder, and add the folder to your system's `Path` environment variable. Refer to appendix D for setup instructions on Windows. For other operating systems, consult the SQLite documentation.

10.3.2 Setting up and connecting to the database

Let's create a booking database called `UkBooking` to store UK destinations, accommodations, and special offers. Figure 10.2 shows the relational diagram for the `UkBooking` database. Each table shows its primary key (PK) and foreign key (FK) columns, with relationships marked by arrows connecting related tables. This setup visually represents the structure and relationships within the database between tables for destinations, accommodations, and offers.

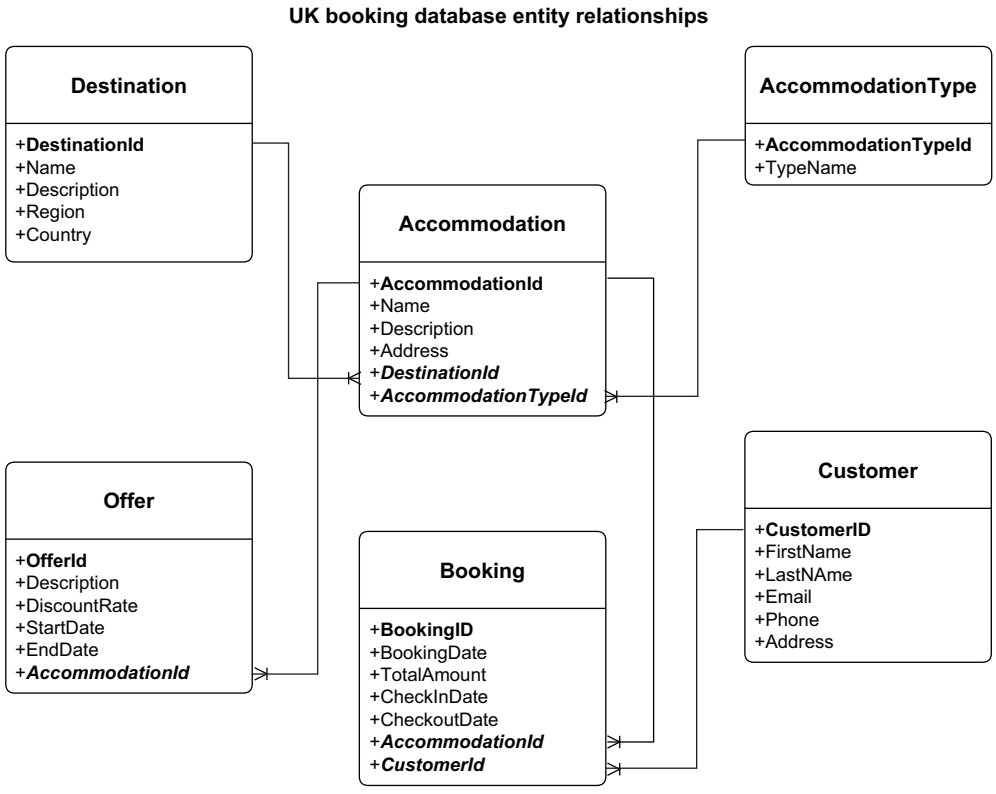


Figure 10.2 Entity-relationship diagram of the `UkBooking` database

Open your operating system shell, navigate to the code folder, and enter the following command to create the `UkBooking` database:

```
C:\Github\building-llm-applications\ch10>sqlite3 UkBooking.db
```

This opens the SQLite terminal:

```
SQLite version 3.46.1 2024-08-13 09:16:08 (UTF-16 console I/O)
Enter ".help" for usage hints.
sqlite>
```

In the SQLite terminal, load the SQL scripts to create and populate the `UkBooking` database. Ensure these files are in `C:\Github\building-llm-applications\ch10`. Download them from GitHub if necessary:

```
sqlite> .read CreateUkBooking.sql
sqlite> .read PopulateUkBooking.sql
```

To confirm the setup, check for records in the `Offer` table:

```
sqlite> SELECT * FROM Offer;
```

You should see output similar to this:

```
1|1|Summer Special|0.15|2024-06-01|2024-08-31
2|2|Weekend Getaway|0.1|2024-09-01|2024-12-31
3|3|Early Bird Discount|0.2|2024-05-01|2024-06-30
4|4|Stay 3 Nights, Get 1 Free|0.25|2024-01-01|2024-03-31
…
```

Now the `UkBooking` database is ready for use with LangChain.

Return to the Jupyter Notebook and import the libraries needed for SQL database connections:

```
from langchain_community.utilities import SQLDatabase
from langchain_community.tools import QuerySQLDataBaseTool
from langchain_classic.chains import create_sql_query_chain
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
import getpass
import os
```

Use the following code to connect to the database and list of available tables:

```
db = SQLDatabase.from_uri("sqlite:///UkBooking.db")
print(db.get_usable_table_names())
```

You should see a list of table names:

```
['Accommodation', 'AccommodationType', 'Booking', 'Customer', 'Destination',
'Offer']
```

Run a sample query to verify the connection:

```
db.run("SELECT * FROM Offer;")
```

The output should display entries from the `Offer` table:

```
"[(1, 1, 'Summer Special', 0.15, '2024-06-01', '2024-08-31'), (2, 2, 'Weekend
Getaway', 0.1, '2024-09-01', '2024-12-31'), [... SHORTENED]
```

Now you're set up to query the `UkBooking` database programmatically using LangChain.

10.3.3 Generating SQL queries from natural language

Now that the setup is complete, you can start generating SQL queries directly from natural language questions. Here's how to test a simple query:

```
llm = ChatOpenAI(openai_api_key=OPENAI_API_KEY, model="gpt-4.1")
sql_query_gen_chain = create_sql_query_chain(llm, db)
response = sql_query_gen_chain.invoke(
    {"question":
     "Give me some offers for Cardiff, including the hotel name"})
```

Printing `response` will show the generated SQL query:

```
'```sql\nSELECT "Offer"."OfferDescription", "Offer"."DiscountRate",
"Accommodation"."Name" \nFROM "Offer" \nJOIN "Accommodation" ON
"Offer"."AccommodationId" = "Accommodation"."AccommodationId" \nJOIN
"Destination" ON "Accommodation"."DestinationId" =
"Destination"."DestinationId" \nWHERE "Destination"."Name" = \'Cardiff\'
\nLIMIT 5;\n```'
```

However, if you attempt to execute this SQL directly against the database, you'll encounter an error due to the backticks (```), which are non-SQL characters:

```
db.run(response)

'Error: (sqlite3.OperationalError) near "```sql\nSELECT
"Offer"."OfferDescription",
```

To clean up the SQL formatting, you can use the LLM to strip unnecessary characters and output a properly formatted SQL statement. The following listing is a simple chain setup for this.

Listing 10.9 Chain to fix the formatting of the generated SQL

```
clean_sql_prompt_template = """You are an expert in SQLite.
You are asked to fix badly formed SQLite queries,
which might contain unneeded prefixes or suffixes.
Given the following unclean SQL statement,
transform it to a clean,
executable SQL statement for SQLite.
Always prefix column names with the table name.
Only return an executable SQL statement which terminates
with a semicolon. Do not return anything else.
Do not include the language name or symbols like ```.

Unclean SQL: {unclean_sql}"""

clean_sql_prompt = ChatPromptTemplate.from_template(
    clean_sql_prompt_template)

clean_sql_chain = clean_sql_prompt | llm

full_sql_gen_chain = sql_query_gen_chain | \
   clean_sql_chain | StrOutputParser()
```

Let's try out this full chain with a sample question and verify the output:

```
question = """Give me some offers for Cardiff,
including the accommodation name"""

response = full_sql_gen_chain.invoke({"question": question})

print(response)
```

The output should be a clean SQL statement:

```
"SELECT Offer.OfferDescription, Offer.DiscountRate, Accommodation.Name \nFROM
Offer \nJOIN Accommodation ON Offer.AccommodationId =
Accommodation.AccommodationId \nJOIN Destination ON
Accommodation.DestinationId = Destination.DestinationId \nWHERE
Destination.Name = 'Cardiff' \nLIMIT 5;"
```

This approach ensures that the SQL statement is correctly formatted and ready to execute against the database.

10.3.4 Executing the SQL query

Now let's create a chain to generate and execute SQL queries.

```
sql_query_exec_chain = QuerySQLDataBaseTool(db=db)

sql_query_gen_and_exec_chain = full_sql_gen_chain \
    | sql_query_exec_chain | StrOutputParser()

response = sql_query_gen_and_exec_chain.invoke(
    {"question":question})
```

Printing `response` should show the following output:

```
"[('Early Bird Discount', 0.2, 'Cardiff Camping')]"
```

This setup allows you to retrieve data from a relational database by using a combined chain (`sql_query_gen_and_exec_chain`) that handles both SQL generation and execution. You can easily integrate this chain within a broader RAG setup, as discussed in earlier sections. The sequence diagram in figure 10.3 gives you a visual idea of what the full RAG with SQL workflow would look like. Try extending this integration as an exercise.

> **TIP** LangChain's `SQLDatabaseChain` class provides a streamlined way to generate SQL queries directly from user questions. This tool uses an LLM and your database connection to automatically create few-shot prompts, similar to those recommended in the Rajkumar paper. Experimenting with `SQLDatabaseChain` can be highly beneficial if you plan to incorporate relational databases into your RAG setup.

10.4 Generating a semantic SQL query

In the previous section, you learned how to generate SQL queries from natural language. However, these queries rely on strict SQL, meaning they depend on exact matching and traditional relational operations. Relational databases operate on record sets using operations such as `SELECT`, `JOIN`, `WHERE`, and `GROUP BY`, where filters are based on exact string matches or numeric comparisons.

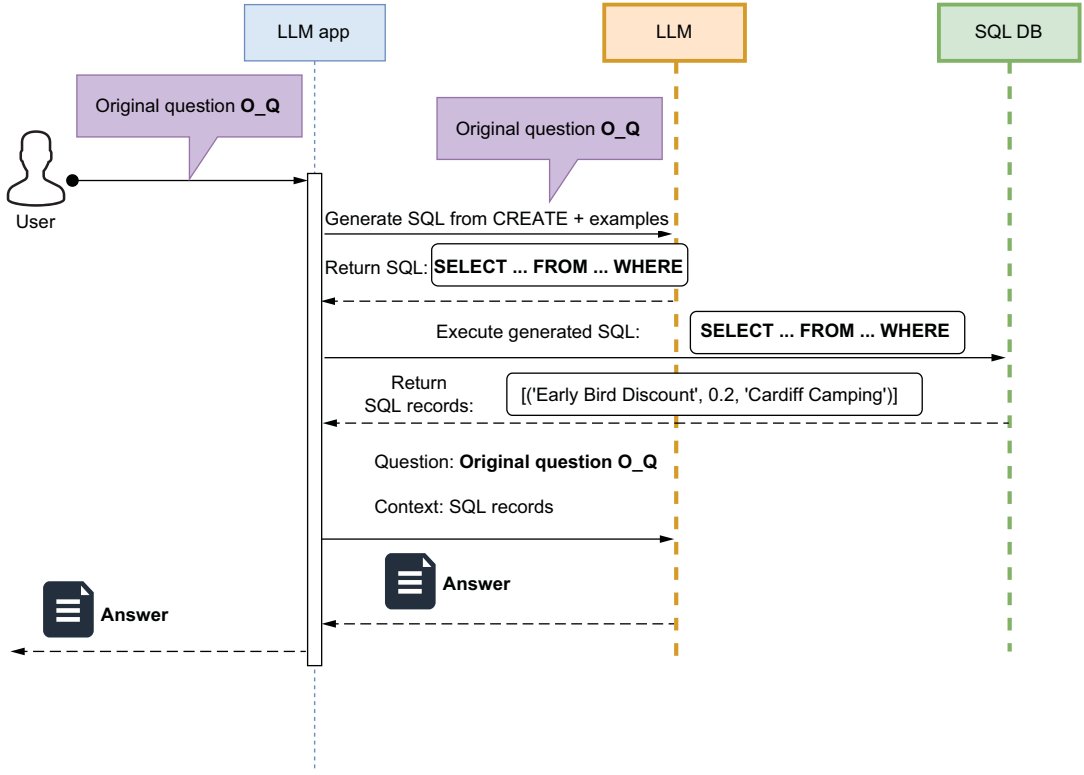


Figure 10.3 RAG with SQL workflow. The LLM converts the natural language question into a SQL query, which is executed on the SQL database. The database returns records that are then processed by the LLM to generate the final answer.

But what if you want to expand the SQL search to include results that are similar in meaning to what the user intended? This requires a shift from standard SQL to a *semantic SQL search*. In this section, I'll provide an overview of how to implement semantic SQL search, a topic that continues to evolve.

10.4.1 Standard SQL query

A standard SQL query filters based on exact matches. For example, to find users with the first name Roberto, you would use this:

```
SELECT first_name, last_name FROM user WHERE first_name = 'Roberto'
```

This query returns only users named Roberto. It won't return records for Robert, Rob, Robbie, Roby, Robin, Roe, Bobby, Bob, or Bert.

You can loosen this search slightly with the `LIKE` operator for partial matching. For instance, to find users with names that start with "Rob," you'd do this:

```
SELECT first_name, last_name FROM user WHERE first_name LIKE 'Rob%'
```

This query will return Roberto, Robert, Rob, Robbie, Roby, and Robin but still won't catch variations such as Roe, Bobby, Bob, or Bert, as they don't contain the string "Rob."

10.4.2 Semantic SQL query

With the rise of LLMs, several relational databases now support semantic search, which enables searches based on embeddings instead of exact matches. An example is *pgvector*, an extension for PostgreSQL that allows vector-based similarity searches using metrics such as Euclidean or cosine distance. This approach enables you to perform searches that return results based on meaning rather than exact text matches. In this section, I'll refer to this approach as semantic SQL search or SQL similarity search interchangeably.

10.4.3 Creating the embeddings

To extend the traditional SQL approach with pgvector's similarity search, you'll need to add vector-based embeddings for any columns you want to use in semantic searches. Here's how to do it:

1. *Add a vector column*. First, add a `VECTOR` type column to the table for each field you want to search by similarity. For example, to enable similarity search on `first_name`, add a column called `first_name_embedding`:

   ```
   ALTER TABLE user  ADD COLUMN first_name_embedding VECTOR
   ```

2. *Calculate embeddings*. Next, compute the embedding values for each `first_name`. You can do this directly within PostgreSQL if you have a function to generate embeddings, or you can compute embeddings externally using an API client such as LangChain:
   - *In-database calculation*—If you supply PostgreSQL with a custom function such as `calculate_my_embedding()` available, you can update the embeddings in place with SQL:

     ```
     UPDATE user
     SET first_name_embedding = calculate_my_embedding(first_name)
     ```

   - *External calculation with LangChain*—If you're using a prebuilt embedding function (e.g., OpenAI's), calculate embeddings externally, and store them using the pgvector API. In listing 10.10, you can see an example of using LangChain's `OpenAIEmbeddings` wrapper to generate embeddings for `first_name` values and update the database (library imports are omitted for brevity).

Listing 10.10 Using LangChain's `OpenAIEmbeddings` wrapper

```
db = SQLDatabase.from_uri(
    YOUR_DB_CONNECTION_STRING)
embeddings_model = OpenAIEmbeddings()
```

Instantiates the database client and embeddings model

```
first_names_resultset_str = db.run('SELECT first_name FROM user')
first_names = [fn[0] for fn in eval(
    first_names_resultset_str)]

first_names_embeddings = embeddings_model.embed_documents(
   first_names)
fn_emb = zip(first_names,
    first_names_embeddings)

for fn, emb in fn_emb:
    sql = f'UPDATE user SET first_name_embeddings =
➥ARRAY{emb} WHERE first_name ="{fn}"'
    db.run(sql)
```

Extracts a list of strings from the SQL result string

Calculates the embedding of each first name

Associates the first names with the related embeddings

By following these steps, you'll enable semantic search on `first_name` or other fields, allowing pgvector to retrieve records based on similarity, rather than exact matches.

10.4.4 Performing a semantic SQL search

After setting up the embeddings (and indexing the related column to guarantee adequate performance on big datasets), you can perform a similarity search as follows:

```
embedded_query= embeddings_model.embed_query("Roberto")
query = (
    'SELECT first_name FROM user WHERE first_name_embeddings IS NOT NULL
    ➥ ORDER BY first_name_embeddings <-> "{embedded_query}"'
)
db.run(query)
```

This query returns variations of Roberto, including Roe, Bobby, Bob, and Bert, by ordering results based on similarity.

10.4.5 Automating semantic SQL search

Now that you understand how to generate embeddings and perform similarity searches in a SQL database, the final step is to create a prompt that can automatically generate SQL similarity queries. This process is similar to what we covered for generating traditional SQL queries. Once you design, implement, and test this prompt—and integrate it into a full chain within LangChain Expression Language (LCEL)—your LLM application will be capable of generating semantic searches on pgvector or any SQL database that supports `ARRAY` (or similar) data types, seamlessly feeding the results to the LLM for synthesis.

10.4.6 Benefits of a semantic SQL search

The simple example here only scratches the surface of semantic SQL's capabilities. You can combine semantic filtering with exact matching or use multiple semantic filters, which are especially powerful in multi-table queries using joins. This approach allows highly nuanced searches, especially when combined with traditional SQL filtering.

Later, I'll show you how to combine metadata and semantic filtering in a vector store, which can achieve similar results. However, using multiple semantic filters in SQL offers greater flexibility, particularly for complex queries.

10.5 Generating queries for a graph database

Graph databases are designed to store, navigate, and query data in a graph format, with nodes representing entities and edges defining relationships, as shown in figure 10.4. They are well-suited for building knowledge graphs, making them ideal for specialized domains that require advanced reasoning, inference, and explainability.

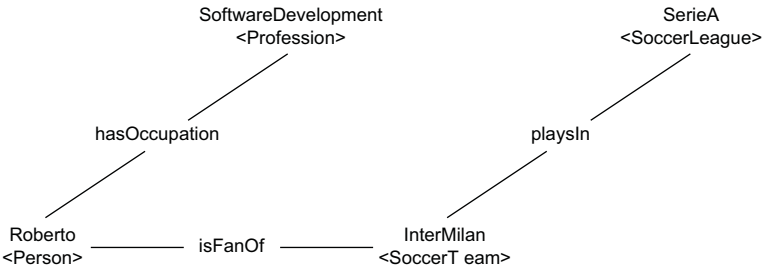


Figure 10.4 Graph representation of data. Nodes represent entities such as `Roberto` and `InterMilan`, while edges such as `hasOccupation` and `isFanOf` depict their relationships.

Unlike relational databases, graph databases don't follow a universal standard. Some use the Resource Description Framework (RDF) to represent data as *triples*—a subject-predicate-object format. For instance, in RDF, you get this:

```
(Roberto, hasOccupation, SoftwareDeveloper)
(Roberto, isFanOf, InterMilan)
(InterMilan, playsIn, SerieA)
```

This triple structure might look like:

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:Roberto rdf:type ex:Person .
ex:Roberto ex:hasOccupation ex:SoftwareDevelopment .
ex:Roberto ex:isFanOf ex:InterMilan .

ex:InterMilan rdf:type ex:SoccerTeam .
ex:InterMilan ex:playsIn ex:SerieA .

ex:SerieA rdf:type ex:SoccerLeague .
```

However, only some graph databases use RDF. Others use proprietary graph representations and query languages such as Cypher (for Neo4j) or Gremlin, rather than RDF and SPARQL.

As you can see in figure 10.4, shown previously, the graph data structure enables powerful, flexible representations of relationships that are difficult to capture with traditional databases. This versatility makes graph databases ideal for knowledge-intensive applications where deep understanding of relationships and reasoning is essential.

Graph databases have been around since the early 2000s, with Neo4j among the first. While they offer powerful, flexible ways to represent and query information, their complexity can be a hurdle. Recently, LLMs have made graph databases more accessible by enhancing several key functions:

- *Entity and relationship extraction*—LLMs can pull entities, relationships, and even full graphs from unstructured text, allowing you to store this data directly in a graph database.
- *Automated query generation*—LLMs can generate complex Cypher or SPARQL queries from natural language questions, easing a task traditionally challenging for less experienced developers. To accomplish this, use a carefully crafted few-shot prompt with examples, which works best with a high-accuracy LLM such as GPT-5.
- *Natural language answers*—LLMs can convert query results (e.g., in RDF) into natural language responses. This process involves feeding the initial question, the generated Cypher or SPARQL query, and the results into a dedicated prompt, which a lower-cost LLM such as GPT-5-nano can handle effectively.

Figure 10.5 shows the resulting Knowledge Graph RAG (KG-RAG) architecture, which closely resembles the setup used for vector store–based RAG.

LangChain supports several graph databases, including Neo4j and Amazon Neptune. While this book doesn't cover KG-RAG in detail, I recommend consulting LangChain's documentation and examples.

Following is a sample prompt template for generating Cypher queries, taken from LangChain's Neo4j QA chain:

```
CYPHER_GENERATION_TEMPLATE = """Task:Generate Cypher
statement to query a graph database.
Instructions:
Use only the provided relationship types and properties in the schema.
Do not use any other relationship types or properties that are not provided.
Schema:
{schema}
Note: Do not include any explanations or apologies in your responses.
Do not respond to any questions that might ask anything else
than for you to construct a Cypher statement.
Do not include any text except the generated Cypher statement.
Examples: Here are a few examples of generated Cypher
for particular questions:
# How many people played in Top Gun?
MATCH (m:Movie {{title:"Top Gun"}})<-[:ACTED_IN]-()
RETURN count(*) AS numberOfActors
The question is:
{question}"""
```

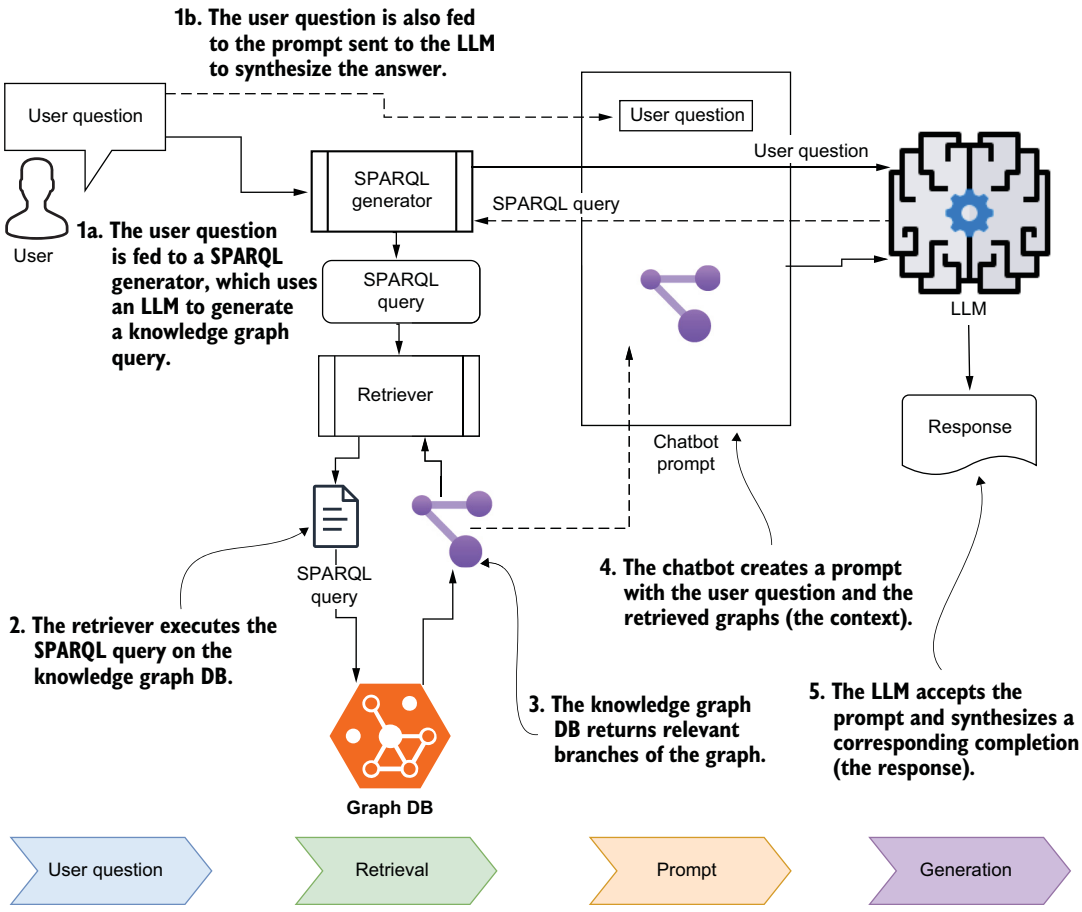


Figure 10.5 The Knowledge Graph RAG architecture (KG-RAG) is similar to vector store–based RAG setups, but it uses a SPARQL generator. The generator converts a natural language question into SPARQL, which is executed on the knowledge graph database. The retrieved graph data is then provided to the LLM, along with the original question, to synthesize the answer.

Graph databases are evolving to meet new LLM-driven use cases, giving rise to *knowledge graph embeddings*. This approach enriches knowledge graphs with textual descriptions and embeddings, supporting semantic search as a complement to traditional graph queries. These tools allow you to use the structured knowledge of graph databases, combined with the adaptability of LLMs, for powerful retrieval-augmented generation solutions.

> **NOTE** For further reading on knowledge graph embeddings, see "A Type-Augmented Knowledge Graph Embedding Framework for Knowledge Graph Completion" (www.nature.com/articles/s41598-023-38857-5). For a comprehensive guide, I recommend *Essential GraphRAG: Knowledge Graph–Enhanced RAG* by Tomaž Bratanič and Oskar Hane (Manning, 2025).

10.6 Chain routing

An application's content might reside in multiple storage types, not just vector stores for unstructured text. You may also use relational databases for structured data, document databases for semi-structured content, and knowledge graph databases for entity relationships. Additionally, the application might need to connect to different LLMs depending on the task, as some LLMs are optimized or more cost-effective for specific functions. As a result, the RAG architecture often branches into a tree structure, with each branch tailored to specific types of queries or tasks, as shown in figure 10.6.

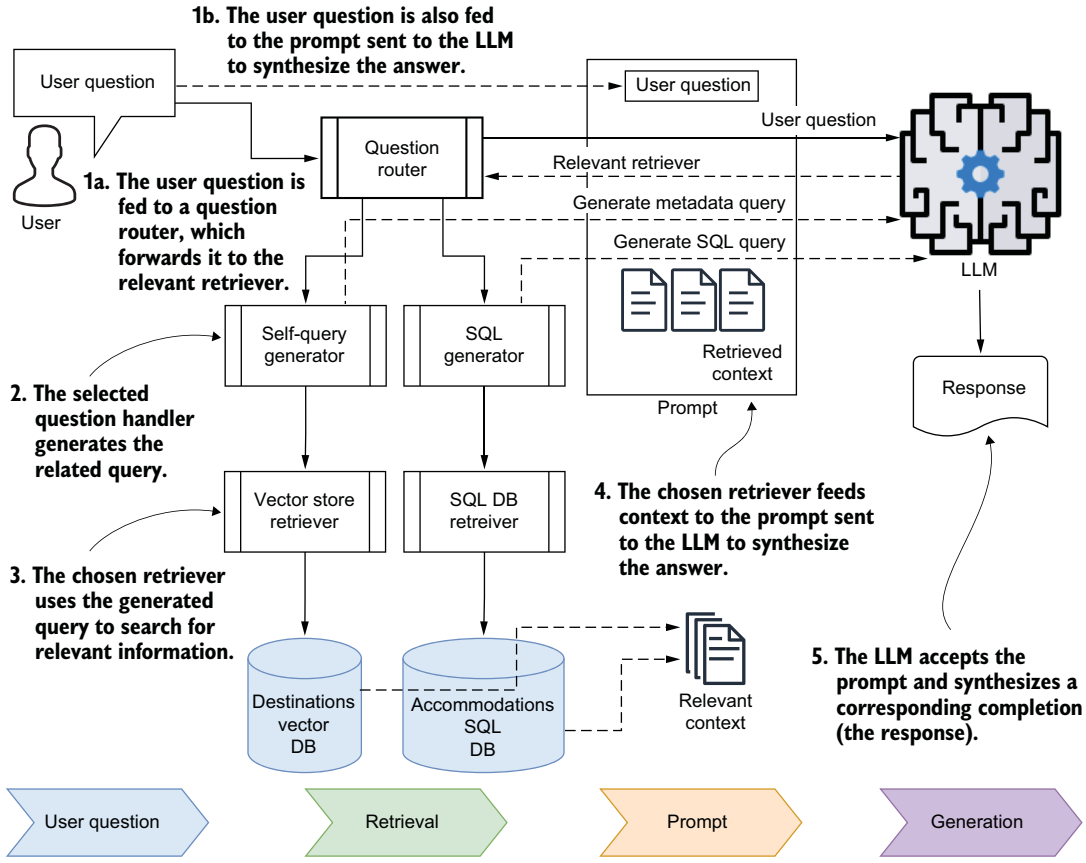


Figure 10.6 Complex RAG architecture with branching pathways, each optimized for specific tasks, such as answering questions about tourist destinations (from a vector store) or accommodation offers (from a relational SQL database)

This figure illustrates a RAG setup with several branches, each designed to handle different application tasks. For example, one branch could address factual questions about tourist destinations, while another branch handles questions about available accommodation offers.

Suppose a user asks about a tourist destination. In that case, you'd likely route this query to a RAG chain based on a vector store. If the user's question is about accommodation offers, you might route it to a RAG chain connected to the `UkBooking` database introduced earlier.

To route each question to the correct chain, you use a routing chain. This chain analyzes the question to determine the best-suited chain for handling it. Let's go over the implementation of a routing chain for this purpose.

10.6.1 Setting up data retrievers

To streamline setup, we'll reuse the vector store and relational database configurations from previous sections. Import the necessary libraries:

```
from typing import Literal
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from pydantic import BaseModel, Field
from langchain_core.runnables import RunnableLambda
```

Now create the corresponding retriever chains:

```
tourist_info_retriever_chain = RunnableLambda(
    lambda x: x['question']) \
       | uk_with_metadata_collection.as_retriever(
           search_kwargs={'k':2})

uk_accommodation_retriever_chain =  full_sql_gen_chain \
    | sql_query_exec_chain | StrOutputParser()
```

These retriever chains direct questions to the appropriate data source. Next, we'll build a router to direct user questions to one of these retriever chains.

10.6.2 Setting up the query router

We'll implement a question router using an LLM. The LLM will analyze each question and determine the best retriever chain based on its content. The prompt will specify the function of each retriever: the vector store for general tourist information and the relational database for accommodation bookings. The router function, defined in listing 10.11, binds the LLM's response to a typed object, instantiating the `datasource` attribute with either `"tourist_info_store"` or `"uk_booking_db"` depending on the question's intent.

Listing 10.11 Routing the query to the correct retriever

```
class RouteQuery(BaseModel):
    """Route a user question to the most relevant datasource."""

    datasource: Literal["tourist_info_store",
        "uk_booking_db"] = Field(
        ...,
        description="""Given a user question,
        route it either to a tourist info vector store
```

```
        or a UK accommodation booking relational database.""",
    )

llm = ChatOpenAI(openai_api_key=OPENAI_API_KEY, model="gpt-5-nano")
structured_llm_router = llm.with_structured_output(
    RouteQuery)
system = """You are an expert at routing a user question
to a tourist info vector store
or to an UK accommodation booking relational database.
The vector store contains tourist information about UK destinations.
Use the vector store for general tourist information questions
on UK destinations.
For questions about accommodation availability or booking,
use the UK Booking database."""
route_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system),
        ("human", "{question}"),
    ]
)

question_router = route_prompt | structured_llm_router
```

Structured router that uses LLM function calls

This setup enables the LLM to intelligently route each question to the appropriate data source, improving response accuracy based on question intent.

TESTING THE ROUTER CHAIN

Let's test the router chain with a question about tourist information and another about accommodation booking:

```
selected_data_source = question_router.invoke(
    {"question": "Have you got any offers in Brighton?"}
)

print(selected_data_source)
```

Expected output:

```
datasource='uk_booking_db'
```

Then, test with a tourist-related question:

```
selected_data_source = question_router.invoke(
    {"question": "Where are the best beaches in Cornwall?"}
)

print(selected_data_source)
```

Expected output:

```
datasource='tourist_info_store'
```

The router correctly identifies the appropriate data source!

SETTING UP THE RETRIEVER CHOOSER

Now let's implement a function to select the correct retriever based on the chosen data source (`'uk_booking_db'` or `'tourist_info_store'`). The following listing shows you how.

Listing 10.12 Retriever chooser function

```
retriever_chains = {
    'tourist_info_store': tourist_info_retriever_chain,
    'uk_booking_db': uk_accommodation_retriever_chain
}

def retriever_chooser(question):
    selected_data_source = question_router.invoke(
        {"question": question})

    return retriever_chains[selected_data_source.datasource]
```

Let's test the retriever chooser function with a sample question:

```
chosen = retriever_chooser("""Tell me about events
or festivals in the UK town of Newquay""")

print(chosen)
```

Expected output:

```
first=RunnableLambda(lambda x: x['question'])
➥ last=VectorStoreRetriever(tags=['Chroma', 'OpenAIEmbeddings'],
➥ vectorstore=<langchain_chroma.vectorstores.Chroma object at
➥ 0x0000022799116010>, search_kwargs={'k': 2})
```

The output confirms that the correct retriever chain instance is selected based on the question's intent. This setup ensures the question is routed to the best-matched data source for accurate results.

10.6.3 Integrating the chain router into a full RAG chain

The final step is to integrate the chain router into a complete RAG chain, enabling the workflow of retriever selection, query execution, and answer synthesis. See the following listing for an example.

Listing 10.13 Full RAG chain for routing, retrieval, and synthesis

```
from langchain_core.runnables import RunnablePassthrough

rag_prompt_template = """
Given a question and some context, answer the question.
If you get a structured context, like a tuple, try to
infer the meaning of the components:
typically they refer to accommodation offers,
and the number is a percentage (0.2 means 20%).
If you do not know the answer, just say I do not know.
```

```
Context: {context}
Question: {question}
"""

rag_prompt = ChatPromptTemplate.from_template(rag_prompt_template)

def execute_rag_chain(question, chosen_retriever):
    full_rag_chain = (
        {
            "context": {"question": RunnablePassthrough()}
                | chosen_retriever,
            "question": RunnablePassthrough(),
        }
        | rag_prompt
        | llm
        | StrOutputParser()
    )

    return full_rag_chain.invoke(question)
```

The context is returned by the retriever after feeding to it the rewritten query.

This is the original user question.

Let's test the RAG chain with both an accommodation query and a tourist information query.

EXAMPLE: ASKING ABOUT ACCOMMODATION OFFERS

In this example, we test how the RAG chain responds when the user asks for accommodation-related information:

```
question = """Give me some offers for Cardiff,
including the accommodation name"""

chosen_retriever = retriever_chooser(question)

answer = execute_rag_chain(question, chosen_retriever)
```

Expected output:

```
One offer for Cardiff is the "Early Bird Discount" at Cardiff Camping, which
provides a 20% discount.
```

EXAMPLE: ASKING ABOUT TOURIST INFORMATION

This example demonstrates how the RAG chain behaves when the query concerns general tourist information rather than accommodation:

```
question_2 = """Tell me about events or festivals
in the UK town of Newquay"""

chosen_retriever_2 = retriever_chooser(question_2)

answer2 = execute_rag_chain(question_2, chosen_retriever_2)
```

Expected output:

```
In Newquay, the **Cornish Film Festival** is held annually each November. It
is a notable event that celebrates film in the region. Additionally, Newquay
```

```
is known for being the UK's surfing capital, with various surfing events,
including the UK championships and the Boardmasters festival, taking place in
the area.
```

In both cases, the RAG chain correctly routes each question to the appropriate retriever, performs the retrieval, and synthesizes a coherent answer by feeding the context and original question to the LLM.

We're nearing the end of our exploration of advanced RAG techniques. One more topic remains: *retrieval postprocessing*, which focuses on refining the chunks sent as context to the LLM, further optimizing the relevance and clarity of responses.

10.7 Retrieval postprocessing

After using the techniques discussed so far to improve the effectiveness and accuracy of RAG retrieval, you'll likely have a list of document chunks (or nodes) from the content store. Before passing these to the LLM for answer synthesis, you may want to perform postprocessing to filter out less relevant content, ensuring the LLM produces a concise and accurate response, as shown in figure 10.7.

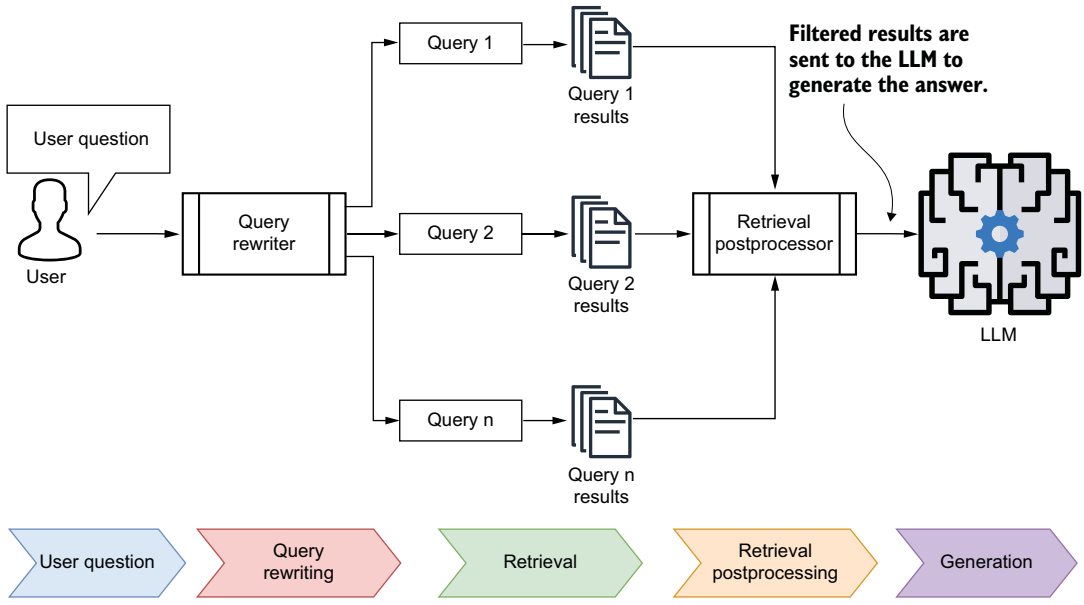


Figure 10.7 Retrieval postprocessing. Retrieved chunks from the vector store are filtered to remove irrelevant content, ensuring only high-quality chunks are sent to the LLM for answering the user's question.

In the following sections, I'll introduce some key postprocessing techniques. Let's start with similarity postprocessors.

10.7.1 Similarity postprocessors

A straightforward way to reduce the number of chunks returned by a similarity retriever (often a vector-based retriever using semantic distance) is to apply a cutoff to similarity scores. Chunks that are below a specific similarity score or that are above a certain distance are discarded.

In LangChain, you can set this similarity threshold before executing the search by instantiating a *score threshold* similarity retriever from the vector store. Set the `search_type` to `"similarity_score_threshold"`, and specify the threshold in `search_kwargs`:

```
score_threshold_similartity_retriever = vector_store.as_retriever(
    search_type="similarity_score_threshold",
    search_kwargs={"score_threshold": 0.6}
)
```

After instantiating this retriever, you can execute the search with its `get_relevant_documents()` method:

```
doc_chunks = score_threshold_similartity_retriever
➥.get_relevant_documents("What are the best beaches in Cornwall?")
```

This retrieves only documents with similarity scores above the specified threshold, ensuring that only highly relevant content is passed to the LLM.

10.7.2 Keyword postprocessors

Another postprocessing approach is to filter retrieved document chunks by keywords. This allows you to include or exclude chunks based on the presence of specific terms.

While LangChain doesn't provide a built-in keyword postprocessor, you can implement one in Python as follows (for demonstration purposes):

```
selected_chunks = [c for c in chunks
             if set(c.split()).intersection(required_keywords)
            and not set(c.split()).intersection(excluded_keywords)]
```

This code filters `chunks` to include only those containing `required_keywords` and excludes any that contain `excluded_keywords`.

10.7.3 Time weighting

You might also want to prioritize chunks based on how recently they were accessed. To do this, include a `last_accessed_at` timestamp in each chunk's metadata, and update it with each access:

```
[Document(page_content='this is some content of a chunk',
➥metadata={'last_accessed_at': datetime.datetime(2024, 01, 02, 14, 18,
➥22, 53225), 'created_at': datetime.datetime(2023, 12, 11, 11, 21, 12,
➥55466), 'buffer_idx': 1})]
```

Once timestamps are in place, you can use a `TimeWeightedVectorStoreRetriever`, which applies a time decay factor to rank chunks by both similarity score and recency:

```
retriever = TimeWeightedVectorStoreRetriever(
    vectorstore=vectorstore,
    decay_rate=1e-10, k=3
)
```

The `TimeWeightedVectorStoreRetriever` adjusts similarity scores based on recency:

```
adjusted_similarity_score = similarity_score
➥ + (1.0 - decay_rate) ** hours_passed
```

This adjusted score allows recent, relevant content to rank higher, ensuring more timely responses. Next, we'll discuss one of the most essential postprocessing techniques.

10.7.4 RAG fusion (Reciprocal Rank Fusion)

In the previous chapter, we discussed multiple query generation, where multiple queries are generated from a user's question, and a subset of relevant results is selected from the combined results across all queries. Using LangChain's `MultiQueryRetriever`, we automated this process to select the most relevant answers. However, if you want finer control over result ranking, consider the *Reciprocal Rank Fusion (RRF) approach.*

RRF, detailed by Cormack, Clarke, and Buttcher in their paper "Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods" (https://mng.bz/Dwe9), reranks retrieved documents based on a specific scoring formula of

```
rrfscore = 1 / (rank + k)
```

where

- `rank` is the document's current rank based on similarity or relevance.
- `k` is a smoothing constant to control the weight of existing ranks.

As each document is processed, its RRF score accumulates across all generated queries. Once all scores are calculated, documents are reranked by their cumulative RRF scores, and the top-ranked results are sent to the LLM for answer synthesis. You can get a visual idea of how the overall process works by following through figure 10.8.

At a high level, implementing RAG fusion involves generating multiple queries, as shown in chapter 9, and ranking the results with the RRF algorithm. This retrieval workflow can then be embedded within a larger RAG chain, as demonstrated in previous examples.

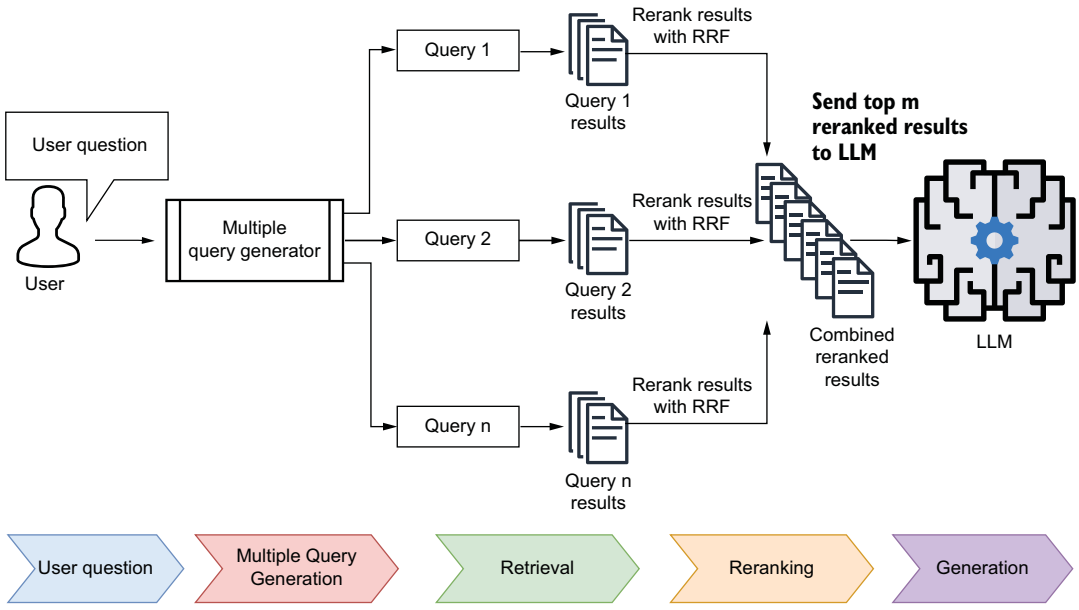


Figure 10.8 Reciprocal Rank Fusion workflow. Multiple queries are generated from the initial user question. Each query retrieves a set of results, such as from a vector store. All results are then reranked using RRF scores, with only the top results sent to the LLM for final answer synthesis.

GENERATING MULTIPLE QUERIES

You can generate multiple queries from an initial question using a chain, as shown in chapter 8. For convenience, the setup is repeated here.

Listing 10.14 Multi-query generation

```
from langchain_core.prompts import ChatPromptTemplate

from typing import List
from langchain_core.output_parsers import BaseOutputParser
from pydantic import BaseModel, Field

multi_query_gen_prompt_template = """
You are an AI language model assistant. Your task is
to generate five different versions of the given user
question to retrieve relevant documents from a vector
database. By generating multiple perspectives on the
user question, your goal is to help
the user overcome some of the limitations of the
distance-based similarity search.
Provide these alternative questions separated by newlines.
Original question: {question}
"""
```

```
multi_query_gen_prompt = ChatPromptTemplate.from_template(
    multi_query_gen_prompt_template)

class LineListOutputParser(BaseOutputParser[List[str]]):
    """Parse out a question from each output line."""

    def parse(self, text: str) -> List[str]:
        lines = text.strip().split("\n")
        return list(filter(None, lines))


questions_parser = LineListOutputParser()

llm = ChatOpenAI(model="gpt-5", openai_api_key=OPENAI_API_KEY)

multi_query_gen_chain = multi_query_gen_prompt | llm | questions_parser
```

With this setup, you can now generate multiple alternative queries from a single question, helping to capture varied perspectives and nuances that improve document retrieval accuracy.

> **NOTE** I've chosen to use GPT-5 instead of GPT-5-mini or GPT-5-nano, as it's more likely to produce higher-quality queries and generate more accurate, well-synthesized responses

Now that multiple queries can be generated, the next step is to implement a ranking mechanism to sort and prioritize the retrieved results. We'll use the RRF algorithm for ranking.

RRF ALGORITHM

The core of this workflow is the RRF algorithm, which assigns scores to documents retrieved by multiple queries. Using the RRF formula, each document is scored based on its rank and then reranked by total RRF score. See the following listing for implementation details.

Listing 10.15 Reciprocal Rank Fusion algorithm

```
def reciprocal_rank_fusion(results_groups:
                           list[list], k=60):
    """ Reciprocal_rank_fusion that takes multiple groups of
        ranked documents and an optional parameter k used in
        the Reciprocal Rank Fusion (RRF) formula """

    indexed_results = {}

    for group_id, results_group in enumerate(
        results_groups):
        for local_rank, doc in enumerate(results_group):
            indexed_results[(group_id, local_rank)] = doc

    fused_scores = {}
```

Based on https://mng.bz/lZjM

Initializes a dictionary to organize results with an index

Indexes the results by (group_id, local_rank)

Initializes a dictionary to hold fused scores for each unique document

```
    for key, doc in indexed_results.items():
        group_id, local_rank = key

        if key not in fused_scores:
            fused_scores[key] = 0

        doc_current_score = fused_scores[key]
        fused_scores[key] += 1 / (local_rank + k)

    reranked_results = [
        (indexed_results[key], score)
        for key, score in sorted(fused_scores.items(),
                                 key=lambda x: x[1], reverse=True)
    ]

    return reranked_results
```

Iterates through the indexed results

Initializes an indexed result with a score of 0 if it hasn't been processed yet

Calculates the new document score with the RRF formula

Reranks the results by RRF score

SETTING UP THE RAG FUSION RETRIEVAL CHAIN

With the RRF algorithm in place, let's create a RAG fusion retrieval chain, as shown here:

```
retriever = uk_with_metadata_collection.as_retriever(
    search_kwargs={'k':3})
top_three_results = RunnableLambda(
    lambda x: x[0:3])

rag_fusion_retrieval_chain =multi_query_gen_chain \
    | retriever.map() | reciprocal_rank_fusion \
    | top_three_results

docs = rag_fusion_retrieval_chain.invoke(
    {"question": question})
len(docs)
```

Selects the top three results

Full RAG fusion retrieval chain

Tests the retrieval_chain_rag_fusion chain

The final step is to integrate this RAG Fusion retrieval chain into a larger RAG chain for end-to-end question routing, retrieval, and answer synthesis.

INCORPORATING RAG FUSION INTO THE RAG CHAIN

As we've seen before, integrating a retrieval chain into a broader RAG chain is straightforward. For completeness, the following listing shows how to incorporate the RAG fusion retrieval chain within a RAG chain.

Listing 10.16 Integrating RAG fusion into the RAG chain

```
rag_prompt_template = """
Given a question and some context, answer the question.
If you do not know the answer, just say I do not know.

Context: {context}
Question: {question}
"""

rag_prompt = ChatPromptTemplate.from_template(rag_prompt_template)
```

```
rag_chain = (
    {
        "context": {"question": RunnablePassthrough()} |
        ➥rag_fusion_retrieval_chain,
        "question": RunnablePassthrough(),
    }
    | rag_prompt
    | llm
    | StrOutputParser()
)
```

The original user question

The context is returned by the retriever after feeding it to the step-back question.

Now let's test the complete RAG chain with an example question:

```
user_question = "Can you give me some tips for a trip to Brighton?"

answer = rag_chain.invoke(user_question)
print(answer)
```

Expected output:

```
Here are some tips for a trip to Brighton:

1. **Visit During Festivals**: If you can, plan your visit in May when the
Brighton Festival and Festival Fringe take place. These events are among the
most popular and showcase a variety of arts and performances.

2. **Enjoy the Beach**: Brighton boasts a beautiful stretch of shingle beach
over 5 miles long. Make sure to spend some time relaxing by the sea,
especially during the summer when the weather is nice.

3. **Explore Local Culture**: Brighton has a vibrant cultural scene with many
activities and events happening year-round. Take the time to explore art
galleries and local events.

4. **Transportation**: Brighton is well-connected by train, making it easy to
get in and out of the city. Consider using public transport or biking to get
around the area.

5. **Seasonal Work Opportunities**: If you're looking for temporary work
while visiting, Brighton is a good spot due to its student population and
seasonal job availability.

6. **Plan for All Budgets**: Whether you're looking for budget options or
willing to splurge, Brighton offers a range of accommodations, dining, and
entertainment options to suit different budgets.

7. **Stay Safe**: Like any city, it's important to stay aware of your
surroundings and follow general safety precautions while exploring.

8. **Local Council Resources**: Check out the Brighton and Hove City Council
website for additional local information and resources to enhance your trip.

Enjoy your trip!
```

Congratulations! You've completed a comprehensive guide to advanced RAG techniques, making you well-equipped to tackle complex RAG tasks.

Summary

- Retrieval-Augmented Generation (RAG) architectures scale beyond single-source retrieval to handle complex information needs across heterogeneous data stores. Combine vector stores, SQL databases, graph databases, and APIs in one system.
- Hybrid search combines dense retrieval (embeddings for semantic similarity) and sparse retrieval (BM25 for keyword matching). Merge results using Reciprocal Rank Fusion (RFF) to balance semantic and lexical relevance.
- SQL databases store structured data for precise queries such as customer orders or financial records. Text-to-SQL systems convert natural language questions into SQL queries, execute them, and return results.
- Graph databases model relationships between entities using nodes and edges. Queries such as "who reports to whom" or "shortest path between concepts" execute efficiently on graph structures.
- Multiple data sources combine in RAG systems. Vector stores handle document content, SQL databases manage transactional data, and graph databases capture entity relationships—all queried through a unified agent interface.
- Query routing uses LLM classification to direct questions to appropriate storage backends. "What did the CEO say in Q3?" routes to a vector store; "How many units sold last quarter?" routes to an SQL database.
- RRF merges ranked results from multiple retrievals. It scores each document based on its position in each ranking (`1/rank`), sums scores across rankings, and re-ranks by total score.
- BM25 requires inverted index data structures that many vector stores don't natively support. LangChain's `BM25Retriever` works with in-memory document lists but doesn't scale to millions of documents.
- Query routing with LLM classification by creating a prompt with descriptions of each data source, using the LLM to classify which source to use, then routing to the appropriate retriever or tool.
- Provide few-shot examples in routing prompts:
  - *Vector store*—What did the CEO say about revenue?
  - *SQL*—How many units sold?
  - *Graph*—Who reports to the VP?
- Implement fallback logic for empty results—if the primary source returns nothing, try alternative sources. For example, a miss in the vector store triggers an SQL database query for structured alternatives.

- When combining BM25 and vector search, retrieve top-k from each, apply RRF to merge results, and then use the combined set for LLM context. This captures both keyword matches and semantic similarity.
- For graph databases, use the Cypher query language with text-to-Cypher conversion similar to text-to-SQL. LangChain provides Neo4j integration for graph database querying.
- Test routing accuracy on labeled queries. Track which percentage route to correct data sources and adjust classification criteria or few-shot examples to improve accuracy.

Part 5

AI agents

This final part brings everything together, taking you from structured workflows to fully capable agents that can think, decide, and act. You'll move beyond static flows and build dynamic, tool-using agents in LangGraph—systems that can choose which tools to call, interpret results, and adapt their next steps based on context. From there, you'll scale up to multi-agent systems that coordinate across specialized roles, routing tasks intelligently and collaborating to solve complex problems.

You'll also connect your agents to the broader AI ecosystem through the Model Context Protocol (MCP), which allows them to discover and use remote tools as seamlessly as local ones. With MCP, your agents gain access to a growing world of interoperable services without extra integration overhead. Finally, you'll learn how to make your agents production-ready by adding memory for long-term context, guardrails for safety, and observability for traceability—all while ensuring your system remains resilient and easy to debug.

The unifying theme here is practicality: building agents that aren't just smart, but also reliable, transparent, and maintainable. By the end of this part, you'll understand how to design agents that can reason effectively, collaborate efficiently, and operate safely in real-world environments—agents that are ready to grow from a single assistant into an entire ecosystem of intelligent, connected AI services.

11 Building tool-based agents with LangGraph

This chapter covers

- Building LLM-powered agents using LangGraph
- Registering and using tools for dynamic agent execution
- Debugging agent execution and tool calls
- Simplifying agents with prebuilt LangGraph components
- Observing agent execution with LangSmith

In chapter 5, we explored the distinction between agentic workflows and agents. You learned that agentic workflows are fundamentally deterministic: their logic is based on flows with conditional paths that depend on the current application state. These workflows can be elegantly modeled using node-based graphs in LangGraph, and you saw a complete, hands-on example of such a system.

Agents, however, operate differently. Rather than following a predetermined flow, agents rely on dynamic, context-sensitive decision-making. With the help of a large language model (LLM), an agent chooses which tools to use—and in what

order—based on the evolving context of the task at hand. These decisions aren't prescripted; instead, they unfold step-by-step, as the agent continually evaluates the outputs of previous actions and adapts accordingly.

In this chapter, you'll put these ideas into practice by building a multi-tool travel information agent. You'll begin simply, implementing an agent that provides destination information using a single tool. From there, you'll extend it into a true multi-tool agent, able to answer questions about both travel destinations and their current weather conditions.

As we progress, I'll introduce you to the core concepts necessary for constructing multi-tool agents, with particular focus on the tool-calling protocol. You'll first implement this protocol from scratch to grasp every detail, and then you'll see how LangGraph's built-in capabilities can streamline and simplify your agent's architecture.

This chapter's multi-tool agent will serve as the foundation for the more advanced, multi-agent systems you'll build in the chapters ahead. Let's dive in—there's a lot to discover.

11.1 Starting simple: Building a single-tool travel info agent

In this section, we'll lay the groundwork for our agent-based applications by building a straightforward travel information agent. This first agent will use just one tool: a vector store retriever that answers questions about Cornwall's destinations and resorts, using content from Wikivoyage (www.wikivoyage.org). The content is split into chunks and stored in a vector store for efficient retrieval.

If you've followed the advanced Retrieval-Augmented Generation (RAG) chapters earlier in this book, you should already be comfortable with sourcing content and populating a vector store. Here, we'll build on that foundation, keeping the focus on agent mechanics.

11.1.1 Project setup

Let's set up a new Python project using Visual Studio Code (VS Code). This works seamlessly with Cursor as well.

CREATING A VIRTUAL ENVIRONMENT AND INSTALLING DEPENDENCIES

First, set up your Python virtual environment and install all necessary dependencies. You'll find the `requirements.txt` file either on the Manning website for this book or in the cloned GitHub repository that accompanies chapter 11.

Open a new PowerShell terminal in VS Code (choose Terminal > New Terminal), navigate to the `ch11` project folder, and then create and activate a new virtual environment:

```
PS C:\Github\building-llm-applications\ch11> python -m venv env_ch11
PS C:\Github\building-llm-applications\ch11> .\env_ch11\Scripts\activate
(env_ch11) PS C:\Github\building-llm-applications\ch11>
```

Once your environment is activated, install the required dependencies:

```
(env_ch11) PS C:\Github\building-llm-applications\ch11>
➥ pip install -r .\requirements.txt
```

Your project environment is now ready for development.

ADDING YOUR OPENAI API KEY

Create an `.env` file in your project root, and add your OpenAI API key:

```
OPENAI_API_KEY=<Your OPENAI_API_KEY>
```

CONFIGURING VS. CODE DEBUGGING

For smooth debugging, add the following `launch.json` to your `.vscode` directory:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Python Debugger: Current File",
            "type": "debugpy",
            "request": "launch",
            "program": "${file}",
            "console": "integratedTerminal"
        }
    ]
}
```

Organize your implementation files by using the naming convention `main_x_y.py` for the scripts. The `x` represents the feature (e.g., `main_01_01.py` for the initial travel info agent, `main_02_01.py` when adding weather), while `y` is the version of that feature as we iterate. This will make it easy for you to compare versions and follow the progression of the implementation.

> **NOTE** The code in these examples is intentionally simplified to focus on core functionality. Error handling and defensive programming are omitted for clarity and learning purposes.

11.1.2 Loading environment variables

Once your `.env` file is ready, create a code file named `main_01_01.py`. You can load your API key at the top of the script, after the import statements (omitted here for brevity—refer to the GitHub repository for the full list), as shown in the following listing.

Listing 11.1 Loading environment variables

```
load_dotenv()
```

Loads environment variables from the .env file

11.1.3 Preparing the travel information vector store

To enable our travel information agent to answer queries about Cornwall destinations, we first need a way to store and efficiently retrieve relevant information. The approach here draws on techniques you saw in chapter 8, sections 8.3 and 8.4, on advanced RAG techniques, but streamlines them for this agent-centric context. At a high level, we'll download travel content for a set of Cornwall destinations from Wikivoyage, break the

text into manageable chunks, embed those chunks into vector representations, and then store everything in a Chroma vector store. We'll also encapsulate the initialization logic in a singleton pattern to ensure that the vector store is only built once during the agent's lifetime. The following code sets up this vector store, making it easy to retrieve relevant travel information as the agent operates.

Listing 11.2 Preparing the travel information vector store

```
UK_DESTINATIONS = [
    "Cornwall",
    "North_Cornwall",
    "South_Cornwall",
    "West_Cornwall",
]

async def build_vectorstore(
    destinations: Sequence[str]) -> Chroma:
    """Download Wikivoyage pages and create
    a Chroma vector store."""
    urls = [f"https://en.wikivoyage.org/wiki/{slug}"
   for slug in destinations]
    loader = AsyncHtmlLoader(urls)
    print("Downloading destination pages ...")
    docs = await loader.aload()

    splitter = RecursiveCharacterTextSplitter(
        chunk_size=1024, chunk_overlap=128)
    chunks = sum([splitter.split_documents([d])
        for d in docs], [])

    print(f"Embedding {len(chunks)} chunks ...")
    vectordb_client = Chroma.from_documents(
        chunks, embedding=OpenAIEmbeddings())
    print("Vector store ready.\n")
    return vectordb_client

_ti_vectorstore_client: Chroma | None = None

def get_travel_info_vectorstore() -> Chroma:
    global _ti_vectorstore_client
    if _ti_vectorstore_client is None:
        if not os.environ.get(
            "OPENAI_API_KEY"):
            raise RuntimeError(
                """Set the OPENAI_API_KEY env
                variable and re-run.""")
        _ti_vectorstore_client = asyncio.run(
            build_vectorstore(UK_DESTINATIONS))
    return _ti_vectorstore_client

ti_vectorstore_client
➥= get_travel_info_vectorstore()
ti_retriever = ti_vectorstore_client.as_retriever()
```

List of target Cornwall destinations—expand as needed

Asynchronous function to build and return the vector store

Downloads Wikivoyage content for each destination

Splits downloaded documents into manageable chunks

Embeds each chunk and stores it in the Chroma vector store

Returns the initialized vector store

Singleton cache for the vector store client

Function to initialize/retrieve the cached vector store

Returns the cached vector store client instance

Creates the vector store client

Creates a retriever from the vector store

This setup starts by defining a list of Cornwall-related destinations that serve as the basis for information retrieval. The `build_vectorstore()` asynchronous function constructs URLs for each destination and uses an asynchronous loader to fetch the corresponding Wikivoyage pages. Once the pages are downloaded, the text is split into overlapping chunks to ensure that information remains contextually meaningful. These chunks are then embedded using OpenAI's embedding models and stored in a Chroma vector store, making them quickly searchable by semantic similarity.

To prevent unnecessary recomputation and data downloads, the singleton pattern is used for the vector store client. The `get_travel_info_vectorstore()` function ensures that the vector store is only built once and is reused for all future retrievals. At the end, the vector store client is instantiated, and a retriever object is created from it—this retriever will be the agent's interface for accessing Cornwall travel information. This foundation allows the agent to efficiently answer user queries about destinations using up-to-date knowledge sourced directly from Wikivoyage.

11.2 Enabling agents to call tools

Now that we have our vector store retriever ready, the next step is to expose this retrieval capability as a tool the agent can use. This is where the concept of tool calling—a major advance in modern agent frameworks—enters the picture.

11.2.1 From function calling to tool calling

LLMs like those from OpenAI initially introduced *function calling*—a mechanism that allows the model to request specific functions, passing structured arguments, based on the needs of a given prompt. This concept quickly evolved into the more general *tool-calling* protocol, now widely supported by major LLM providers. With tool calling, models can invoke not just custom functions but also a variety of built-in or external "tools," handling everything from code execution to external API lookups.

> **DEFINITION** A *tool* is any external function, service, or capability that an LLM can call through the tool-calling protocol. Tools extend the model's abilities beyond text generation—for example, running code, querying a database, or calling an API—by letting the model pass structured inputs and receive structured outputs.

You might wonder why tool calling matters. Why not just let agents access functionality directly through standard REST API endpoints?

The key reason is flexibility. We turn to agents—rather than fixed agentic workflows—when we can't predict in advance which tools will be needed, in what sequence they should be called, or how a user's question will map to tool parameters. In these cases, hardcoding API calls isn't enough.

An agent's strength lies in making those decisions dynamically: selecting the right tool, determining the correct order of calls, and shaping inputs to match tool parameters. Modern LLMs are explicitly trained for this purpose, with tool calling built into their response protocols (e.g., the OpenAI Responses API). In short, tool calling unlocks the adaptive reasoning that makes multi-tool agents possible.

This evolution has made agent implementations dramatically simpler and more robust, particularly when using the *ReAct* (Reasoning and Acting) design pattern. The ReAct pattern, introduced by Yao et al. in their 2022 paper "ReAct: Synergizing Reasoning and Acting in Language Models" (https://arxiv.org/abs/2210.03629) enables LLMs to interleave reasoning steps ("thoughts") with tool invocations ("actions"). In effect, the model can break a task into smaller pieces, use tools where appropriate, and synthesize an answer step-by-step. Figure 11.1 shows a simplified view of the ReAct pattern as a process diagram.

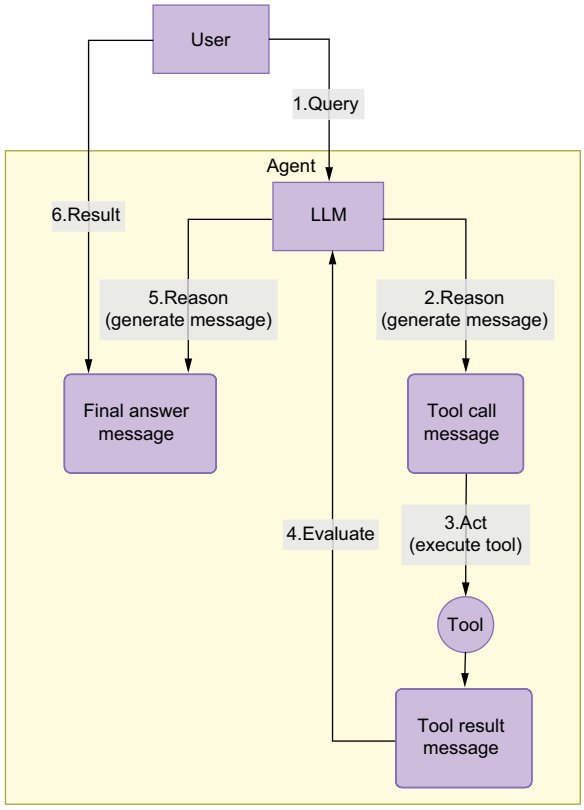


Figure 11.1 The ReAct pattern alternates between reasoning and action, enabling the agent to process a user question by thinking, calling tools as needed, and following up with further reasoning before delivering the final answer.

The ReAct pattern starts with a user question, as shown in the diagram. The agent first enters the Reasoning phase, where the LLM analyzes the input and decides on the next steps. If further information or actions are required, the agent moves into the Act phase, calling one or more tools—such as running a semantic search or invoking an API.

Once the tool returns results, the agent cycles back into Reasoning, incorporating the new information into its thought process. This may trigger additional tool calls, or the agent may determine that it has enough context to answer. The process concludes when the agent produces a Final Answer for the user.

This stepwise interplay between reasoning and acting embodies the core of the ReAct pattern, allowing agents to dynamically combine thought and action as they work toward solving the user's query.

> **NOTE** With the introduction of the OpenAI Responses API, the transition from function calling to tool calling has accelerated. The Responses API is designed specifically for structured tool use and has largely superseded the older Completion API for agentic applications.

11.2.2 How tool calling works with LLMs

OpenAI's tool calling supports several types of tools—including user-defined functions, code interpreters, and native capabilities such as web browsing. At a high level, when you register a tool with the LLM, you expose both the function signature (name and parameters) and a textual description. The model then decides, at runtime, when and how to use each tool based on the user's input and the context of the conversation.

In LangGraph, tools can be registered using either a class-based or a decorator-based approach. We'll use the decorator-based approach for clarity and conciseness. Let's implement our semantic search tool as a function decorated with `@tool`, making it available to the agent for tool calling, as shown in the following listing.

Listing 11.3 LangGraph attribute-based tool definition

```
@tool
def search_travel_info(query: str) -> str:
    """Search embedded Wikivoyage content for
    information about destinations in England."""
    docs = ti_retriever.invoke(query)
    top = docs[:4] if isinstance(docs, list) else docs
    return "\n---\n".join(
        d.page_content for d in top)
```

Defines the tool using the @tool decorator

Defines the tool function, which takes a query, performs a semantic search, and returns a string response from the vector store

Performs a semantic search on the vector store and returns the top four results

Joins the top four results into a single string

This decorated function, `search_travel_info()`, is now recognized as a tool: it takes a user query, searches the vector store for relevant Wikivoyage content, and returns up to four top results as a single string. The `@tool` decorator ensures that the function's name, description, and parameter schema are all available to the LLM for tool calling.

11.2.3 Registering tools with the LLM

To enable the agent to use our semantic search tool, we must register it with the LLM so that the model knows both the function's signature and its intended purpose. In LangChain, this is achieved using the `bind_tools` protocol, but it's instructive to see how this works at the OpenAI API level first.

With the introduction of the OpenAI Responses API, the way tools and functions are registered has become more standardized and explicit. Now you provide a

structured definition of your tool—specifying its name, description, and parameter schema. The model can then return responses that explicitly request tool invocations, passing arguments as needed. For further details, you can consult the OpenAI function calling documentation (https://mng.bz/Bzag).

EXAMPLE: MANUAL TOOL REGISTRATION WITH OPENAI API

Suppose you wanted to expose our `search_travel_info` function directly to the OpenAI API (without using LangChain). You would define the tool's schema as shown in listing 11.4. Note that this example is for illustration only and isn't included in the book's provided source code.

Listing 11.4 Manual tool registration with the OpenAI API

```
search_travel_info_tool = {
    "type": "function",
    "function": {
        "name": "search_travel_info",
        "description": "Search embedded Wikivoyage content for information
        ➥about destinations in England.",
        "parameters": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",
                    "description": "A natural language
                    ➥ query about a destination in
                    ➥ England."
                }
            },
            "required": ["query"]
        }
    }
}

response = client.chat.completions.create(
    model="gpt-5",
    messages=[
        {"role": "user", "content": "Tell me about surfing in
        ➥Cornwall."}
    ],
    tools=[search_travel_info_tool],
    tool_choice="auto",
)
```

Specifies that this object is a function tool (as per OpenAI's tool-calling schema)

Name of the tool/function being registered

Human-readable description of what the tool does

The parameters accepted by the function are described in this dictionary.

The parameters are passed as an object (i.e., a dictionary of named arguments).

The function accepts a single parameter named "query."

Description of what the "query" parameter should contain

"query" is a required argument for the tool.

Specifies the OpenAI model to use for the completion

Provides the initial user message to the chat (the question to answer)

Registers the tool so the model knows it can call it if needed

Allows the model to automatically decide whether and when to use the tool

In this example, you explicitly define the tool's metadata and parameters and then pass it in the `tools` argument of your API call. When the model decides it needs to call `search_travel_info`, it will return a structured tool call in its response. Your application must then handle the invocation of the Python function, passing the model-generated arguments, and send the results back to the LLM if the conversation continues.

REGISTERING TOOLS IN LANGCHAIN

LangChain automates much of this process. You simply define your tool as a decorated Python function (as you saw earlier in listing 11.3) and then register it with the LLM. You can see how this looks in code in the following listing.

Listing 11.5 Registering tools in LangChain

```
TOOLS = [search_travel_info]

llm_model = ChatOpenAI(
    model="gpt-5-mini",
  use_responses_api=True)
llm_with_tools = llm_model.bind_tools(TOOLS)
```

Defines the tools list (in our case, only one tool)

Instantiates the LLM model with the GPT-5-mini model and the responses API

Binds the tools to the LLM model, which will generate a response with the tool calls

Here, we list the available tools (currently just `search_travel_info`), instantiate the `gpt-5-mini` chat model (with Responses API support), and use `.bind_tools(TOOLS)` to expose those tools to the model for tool calling.

LangChain handles the translation between your Python code and the OpenAI function/tool-calling protocol, including automatically generating the appropriate JSON schema from your function signature and docstring. This setup ensures that whenever the model receives a user message, it can autonomously decide if and when to call any registered tool, structuring its responses according to the tool-calling protocol.

> **NOTE** If you're using a model other than OpenAI, or you prefer not to use the Responses API, tool calling can still work—though the capabilities and structure of the responses may differ. The older Completion API is still available, but for most agentic use cases, the newer Responses API is recommended for its clarity, power, and alignment with evolving best practices.

11.2.4 Agent state: Tracking the conversation

In our implementation, the agent's state is simply a collection of LLM messages that track the entire conversation. This is defined as follows:

```
class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], operator.add]
```

Here, `AgentState` only contains the sequence of messages exchanged between the user, the agent, and any tool responses. This keeps the design simple.

11.2.5 Executing tool calls

The core of the tool execution logic is implemented in a node that examines the LLM's most recent message, extracts any tool calls requested by the model, and invokes the corresponding functions with the provided arguments. Each tool's output is then wrapped in a message and appended to the conversation state. A simplified implementation is shown in the following listing.

Listing 11.6 Tool execution implemented from scratch

```
class ToolsExecutionNode:
    """Execute tools requested by the LLM in the last AIMessage."""

    def __init__(self, tools: Sequence):
        self._tools_by_name = {t.name: t for t in tools}

    def __call__(self, state: dict):
        messages: Sequence[BaseMessage] = state.get("messages", [])

        last_msg = messages[-1]
        tool_messages: list[ToolMessage] = []
        tool_calls = getattr(last_msg,
            "tool_calls", [])

        for tool_call in tool_calls:
            tool_name = tool_call["name"]
            tool_args = tool_call["args"]
            tool = self._tools_by_name[tool_name]
            result = tool.invoke(tool_args)
            tool_messages.append(
                ToolMessage(
                    content=json.dumps(result),
                    name=tool_name,
                    tool_call_id=tool_call["id"],
                )
            )
        return {"messages": tool_messages}

tools_execution_node = ToolsExecutionNode(TOOLS)
```

Defines the tools execution node

Initializes the tools execution node with the tools list

Defines the __call__ method, which is called when the node is invoked

Gets the last message from the messages list

Initializes the tool messages list to gather the results of the tool calls

Gets the tool calls from the last message

Iterates over the tool calls

Gets the tool name from the tool call

Gets the tool arguments from the tool call

Gets the tool from the tools list

Invokes the tool with the arguments

Adds the tool result to the tool messages list

Returns the tool messages list, which contains the results of the tool calls

Instantiates the tools execution node to be used as a node in the LangGraph graph

This pattern allows the agent to handle multiple tool calls in a single step. The `ToolsExecutionNode` inspects the LLM's latest response, invokes each requested tool by name, collects the results, and formats them for the next stage in the conversation. In a typical agent workflow, these tool results are sent back to the LLM, which incorporates the new information to either reason further or generate a direct answer to the user's query.

The built-in ToolNode class

You usually won't need to write this logic yourself in practice because LangGraph provides a built-in class, `ToolNode`, which performs the same function as our custom `ToolsExecutionNode`. For most applications, you can use

```
tools_execution_node = ToolNode(TOOLS)
```

This saves you from having to implement the tools execution logic manually, streamlining your agent development process.

Now that you understand how tool execution is managed, let's explore how the agent, guided by the LLM, determines which tools to call—or whether it already has enough information to generate the final answer.

11.2.6 The LLM node: Coordinating reasoning and action

Next, we add an LLM node to our LangGraph workflow, as shown in the following listing.

Listing 11.7 LLM node

```
def llm_node(state: AgentState):
    """LLM node that decides whether
    to call the search tool."""
    current_messages = state["messages"]
    response_message = llm_with_tools.invoke(
        current_messages)

    return {"messages": [response_message]}
```

Defines the LLM node

Gets the current messages from the agent state

Invokes the LLM model with the current messages. The LLM will decide whether to call the search tool or return an answer.

Returns the response message, which contains the tool call or the answer

This node takes the agent state (a list of messages) and forwards them to the LLM for processing. Because we're using an LLM with tool calling enabled (`llm_with_tools`), the model automatically understands when to issue tool calls and when to produce a final answer:

- If the last message is a user question, the LLM can decide to request a tool call (or multiple calls) to retrieve relevant information.
- If the last message(s) are tool results, the LLM integrates those responses, reasons further, and may either produce a final answer or request additional tool calls if needed.

The model returns either an `AIMessage` with a final answer in the content field, or additional tool calls in the `tool_calls` field, depending on the situation. This flexible, reactive loop is at the heart of all modern agent implementations, and it's what makes today's LLM-based agents so capable.

11.3 Assembling the agent graph

With our LLM node and tool execution node ready, the next step is to assemble them into a working agent graph. In LangGraph, an agent is structured as a directed graph, where each node represents a component (e.g., an LLM or a tool executor), and edges represent the possible flow of data and control between these nodes.

The code in listing 11.8 demonstrates how we assemble our single-tool travel information agent as a graph. Each node in the graph corresponds to either the LLM or the tool execution logic.

Listing 11.8 Graph of the tool-based agent

```
builder = StateGraph(AgentState)
builder.add_node("llm_node", llm_node)
builder.add_node("tools", tools_execution_node)

builder.add_conditional_edges("llm_node",
    tools_condition)

builder.add_edge("tools", "llm_node")

builder.set_entry_point("llm_node")
travel_info_agent = builder.compile()
```

Defines the graph builder

Adds the LLM node and the tools node to the graph

Adds the conditional edges to the graph to decide whether to execute the tool calls or return an answer and exit the graph

Adds the edge from the tools node to the LLM node

Sets the entry point to the LLM node

Compiles the graph

11.4 Understanding the agent graph structure

Our agent graph consists of two main nodes, as shown in figure 11.2: the LLM node, responsible for reasoning and generating tool calls, and the tools node, responsible for executing the requested tools and returning results. The flow of the conversation alternates between these two nodes, reflecting the ReAct pattern described earlier in this chapter.

A critical aspect of this setup is the conditional edge that connects the LLM node to the next step. This is controlled by the `tools_condition` function—a prebuilt utility in LangGraph. This function examines the latest message from the LLM:

- If the message contains the `tool_calls` property (meaning the LLM is requesting one or more tool invocations), the graph routes the flow to the node named `"tools"`.
- If there are no tool calls present, the flow is directed to the `END` node, terminating the graph and producing a final answer for the user.

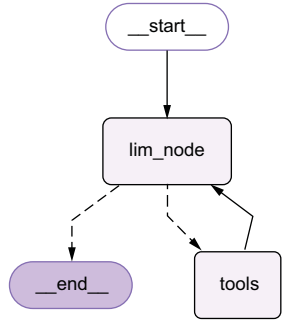


Figure 11.2 Conditional graph logic routes each user query through the LLM node and then dynamically directs flow to the tools node or ends the process, depending on whether tool calls are required.

This mechanism lets the agent dynamically decide at each turn whether to continue reasoning, take action, or conclude the interaction.

We explicitly set the entry point of the graph to `"llm_node"`, ensuring that each user question is first processed by the language model. (As an alternative, you could achieve the same effect by adding an edge from the `START` node to `"llm_node"` with `graph_builder.add_edge(START, "llm_node")`.) By compiling the graph with `builder.compile()`, we finalize our travel information agent, ready to receive queries and intelligently use its tool to find relevant travel information.

11.5 Running the agent chatbot: The Read-Eval-Print Loop

The final step in building our agent-powered chatbot is to implement the user interface—a simple loop that continuously accepts user questions and returns answers until the user chooses to exit. This classic pattern, known as a Read-Eval-Print Loop (REPL), is the main bridge between the user and your travel information agent.

At a high level, the chat loop listens for input, wraps the user's question in a message structure, invokes the agent graph, and prints the assistant's reply. This interaction continues indefinitely, enabling a true conversational experience. The following listing shows how you can implement this chat loop in Python.

Listing 11.9 Chatbot REPL

```
def chat_loop():
    print("UK Travel Assistant (type 'exit' to quit)")
    while True:
        user_input = input(
            "You: ").strip()
        if user_input.lower() in \
            {"exit", "quit"}:
                break
        state = {"messages":
            [HumanMessage(content=user_input)]}
        result = travel_info_agent.invoke(
            state)
        response_msg = result["messages"][-1]
        print(
           f"Assistant: {response_msg.content}\n")

if __name__ == "__main__":
    chat_loop()
```

Defines the chat loop

Gets the user input

Checks if the user input is "exit" or "quit" to exit the loop

Creates the initial state with a HumanMessage containing the user input

Invokes the graph with the initial state

Gets the last message from the result, which contains the final answer

Prints the assistant's final answer from the content of the preceding message

This loop welcomes the user, waits for their question, and continues processing until the user types `exit` or `quit`. Each input is packaged as a `HumanMessage` and passed to the travel information agent you just built. The agent's reply is extracted from the graph's output and displayed back to the user.

With this in place, you're now ready to run your first agent-based chatbot. Try asking it about destinations or activities in Cornwall, and experience how the agent reasons, retrieves information, and converses, all within the framework you've just constructed, as we're about to see.

11.6 Executing a request

Now that your travel information agent is ready, let's step through the agent in action by running and debugging your implementation. This hands-on walkthrough will show you how the agent orchestrates the flow between the LLM and the tool, helping you see the LangGraph framework in motion.

11.6.1 Step-by-step debugging

Begin by opening your `main_01_01.py` file and running it in debug mode, using the Python debug configuration you set up in your `launch.json` earlier. To trace the agent's flow, place breakpoints at the beginning of the following functions:

- `search_travel_info()`
- `ToolsExecutionNode.__call__()`
- `llm_node()`

Ready? Press F5 (or click the Play icon in your IDE), and let's walk through the agent's workflow together.

VECTOR STORE CREATION

When you start the script, you'll see the vector store being created and populated with travel information. In your debug console, you'll see the output similar to that in figure 11.3.

```
USER_AGENT environment variable not set, consider setting it to identify your requests.
Downloading destination pages ...
Fetching pages: 100%|#############################################################################################
#########| 4/4 [00:01<00:00,  3.96it/s]
Embedding 1118 chunks ...
Vector store ready.

UK Travel Assistant (type 'exit' to quit)
```

Figure 11.3 Output at startup during vector store creation

CHATBOT LOOP LAUNCH

Next, the chatbot loop starts up, waiting for your input:

```
UK Travel Assistant (type 'exit' to quit)
You:
```

Enter your question, and press Enter:

```
You: Suggest three towns with a nice beach in Cornwall
```

LLM NODE ACTIVATION

Your breakpoint inside `llm_node()` will trigger. Inspect `state["messages"]`:

```
HumanMessage (content='Suggest three towns with a nice beach in Cornwall',
additional_kwargs={}, response_metadata={})
```

Step over (press F10) to the next line to send this message to the LLM. Because the LLM is configured for tool calling, examine the resulting `response_message`:

```
   AIMessage(content=[],
...
tool_calls=[
```

```
{'name': 'search_travel_info', 'args': {'query': 'beach towns in Cornwall'},
'id': 'call_L4PwmeyLkrkYX2PfC2gY6ri6', 'type': 'tool_call'},
{'name': 'search_travel_info', 'args': {'query': 'best beaches in Cornwall'},
'id': 'call_XPNctNtyIKVetJa3ruM9z7d5', 'type': 'tool_call'},
{'name': 'search_travel_info', 'args': {'query': 'top seaside towns
Cornwall'}, 'id': 'call_RAWMLwdFVALuIbJxWKfta5yp', 'type': 'tool_call'}],
...)
```

Here, the LLM has generated three tool calls—each targeting your semantic search tool with a slightly different query. Notice how the model rewrites the queries to maximize information coverage, as discussed in chapter 9. You don't need to manually handle query rewriting—the LLM handles it.

TOOLS EXECUTION NODE

Continue execution (press F5). The list of tool calls is added to the message list. The conditional edge's `tools_condition` detects tool calls, routing execution to `ToolsExecutionNode.__call__()`. Your breakpoint will trigger there.

Next, inspect `state["messages"]`:

```
[
HumanMessage(content='Suggest three towns with a nice beach in Cornwall',
additional_kwargs={}, response_metadata={}),
AIMessage(content=[], ... ,
tool_calls=[
{'name': 'search_travel_info', 'args': {'query': 'beach towns in Cornwall'},
'id': 'call_L4PwmeyLkrkYX2PfC2gY6ri6', 'type': 'tool_call'},
{'name': 'search_travel_info', 'args': {'query': 'best beaches in Cornwall'},
'id': 'call_XPNctNtyIKVetJa3ruM9z7d5', 'type': 'tool_call'},
{'name': 'search_travel_info', 'args': {'query': 'top seaside towns
Cornwall'}, 'id': 'call_RAWMLwdFVALuIbJxWKfta5yp', 'type': 'tool_call'}] ,
...)
]
```

The last message (from the LLM) has no content (the LLM hasn't answered yet because it needs more information), but it contains the tool calls that must be executed. The node extracts these, then iterates through each one, extracting the tool name and arguments, and invokes the corresponding tool:

```
result = tool.invoke(tool_args)
```

Step through this to watch `search_travel_info()` execute. Each semantic search result (a document returned from the vector store) is collected as a `ToolMessage` and added to the list for the next LLM step.

TOOL CALL RESULTS PASSED BACK TO LLM

Continue (press F5), and you'll return to `llm_node()`. Now `state["messages"]` contains your original question, the LLM's tool call instructions, and the results from each tool execution:

```
\[HumanMessage(content='Suggest three towns with a nice beach in
Cornwall', additional\_kwargs={}, response\_metadata={}),
```

```
   AIMessage(content=\[], ...
   tool\_calls=\[{'name': 'search\_travel\_info', 'args': {'query': 'beach
   towns in Cornwall'}, ...}, ...]),
   ToolMessage(content='...', name='search\_travel\_info',
   tool\_call\_id='call\_L4PwmeyLkrkYX2PfC2gY6ri6'),
   ToolMessage(content='...', name='search\_travel\_info',
   tool\_call\_id='call\_XPNctNtyIKVetJa3ruM9z7d5'),
   ToolMessage(content='...', name='search\_travel\_info',
   tool\_call\_id='call\_RAWMLwdFVALuIbJxWKfta5yp')]
```

The content of each tool message gives the LLM the facts it needs to synthesize a final answer. Step over the following line:

```
response\_message = llm\_with\_tools.invoke(current\_messages)
```

Inspect `response_message`:

```
AIMessage(content=\[{'type': 'text', 'text': "Three towns in Cornwall with
nice beaches are:\n\n1. Newquay - Known as the UK's surfing capital with
popular beaches like Fistral Beach.\n2. St Ives - A picturesque town with
beautiful sandy beaches.\n3. Falmouth - Located on the south coast with
beaches like Gyllyngvase beach.\n\nThese towns offer great beach experiences
along with other attractions.", 'annotations': \[]}], ...)
```

Now the `content` field is populated with the LLM's answer. The `tool_calls` field is gone—the LLM no longer needs external tools and has synthesized a response.

COMPLETING THE REQUEST

As execution leaves `llm_node()`, the `tools_condition` on the conditional edge checks for further tool calls. Finding none, it ends the conversation. When your main invocation returns

```
result = travel\_info\_agent.invoke(state)
```

the `result.messages` list concludes with the final `AIMessage` containing the answer:

```
   {'messages': [
   HumanMessage(content='Suggest three towns with a nice beach in Cornwall',
...),
   AIMessage(content=[],  tool_calls=[...]),
   ToolMessage(...), ToolMessage(...), ToolMessage(...),
   AIMessage(content=[{'type': 'text', 'text': "Three towns in Cornwall with
nice beaches are:\n\n1. Newquay - Known as the UK's surfing capital with
popular beaches like Fistral Beach.\n2. St Ives - A picturesque town with
beautiful sandy beaches.\n3. Falmouth - Located on the south coast with
beaches like Gyllyngvase beach.\n\nThese towns offer great beach experiences
along with other attractions.", 'annotations': []}], ...)
   ]
```

the chatbot will now display the answer and prompt for your next question:

```
UK Travel Assistant (type 'exit' to quit)
You: Suggest three towns with a nice beach in Cornwall
Assistant: [{'type': 'text', 'text': "Three towns in Cornwall with nice
beaches are:\n\n1. Newquay - Known as the UK's surfing capital with popular
```

```
beaches like Fistral Beach.\n2. St Ives - A picturesque town with beautiful
sandy beaches.\n3. Falmouth - Located on the south coast with beaches like
Gyllyngvase beach.\n\nThese towns offer great beach experiences along with
other attractions.", 'annotations': []}]

You:
```

You've now observed, step-by-step, how your agent processes a user request: interpreting the question, generating tool calls, executing them, and synthesizing a final, well-informed answer. This debug-driven walkthrough offers deep insight into the mechanics of agentic reasoning and action in LangGraph.

11.7 Expanding your agent: Adding a weather forecast tool

So far, our agent has been able to answer travel-related queries using semantic search over curated travel content. But real-world travel advice often depends on dynamic, real-time information—like the weather! In this section, you'll learn how to extend your agent by adding a second tool, enabling it to respond with context-aware answers based on current conditions.

11.7.1 Implementing a mock weather service

To illustrate the process, start by copying your `main_01_01.py` file to a new script called `main_02_01.py`. This will help you track each evolutionary step in your agent's development.

First, let's introduce a mock weather service. This service will simulate real-time weather data for any given town, returning both a weather condition (e.g., sunny or rainy) and a temperature. You can see the implementation of the `WeatherForecastService` in the following listing.

Listing 11.10 `WeatherForecastService`

```
class WeatherForecast(TypedDict):
    town: str
    weather: Literal["sunny", "foggy", "rainy", "windy"]
    temperature: int

class WeatherForecastService:
    _weather_options = ["sunny", "foggy", "rainy", "windy"]
    _temp_min = 18
    _temp_max = 31

    @classmethod
    def get_forecast(cls, town: str) \
        -> Optional[WeatherForecast]:

        weather = random.choice(cls._weather_options)
        temperature = random.randint(cls._temp_min, cls._temp_max)
        return WeatherForecast(town=town,
             weather=weather,
             temperature=temperature)
```

Defines the get_forecast method, which returns a WeatherForecast object

This mock service chooses a random weather condition and temperature within a typical summer range for Cornwall. Later, you can swap this out for a real-world weather API if desired.

11.7.2 Creating the weather forecast tool

With the mock service in place, the next step is to create a tool that wraps it, making weather data available to the agent. Here's how you can define the tool function and add it to your agent's toolkit:

```
@tool
def weather_forecast(town: str) -> dict:
    """Get a mock weather forecast for a given town. Returns a
    ➥ WeatherForecast object with weather and temperature."""
    forecast = WeatherForecastService.get_forecast(town)
    if forecast is None:
        return {"error": f"No weather data available for '{town}'."}
    return forecast
```

The `weather_forecast` tool provides a mock weather forecast for a given town. When called with a town name, it returns a dictionary containing the weather conditions (e.g., sunny, foggy, rainy, or windy) and temperature for that location. If no data is available, it returns an error message instead. This tool allows the agent to incorporate simulated real-time weather information into its responses.

11.7.3 Updating the agent for multi-tool support

Finally, make sure your LLM model is set up to use tool calling and recognizes both available tools:

```
TOOLS = [search_travel_info, weather_forecast]

llm_model = ChatOpenAI(
    model="gpt-5-mini",
    use_responses_api=True
)
llm_with_tools = llm_model.bind_tools(TOOLS)
```

Defines the tools list (in our case, search_travel_info and weather_forecast)

Instantiates the LLM model with the GPT-5-mini model and enables the Responses API for tool calling

Binds the tools to the LLM model, enabling tool-calling support

With this setup, your agent is now equipped to handle both travel queries and weather checks, giving it the foundation to provide much more accurate and useful responses. In the next sections, you'll see how to guide the LLM to use both tools effectively, and you'll observe the agent's new capabilities in action. This workflow not only shows how to extend your agent's toolset but also demonstrates the modular, incremental approach that makes agentic systems with LangGraph and LangChain so powerful.

11.8 Executing the multi-tool agent

With your weather forecast tool registered, your agent can now synthesize answers that combine travel information and real-time weather conditions. The beauty of this

modular approach is that adding a new tool requires no changes to your agent's graph structure. All orchestration is handled by the LLM and the tool-calling protocol.

11.8.1 Running the multi-tool agent (initial behavior)

Let's put the new capabilities to the test. Set the same breakpoints as before—especially in the tool execution and LLM nodes—and add a breakpoint at the start of `weather_forecast()`. Run `main_02_01.py` in debug mode (press F5), and enter the following prompt:

```
You: Suggest two Cornwall beach towns with nice weather
```

After submitting your query, your first breakpoint will hit in `llm_node()`. Step through to the last line, and inspect the value of `response_message.tool_calls`:

```
[{'name': 'weather_forecast', 'args': {'town': 'Newquay'}, 'id':
'call_3nIgMLIFgMZBvVvrTHbRh2lj', 'type': 'tool_call'},
{'name': 'weather_forecast', 'args': {'town': 'Falmouth'}, 'id':
'call_Qfv5ENGOXhEUUcAAEamDyoQU', 'type': 'tool_call'}]
```

Here, the LLM has simply picked two towns it "knows" (Newquay and Falmouth) and asked for their weather (you might get a different combination of towns), without consulting your semantic search tool at all. This is typical of a model using its internal knowledge base rather than the tools provided—something we want to avoid for reliability and accuracy.

Why does this happen? The LLM is acting on its pretraining and defaulting to what it already "knows" about Cornwall, rather than querying your up-to-date data.

11.8.2 Improving LLM tool usage with system guidance

To nudge the LLM toward tool use and away from hallucinations, let's make its instructions and tool descriptions clearer. Make a copy of your script as `main_02_02.py`, and update the tool definitions by adding a description of each tool:

```
@tool(description="""Search travel information
about destinations in England.""")
def search_travel_info(query: str) -> str:
    ...

@tool(description="Get the weather forecast, given a town name.")
def weather_forecast(town: str) -> dict:
    ...
```

Next, introduce a guiding `SystemMessage` in your `llm_node()`:

```
system_message = SystemMessage(content="""You are a helpful assistant
    that can search travel information and get the weather forecast.
    Only use the tools to find the information
    you need (including town names).""")
current_messages.append(system_message)
```

You can see the amended `llm_node()` function in the following listing.

Listing 11.11 Guiding tool selection

```
def llm_node(state: AgentState):
    """LLM node that decides whether to call the search tool."""
    current_messages = state["messages"]
    system_message = SystemMessage(content="""You are a helpful assistant
    that can search travel information and get the weather forecast.
    Only use the tools to find the information
    you need (including town names).""")

    current_messages.append(system_message)
    response_message = llm_with_tools.invoke(
        current_messages)

    return {"messages": [response_message]}
```

Defines the LLM node

Gets the current messages from the agent state

Adds a system message to the current messages to set the behavior of the assistant

Invokes the LLM model with the current messages. The LLM will decide whether to call the search tool or return an answer.

Appends the system message to the current messages

Returns the response message, which contains the tool call or the answer

Restart your application in debug mode, and ask the following:

```
You: Suggest two Cornwall beach towns with nice weather
```

At your first breakpoint in `llm_node()`, examine `current_messages` before and after appending the system message. Then, check `response_message`—now you'll see the following tool call:

```
{'name': 'search_travel_info', 'args': {'query': 'beach towns in Cornwall'},
'id': 'call_rJrYwfFG4BwaUPWBaQeUrn7o', 'type': 'tool_call'}
```

What does this mean? The LLM is now forced to use your semantic search tool for candidate beach towns and won't pick them from its own "knowledge." This minimizes hallucinations and guarantees the data used comes from your knowledge base.

Continue stepping through the code (press F5). Right before you submit the `current_messages` to the LLM again, your messages will look like this:

```
[HumanMessage(content='Suggest 2 Cornwall beach towns with nice weather',
additional_kwargs={}, response_metadata={}),
SystemMessage(content='You are a helpful assistant that can search travel
information and get the weather forecast. Only use the tools to find the
information you need (including town names).', additional_kwargs={},
response_metadata={}),
AIMessage(content=[], ..., tool_calls=[{'name': 'search_travel_info', 'args':
{'query': 'beach towns in Cornwall'}, 'id': 'call_rJrYwfFG4BwaUPWBaQeUrn7o',
'type': 'tool_call'}], ...),
ToolMessage(content='"<p id=\\"mwrg\\">Cornwall, in particular Newquay, is
the UK\'s <b id=\\"mwrw\\"><a rel=\\"mw:WikiLink\\" href=\\"//
```

```
en.wikivoyage.org/wiki/Surfing\\" title=\\"Surfing\\" id=\\"mwsA\\">surfing</
a></b> capital, with equipment hire and surf schools present on many of the
county\'s beaches, and events like the UK championships or Boardmasters
festival.</p>\\n---\\n...</section><section ...', name='search_travel_info',
tool_call_id='call_rJrYwfFG4BwaUPWBaQeUrn7o'),
SystemMessage(content='You are a helpful assistant that can search travel
information and get the weather forecast. Only use the tools to find the
information you need (including town names).', additional_kwargs={},
response_metadata={})]
```

From the `ToolMessage`, the LLM now receives a list of possible beach towns from your vector store.

Next, as you step through and reach `llm_node()` again, the LLM will issue calls for the weather forecast in two of these towns:

```
AIMessage(content=[], ..., tool_calls=[
{'name': 'weather_forecast', 'args': {'town': 'Newquay'}, 'id':
'call_OoDb7UwfrWF8c79DrfK2w9mp', 'type': 'tool_call'},
{'name': 'weather_forecast', 'args': {'town': 'St Ives'}, 'id':
     'call_4Q4IaMIU9qO6sgfp4ls2Lwxw', 'type': 'tool_call'}], ...)
```

The towns selected by the LLM might differ in your run, but they'll always come from the results of your semantic search tool.

Step through the weather tool calls. Each `ToolMessage` you inspect should look like this:

```
ToolMessage(content='{"town": "Newquay", "weather": "foggy", "temperature":
20}', name='weather_forecast', tool_call_id='...')
ToolMessage(content='{"town": "St Ives", "weather": "windy", "temperature":
22}', name='weather_forecast', tool_call_id='...')
```

What if the weather isn't good? If the weather is less than ideal in those towns, continue to the next LLM response, and you'll see the following:

```
AIMessage(content=[], ...,
tool_calls=[
{'name': 'weather_forecast', 'args': {'town': 'Perranporth'}, 'id':
'call_LOqaMnreszfHb5vItFCapRSk', 'type': 'tool_call'},
{'name': 'weather_forecast', 'args': {'town': 'Falmouth'}, 'id':
     'call_iaOeeYAIxK1lyShjwPdLWQ6b', 'type': 'tool_call'}], ...)
```

Here, the LLM is asking for the weather in two more towns, likely because the previous results didn't meet the "nice weather" requirement. This process continues until the agent finds two towns with suitable conditions. After the final tool calls, the LLM generates a synthesized, fact-based answer:

```
AIMessage(content=[{'type': 'text', 'text': 'Two beach towns in Cornwall with
nice weather currently are:\n\n1. Perranporth - The weather is sunny with a
temperature of 31°C.\n2. Falmouth - The weather is windy but still warm with
a temperature of 28°C.\n\nWould you like more information about these towns
or other beach towns in Cornwall?', 'annotations': []}]
```

This appears to the user as

```
UK Travel Assistant (type 'exit' to quit)
You: Suggest two Cornwall beach towns with nice weather
Assistant: [{'type': 'text', 'text': 'Two beach towns in Cornwall with nice
weather currently are:\n\n1. Perranporth - The weather is sunny with a
temperature of 31°C.\n2. Falmouth - The weather is windy but still warm with
a temperature of 28°C.\n\nWould you like more information about these towns
or other beach towns in Cornwall?', 'annotations': []}]
```

In summary, by enhancing tool descriptions and providing explicit instructions via system prompts, you can guide the LLM to chain tool use in a multi-step, fact-grounded workflow—first searching for beach towns, then filtering by real-time weather. This produces answers that are both dynamic and reliable.

> **EXERCISE** Try replacing the mock weather tool with a real API, such as `OpenWeatherMap`, using LangChain's `OpenWeatherMap` integration. This will make your agent truly real time!

11.9 Using prebuilt components for rapid development

Up to this point, you've built an agent from the ground up: wiring together the graph, orchestrating tool calls, and stepping through every detail in the debugger. You now have a solid grasp of how tool calling works under the hood.

However, for most production scenarios, you'll want to reduce boilerplate and move faster—while still retaining transparency and observability when needed. The LangGraph library provides prebuilt agent components, such as the ReAct agent, that encapsulate much of this orchestration logic for you. Now let's see how dramatically you can simplify your agent by switching to a prebuilt approach.

11.9.1 Refactoring to use the LangGraph ReAct agent

Start by copying your previous script (`main_02_02.py`) to a new file: `main_03_01.py`. This refactoring not only removes low-level orchestration code but also ensures that your agent follows a proven, well-tested interaction pattern designed specifically for reliable tool use.

IMPORTING THE REMAININGSTEPS UTILITY

At the top of your script, add the following import:

```
from langgraph.managed.is_last_step import RemainingSteps
```

REMOVING MANUAL TOOL BINDING

You can now delete the line where you bound tools to the LLM:

```
llm_with_tools = llm_model.bind_tools(TOOLS)
```

The prebuilt agent will handle this for you internally.

UPDATING THE AGENTSTATE

Modify your `AgentState` definition to include a `remaining_steps` field. This field allows the agent to manage how many tool-calling rounds are left in a controlled way:

```
class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], operator.add]
    remaining_steps: RemainingSteps
```

REMOVING NODE AND GRAPH CONSTRUCTION

Now for the biggest simplification: delete your custom `ToolsExecutionNode` and `llm_node`, as well as any explicit graph wiring code. Replace it all with a single instantiation of the built-in LangGraph ReAct agent:

```
travel_info_agent = create_react_agent(
    model=llm_model,
    tools=TOOLS,
    state_schema=AgentState,
    prompt="""You are a helpful assistant that
can search travel information and get the weather forecast.
Only use the tools to find the information you need
(including town names).""")
```

That's it! The ReAct agent now orchestrates the flow, tool calling, and synthesis for you.

11.9.2 Running the prebuilt agent

When you run `main_03_01.py` and ask your usual test question

```
You: Suggest two Cornwall beach towns with nice weather
Assistant: [{'type': 'text', 'text': 'Two beach towns in Cornwall with nice
weather are:\n\n1. St Ives - It has sunny weather with a temperature around
26°C.\n2. Newquay - It also enjoys sunny weather with a temperature around
22°C.\n\nBoth towns are popular for their beautiful beaches and pleasant
weather.', 'annotations': []}]
```

you get the correct, grounded answer—with much less code.

11.9.3 Observing and debugging with LangSmith

A common concern when switching to high-level abstractions is loss of visibility: How do you know the agent is actually following the right reasoning steps? While you can still debug tool functions directly, the flow inside the agent itself is less exposed. This is where LangSmith comes in. LangSmith enables full tracing and inspection of agent behavior, including tool calls, LLM reasoning, and intermediate states.

11.9.4 Enabling LangSmith tracing

To enable tracing, add the following lines to your `.env` file:

```
LANGSMITH_TRACING=true
LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
```

```
LANGSMITH_API_KEY="<your-langsmith-api-key>"
LANGSMITH_PROJECT="langchain-in-action-react-agent"
```

After rerunning your application and submitting a question, you can log into LangSmith (https://smith.langchain.com):

1. Select Tracing Projects in the Observability menu (left-hand sidebar).
2. Click your project (`langchain-in-action-react-agent`) in the right-hand panel.
3. Open the latest trace.

You'll see the full execution trace, as shown in figure 11.4.

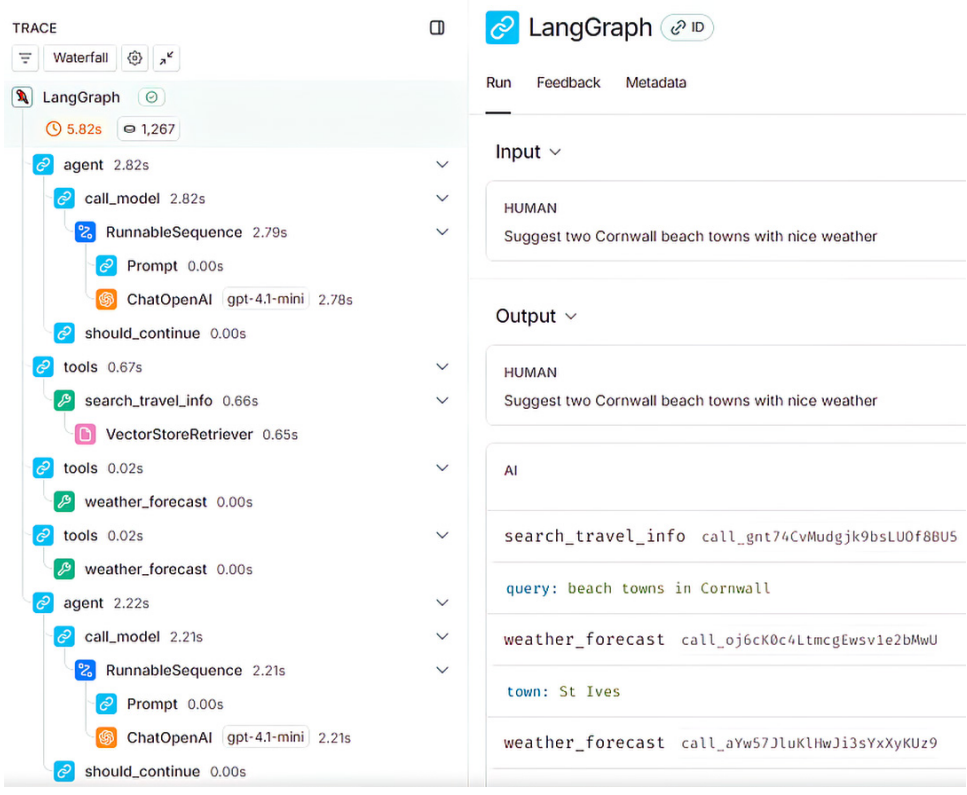


Figure 11.4 LangSmith agent execution trace. This trace visualizes each step of the agent's workflow—including LLM calls, tool invocations, and message flow—when answering the prompt, "Suggest two Cornwall beach towns with nice weather."

The LangSmith graphical trace shows every tool call, every LLM step, and the flow of messages—so even when you use a prebuilt agent, you retain the ability to audit, debug, and understand exactly what the agent is doing at every stage.

In summary, by combining prebuilt LangGraph agent components with LangSmith for observability, you can quickly build robust, production-ready agentic applications—while still maintaining full transparency and control when needed. This workflow has become the foundation for many modern AI agent systems in real-world use today.

Summary

- LangGraph builds agents using node-based architecture (explicit graphs with hardcoded conditional routing) or ReAct agents (built-in reasoning-action loops with automatic tool selection).
- Tool registration defines Python functions with type hints and docstrings that LLMs can discover and invoke. The docstring becomes the tool description that guides LLM selection.
- The LLM reads tool descriptions and decides which to call based on the user's question. Write clear, specific tool descriptions that explain when to use each tool and what inputs it expects.
- State inspection reveals which tool the LLM chose, what arguments it passed, what the tool returned, and how the agent used that output to continue or conclude. This aids in debugging and refinement.
- Tool descriptions must specify input parameters, expected outputs, and when to use the tool. For example: `"search_customer(name: str) -> dict: Finds customer records by name. Use when user asks about specific customer details."`
- Multi-tool workflows chain outputs across tools with the LLM orchestrating the sequence. For example, `search_customer → get_orders → calculate_total` happens automatically based on the user's request.
- The LLM manages the sequence without hardcoded logic. Trust the LLM to chain tools appropriately by providing clear tool descriptions and validating outputs through testing.
- The ReAct agent in LangGraph provides built-in tool calling, error management, and retry logic. This simplifies agent creation compared to manually building graphs with conditional edges.
- LangSmith traces capture the full execution flow. They show LLM reasoning steps, tool selection decisions, intermediate outputs, and final responses in a visual timeline.
- Each trace includes token counts, latency, and error states for debugging failed agent runs.
- Tool-calling agents form the foundation for multi-agent systems. Specialized agents (researcher, writer, reviewer) each have distinct tool sets and are coordinated by supervisor agents.
- Tool functions should return structured data (dictionaries, lists) not strings when possible. Structured returns enable the LLM to extract specific fields rather than parsing unstructured text.

- Tool selection depends on description quality. Test descriptions by running sample queries and checking if the agent selects the correct tool. Revise descriptions that cause selection errors.
- Implement custom error handling in tools by wrapping tool logic in `try-except` and returning structured error messages such as `{"error": "Customer not found", "details": "..."}`.

12 Multi-agent systems

This chapter covers

- Connecting tools to data sources
- Composing multi-agent systems using router and supervisor patterns
- Debugging, testing, and tracing multi-agent interactions

In chapter 11, we explored the foundations of building AI agents by creating a travel information agent capable of answering user queries about destinations, routes, and transportation options. While a single, specialized agent can be powerful, real-world applications often require the coordination of multiple agents, each handling a distinct area of expertise. In this chapter, we'll embark on that journey—transforming our travel information agent into a robust, multi-agent travel assistant system.

Imagine planning a trip where you not only need up-to-date travel information but also want to seamlessly book your accommodation. Our enhanced multi-agent travel assistant will do just that: it will be able to answer travel questions and help you

reserve hotels or bed and breakfasts (B&Bs) in your chosen destination. To achieve this, we'll begin by building a new agent—the accommodation booking agent.

The accommodation booking agent will empower users to book lodgings from two different sources. First, it will interface with a local accommodation database, which mainly features hotel deals and is exposed via a dedicated tool. Second, it will connect to an external B&B REST API, providing access to a wider selection of B&B options, also accessible through its own tool. Depending on user requests, the agent will use one or both of these tools to deliver relevant accommodation options.

Once we have our new agent in place, we'll combine it with the travel information agent from the previous chapter. The result will be a unified, multi-agent travel assistant capable of fielding a wide variety of travel-related queries, handling both information requests and accommodation bookings, and even combining both for a more streamlined experience. Let's begin by constructing our new accommodation booking agent.

12.1 Building an accommodation booking agent

To build a practical, helpful travel assistant, we need more than just information retrieval—we need the ability to act. In this section, we'll develop an accommodation booking agent from the ground up, starting by building the tools it needs: one for hotel bookings based on a local room availability database, and another for B&B bookings from an external REST API. By the end of this section, you'll have a ReAct-style agent that can check and book both hotel and B&B rooms in Cornwall.

12.1.1 Hotel booking tool

Let's start by creating the hotel booking tool. To enable our agent to retrieve hotel offers and availability, we'll use LangChain's SQL Database toolkit, which exposes a SQL database as a set of agent tools. This toolkit makes it straightforward for an agent to run queries, retrieve hotel details, and check room availability—all through tool calls, without needing to write raw SQL in your prompts.

The hotel data, including current offers and availability, is stored in a local SQLite database called `cornwall_hotels.db`, which is kept up-to-date by our backend partners. We don't need to worry about how the data is pushed—just trust that it's there and refreshed as needed.

> **NOTE** I recommend using the plain Command Prompt (cmd) rather than PowerShell to launch the SQLite shell, as PowerShell may return errors. This section assumes SQLite is already installed. If not, see section 10.3.1 in chapter 10, and see appendix D for installation steps, including how to add the installation folder to your system's `Path` environment variable.

First, copy the latest script, `main_03_01.py`, to a new script, `main_04_01.py`. Then, prepare your environment by following these steps:

1. Create a folder named `hotel_db`.
2. Place the provided SQL schema file `cornwall_hotels_schema.sql` into that folder.

3 Open a plain Command Prompt (inside Visual Studio Code [VS Code] or standalone using `cmd` from the Windows search box), navigate to the folder, and create the database with the following (I've omitted the root of the `ch11` folder for convenience):

```
\ch11>cd hotel_db
\ch11\hotel_db>sqlite3 cornwall_hotels.db < cornwall_hotels_schema.sql
```

Now let's check that the database is working. Open the SQLite shell:

```
\ch11\hotel_db>sqlite3 cornwall_hotels.db
```

Within the SQLite shell, run these queries to verify your setup:

```
sqlite> .tables

sqlite> SELECT * FROM hotels;

sqlite> SELECT * FROM hotel_room_offers;
```

With the database ready, let's move on to the Python implementation. Import the necessary LangChain SQL integration libraries:

```
from langchain_community.utilities.sql_database import SQLDatabase
from langchain_community.agent_toolkits import SQLDatabaseToolkit
```

Instantiate the SQLite database:

```
hotel_db = SQLDatabase.from_uri("sqlite:///hotel_db/cornwall_hotels.db")
```

Now create an instance of the SQL Database toolkit:

```
hotel_db_toolkit = SQLDatabaseToolkit(db=hotel_db, llm=llm_model)
```

That's it! Now you can access the toolkit's tools with

```
hotel_db_toolkit_tools = hotel_db_toolkit.get_tools()
```

12.1.2 B&B booking tool

Next, let's create a B&B booking tool. This tool will retrieve B&B room availability from a REST service. For development and testing, we'll mock this service.

First, we'll define the return type for our tool, and then create a mock implementation of the `BnBBookingService`, as shown in listing 12.1. (For convenience, the example here uses a reduced set of mock data. You can find the complete implementation in the code files provided with this book.).

Listing 12.1 BnBBookingService

```
class BnBBookingService:                          <-- Defines the B&B availability tool
    @staticmethod
    def get_offers_near_town(town: str, num_rooms: int) \
        -> List[BnBOffer]:                        <-- Calls the B&B booking service to get the offers
```

```
        mock_bnb_offers = [                                    <-- Mocked B&B offers
            {"bnb_id": 1, "bnb_name": "Seaside BnB",
            "town": "Newquay", "available_rooms": 3,
            "price_per_room": 80.0},
            {"bnb_id": 2, "bnb_name": "Surfside Guesthouse",
            "town": "Newquay", "available_rooms": 2,
            "price_per_room": 85.0},
            # Falmouth
            {"bnb_id": 3, "bnb_name": "Harbour View BnB",
            "town": "Falmouth", "available_rooms": 4,
            "price_per_room": 78.0},
            {"bnb_id": 4, "bnb_name": "Seafarer's Rest",
            "town": "Falmouth", "available_rooms": 1,
            "price_per_room": 90.0},,
            ...
            # Port Isaac
            {"bnb_id": 27, "bnb_name": "Port Isaac View BnB",
            "town": "Port Isaac", "available_rooms": 2,
            "price_per_room": 99.0},
            {"bnb_id": 28, "bnb_name": "Fisherman's Cottage BnB",
            "town": "Port Isaac", "available_rooms": 2,
            "price_per_room": 101.0},
            # Fowey
            {"bnb_id": 29, "bnb_name": "Fowey Quay BnB",
            "town": "Fowey", "available_rooms": 2,
            "price_per_room": 94.0},
            {"bnb_id": 30, "bnb_name": "Riverside Rest BnB",
            "town": "Fowey", "available_rooms": 2,
            "price_per_room": 96.0},
        ]
        offers = [offer for offer in
            mock_bnb_offers
            if offer["town"].lower() == town.lower()
               and offer["available_rooms"] >= num_rooms]
        return offers
```

Now we can define the `check_bnb_availability` tool in the following listing.

Listing 12.2 `check_bnb_availability` tool

```
@tool(description="""Check BnB room availability and
price for a destination in Cornwall.""")                <-- Defines the B&B availability tool
def check_bnb_availability(destination: str, num_rooms: int) \
  -> List[Dict]:                                        <-- Defines the input and return type of the B&B availability tool

    offers = BnBBookingService.get_offers_near_town(
        destination, num_rooms)
    if not offers:
        return [{"error": f"No available BnBs found
        ➥in {destination} for {num_rooms} rooms."}]
    return offers
    return offers
```

12.1.3 ReAct accommodation booking agent

With both the hotel and B&B booking tools ready, it's time to build the ReAct accommodation booking agent. This agent will use both tools in response to user requests. If the user doesn't specify a preference, the agent will search both hotels and B&Bs for available rooms:

```
BOOKING_TOOLS = hotel_db_toolkit_tools + \
   [check_bnb_availability]

accommodation_booking_agent = create_react_agent(
    model=llm_model,
    tools=BOOKING_TOOLS,
    state_schema=AgentState,
    prompt="""You are a helpful assistant that can check
    hotel and BnB room availability and price for a
    destination in Cornwall. You can use the tools to
    get the information you need. If the users does
    not specify the accommodation type, you should
    check both hotels and BnBs.""",
)
```

Defines the booking tools, which are the tools from the hotel database toolkit and the B&B availability tool

Creates the accommodation booking agent

You can now try out the agent by replacing the agent line in `chat_loop()` with

```
...
result = accommodation_booking_agent.invoke(state)
...
```

Let's run the `main_04_01.py` script in debug mode and ask the following question:

```
UK Travel Assistant (type 'exit' to quit)
You: Are there any hotel or BnB rooms available in Penzance?
```

After you press Enter, you might see an answer similar to the following (I'm reporting only the context of the `text` property):

```
I have found Penzance Pier BnB with available rooms at £95 per room, and
Cornish Charm BnB with 3 available rooms at £87 per room.

For hotels, Penzance Palace has 3 available rooms with prices of £130 for a
single room and £200 for a double room. Would you like to book a room or need
more information?
```

As you can see, the agent used both tools to retrieve results from both the hotel database and the mock B&B service.

At this point, your accommodation booking agent is working as expected. It's strongly recommended to debug the execution and inspect the LangSmith traces to better understand how the agent is reasoning and acting step-by-step.

Although you now have both a travel information agent and an accommodation booking agent, they are still disconnected. You can use either one or the other but not both in a unified experience. In the next section, we'll build a multi-agent travel

assistant that brings these capabilities together, providing a seamless experience for travel planning and accommodation booking.

12.2 Building a router-based travel assistant

So far, we've developed two independent agents—a travel information agent and an accommodation booking agent—each with specialized capabilities. While this modular approach is powerful, it raises an essential design question: How can we combine these agents to deliver a seamless user experience—one that can answer travel information queries and handle accommodation bookings in a single conversation?

A common and effective solution is to introduce a router agent. This agent acts as an intelligent entry point: it receives the user's message, determines which specialized agent should handle the request, and dispatches the task accordingly.

12.2.1 Designing the router agent

To implement our multi-agent travel assistant, begin by copying your previous script, `main_04_01.py`, to `main_05_01.py`. Next, we need to bring in some extra libraries to support graph-based workflows:

```
from langgraph.graph import StateGraph, END
from langgraph.types import Command
```

The next step is to clearly define the set of agents available for routing. We do this by declaring an enumeration for the two agent types:

```
class AgentType(str, Enum):
    travel_info_agent = "travel_info_agent"
    accommodation_booking_agent = "accommodation_booking_agent"
```

To ensure our router agent receives clear and structured decisions from the LLM, we define a Pydantic model that captures the LLM's output—specifying which agent should handle each query:

```
class AgentTypeOutput(BaseModel):
  agent: AgentType = Field(...,
    description="Which agent should handle the query?")
```

By configuring the OpenAI LLM client to produce responses in this structured format, we eliminate any need for string parsing or manual postprocessing:

```
llm_router = llm_model.with_structured_output(
    AgentTypeOutput)
```

The router will always produce a result of either `travel_info_agent` or `accommodation_booking_agent`.

12.2.2 Routing logic

The heart of the router agent is its system prompt, which concisely instructs the LLM how to classify each user request:

```
ROUTER_SYSTEM_PROMPT = (
    """You are a router. Given the following user message,
    decide if it is a travel information question
    (about destinations, attractions, or general travel info) """
    """or an accommodation booking question (about hotels,
    BnBs, room availability, or prices).\n"""
    """If it is a travel information question,
    respond with 'travel_info_agent'.\n"""
    """If it is an accommodation booking question,
    respond with 'accommodation_booking_agent'."""
)
```

With this system prompt, the router agent evaluates each user input and decides which specialist agent should take over. The router is implemented as the entry node of our LangGraph workflow, with the travel information agent and the accommodation booking agent as subsequent nodes. You can see the router implementation in the following listing.

Listing 12.3 Router agent node

```
def router_agent_node(state: AgentState) -> Command[AgentType]:
    """Router node: decides which agent
    should handle the user query."""
    messages = state["messages"]
    last_msg = messages[-1] if messages else None
    if isinstance(last_msg, HumanMessage):
        user_input = last_msg.content
        router_messages = [
            SystemMessage(content=ROUTER_SYSTEM_PROMPT),
            HumanMessage(content=user_input)
        ]
        router_response = llm_router.invoke(
            router_messages)
        agent_name = router_response.agent.value
        return Command(update=state,
            goto=agent_name)

    return Command(update=state,
        goto=AgentType.travel_info_agent)
```

Gets the messages from the state

Gets the last message from the messages list

Checks if the last message is a HumanMessage

Gets the content of the last message

Creates the router messages, including the system prompt and the user input

Invokes the router model, which returns the relevant agent name

Gets the agent name from the router response

Returns the command to update the state and go to the agent

If the last message isn't a HumanMessage, returns the command to update the state and go to the travel_info_agent (default agent)

If you examine the implementation in listing 12.3, you'll notice that the router extracts the user's message and submits it to the LLM along with the system prompt. The LLM returns a structured output of type `AgentTypeOutput`, which contains the agent name to which the request should be routed. The router then uses a `Command` to redirect the conversation flow to the selected agent node in the graph. In simple workflows like this one, the `Command` can hand off the unchanged state to the new node, but it also allows for state updates in more complex flows.

12.2.3 Building the multi-agent graph

At this point, you have all the components needed to assemble the graph-based multi-agent system. You can see the graph implementation in the following listing.

Listing 12.4 Router-based multi-agent travel assistant graph

```
graph = StateGraph(AgentState)
graph.add_node("router_agent", router_agent_node)
graph.add_node("travel_info_agent",
    travel_info_agent)
graph.add_node("accommodation_booking_agent",
    accommodation_booking_agent)

graph.add_edge("travel_info_agent", END)
graph.add_edge("accommodation_booking_agent", END)

graph.set_entry_point("router_agent")
travel_assistant = graph.compile()
```

Defines the graph
Adds the router agent node
Adds the travel info agent node
Adds the accommodation booking agent node
Adds the edge from the travel info agent to the end
Adds the edge from the accommodation booking agent to the end
Sets the entry point to the router agent
Compiles the graph

The workflow graph connects the router agent to the two specialized agents. Notably, the only explicit edges you define are those from the travel information agent and the accommodation booking agent to the end of the workflow. The connection from the router to the specialized agents is determined dynamically at runtime by the LLM's response and is handled via `Command`.

Figure 12.1 is a graphical representation of the current multi-agent travel assistant graph. The router agent dispatches user queries to either the travel information agent or the accommodation booking agent, each equipped with their own specialized tools.

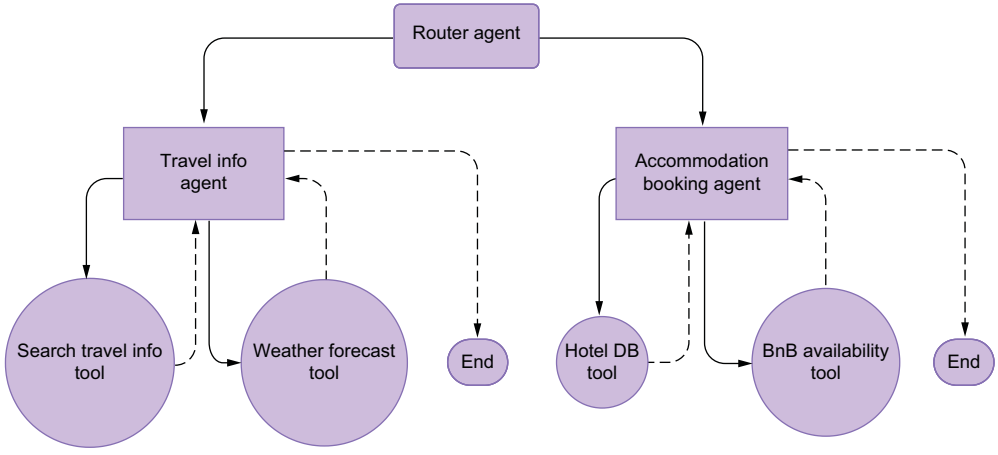


Figure 12.1 Router-based multi-agent travel assistant

As the diagram shows, this is a hybrid architecture. At the top, the system exhibits a deterministic, workflow-driven routing logic. At the lower level, each specialist agent uses its own set of tools (e.g., travel data APIs or accommodation booking interfaces) and follows a tool-based decision process, which is inherently more flexible and dynamic.

12.2.4 Trying out the router agent

To see the system in action, run your multi-agent travel assistant by starting `main_05_01.py` in debug mode, and try the following two user queries:

- What are the main attractions in St Ives?
- Are there any rooms available in Penzance this weekend?

One important thing to note with this design is that each user question is routed to a single agent for handling—in other words, each query takes a one-way ticket through the workflow. The router makes a clean and unambiguous handoff, and the selected agent responds directly to the user before the workflow ends. For example, when you ask

> What are the main attractions in St Ives (Cornwall)?

the request is routed to the travel information agent. You can see the related LangSmith execution trace in figure 12.2.

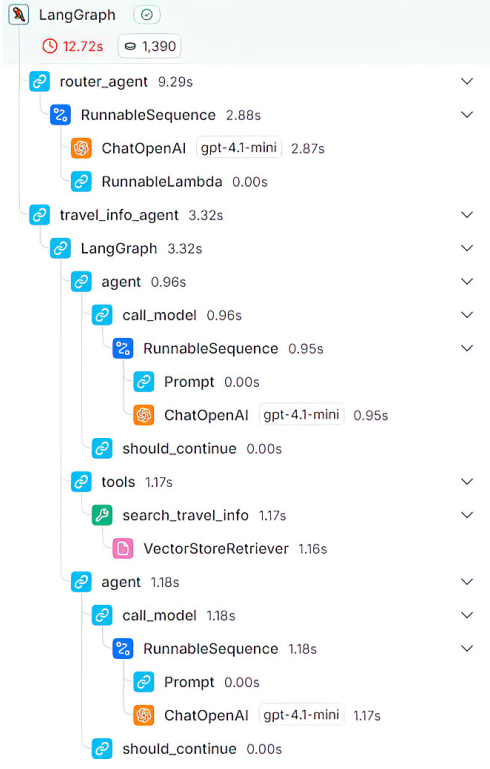


Figure 12.2 LangSmith execution trace of a travel information question

If you ask

> Are there any hotel or BnB rooms available in Penzance this weekend?

the system dispatches the query to the accommodation booking agent.

In both cases, the workflow (see figure 12.2) is clear: the router agent evaluates the intent and hands the query off to the most appropriate specialist agent, which then handles the response and ends the session. This modular, graph-based design provides a strong foundation for more advanced workflows. In later sections, you'll see how you can evolve this system to handle more complex, multi-step, or even collaborative agentic scenarios.

12.3 Handling multi-agent requests with a Supervisor component

The workflow-based multi-agent architecture we developed in the previous section works well for simple, single-purpose queries—questions that can be clearly routed to either the travel information agent or the accommodation booking agent. But what happens when a user asks for something that spans both domains? With our previous router-based design, a question such as

> Can you find a nice seaside Cornwall town with good weather right now and find availability and price for one double hotel room in that town?"

can't be answered effectively, as it requires both agents to work together and share intermediate results.

To solve this, we need to shift our architecture toward a more flexible, collaborative agent system—one where multiple specialized agents can be coordinated as subtools under a higher-level manager. In LangGraph, this is exactly the use case for the Supervisor: a built-in component designed to orchestrate multiple agents, allowing them to collaborate on complex requests.

12.3.1 The Supervisor pattern: An agent of agents

Conceptually, the Supervisor is an "agent of agents": it acts as an orchestrator, managing a collection of other agents (which may themselves use tools) and deciding which agent to activate, possibly multiple times in a single workflow. Each agent acts as a specialized tool that the Supervisor can invoke as needed. Let's see how to set up this pattern in your multi-agent travel assistant. Start by copying one of your previous implementations, `main_04_01.py`, to a new script, `main_06_01.py`. Next, install the necessary package:

```
pip install langgraph-supervisor
```

Then, import the Supervisor:

```
from langgraph_supervisor.supervisor import create_supervisor
```

When defining agents to be managed by the Supervisor, it's important to assign each a unique name. You can see how to instantiate your agents with names in the following listing.

Listing 12.5 Setting up the leaf agents

```
travel_info_agent = create_react_agent(
    model=llm_model,
    tools=TOOLS,
    state_schema=AgentState,
    name="travel_info_agent",
    prompt="""You are a helpful assistant that can search
    travel information and get the weather forecast.
    Only use the tools to find the information you
    need (including town names).""",
)

accommodation_booking_agent = create_react_agent(
  model=llm_model,
  tools=BOOKING_TOOLS,
  state_schema=AgentState,
  name="accommodation_booking_agent",
  prompt="""You are a helpful assistant that can check
    hotel and BnB room availability and price for a
    destination in Cornwall. You can use the tools
    to get the information you need. If the user
    does not specify the accommodation type, you
    should check both hotels and BnBs.""",
)
```

Now you can implement your travel assistant as a Supervisor, as shown in the following listing.

Listing 12.6 Setting up the Supervisor agent

```
travel_assistant = create_supervisor(
    agents=[travel_info_agent,
        accommodation_booking_agent],
    model= ChatOpenAI(model="gpt-5",
        use_responses_api=True),
    supervisor_name="travel_assistant",
    prompt=(
        """You are a supervisor that manages two agents:
        a travel information agent and an accommodation booking agent. """
        """You can answer user questions that might require
        calling both agents when needed. """
        """Decide which agent(s) to use for each user request
        and coordinate their responses."""
    )
).compile()
```

Creates the Supervisor

Defines the agents to be used by the Supervisor

Defines the LLM to be used by the Supervisor as a high-grade model such as GPT-5

Defines the prompt for the Supervisor (the system prompt) that will be used to guide the Supervisor's behavior

Compiles the Supervisor, which is a LangGraph graph

You'll notice that configuring a Supervisor is much like setting up a ReAct agent, but instead of passing a list of tools, you provide a list of agents. Because the Supervisor needs to analyze complex multi-step requests and coordinate several agents, it's best to use a more powerful LLM (e.g., GPT-5) to maximize accuracy and task decomposition.

> **TIP** Try experimenting with different models for the Supervisor, such as GPT-5, and compare how well the assistant handles increasingly complex, multifaceted questions.

As in previous designs, simply update your chat loop to invoke the Supervisor-based travel assistant:

```
result = travel_assistant.invoke(state)
```

12.3.2 From "one-way" to "return ticket" interactions

Unlike the workflow-based router—where every user question was routed once and only once to a specific agent (a one-way ticket)—the Supervisor enables a much richer interaction. The Supervisor can invoke each agent as needed, potentially revisiting agents multiple times (return tickets) in a single session to collect, combine, and reason over intermediate results. This enables the system to handle more sophisticated, open-ended, and multipart queries.

In figure 12.3, you can see a diagram representing this Supervisor-based architecture in which both the high-level (Supervisor) and low-level (Agent/Tool) orchestration follow a tool-based approach, maximizing flexibility and composability. The

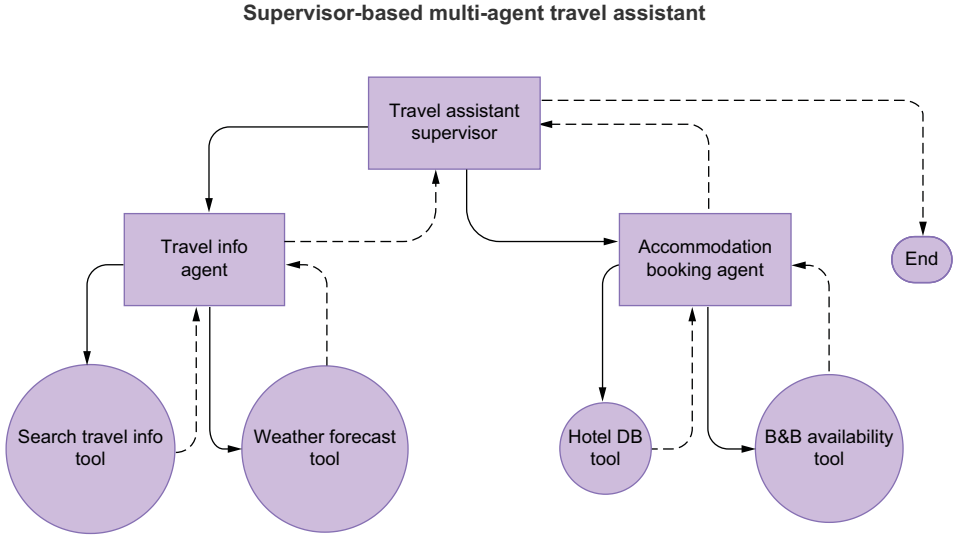


Figure 12.3 The router agent dispatches user queries to either the Travel Information Agent or the Accommodation Booking Agent, each equipped with their own specialized tools.

Supervisor becomes the central decision-maker, ensuring the right agent (or sequence of agents) is activated for every complex travel request.

This Supervisor-driven architecture unlocks a new level of multi-agent collaboration, laying the groundwork for more advanced, open-ended AI travel assistants capable of addressing real-world, multi-step needs.

12.3.3 Trying out the Supervisor agent

Now run the travel assistant by starting `main_06_01.py` in debug mode (with LangSmith tracing enabled), and try entering a complex, multi-part question such as the following:

```
UK Travel Assistant (type 'exit' to quit)
You: Can you find a nice seaside Cornwall town with good weather right now
and find availability and price for one double hotel room in that town?
```

When you examine the LangSmith trace, you'll notice a more intricate agent utilization trajectory, similar to that in figure 12.4, in which the `travel_assistant` is the Supervisor agent.

In the following, I've summarized the key steps from the execution trace so you can understand the flow better (remember the `travel_assistant` is the Supervisor):

- travel assistant
  - tools
    transfer_to_travel_info_agent
- travel_info_agent
  - tools
    search_travel_info
  - tools
    weather_forecast
- travel_assistant
  - tools
    transfer_to_accommodation_booking_agent
- accommodation_booking_agent
  - tools
    check_bnb_availability
  - tools
    sql_db_query

This shows that the Supervisor-based travel assistant was able to coordinate both agents, using each one and its underlying tools as needed to fully answer the user's question. Technically, agents are now orchestrated as tools, and the Supervisor manages this collaboration dynamically.

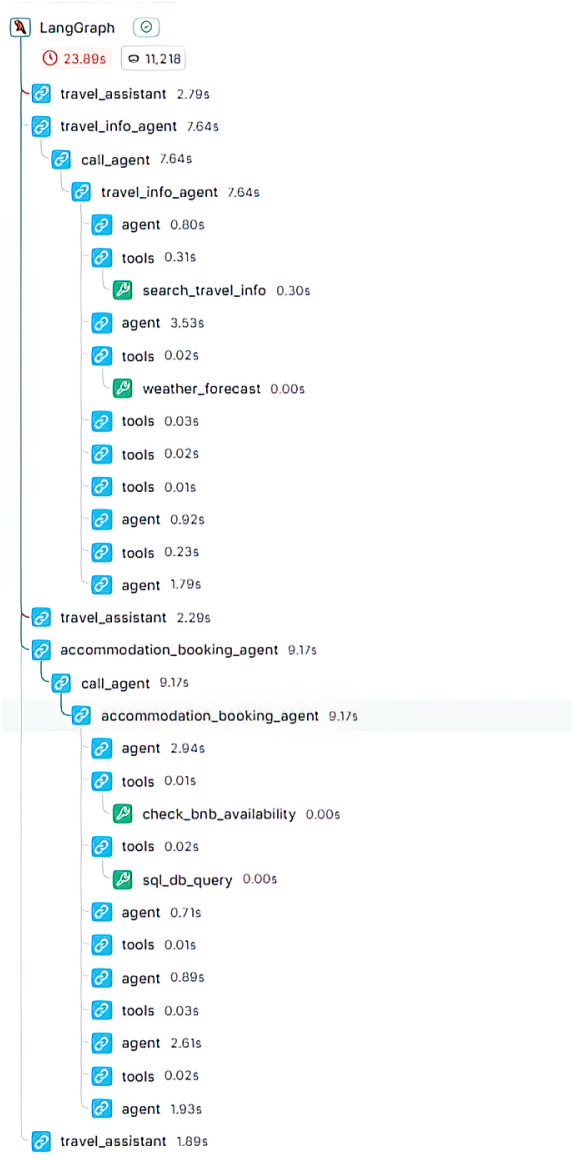


Figure 12.4 LangSmith execution trace of a combined travel information and booking question

Summary

- Agent tools extend beyond information retrieval to perform actions. They query SQL databases for customer records, send emails, create Jira tickets, or execute trading orders through broker APIs.
- Router agents classify incoming queries using LLM-based classification and delegate to specialist agents. A customer service router sends billing questions to the billing agent and technical issues to the support agent.

- Supervisor agents coordinate multi-agent workflows by decomposing complex requests into subtasks. A research supervisor might delegate web search to one agent, database queries to another, and synthesis to a third.
- LangSmith traces display decision trees. They show which agent received each query, what tools it called, intermediate results, and final handoffs to other agents or back to the user. This enables identifying bottlenecks such as slow tool calls or agents selecting incorrect tools for specific query types.
- Multi-agent systems require careful prompt engineering for each agent's role. Each agent should have a clearly defined scope, tools list, and examples of queries it should handle.
- Router agents should include explicit criteria for edge cases such as ambiguous queries. Define a default route or request clarification when classification confidence is below a threshold.
- Supervisor agents need visibility into specialist agent capabilities. Provide tool lists and capability descriptions from each specialist agent to the supervisor's planning prompt.
- When agents pass data between each other, use structured formats (JavaScript Object Notation [JSON]) not natural language. Parsing unstructured inter-agent communication increases error rates.
- To create a router with tool calling, define routing as a tool where the LLM selects which specialist agent to invoke based on query classification.
- When implementing the supervisor pattern, the supervisor agent has tools that invoke other agents. Each specialist agent is wrapped as a tool that the supervisor can call.
- To track agent handoffs in state, include `current_agent` and `agent_history` fields in your state object to monitor which agent handled each step of the workflow.
- To test router accuracy, create a labeled dataset of queries with correct agent assignments, measure classification accuracy, and adjust routing logic based on the error analysis.
- The Supervisor pattern goes further, orchestrating collaboration among agents, so multiple agents can be used together to answer complex, multipart questions.
- Tracing and debugging with LangSmith provides valuable visibility into agent decisions and tool usage, making it easier to optimize and extend your system.

13 Building and consuming MCP servers

This chapter covers

- Understanding the purpose and architecture behind the Model Context Protocol (MCP)
- Building and exposing your own MCP server, with a practical weather tool example
- Testing and consuming MCP servers and related tools in applications
- Integrating remote MCP tools into agents alongside local tools

Building AI agents that can reliably access and use external context is one of the central challenges for application developers. Until recently, integrating data from multiple sources meant wrapping each one as a tool, often using different protocols, resulting in a repetitive, time-consuming task duplicated across teams.

The *Model Context Protocol* (MCP), introduced by Anthropic, solves this problem by defining a unified way for services to expose tools through MCP servers. Agents, or MCP hosts, connect to these servers via MCP clients, discovering and invoking remote tools as easily as local ones. This shifts integration work to where it

belongs—at the source—so developers can focus on building capable agents rather than re-implementing the same wrappers. Once connected, MCP tools slot seamlessly into existing agent architectures.

Since its release in late 2024, MCP has quickly become a de facto standard. Major large language model (LLM) providers such as OpenAI and Google have adopted it in their APIs and SDKs, and a growing ecosystem of companies and communities are publishing services as MCP servers. Thousands of tools are already available on public MCP portals, ready to integrate into new or existing projects with minimal effort.

In this chapter, you'll get hands-on with MCP. We'll cover the core protocol and architecture, explore the expanding ecosystem, and point you to both official and community MCP servers. You'll then build your own MCP server—starting with a weather data example using AccuWeather—test it, and integrate it into an agent application. Finally, we'll look at combining remote MCP tools with local logic, configuring clients, and following best practices for reliable integration.

As MCP adoption grows, the ability to build and consume MCP servers is becoming essential for both developers and service providers. Let's dive in and see how MCP is shaping the next generation of context-rich AI applications.

13.1 Introduction to MCP servers

Modern AI agents depend on external sources of context—databases, APIs, and specialized services. Traditionally, each of these had to be wrapped as a tool that agents could call, usually via ad hoc protocols supported by LLMs. This approach works but comes at a cost: every team ends up writing and maintaining similar wrappers, leading to duplicated effort and inconsistent integrations across the ecosystem.

MCP takes a different path. Instead of forcing every developer to reinvent the wheel, it allows context providers themselves to expose their data and services as tools through a common protocol. Agents can then consume these tools directly, without custom glue code. As illustrated in figure 13.1, this unified approach eliminates much of the friction in agent development and fosters a richer, more reusable ecosystem.

13.1.1 The problem: Context integration at scale

As we saw in previous chapters, LLM agents typically consume external data via tools injected into their requests. Each new context source—a weather API, a document database, a search service—means creating yet another wrapper tool and integrating it with the agent, often following slightly different conventions and protocols. This process is not only repetitive but also results in the same work being duplicated by countless teams.

Imagine every agent developer spending time wrapping the same set of public APIs, or every organization hand-rolling integrations for standard services. This approach simply doesn't scale, especially as the range of possible tools and context sources grows.

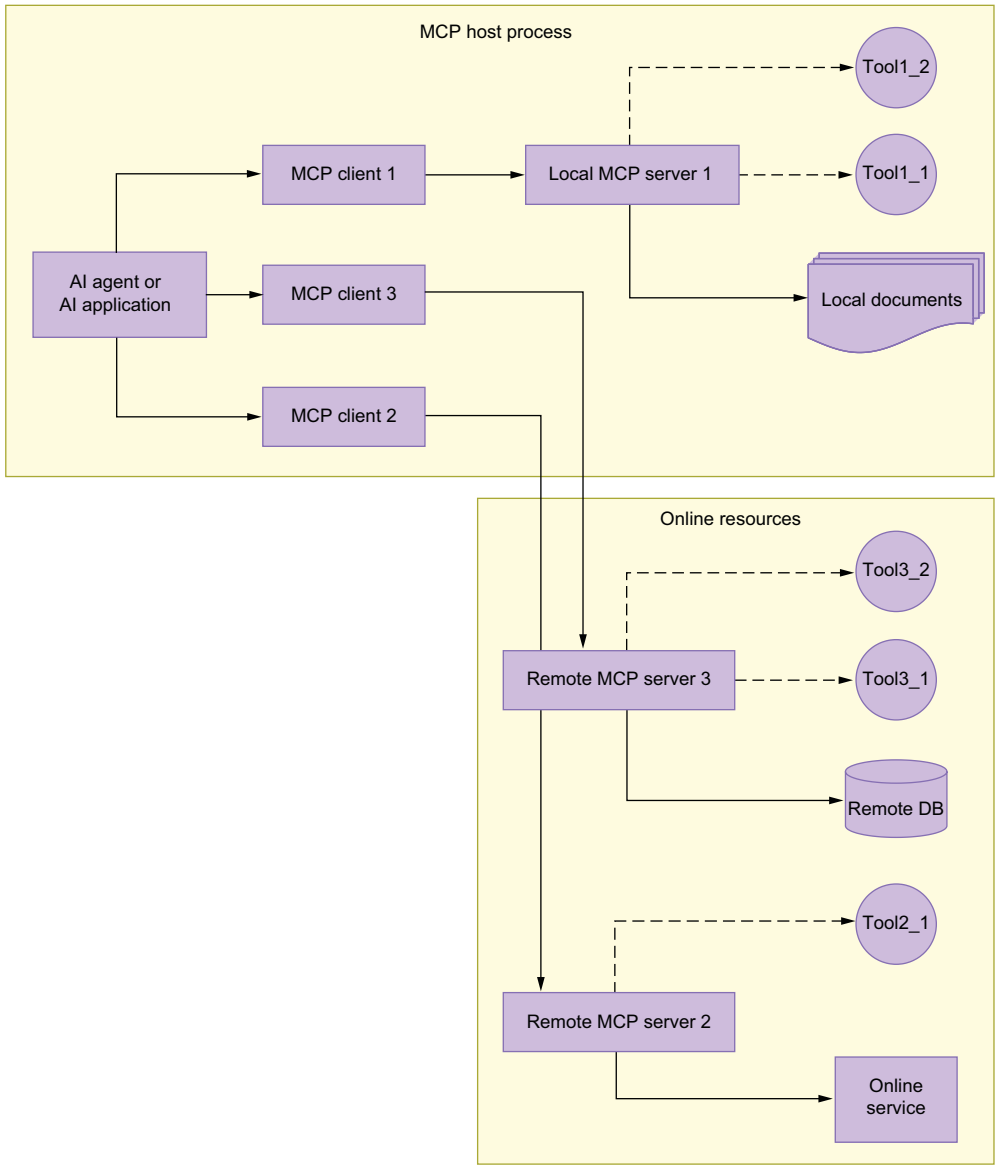


Figure 13.1 MCP host process connecting to multiple local and remote MCP servers via MCP clients, each exposing tools backed by various resources. This architecture enables agents to flexibly access both local documents and online services through standardized tool interfaces.

13.1.2 The solution: The Model Context Protocol

MCP addresses this challenge by providing a standardized way for data and service providers to expose tools via MCP servers. Instead of each agent individually wrapping every service, developers can simply "subscribe" to MCP servers and use the tools they

expose, with minimal additional work. The protocol defines a classic client/server architecture, as you can see in figure 13.1, shown previously.

Here, the MCP host process—the agent or application—connects to one or more MCP servers using an MCP client. Each MCP server can expose a collection of tools, and the agent can use only the tools it needs. This architecture is similar to that of REST APIs and WebSockets, which expose endpoints for consumption through a client and makes it easy to add or swap out new sources of context as requirements change.

As you also saw earlier in figure 13.1, MCP servers can be either remote (accessed over Streamable HTTP, usually in production) or local (accessed via standard input/output [STDIO], typically for development or for accessing local resources such as files). In most cases, and throughout this chapter, we'll assume you're working with remote MCP servers (even if they are running on your computer).

Once configured, tools from MCP servers integrate with your agent in exactly the same way as local tools, adhering to the same tool calling protocols you learned in earlier chapters. If your agent is modular and well-architected, you often don't need to change any logic at all to support remote MCP tools.

> **NOTE** MCP goes beyond just tools—it can also standardize how prompts, files, and other resources are shared. For our purposes, we'll focus on tool exposure, but for a deeper dive, see the original "Introducing the Model Context Protocol" article by Anthropic (www.anthropic.com/news/model-context-protocol).

13.1.3 The MCP ecosystem

Since its introduction in late 2024, MCP has rapidly gained traction. Leading LLM providers such as OpenAI and Google have both adopted MCP, integrating support into their APIs and agent SDKs. On the provider side, companies are increasingly wrapping their services and data with MCP servers, making them instantly "AI-ready" for a wide range of agents and applications.

A number of public MCP server portals have emerged, making it easy to find tools for almost any need. Table 13.1 highlights some of the most prominent directories.

Table 13.1 MCP server portals

Portal	Description
https://github.com/modelcontextprotocol/servers	Anthropic's official MCP portal, listing both official and community servers
https://mcp.so/	A community-driven directory featuring more than 16,000 servers
https://smithery.ai/	A portal with more than 5,000 tools, mostly MCP-compliant
https://mcpservers.org/	A collection of around 1,500 servers

With such a broad and growing ecosystem, agents can increasingly draw on a shared library of tools—whether built in-house, offered by major companies, or shared by the community.

After this overview of the motivation, architecture, and ecosystem behind MCP servers, we're ready to look at how to build, expose, and consume these tools in real-world applications. In the next sections, we'll explore how to create your own MCP server and how to integrate MCP tools seamlessly into your agents.

13.2 How to build MCP servers

With a clear understanding of what MCP servers are and their role in the modern AI agent ecosystem, the next step is to learn how to actually build, deploy, and integrate MCP servers. This section will guide you through the key resources, official tools, and best practices to help you create robust MCP servers and make them available to agents and applications.

13.2.1 Essential resources for MCP server development

Before diving into code, it's important to familiarize yourself with the foundational documentation and tools for MCP development. The official hub for the protocol is https://modelcontextprotocol.io, which provides a wealth of information on the architecture, design principles, tutorials, and complete protocol specification. Whether you're new to MCP or looking for advanced features, this site should be your starting point.

> **TIP** Pay particular attention to the MCP protocol specification (https://modelcontextprotocol.io/specification), which is regularly updated and details every aspect of the protocol—including transport mechanisms, security considerations, and best practices. The specification itself is technology-agnostic, making it valuable no matter which language or platform you use.

While understanding the protocol is crucial, implementing MCP from scratch in your own application isn't recommended. Doing so would mean duplicating a significant amount of effort, and you'd likely end up reimplementing features already solved in mature SDKs. Instead, you should use one of the official language-specific SDKs available for MCP.

13.2.2 Official language-specific MCP SDKs

The official MCP GitHub repository aggregates a variety of resources, with particular emphasis on the official SDKs for several major programming languages—including Python, JavaScript, Java, Kotlin, and C#. This book will primarily focus on Python, but the same general principles apply to the other supported languages.

FastMCP 1

The initial Python SDK, often referred to as FastMCP 1, is available at https://mng.bz/dWpX. This library provided the community with the first robust framework for building and consuming MCP servers in Python.

FastMCP 2

Building on the experience with FastMCP 1, the MCP community released FastMCP 2—a major upgrade that addresses limitations of the original implementation and aligns more closely with the latest protocol specifications. FastMCP 2 offers significant improvements:

- Easier deployment and server composition
- Enhanced security features
- Improved client connectivity and advanced capabilities, such as dynamic tool rewriting
- Built-in testing utilities and integration hooks for other libraries

FastMCP 2 is actively maintained at https://github.com/jlowin/fastmcp, and you'll find comprehensive documentation and tutorials at https://gofastmcp.com/getting-started/welcome. We'll use FastMCP 2 for hands-on examples, so keep these resources handy as you follow along.

13.2.3 Consuming MCP servers in LLM applications and agents

While the previous sections focused on building MCP servers, it's equally important to understand how to consume these servers in your own AI-powered applications. Thanks to wide adoption, integrating MCP tools has become remarkably straightforward—especially with leading platforms such as OpenAI.

OpenAI's API natively supports tools provided by public MCP servers via the Responses API. Not only can you discover and reference these tools, but OpenAI will also execute them for you—eliminating the need for manual client code in many cases.

> **TIP** Review the OpenAI documentation on remote MCP tool integration. The process is simple, but it's recommended to carefully consider your authorization strategy—deciding whether to approve such calls automatically or interactively.

In many enterprise environments, MCP servers might be available only within organizational boundaries, so you can't expect the OpenAI Responses API to execute the remote tools for you. For these scenarios, there are two primary approaches:

- *Use the FastMCP client.* The official FastMCP SDK provides client facilities to connect, authenticate, and consume tools from MCP servers directly.
- *Take advantage of LangChain/LangGraph integrations.* If you're developing agents with LangGraph or LangChain, you can use the LangChain MCP client library—particularly the `MultiServerMCPClient` class—to easily aggregate and consume tools from multiple MCP servers through a simple configuration interface.

In the following sections, we'll demonstrate both approaches in practice. You'll learn how to build, test, and integrate a practical MCP server into an agent workflow—

whether you're working directly with SDKs or using modern agent frameworks such as LangChain.

13.3 Building a weather MCP server

After learning about what MCP Servers are, their purpose, and the available libraries and ecosystem, it's time to build one! In this section, we'll replace the mockup weather tool that you used to build agents in previous chapters with a real-world weather MCP server based on the AccuWeather REST API. We'll also integrate this server into one of the agent-based solutions built earlier. Step-by step, you'll see how to build, test, and connect the MCP server to your agent.

13.3.1 Implementing the MCP server

We begin by replacing our previous mock weather tool with a proper MCP server that exposes live weather data from AccuWeather. Before implementing the code, go to the AccuWeather developer portal and register for free at https://developer.accuweather.com/signup to claim an API key.

After completing registration, you'll be redirected to the subscriptions page (https://developer.accuweather.com/subscriptions), where you'll see your Default App and its associated API key. Copy this key. Next, add it to your project's `.env` file, as shown here (replacing the placeholder with your actual key):

```
ACCUWEATHER_API_KEY=<Your API key>
```

You're now ready to implement a real MCP server that exposes the weather service. Create a folder named `mcp` within the `ch11` folder, and then create an empty Python script called `accuweather_mcp.py`. You can see the MCP server implementation in the following listing, adapted from this GitHub repository: https://github.com/adhikasp/mcp-weather.

Listing 13.1 AccuWeather MCP server

```
import os
import json
from typing import Dict
from fastmcp import FastMCP
from dotenv import load_dotenv
from aiohttp import ClientSession

load_dotenv()                                    # Loads environment variables

mcp = FastMCP("mcp-accuweather")                 # Initializes FastMCP

@mcp.tool(description="""Get weather conditions  # Defines the MCP tool
for a location.""")
async def get_weather_conditions(location: str) -> Dict:
    """Get weather conditions for a location."""
    api_key = os.getenv("ACCUWEATHER_API_KEY")   # Gets the AccuWeather API key
    base_url = "http://dataservice.accuweather.com"
```

```
    async with ClientSession() as session:
        location_search_url = f"{base_url}/locations/v1/cities/search"
        params = {                                          Parameters for
            "apikey": api_key,                              location search
            "q": location,
        }
        async with session.get(location_search_url,
            params=params) as response:
            locations = await response.json()                    Gets locations
            if response.status != 200:
                raise Exception(f"""Error fetching location
                data: {response.status}, {locations}""")
            if not locations or len(locations) == 0:
                raise Exception("Location not found")
        location_key = locations[0]["Key"]                  Gets the
                                                            location key
        current_conditions_url =
        ➥f"{base_url}/currentconditions/v1/{location_key}"
        params = {                                          Current conditions
            "apikey": api_key,                              parameters
            "details": "true"
        }
        async with session.get(current_conditions_url,
        params=params) as response:
            current_conditions = \                          Gets current
               await response.json()                        conditions

        if current_conditions and len(current_conditions) > 0:
            current = current_conditions[0]                 Formats current
            current_data = {                                conditions
                "temperature": {
                    "value": current["Temperature"]["Metric"]["Value"],
                    "unit": current["Temperature"]["Metric"]["Unit"]
                },
                "weather_text": current["WeatherText"],
                "relative_humidity": current.get("RelativeHumidity"),
                "precipitation": current.get("HasPrecipitation", False),
                "observation_time": current["LocalObservationDateTime"]
            }
        else:
            current_data = "No current conditions available"
                                                                Returns
                                                                structured content
        return {
            "location": locations[0]["LocalizedName"],
            "location_key": location_key,
            "country": locations[0]["Country"]["LocalizedName"],
            "current_conditions": current_data,
        }

if __name__ == "__main__":
    mcp.run(transport="streamable-http",                    Runs the
        host="127.0.0.1",                                   MCP server
        port=8020, path="/accu-mcp-server")
```

This implementation relies on the `fastmcp` package (i.e., FastMCP 2), which should already be installed in your virtual environment, as it's listed in the `requirements.txt` file. The core logic of the implementation is simple: it searches the underlying AccuWeather locations REST endpoint to resolve the user's input and then queries current weather conditions using AccuWeather's API.

Now open a new terminal (within Visual Studio Code [VS Code] or otherwise), activate your virtual environment, move into the `mcp` folder, and run the server:

```
C:\Github\building-llm-applications\ch11>env_ch11\Scripts\activate
(env_ch11) C:\Github\building-llm-applications\ch11>cd mcp
(env_ch11) C:\Github\building-llm-applications\ch11\mcp>
➥python accuweather_mcp.py
```

You'll see the MCP server starting up, and finally, you'll see this output:

```
←[32mINFO←[0m:     Started server process [←[36m20712←[0m]
←[32mINFO←[0m:     Waiting for application startup.
←[32mINFO←[0m:     Application startup complete.
←[32mINFO←[0m:     Uvicorn running on ←[1mhttp://0.0.0.0:8020←[0m
                   (Press CTRL+C to quit)
```

Congratulations! Your first MCP server is up and running!

13.3.2 Trying out the MCP server with MCP Inspector

One of the fastest ways to interactively test your newly created MCP server is by using MCP Inspector. This tool provides a user-friendly interface that lets you connect to any MCP server, explore its tools, and run live queries without needing to write any client code. The process is straightforward, and MCP Inspector is a great way to build confidence before integrating the server into your applications.

INSTALLING MCP INSPECTOR

To get started, you'll need to install MCP Inspector on your computer. MCP Inspector is a Node.js application, so ensure you have Node.js installed. If not, you can download and install it from https://nodejs.org.

Once Node.js is ready, open a new command prompt or terminal. It's a good practice to keep your tools organized, so create a new folder under your project directory—such as `mcp-inspector` inside `ch11`. Then, run the following command to launch MCP Inspector using `npx`:

```
c:\Github\building-llm-applications\ch11\mcp-inspector>
➥npx @modelcontextprotocol/inspector
```

During installation, you'll be asked to confirm before proceeding (you might get a slightly different version number):

```
Need to install the following packages:
@modelcontextprotocol/inspector
Ok to proceed? (y) y
```

After confirmation, the installation will proceed, and MCP Inspector will automatically launch in your browser:

```
Starting MCP inspector...
⚙ Proxy server listening on localhost:6277
? Session token:
8722a69c8ccf491da9862c288957a8cb4451b88d1f5e0539767d2b601d42c1e6
   Use this token to authenticate requests or set DANGEROUSLY_OMIT_AUTH=true
   to disable auth

? MCP Inspector is up and running at:
   http://localhost:6274/?MCP_PROXY_AUTH_TOKEN=8722a69c8ccf491da9862c288957a8
   cb4451b88d1f5e0539767d2b601d42c1e6

? Opening browser...
```

Connecting to your Weather MCP server

Once MCP Inspector is running, your browser should automatically open a new tab with the MCP Inspector interface, similar to what is shown in figure 13.2 (it might have changed a bit by the time the book is published).

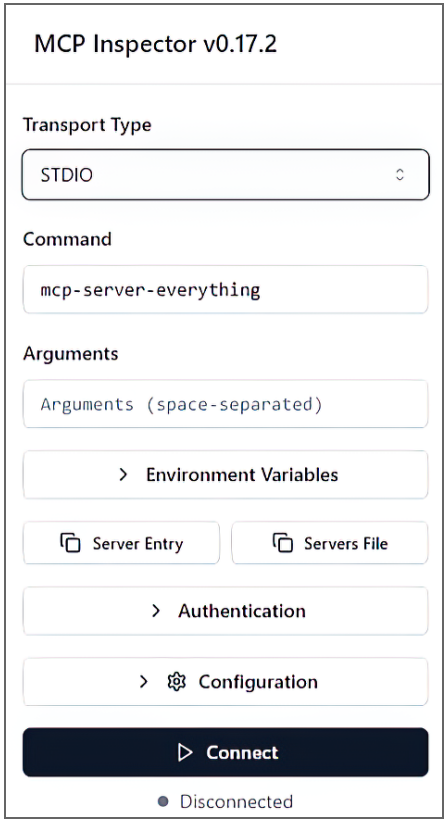


Figure 13.2 MCP Inspector after installation

In the interface, you'll see several options on the left (see figure 13.3). To connect to your Weather MCP server, make the following configurations:

- *Transport Type*: Enter `Streamable HTTP`.
- *URL*: Enter `http://127.0.0.1:8020/accu-mcp-server`.
- *Connection Type*: Select Via Proxy.
- *Authentication*: Click the Authentication panel, and disable authentication by switching the toggle to the left.

When finished, click the Connect button. The MCP Inspector will now attempt to connect to your MCP server. When the connection is successful, you'll see a green Connected status on the left-hand panel, as shown in the figure, confirming that your MCP server is up and running.

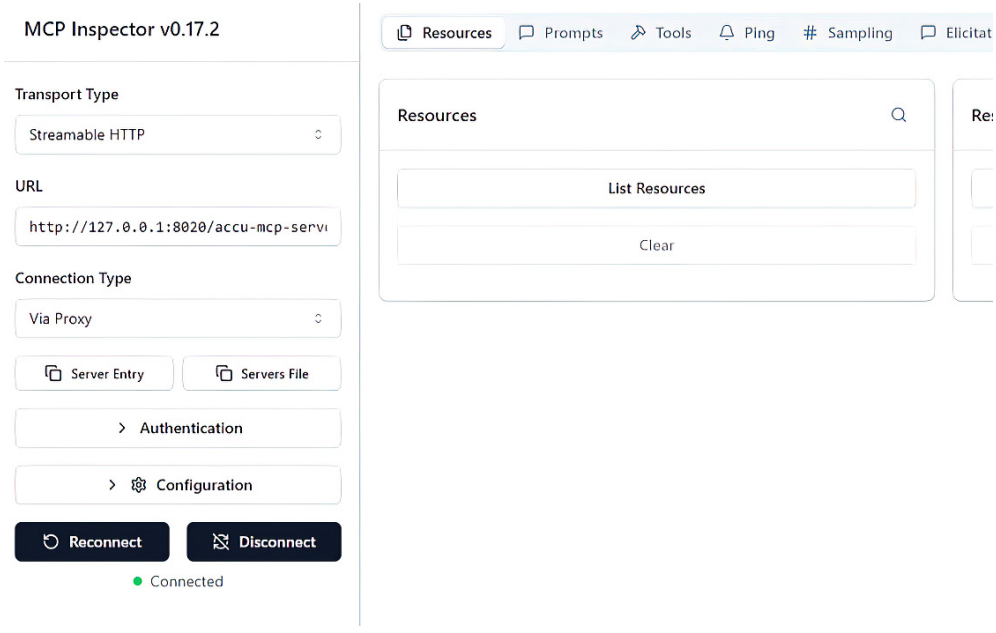


Figure 13.3 The MCP Inspector after connecting to the Weather MCP server

NOTE Depending on your system, you may need to use http://localhost:8020/accu-mcp-server instead of http://127.0.0.1:8020/accu-mcp-server.

EXPLORING AND TESTING THE WEATHER TOOL

The MCP Inspector provides tabs to browse the different features an MCP server may expose: Tools, Prompts, and Resources. For your Weather MCP server, the focus is on the Tools tab:

1. Click the Tools tab at the top of the screen.
2. Click the List Tools button.

You should see the `get_weather_conditions` tool appear in the list. If you click it, a dedicated panel for this tool will appear on the right, as shown in figure 13.4.

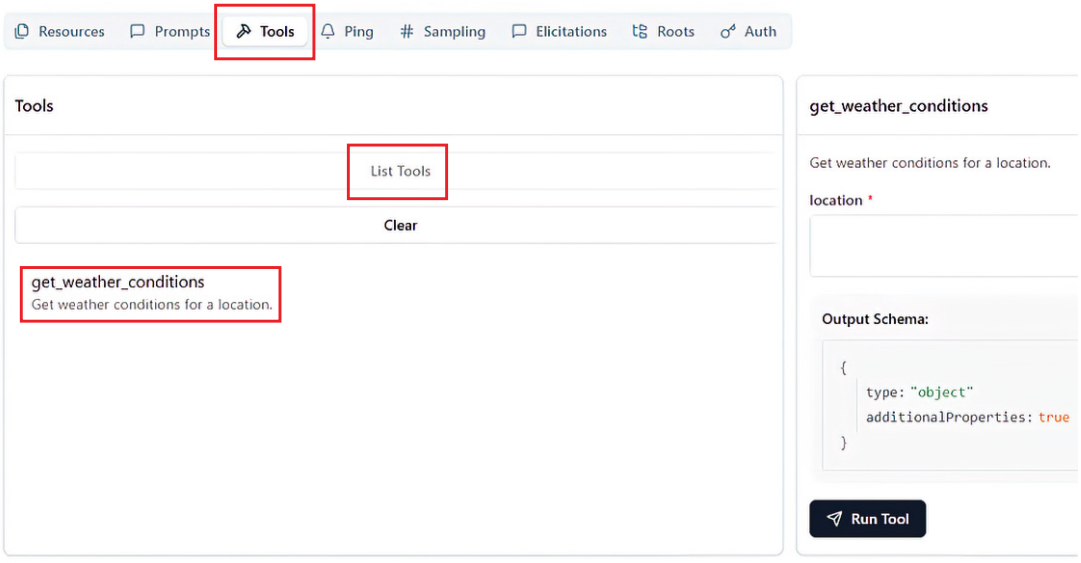


Figure 13.4 The MCP Inspector showing the get_weather_conditions tool and its corresponding panel on the right

To try out your new tool, follow these steps:

1. In the `get_weather_conditions` panel, enter a location such as `Penzance, UK` in the Location text box.
2. Click Run Tool.

The tool will run and display the current weather conditions for the location you specified, as shown in figure 13.5.

As you can see, the MCP Inspector makes it easy to verify that your MCP server and its exposed tool are functioning correctly, all without having to write any additional client code.

Next steps after using the MCP Inspector

Now that you've validated your MCP server using the MCP Inspector and confirmed that your tool is operational, you're ready to move forward and implement a test client application. This will allow you to connect to your MCP server programmatically and further integrate it into your agent solutions.

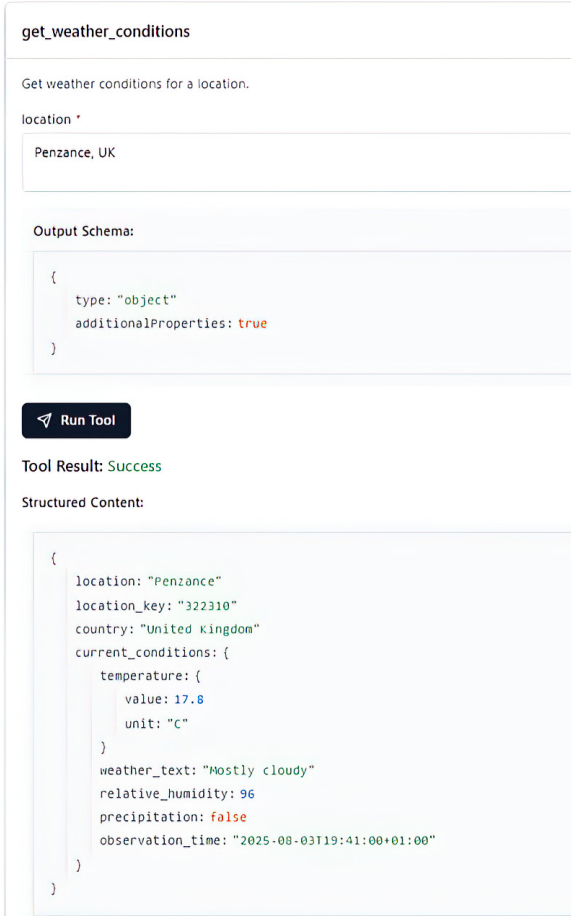


Figure 13.5 The MCP Inspector output showing the current weather conditions for Penzance, UK, and confirming that the tool works as expected

13.3.3 Consuming the MCP server from a test MCP host

Once you've verified your MCP server using the MCP Inspector, the next step is to interact with it programmatically. To do this, you'll implement a simple MCP host—essentially a lightweight client—that connects to your weather MCP server and exercises its functionality. Create a new Python script named `test_accuweather_mcp.py` inside the `mcp` folder, and copy in the code from listing 13.2. This approach allows you to confirm, via code, that your MCP server is discoverable and responds correctly to tool calls.

Listing 13.2 Test the MCP host for the Weather MCP server

```
from fastmcp import Client
from fastmcp.client.transports import StreamableHttpTransport
import asyncio
```

```
transport = StreamableHttpTransport(
    url="http://localhost:8020/accu-mcp-server")
client = Client(transport)
async def main():
    async with client:
        print(f"Client connected: {client.is_connected()}")
        tools = await client.list_tools()
        print(f"Available tools: {tools}")
        if any(tool.name == "get_weather_conditions"
            for tool in tools):
            result = await client.call_tool("get_weather_conditions",
                {"location": "Penzance, UK"})
            print(f"Call result: {result}")

    print(f"Client connected: {client.is_connected()}")

if __name__ == "__main__":
    asyncio.run(main())
```

Sets up the transport as a Streamable HTTP server against the MCP server running on port 8020

Creates the MCP client

Lists the tools exposed by the MCP server

Checks if the tool exists

Calls the tool

Prints the result of the call

Runs the main function

If you analyze the listing, you'll see that the script instantiates an MCP client and binds it to the MCP server endpoint you implemented earlier. It lists the available tools and then executes a call against the weather tool exposed by the server.

To run the test host, open another terminal, activate your virtual environment, step into the `mcp` folder, and run the script:

```
C:\Github\building-llm-applications\ch11>env_ch11\Scripts\activate
(env_ch11) C:\Github\building-llm-applications\ch11>cd mcp
(env_ch11) C:\Github\building-llm-applications\ch11\mcp>
➥python test_accuweather_mcp.py
```

Alternatively, you can simply run `test_accuweather_mcp.py` in debug mode. You'll get the following output, showing the returned weather data (in Celsius):

```
Client connected: True
Available tools: [Tool(name='get_weather_conditions', title=None,
description='Get weather conditions for a location.',
inputSchema={'properties': {'location': {'title': 'Location', 'type':
'string'}}, 'required': ['location'], 'type': 'object'},
outputSchema={'additionalProperties': True, 'type': 'object'},
annotations=None, meta=None)]
Call result: CallToolResult(content=[TextContent(type='text',
text='{"location":"Penzance","location_key":"322310","country":"United
Kingdom","current_conditions":{"temperature":{"value":23.0,"unit":"C"},"weath
er_text":"Sunny","relative_humidity":71,"precipitation":false,"observation_ti
me":"2025-07-13T10:56:00+01:00"}}', annotations=None, meta=None)],
structured_content={'location': 'Penzance', 'location_key': '322310',
'country': 'United Kingdom', 'current_conditions': {'temperature': {'value':
23.0, 'unit': 'C'}, 'weather_text': 'Sunny', 'relative_humidity': 71,
'precipitation': False, 'observation_time': '2025-07-13T10:56:00+01:00'}},
data={'location': 'Penzance', 'location_key': '322310', 'country': 'United
Kingdom', 'current_conditions': {'temperature': {'value': 23.0, 'unit': 'C'},
```

```
'weather_text': 'Sunny', 'relative_humidity': 71, 'precipitation': False,
'observation_time': '2025-07-13T10:56:00+01:00'}}, is_error=False)
Client connected: False
```

Notice how the available tool is exposed and how the result of the call to `get_weather_conditions` is wrapped in a `CallToolResult`, following the standard protocol seen earlier. This confirms that results from MCP servers comply with the expected tool-calling protocol, as covered in previous chapters. You may also want to look at the MCP server output in the terminal:

```
←[32mINFO←[0m:     Uvicorn running on ←[1mhttp://127.0.0.1:8020←[0m (Press
                   CTRL+C to quit)
←[32mINFO←[0m:     127.0.0.1:60339 - "←[1mPOST /accu-mcp-server HTTP/1.1←[0m"
←[33m307 Temporary Redirect←[0m
←[32mINFO←[0m:     127.0.0.1:60339 - "←[1mPOST /accu-mcp-server/ HTTP/
                   1.1←[0m" ←[32m200 OK←[0m
←[32mINFO←[0m:     127.0.0.1:60342 - "←[1mPOST /accu-mcp-server HTTP/1.1←[0m"
←[33m307 Temporary Redirect←[0m
```

13.4 Integrating the Weather MCP tool into an agent

With your weather MCP server up and running, it's time to put it to practical use within a real agent application. In this section, you'll learn how to upgrade the travel assistant agent from previous chapters by replacing its mock weather tool with a fully functional, MCP-powered version. We'll focus on integrating the live weather tool from the remote MCP server into the travel information agent, allowing it to consume real AccuWeather data alongside its existing local capabilities, as shown in figure 13.6.

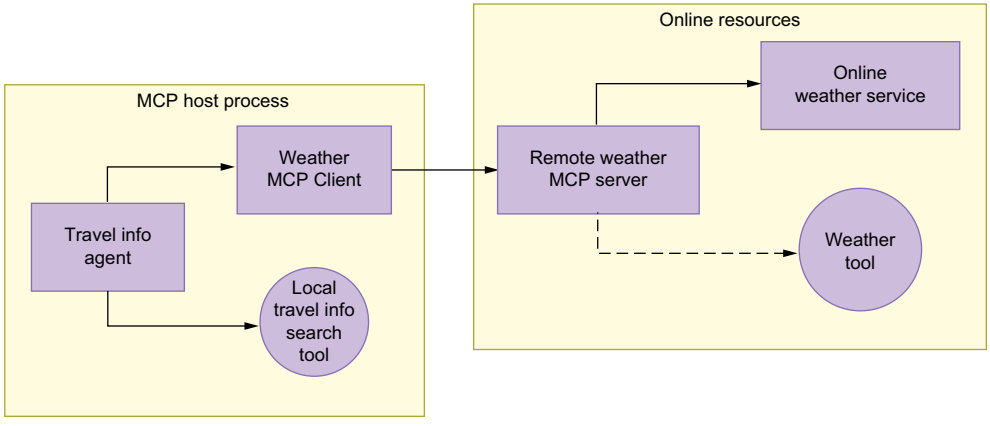


Figure 13.6 The travel information agent enhanced with a real weather tool from a remote MCP server, replacing the previous local mock weather tool

13.4.1 Preparing the travel agent for live weather data

We'll modify the travel information agent to use the new AccuWeather MCP server instead of the local mock weather service. To do so, we follow these steps:

1. Remove the implementation of the `weather_forecast` tool and the `WeatherForecastService` class (along with its return type).
2. Replace them with a client that connects to the AccuWeather MCP server and retrieves the remote tool dynamically.

To begin, let's take a copy of the initial travel agent code from `main_03_01.py`, save it as `main_07_01.py`, and run it in debug mode to recall the expected behavior from the mock tool. For example, asking for the weather in Penzance:

```
UK Travel Assistant (type 'exit' to quit)
You: What's the weather in Penzance?
...
Assistant: [{'type': 'text', 'text': 'Currently, the weather in Penzance is
windy with a temperature of 21°C. If you need a detailed forecast or more
travel information, feel free to ask!', 'annotations': []}]
```

As you can see, the mock data provided a made-up response. Now let's wire up the agent to use live weather information from the MCP server.

13.4.2 Integrating the AccuWeather MCP tool

Start by implementing an asynchronous function to instantiate a client for the AccuWeather MCP server. This will return the tools exposed by the server (in this case, just one):

```
async def get_accuweather_tools():
    mcp_client = MultiServerMCPClient({
        "accuweather": {
            "url": "http://127.0.0.1:8020/accu-mcp-server",
            "transport": "streamable_http"
        }
    })
    return await mcp_client.get_tools()
```

Defines the function to get the AccuWeather tools as an async function

Instantiates MultiServerMCPClient

Registers the AccuWeather MCP server

Returns the AccuWeather tools exposed by the MCP server

13.4.3 Updating the agent chat loop

Because the agent now calls out to remote tools, you'll need to adapt the main chat loop to support asynchronous tool invocation:

```
async def chat_loop(agent):
    print("UK Travel Assistant (type 'exit' to quit)")
    while True:
        user_input = input("You: ").strip()
        if user_input.lower() in {"exit", "quit"}:
            break
        state = {"messages": [HumanMessage(
```

Defines the chat loop as an async function

Starts the chat loop

Gets the user input

Checks to see if the user input is "exit" or "quit" to exit the loop

```
        content=user_input)]}
    result = await agent.ainvoke(state)
    response_msg = result["messages"][-1]
    print(
        f"Assistant: {response_msg.content}\n")
```

Creates the initial state with a HumanMessage containing the user input

Invokes the agent with the initial state, asynchronously

Gets the last message from the result, which contains the final answer

Prints the assistant's final answer from the content of the last message

13.4.4 Combining local and remote tools

In your async `main()` function, you'll now retrieve the AccuWeather tools and combine them with your local semantic search tool, as shown in the following listing.

Listing 13.3 Combining local and remote tools

```
class AgentState(TypedDict):
    messages: Annotated[Sequence[BaseMessage], operator.add]
    remaining_steps: RemainingSteps

async def main():
    accuweather_tools = \
        await get_accuweather_tools()
    tools = [search_travel_info,
        *accuweather_tools]
    llm_model = ChatOpenAI(
        model="gpt-5-mini",
        use_responses_api=True)

    travel_info_agent = create_react_agent(
        model=llm_model,
        tools=tools,
        state_schema=AgentState,
        name="travel_info_agent",
        prompt="""You are a helpful assistant that can
        search travel information and get the weather forecast.
        Only use the tools to find the information you need
        (including town names).""",
    )
    await chat_loop(travel_info_agent)

if __name__ == "__main__":
    asyncio.run(main())
```

Defines the AgentState class

Gets the AccuWeather MCP server tools

Combines the local search_travel_info tool with the AccuWeather MCP server tools

Instantiates the LLM model

Creates the travel_info_agent

Starts the chat loop

Runs the main function asynchronously

NOTE The main function and chat loop are both now asynchronous, allowing your agent to use local and MCP tools with minimal effort.

13.4.5 Testing and verification

With the updated code in place, you can now run the `main_07_01.py` script in debug mode. Place a breakpoint at the line where the language model (`llm_model`) is instantiated, and inspect the tools list. You should see both the local and remote tools in the output:

```
[StructuredTool(name='search_travel_info', description='Search travel
information about destinations in England.', args_schema=<class
'langchain_core.utils.pydantic.search_travel_info'>, func=<function
search_travel_info at 0x000002141393D620>),
Tool(name='get_weather_conditions', title=None, description='Get weather
conditions for a location.', inputSchema={'properties': {'location':
{'title': 'Location', 'type': 'string'}}, 'required': ['location'], 'type':
'object'}, outputSchema={'additionalProperties': True, 'type': 'object'},
annotations=None, meta=None)]
```

Continue running the code, and try asking the assistant for the weather in Penzance (after you are shown `You:`):

```
UK Travel Assistant (type 'exit' to quit)
You: What's the weather in Penzance?
Assistant: [{'type': 'text', 'text': 'The current weather in Penzance is
light rain with a temperature of 17°C. The humidity is quite high at 94%.',
'annotations': []}]
```

As you can see, the AccuWeather MCP server is now invoked by your agent. You can also check the terminal in which the MCP server is running and inspect the LangSmith trace to confirm the tool was called as expected.

13.4.6 Using the agent for complex queries

Finally, experiment with more advanced reasoning-based queries that combine travel information with live weather data. Be sure to adapt the questions to the current season. For example:

```
You: Suggest two beach Cornwall towns with nice weather
Assistant: [{'type': 'text', 'text': 'Two beach towns in Cornwall are Newquay
and St Ives. However, currently, Newquay is experiencing light rain with a
temperature of 17°C, and St Ives has hazy sunshine with a temperature of 6°C.
If you prefer nicer weather, St Ives would be the better choice at the
moment.', 'annotations': []}]
```

You can continue experimenting with queries like this:

```
You: Suggest two beach Cornwall towns with nice weather; keep trying until
you find two with nice weather
```

By integrating the weather tool exposed by your MCP server, you've made your agent capable of delivering genuinely real-time, actionable information. This not only demonstrates the power of MCP but also how external tools can be combined with local agent skills for richer, more useful applications.

Summary

- Model Context Protocol (MCP) provides standardized tool interfaces that agents can discover and consume without custom integration code. Servers expose capabilities such as database access, file operations, or API calls through a uniform JavaScript Object Notation–Remote Procedure Call (JSON-RPC) protocol.

- Vendors themselves often provide these servers. Check if your target service already has an official MCP server before building custom integrations (e.g., GitHub, Slack, Google Drive).
- MCP uses a client/server architecture. Agents (hosts) send JSON-RPC 2.0 requests to MCP servers that handle the actual operations and return results in a standardized format.
- A single agent can connect to multiple MCP servers simultaneously, treating them as a unified tool ecosystem. The agent discovers available tools from each server and decides which to invoke.
- FastMCP 2 (Python) and the TypeScript MCP SDK enable building MCP servers. They support decorator-based tool definition, automatic schema generation from type hints, and built-in error handling.
- MCP Inspector provides a web UI for testing MCP servers interactively before production integration. This validates server implementations before integrating them into agent workflows.
- OpenAI's ChatGPT and API natively support MCP tool calling. Agents can invoke MCP-exposed tools using standard function calling without additional bridging layers. This eliminates custom bridging code between LLM providers and MCP servers. Configure your agent to connect to MCP servers, and the LLM will automatically discover and call available tools.
- LangChain and LangGraph convert MCP tools into native framework tools automatically. Agents can mix MCP-provided tools with framework-native tools seamlessly.
- An agent might call a local calculation function, then an MCP-served database query, and then a local formatting function—all in one workflow without distinguishing tool sources.
- MCP adoption is growing as the standard for exposing services to AI agents. Major platforms such as Anthropic, GitHub, and Slack are releasing MCP servers for their APIs.
- Learning MCP architecture and server development enables building agents that connect to this expanding ecosystem without reinventing integration patterns for each service.
- MCP servers run as separate processes communicating via standard input/output (STDIO) or HTTP. For local development, STDIO is simpler; for production across networks, use HTTP with proper authentication.

14 Productionizing AI agents: Memory, guardrails, and beyond

This chapter covers:

- Adding short-term memory with LangGraph checkpoints
- Implementing guardrails at multiple workflow stages
- Additional considerations for production deployment

Building AI agents that behave reliably in real-world environments is about more than just connecting a language model to some tools. Production systems need to maintain context across turns, respect application boundaries, handle edge cases gracefully, and keep operating even when something unexpected happens. Without these safeguards, even the most capable model will eventually produce errors, off-topic answers, or inconsistent behavior that undermines user trust.

In this chapter, we'll focus on two of the most important capabilities for making AI agents production-ready: memory and guardrails. Memory allows an agent to "remember" past interactions, enabling it to hold natural conversations, answer follow-up questions, and recover from interruptions. Guardrails keep the agent within its intended scope and policy framework, filtering out irrelevant or unsafe

requests before they reach the model—and, if needed, catching inappropriate responses after the model has generated them.

We'll explore how these features work in LangGraph, using our travel information assistant as the running example. You'll see how to persist conversation state using checkpoints, enforce scope restrictions at both the router and agent level, and extend the design to include post-model checks and human-in-the-loop review. By the end of the chapter, you'll have the tools to build assistants that are not only smart but also safe, focused, and resilient.

14.1 Memory

When designing AI agent-based systems—especially those exposed via chatbots—one of the most important steps toward production readiness is adding memory. Memory allows the system to maintain context between user interactions, enabling stateful workflows and natural conversations.

Without memory, each interaction between the user and the large language model (LLM) starts fresh, with no knowledge of what was said before. For simple Q&A, this might be acceptable, but for anything that requires follow-up, refinement, or reference to past turns, it quickly becomes frustrating.

LangGraph provides a powerful and flexible mechanism for implementing memory: checkpoints. In this section, we'll explore short-term memory using checkpoints and see how to integrate them into our existing `travel_assistant` agent.

14.1.1 Types of memory

In AI agents, memory can exist at different scopes. The three different scopes are as follows:

- *Short-term memory*—Context retained during a single session between a user and the LLM and typically stored in-memory or in a session-scoped store. This is ideal for ongoing conversations until the user achieves their goal.
- *Long-term user memory*—Persistent across multiple sessions for the same user, enabling the system to remember preferences or past activities.
- *Long-term application-level memory*—Persistent across all users and sessions, storing general knowledge useful for everyone (e.g., current exchange rates).

Long-term user and application memory tend to be highly system-specific, so in this section, we'll focus exclusively on short-term conversational memory.

14.1.2 Why short-term memory is needed

In a conversational application that uses the tool-calling protocol (as described in earlier chapters), a typical interaction works like this:

1. The user sends a message.
2. The LLM may issue tool calls.
3. The application executes these tools and returns results to the LLM.
4. The LLM synthesizes a final answer.

If the user follows up with a clarifying or related question, a stateless system would lose all prior context, forcing the conversation to restart. This is both inefficient and unnatural.

The solution is to store the entire conversation history after each interaction and feed it back to the LLM on subsequent turns. This makes the system stateful and allows the model to resolve references like "same town" or "that hotel" based on prior turns.

14.1.3 Checkpoints in LangGraph

Although you could implement short-term memory simply by storing the list of exchanged messages after each LLM response, LangGraph provides a more powerful and generalized mechanism: the `checkpointer`. A *checkpoint* is a snapshot of the graph's execution state taken at a specific moment in the flow. In practice, LangGraph takes these snapshots at each node in the graph—starting from the entry point (also called the *START* node) and continuing until the exit point (*END* node).

This approach is more flexible than storing messages alone because it preserves all aspects of the graph state, not just the conversation text. That flexibility enables a range of use cases beyond chat memory:

- *State rehydration after failure*—If an execution fails partway through, you can resume from the last successful checkpoint without rerunning the entire process. This is especially valuable when some steps are expensive or slow.
- *Human-in-the-loop workflows*—You can pause at a checkpoint to collect manual input or approval and then resume exactly where you left off using the stored state.
- *Multi-turn conversational context*—For chatbots, checkpoints ensure the conversation history is preserved across turns without having to reconstruct it manually.

In this chapter, we'll focus solely on the last use case—maintaining conversational state for the entire duration of a user–LLM session—while keeping in mind that the same mechanism is equally capable of powering the others.

What is a checkpoint?

A checkpoint represents the saved state of the graph at a given super-step where the following is true:

- *A super-step* corresponds to the execution of a single graph node.
- *Parallel nodes* processed within the same step are grouped together.

By saving a checkpoint at each super-step, we can do the following:

- Reuse the current execution state in a follow-up question
- Replay the persisted execution up to a specific point—useful when recovering from errors or resuming after manual intervention

How checkpoints work in LangGraph

LangGraph's checkpointing system is built around two main concepts:

- *Checkpointer*—The component responsible for capturing and storing state snapshots. After each super-step (i.e., completion of a node), the `checkpointer`

records the current graph state, as illustrated in figure 14.1. This snapshot can include conversation history, tool outputs, intermediate variables, and execution metadata—ensuring the workflow can be resumed or inspected at any point.

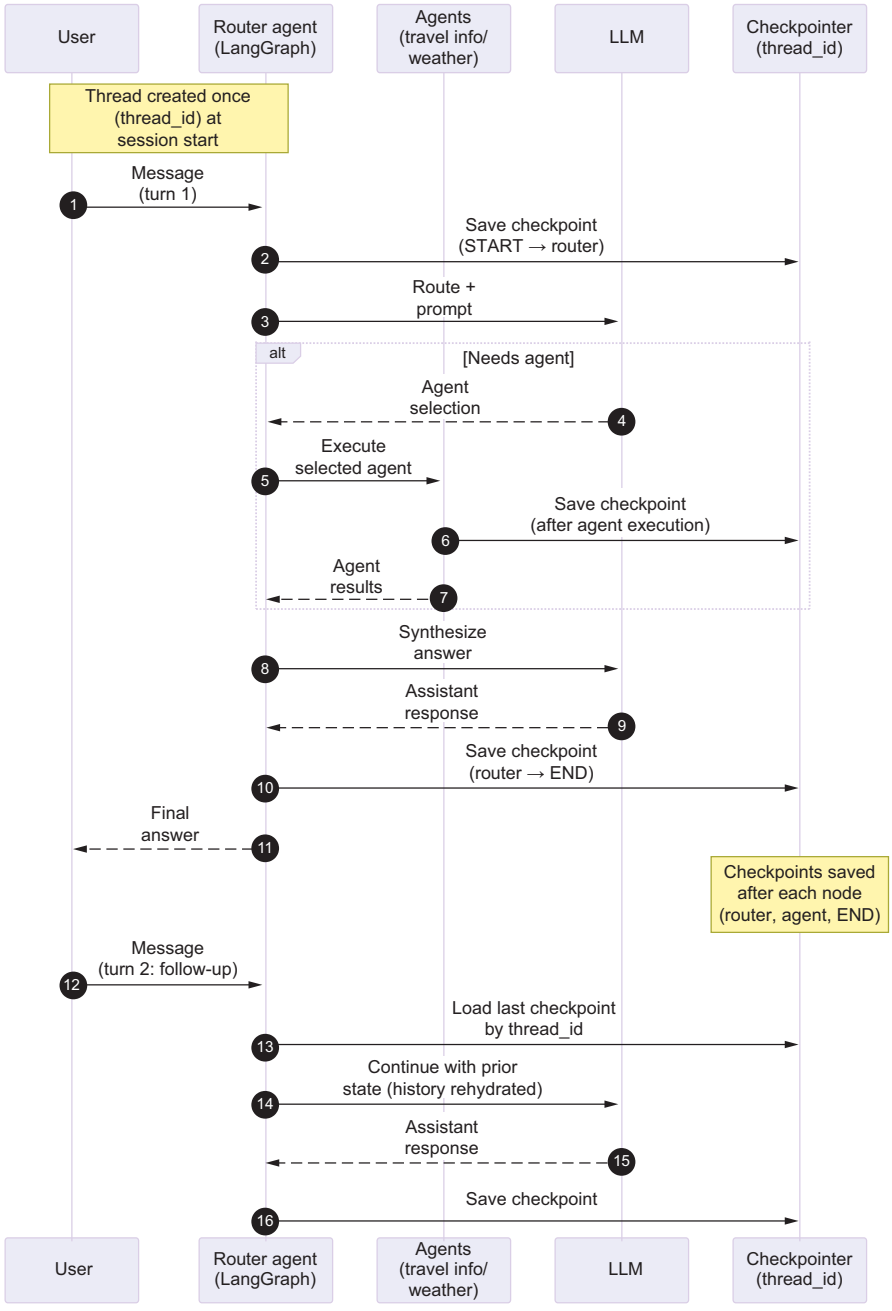


Figure 14.1 Sequence diagram of the travel assistant with checkpoints saved after each node execution, allowing state restoration across turns using a shared `thread_id`

- *Configuration*—Defines the session that checkpoints belong to. In LangGraph, a session is called a *thread*:
  - Each thread is identified by a thread ID (a unique value, typically a Universally Unique Identifier [UUID]).
  - A single user may have multiple threads (sessions) active or saved at different times.
  - The configuration links the checkpoint to a specific thread so the system knows which session state to load later.

When you pass a thread ID to the graph on subsequent invocations, LangGraph retrieves the stored state from the last checkpoint in that thread and resumes execution from there. Alternatively, in a conversational context, LangGraph uses it to provide the LLM with the correct history.

14.1.4 Adding short-term memory to our travel assistant

To demonstrate how LangGraph's persistence and checkpointing work, we'll enhance our router-based `travel_assistant` from section 12.2 with short-term memory. For clarity, we'll walk through the changes step-by-step. Start by making a copy of your existing `main_05_01.py` and naming it `main_08_01.py`. We'll add persistence features to this copy so you can compare it with the original.

STEP 1: THE ORIGINAL STATELESS CHAT LOOP

First, let's revisit the existing chat loop in listing 14.1. In this version, each time the user interacts, we only pass the new message as the state to `travel_assistant`. There's no concept of session or continuity—each turn is treated as completely separate.

Listing 14.1 Original chat loop from router-based agent solution

```
def chat_loop():                                          <- Defines the chat loop
    print("UK Travel Assistant (type 'exit' to quit)")
    while True:
        user_input = input("You: ").strip()               <- Gets the user input
        if user_input.lower() in {"exit", "quit"}:        <- Checks if the user input is "exit" or "quit" to exit the loop
            break
        state = {"messages":
           [HumanMessage(content=user_input)]}            <- Creates the initial state with a HumanMessage containing the user input
        result = travel_assistant.invoke(state)           <- Invokes the graph with the initial state
        response_msg = result["messages"][-1]             <- Gets the last message from the result, which contains the final answer
        print(f"Assistant: {
            response_msg.content}\n")                     <- Prints the assistant's final answer from the content of the last message
```

In this approach, `travel_assistant.invoke()` receives only the state—a fresh message each time:

```
state = {"messages": [HumanMessage(content=user_input)]}
result = travel_assistant.invoke(state)
```

STEP 2: INTRODUCING A THREAD ID

To persist state across turns, we need a way to uniquely identify the conversation. LangGraph does this using a thread ID, passed in a `RunnableConfig`. We can generate one at the start of the chat loop:

```
import uuid
thread_id = uuid.uuid1()
config = {"configurable": {"thread_id": thread_id}}
```

The following listing shows the revised chat loop implementation, where the thread ID is created once, printed, and passed with every `invoke()` call.

Listing 14.2 Revised chat loop implementation with `thread_id`

```
def chat_loop():
    thread_id=uuid.uuid1()
    print(f'Thread ID: {thread_id}')
    config={"configurable":
        {"thread_id": thread_id}}

    print("UK Travel Assistant (type 'exit' to quit)")
    while True:
        user_input = input("You: ").strip()
        if user_input.lower() in {"exit", "quit"}:
            break
        state = {"messages":
            [HumanMessage(content=user_input)]}
        result = travel_assistant.invoke(state,
            config=config)
        response_msg = result["messages"][-1]
        print(
           f"Assistant: {response_msg.content}\n")
```

Defines the chat loop

Creates a unique session ID

Checks if the user input is "exit" or "quit" to exit the loop

Creates the initial state with a HumanMessage containing the user input

Sets the state with the HumanMessage

Invokes the graph with the state and the config (which includes the session ID)

Gets the last message from the result, which contains the final answer

Prints the assistant's final answer from the content of the last message

Now, instead of just passing `state`, we pass two arguments:

```
result = travel_assistant.invoke(state, config)
```

The config ensures that all checkpoints and state belong to this specific session (`thread_id`).

STEP 3: ADDING A CHECKPOINTER

With the thread ID in place, we can add an in-memory `checkpointer` to store state snapshots:

```
from langgraph.checkpoint.memory import InMemorySaver
checkpointer = InMemorySaver()
```

When compiling the graph, pass the `checkpointer`, as shown in listing 14.3.

> **NOTE** `InMemorySaver` is ideal for development, testing, and quick proof-of-concepts. In a production environment, you should use a persistent storage

backend to ensure state survives restarts and can be shared across multiple application instances. LangGraph provides built-in options such as a `SqliteSaver` (from the `langgraph-checkpoint-sqlite` package, backed by a SQLite database) and a `PostgresSaver` (from the `langgraph-checkpoint-postgres` package, backed by PostgreSQL). It also provides the async versions, `SqliteSaverAsync` and `PostgresSaverAsync` (from the same packages). For production deployments, the PostgreSQL-based `checkpointer` is generally recommended for its scalability, reliability, and concurrency support.

Listing 14.3 Graph enriched with `checkpointer`

```
graph = StateGraph(AgentState)
graph.add_node("router_agent", router_agent_node)
graph.add_node("travel_info_agent",
    travel_info_agent)
graph.add_node("accommodation_booking_agent",
    accommodation_booking_agent)

graph.add_edge("travel_info_agent", END)
graph.add_edge("accommodation_booking_agent", END)

graph.set_entry_point("router_agent")

checkpointer = InMemorySaver()
travel_assistant = graph.compile(
    checkpointer=checkpointer)
```

Defines the graph
Adds the router agent node
Adds the travel info agent node
Adds the accommodation booking agent node
Adds the edge from the travel info agent to the end
Adds the edge from the accommodation booking agent to the end
Sets the entry point to the router agent
Instantiates the in-memory checkpointer
Compiles the graph with the in-memory checkpointer

This enables the graph to save a snapshot after every node execution so it can resume with full context on the next turn.

Step 4: Configuring the LLM for conversation continuity

Finally, we need to configure the LLM to reference previous turns without resending the entire conversation history every time. OpenAI's Responses API allows this using the `use_previous_response_id` flag:

```
llm_model = ChatOpenAI(
    model="gpt-5",
    use_responses_api=True,
    use_previous_response_id=True)
```

Instantiates the LLM model with the GPT-5 model
Uses the Responses API
Uses the previous response ID to continue the conversation

With `use_previous_response_id=True`, the LangChain ChatOpenAI wrapper sends only the ID of the previous response, and OpenAI rehydrates the full history internally.

NOTE If you enable `use_responses_api=True` but not `use_previous_response_id=True`, LangChain will try to resend the full history on every turn, instead of just its ID. The Responses API treats this as a duplicate submission and returns an error. When using LangGraph memory with the Responses API, enabling `use_previous_response_id` is mandatory.

14.1.5 Executing the checkpointer-enabled assistant

With the `checkpointer` integrated into our `travel_assistant`, we can now see short-term conversational memory in action. Run the updated router-based assistant, and enter a typical question:

```
Thread ID: e683b337-752b-11f0-84a9-34f39a8d3195
You: What's the weather like in Penzance?
Assistant: [{'type': 'text', 'text': 'The weather in Penzance is currently
     sunny with a temperature of 19°C.', 'annotations': []}]
```

Now enter a follow-up question that refers back to the previous turn (remember, this script is based on a copy of `main_05_01.py`, which uses a mock weather service that returns random conditions):

```
You: What's the weather in the same town now?
Assistant: [{'type': 'text', 'text': 'Current weather in Penzance: foggy,
around 28°C.', 'annotations': []}]
```

Notice how the assistant correctly resolves "same town" to Penzance. It can do this because the LangGraph `checkpointer` provides the LLM with the full conversation history, not just the latest user input. (As mentioned earlier, because we're using a mock weather service, the second call will return different weather conditions.)

Congratulations—you've just implemented a stateful conversational chatbot! Although this is satisfying, it's worth taking a deeper look at what's happening under the hood so you can understand exactly how the `checkpointer` maintains short-term memory.

14.1.6 Rewinding the state to a past checkpoint

To better understand how LangGraph manages conversational memory, we'll simulate what happens internally when restoring from a checkpoint as follows:

1. Ask the chatbot a question.
2. Retrieve the last checkpoint from the `checkpointer`.
3. Rehydrate the graph state to that checkpoint.
4. Ask a follow-up question that depends on that restored context.

This is effectively what LangGraph does automatically when you pass the same `thread_id` on subsequent turns.

STEP 1: UPDATING THE CHAT LOOP FOR STATE INSPECTION

Create a copy of `main_08_01.py`, and name it `main_08_02.py`. Replace the `chat_loop()` with the implementation shown in the following listing.

Listing 14.4 Step-by-step state inspection

```
def chat_loop():
    thread_id=uuid.uuid1()          <-- Creates a unique thread ID
    print(f'Thread ID: {thread_id}')
```

```
config={"configurable":
   {"thread_id": thread_id}}

user_input = input("You: ").strip()

question = {"messages":
    [HumanMessage(content=user_input)]}
result = travel_assistant.invoke(
    question, config=config)
response_msg = result["messages"][-1]
print(
   f"Assistant: {response_msg.content}\n")

state_history = travel_assistant.get_state_history(
    config)
state_history_list = list(state_history)
print(f'State history: {state_history_list}')

last_snapshot = list(state_history_list)[0]
print(f'Last snapshot: {last_snapshot.config}')

thread_id = last_snapshot.config[
    "configurable"]["thread_id"]
last_checkpoint_id = last_snapshot.config[
    "configurable"]["checkpoint_id"]

new_config = {"configurable":
          {"thread_id": thread_id,
           "checkpoint_id": last_checkpoint_id}}

retrieved_snapshot = travel_assistant.get_state(
    new_config)
print(
   f'Retrieved snapshot: {retrieved_snapshot}')

travel_assistant.invoke(None,
config=new_config)

new_question = {"messages": [HumanMessage(
    content="What is the weather in the same town?")]}
result = travel_assistant.invoke(new_question,
    config=new_config)
response_msg = result["messages"][-1]

print(
   f"Assistant: {response_msg.content}\n")
```

Creates a config with the thread ID

Creates the initial state with a HumanMessage containing the user input

Sets the state with the HumanMessage

Invokes the graph with the state and the config

Gets the last message from the result, which contains the final answer

Gets the state history from the graph

Prints the assistant's final answer from the content of the last message

Prints the state history

Gets the last snapshot from the state history

Gets the thread ID from the last snapshot

Gets the checkpoint ID from the last snapshot

Creates a new config with the thread ID and the checkpoint ID

Gets the snapshot from the graph with the new config referencing the last checkpoint

Prints the retrieved snapshot

Rewinds the graph to the last checkpoint

Invokes the graph with the new question referencing content from the last checkpoint

Gets the last message from the result, which contains the final answer

Prints the assistant's final answer from the content of the last message

STEP 2: RUNNING THE EXAMPLE AND VIEWING STATE HISTORY

Run `main_08_02.py` in debug mode, and enter the usual question (remember, the weather is still randomized):

```
You: What's the weather like in Penzance?
Assistant: [{'type': 'text', 'text': 'It's currently foggy in Penzance with a
temperature around 27°C.', 'annotations': []}]
```

The script in the previous listing 14.4 continues and retrieves the state history:

```
state_history = travel_assistant.get_state_history(config)
state_history_list = list(state_history)
print(f'State history: {state_history_list}')
```

You'll see multiple `StateSnapshot` entries, each representing a checkpoint, starting from the most recent and moving backward to the initial `START` node. Each snapshot contains the following:

- The messages exchanged so far
- Tool call information (if any)
- Metadata about the graph execution step

STEP 3: REHYDRATING FROM A SPECIFIC CHECKPOINT

Let's take a look at what the rest of the script does, starting from the most recent snapshot:

```
last_snapshot = list(state_history_list)[0]
print(f'Last snapshot: {last_snapshot.config}')
```

Extract the `thread_id` and `checkpoint_id`:

```
thread_id = last_snapshot.config["configurable"]["thread_id"]
last_checkpoint_id = last_snapshot.config["configurable"]["checkpoint_id"]
```

Build a new config pointing to that checkpoint:

```
new_config = {"configurable":
              {"thread_id": thread_id,
               "checkpoint_id": last_checkpoint_id}}
```

Retrieve the state at this checkpoint to confirm it matches expectations:

```
retrieved_snapshot = travel_assistant.get_state(new_config)
print(f'Retrieved snapshot: {retrieved_snapshot}')
```

You should see the full conversation history up to that checkpoint, including user messages, tool outputs, and assistant responses.

STEP 4: RESUMING FROM THE RESTORED STATE

To rewind the graph to that point, use the following:

```
travel_assistant.invoke(None, config=new_config)
```

Now ask a follow-up that depends on that past context:

```
new_question = {"messages": [HumanMessage(
   content="What is the weather in the same town?")]}
result = travel_assistant.invoke(new_question, config=new_config)
response_msg = result["messages"][-1]
```

You might see output like this:

```
Assistant: [{'type': 'text', 'text': 'In Penzance it's currently windy with a
temperature around 20°C.', 'annotations': []}]
```

The assistant correctly infers that "same town" is Penzance, confirming that the rehydrated state was passed back to the LLM.

By stepping through this manual rewind, you now have a low-level understanding of how LangGraph's `checkpointer` powers short-term conversational memory—storing the entire execution context at each node and restoring it later to continue exactly where the conversation left off.

14.2 Guardrails

Guardrails are application-level mechanisms that keep an AI agent operating within a defined scope, policy framework, and intended purpose. They serve as the "rules of the road" for agent behavior, inspecting and validating both inputs and outputs at critical points to ensure the system stays safe, relevant, compliant, and efficient. Without guardrails, an agent might drift into unrelated subject matter, produce unsafe or noncompliant responses, or waste processing resources on unnecessary actions.

A strong guardrail design can stop a travel assistant from giving stock market tips, prevent a support bot from revealing confidential information, or block poorly formed requests before they reach an expensive language model. In practice, guardrails often fall into three broad categories:

- *Rule-based*—These guardrails are explicit conditions or regular expressions that catch prohibited patterns or topics.
- *Retrieval-based*—These guardrails check against approved data sources to confirm that a request is relevant and in scope.
- *Model-based*—These guardrails are compact classification or moderation models that assess intent, safety, or adherence to policy

These controls can be introduced at multiple points in the agent workflow:

- *Pre-model checks*—Reject invalid or irrelevant queries before the LLM runs.
- *Post-model checks*—Verify generated responses against policy or safety guidelines.
- *Routing-stage checks*—Decide whether a query should trigger certain tools or branches.
- *Tool-level checks*—Block unsafe or unauthorized tool actions.

Functionally, guardrails act like validation layers—if a check fails, the agent's normal flow is adjusted, whether by refusing the request, asking for clarification, or redirecting to a safer response path.

In this section, we'll integrate custom guardrails into the router node of the travel information assistant we built earlier (now enhanced with memory) so it can immediately reject irrelevant or out-of-scope requests—such as nontravel queries. We'll also

explore LangGraph's pre-model hook, which lets us enforce guardrails before any LLM call, ensuring that both the travel and weather agents remain within coverage boundaries—for example, only handling destinations in Cornwall.

14.2.1 Implementing guardrails to reject nontravel-related questions

The first and most obvious guardrail for our UK travel information assistant is a domain relevance check—a prefilter that screens the user's question before any agent reasoning occurs. If the question falls outside the assistant's remit, we intercept it early and politely refuse to answer. This prevents the system from attempting to handle queries about unrelated topics such as sports results, financial markets, or celebrity gossip. Introducing this guardrail delivers two key benefits:

- *Improved accuracy*—Our agents are trained and configured only for travel and weather information. If we attempt to answer unrelated questions, the results will almost certainly contain inaccuracies or hallucinations. By rejecting irrelevant queries outright, we keep the conversation aligned with the agent's actual capabilities.
- *Cost control*—Without this filter, some users might deliberately use our assistant as a free gateway to an expensive LLM, bypassing subscription costs for unrelated questions. By blocking nontravel topics early, we prevent this form of resource abuse and avoid unnecessary processing costs.

DEFINING THE GUARDRAIL POLICY

Our first task is to clearly define what qualifies as in-scope for this assistant. This ensures the guardrail has unambiguous decision criteria.

To begin, make a copy of the `main_08_01.py` script from the previous section, and save it as `main_09_01.py`. Then, implement the guardrail policy shown in the following listing.

Listing 14.5 Restricting questions to travel-related topics

```
class GuardrailDecision(BaseModel):
    is_travel: bool = Field(
        ...,
        description=(
            """True if the user question is about travel information:
            destinations, attractions,
            lodging (hotels/BnBs), prices, availability,
            or weather in Cornwall/England."""
        ),
    )
    reason: str = Field(...,
        description="Brief justification for the decision.")

GUARDRAIL_SYSTEM_PROMPT = (
    """You are a strict classifier. Given the user's
```

Defines the GuardrailDecision model

Defines the GUARDRAIL_SYSTEM_PROMPT, which constrains the model to only answer travel-related questions

```
    last message, respond with whether it is
    travel-related. Travel-related queries
    include destinations, attractions, lodging (hotels/BnBs),
    room availability, prices, or weather in Cornwall/England."""
)

REFUSAL_INSTRUCTION = (
    """You can only help with travel-related questions
    (destinations, attractions, lodging, prices,
    availability, or weather in Cornwall/England).
    The user's request is not travel-related.
    Politely refuse and briefly explain what
    topics you can help with."""
)

llm_guardrail = llm_model.with_structured_output(
    GuardrailDecision)
```

Defines the REFUSAL_INSTRUCTION, which is used to politely refuse to answer nontravel-related questions

Uses the same base model with structured output for fast, lightweight classification

This snippet defines a guardrail policy that uses a lightweight LLM classifier to determine whether a user's question is travel-related. The guardrail setup consists of several key components, each serving a distinct role in enforcing domain-specific constraints and guiding model behavior:

- `GuardrailDecision` is a Pydantic model that structures the classification output. The `is_travel` flag indicates whether the request falls within the travel domain, and `reason` provides a brief justification for the decision.
- `GUARDRAIL_SYSTEM_PROMPT` instructs the model to classify strictly, giving a precise definition of "travel-related" for our purposes.
- `REFUSAL_INSTRUCTION` contains a fixed, polite instruction to explain to the user why their question can't be answered.
- `llm_guardrail` wraps the base LLM with structured output formatting, enabling fast, consistent decision-making before the main routing logic runs.

Updating the router graph

At a high level, we want irrelevant queries to exit the workflow immediately—without touching any of the downstream agents. That means introducing a dedicated `guardrail_refusal` node in our LangGraph routing structure. This node simply redirects to the graph's `END` without performing any work. The updated workflow is shown in figure 14.2.

The router agent now checks whether a question is in scope. If it is, the query is routed to either the travel information agent or the weather agent as before. If it isn't, the router sends it to the new `guardrail_refusal` node, which is linked directly to the `END` node. The corresponding graph definition is shown in listing 14.6.

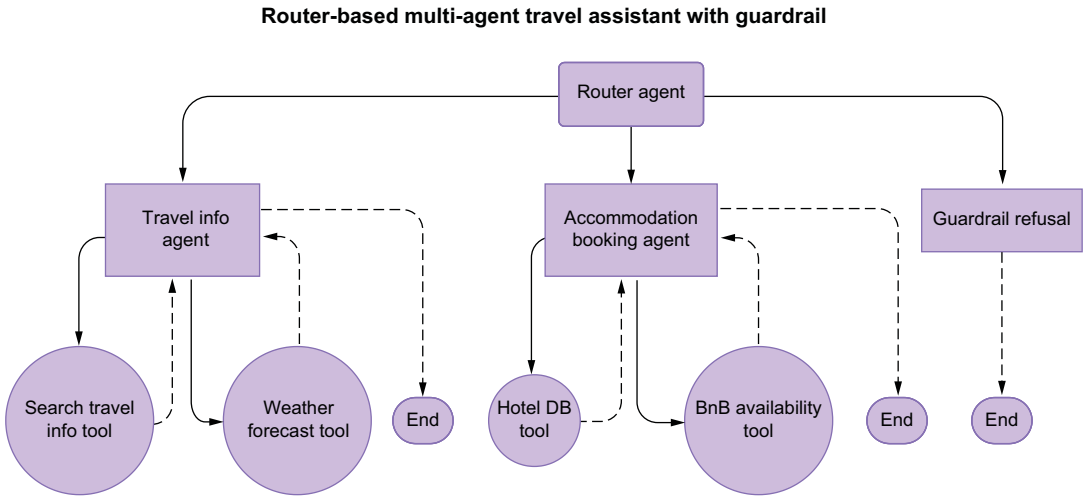


Figure 14.2 Updated travel assistant workflow. The guardrail introduces an early-exit path, allowing out-of-scope queries to terminate before reaching downstream agents.

Listing 14.6 Adding a `guardrail_refusal` node to the router graph

```
def guardrail_refusal_node(state: AgentState):
    return {}

graph = StateGraph(AgentState)
graph.add_node("router_agent", router_agent_node)
graph.add_node("travel_info_agent", travel_info_agent)
graph.add_node("accommodation_booking_agent", accommodation_booking_agent)
graph.add_node("guardrail_refusal",
    guardrail_refusal_node)

graph.add_edge("travel_info_agent", END)
graph.add_edge("accommodation_booking_agent", END)
graph.add_edge("guardrail_refusal", END)

graph.set_entry_point("router_agent")

checkpointer = InMemorySaver()
travel_assistant = graph.compile(checkpointer=checkpointer)
```

Defines the guardrail refusal node, which is a no-op node that is used to shortcut to END

Adds the guardrail refusal node

Adds the edge from the guardrail refusal node to the end

As you can see, the `guardrail_refusal` node is intentionally a no-op—its only role is to create a clean shortcut to `END`.

UPDATING THE ROUTER AGENT

With the guardrail policy defined and the graph updated, the final step is to implement the logic that enforces it. Previously, our router agent only decided whether to send a query to the travel or weather agent. Now it must first run the guardrail check, as illustrated in the diagram of figure 14.3.

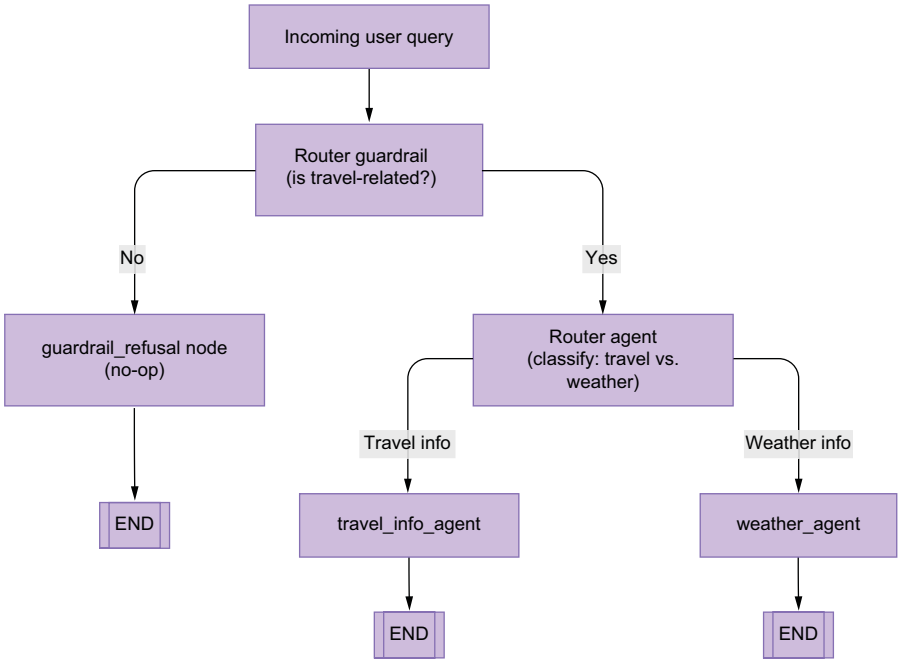


Figure 14.3 Flowchart of updated router logic with guardrail check: queries first pass a relevance filter, and irrelevant ones are sent with a refusal message directly to the guardrail_refusal node.

If the guardrail check fails—meaning the question is judged irrelevant to the travel assistant—the router generates a refusal message and routes execution directly to the `guardrail_refusal` node, as shown in figure 14.3. This logic is implemented in the following listing.

Listing 14.7 Router agent with guardrail enforcement

```
def router_agent_node(state: AgentState) -> Command[AgentType]:
    """Router node: decides which agent should handle the user query."""
    messages = state["messages"]
    last_msg = messages[-1] if messages else None
    if isinstance(last_msg, HumanMessage):
        user_input = last_msg.content

        # Guardrail classification at routing time
        classifier_messages = [
            SystemMessage(content=GUARDRAIL_SYSTEM_PROMPT),
            HumanMessage(content=user_input),
        ]
        decision = llm_guardrail.invoke(
            classifier_messages)
        if not decision.is_travel:
```

Defines the guardrail decision prompt

Invokes the guardrail model, which returns a GuardrailDecision object

Checks if the decision isn't travel-related

Defines the refusal text

```
            refusal_text = (
                """Sorry, I can only help with travel-related
                questions (destinations, attractions, lodging,
                prices, availability, or weather in Cornwall/England).
                Please rephrase your request to be travel-related."""
            )
            return Command(
                update={"messages": [AIMessage(content=refusal_text)]},
                goto="guardrail_refusal",
            )

        router_messages = [
            SystemMessage(content=ROUTER_SYSTEM_PROMPT),
            HumanMessage(content=user_input)
        ]
        router_response = llm_router.invoke(router_messages)
        agent_name = router_response.agent.value
        return Command(update=state, goto=agent_name)

    return Command(update=state, goto=AgentType.travel_info_agent)
```

Returns the command to set a refusal message in the state and goes to the guardrail refusal node

In this revised logic

- The router first invokes `llm_guardrail` with the latest user query.
- If the classification result indicates the question isn't travel-related, the router constructs a fixed refusal message, stores it in the state, and sends execution to the `guardrail_refusal` node—bypassing all normal routing.
- If the check passes, the router continues with its usual process of selecting the most appropriate agent.

TESTING THE GUARDRAIL

To see it in action, run `main_09_01.py` in debug mode, and place a breakpoint on the line where `llm_guardrail` is invoked. Then, enter the following query:

```
UK Travel Assistant (type 'exit' to quit)
You: Can you give me the latest results of Inter Milan?
```

When execution stops at the breakpoint, inspect `decision.is_travel`. You should see `False` because soccer scores are outside the allowed travel domain. Resuming execution (press F5) will produce the following:

```
Assistant: Sorry, I can only help with travel-related questions
(destinations, attractions, lodging, prices, availability, or weather in
Cornwall/England). Please rephrase your request to be travel-related.
```

Congratulations—you've successfully implemented your first guardrail! However, our work isn't done. Remember, one of our agents only has coverage for Cornwall, not the whole of the UK. This means we'll want to implement further scope restrictions at the agent level, which we'll tackle in the next section.

14.2.2 Implementing more restrictive guardrails at the agent level

In traditional software development, it's considered best practice for each class or component to validate its own data rather than relying solely on validations at higher levels such as the UI. The same principle applies to agent-based systems: each agent should enforce its own input guardrails, even if broader checks are already in place at the chatbot entry point.

These agent-level guardrails are often more restrictive than system-wide ones because they can account for the specific capabilities and data scope of the individual agent. In our case, the following is true:

- The travel information agent can only handle queries about Cornwall because its vector store contains data exclusively from that region.
- The accommodation booking agent will also be limited to Cornwall for now to keep the assistant's scope consistent.

We have two levels of guardrails:

- *Router-level*—This guardrail acts as an early fail-fast filter before any agent logic or tool invocation.
- *Agent-level*—These guardrails provide a "belt-and-suspenders" safeguard to catch out-of-scope requests if the agent is ever called directly or reused in a different context.

Defining the Cornwall-restricted guardrail policy

We start by explicitly defining the policy: only travel-related questions about Cornwall are permitted. This ensures that both travel and accommodation booking agents reject queries for other regions or countries.

Make a copy of `main_09_01.py`, and rename it `main_09_02.py`. Then, add the following system prompts to define the classification and refusal behavior.

Listing 14.8 System prompts for classification and refusal behavior

```
AGENT_GUARDRAIL_SYSTEM_PROMPT = (
    """You are a strict classifier. Given the user's last message,
    respond with whether it is travel-related. Travel-related
    queries include destinations, attractions, lodging
    (hotels/BnBs), room availability, prices, or weather in
    Cornwall/England. Only accept travel-related questions covering
    Cornwall (England) and reject any questions from other areas in
    England and from other countries"""
)

AGENT_REFUSAL_INSTRUCTION = (
    """You can only help with travel-related questions
    (destinations, attractions, lodging, prices,
    availability, or weather in Cornwall/England). The user's
    request is not travel-related. Or it might be a travel
    related question but not focusing on Cornwall (England).
```

```
    Politely refuse and briefly explain what
    topics you can help with."""
)
```

CREATING THE AGENT-LEVEL GUARDRAIL FUNCTION

The agent guardrail is implemented as a Python function that takes the current graph state and returns either an unchanged state (for valid input) or one modified to instruct the LLM to issue a refusal. The code is given in the following listing.

Listing 14.9 Agent-level guardrail function

```
def pre_model_guardrail(state: dict):
    messages = state.get("messages", [])
    last_msg = messages[-1] if messages else None
    if not isinstance(last_msg, HumanMessage):
        return {}

    user_input = last_msg.content
    classifier_messages = [
        SystemMessage(content=AGENT_GUARDRAIL_SYSTEM_PROMPT),
        HumanMessage(content=user_input),
    ]
    decision = llm_guardrail.invoke(classifier_messages)

    if decision.is_travel:
        # Allow normal flow; do not modify inputs
        return {}

    return {"llm_input_messages":
        [SystemMessage(content=AGENT_REFUSAL_INSTRUCTION),
        *messages]}
```

Verifies that the last message is user input

Builds the classification prompt

If valid, proceeds without modification

If invalid, prepends refusal instructions to the LLM input

The `pre_model_guardrail()` function works as a preprocessing filter before the LLM sees the user's query by doing the following:

1. It verifies that the latest message is indeed from the user.
2. It sends the query, along with a strict classification system prompt, to the guardrail LLM.
3. If the query is in scope (travel-related and Cornwall-specific), it passes through unchanged. Otherwise, the function prepends a refusal instruction so the agent politely declines the request.

INJECTING THE GUARDRAIL INTO THE AGENTS

LangGraph's ReAct agents support pre-model hooks (`pre_model_hook`) and post-model hooks (`post_model_hook`), allowing you to intercept and manipulate inputs or outputs. While these hooks can be used for tasks such as summarizing long inputs or sanitizing outputs, here we'll focus solely on input-side guardrails. To enable the Cornwall restriction, we simply pass `pre_model_guardrail` to both the travel information agent and the accommodation booking agent, as shown in the following listings.

Listing 14.10 Travel information agent with Cornwall guardrail

```
travel_info_agent = create_react_agent(
    model=llm_model,
    tools=TOOLS,
    state_schema=AgentState,
    prompt="""You are a helpful assistant that can search travel
    information and get the weather forecast. Only use the tools
    to find the information you need (including town names).""",
    pre_model_hook=pre_model_guardrail,
)
```

Guardrail to check if the user input is travel-related and focusing on Cornwall (England)

Listing 14.11 Accommodation booking agent with Cornwall guardrail

```
accommodation_booking_agent = create_react_agent(
    model=llm_model,
    tools=BOOKING_TOOLS,
    state_schema=AgentState,
    prompt="""You are a helpful assistant that can check hotel
    and BnB room availability and price for a destination in
    Cornwall. You can use the tools to get the information you
    need. If the users does not specify the accommodation type,
    you should check both hotels and BnBs.""",
    pre_model_hook=pre_model_guardrail,
)
```

Guardrail to check if the user input is travel-related and focusing on Cornwall (England)

TESTING THE CORNWALL GUARDRAIL

Run `main_09_02.py` in debug mode, placing a breakpoint on the `llm_guardrail` invocation inside `pre_model_guardrail()`. Then, try the following:

```
UK Travel Assistant (type 'exit' to quit)
You: Can you give me some travel tips for Liverpool (UK)?
```

When paused at the breakpoint, inspect `decision.is_travel`—it should be `False` because the query isn't Cornwall-specific. Execution will then prepend the refusal instruction, resulting in output like this:

```
Assistant: [{'type': 'text', 'text': 'Sorry—I can only help with travel
questions focused on Cornwall (England), such as destinations, attractions,
lodging, prices/availability, and local weather. If you'd like tips for places
like St Ives, Newquay, Falmouth, Penzance, Padstow, or Truro, tell me your
interests and dates/budget and I'll tailor suggestions.', 'annotations': []}]
```

With this, we now have two layers of defense:

- A router-level guardrail that quickly rejects any nontravel queries
- Agent-level guardrails that enforce Cornwall-specific scope for travel and accommodation requests

Our agentic workflow is now protected from both irrelevant and out-of-coverage queries, making the system safer, more reliable, and potentially more cost-efficient by preventing misuse through questions the chatbot isn't designed to handle.

14.3 Beyond this chapter

By this point, you've seen how to equip AI agents with memory and guardrails—two of the most critical building blocks for making them production-ready. But depending on your application's domain, scale, and compliance requirements, there are additional considerations you may need to address before deploying to real users.

Some of these areas have been mentioned in passing throughout the book but not explored in depth, either because they are highly domain-specific or because they deserve their own dedicated treatment. In the following sections, you'll find a few directions worth exploring further.

14.3.1 Long-term user and application memory

In this chapter, we focused on short- to medium-term memory—keeping track of relevant state within a single conversation or across a limited session history. However, production agents often benefit from persistent, long-term memory that stores user preferences, past interactions, and contextual information across weeks, months, or even years. This might involve the following:

- Dedicated vector stores for each user
- Periodic summarization and pruning to keep memory manageable
- Privacy and compliance controls for personally identifiable information (PII)
- Long-term memory can dramatically improve personalization, but it also brings engineering, scaling, and regulatory challenges you'll need to plan for.

Table 14.1 summarizes the various types of memory your AI agent-based system might need to accommodate user needs.

Table 14.1 Type of memory in AI agents

Memory type	Scope	Persistence	Example (travel assistant)	Challenges
Short-term	Single user session	Until session ends	Remembering "same town" in a weather follow-up within one conversation	Limited continuity; lost when session closes
Long-term (user)	Across multiple sessions for one user	Weeks, months, or years	Remembering a user's preferred Cornwall destinations or accommodation types	Privacy compliance and managing the persisted data
Long-term (application)	Across all users and sessions	Ongoing	Storing general travel updates (e.g., Cornwall event calendar, seasonal attraction schedules)	Keeping data fresh; avoiding outdated or incorrect information

14.3.2 Human-in-the-loop

Even a well-designed travel information assistant will encounter situations where automated handling isn't enough—cases where the query is ambiguous, the available data is incomplete, or a decision could have a significant real-world impact. A

human-in-the-loop approach allows such requests to be escalated to a human travel expert for review before a response is sent. For our Cornwall-focused travel assistant, common human-in-the-loop scenarios could include the following:

- Requests for personalized itineraries involving unusual combinations of activities where safety, feasibility, or timing is uncertain
- Queries about real-time disruptions—such as severe weather, transport strikes, or event cancellations—where up-to-date human judgment is needed
- Special accommodation or accessibility requests that require confirming details with local providers

In early production deployments, human-in-the-loop is especially valuable, as it helps ensure accuracy, prevents reputational damage, and provides real-world feedback to refine the assistant's automated policies. Over time, insights from human reviews can be incorporated into improved guardrails, better prompts, or expanded knowledge bases—reducing the number of cases that need escalation.

14.3.3 Post-model guardrails

In this chapter, we focused on guardrails that run before invoking the LLM—screening queries for relevance and redirecting out-of-scope requests. In a production travel information assistant, you may also want post-model guardrails that inspect the model's output before it's shown to the user or sent to a downstream service (e.g., a booking API). For our Cornwall-focused assistant, post-model guardrails could include the following:

- Filtering outdated or incorrect details, for example, removing references to attractions that have permanently closed or events that have already passed
- Redacting sensitive information, such as private contact numbers for small B&Bs that should only be shared after confirmed bookings
- Enforcing brand tone and style, ensuring that all travel advice is given in a warm, welcoming, and concise manner consistent with the assistant's persona
- Verifying structured output, making sure any booking recommendations, price quotes, or itineraries follow the correct format expected by other systems

Post-model guardrails act as a final safety net, catching cases where the LLM's answer might look reasonable but contains factual errors, tone mismatches, or details that are inappropriate for immediate user delivery.

14.3.4 Evaluation of AI agents and applications

One of the most overlooked—but absolutely essential—steps in preparing an agent for production is systematic evaluation. This is a broad and evolving discipline, complex enough to deserve an entire book of its own, but it's critical for ensuring that your travel assistant remains accurate, safe, and efficient once deployed. For our Cornwall-focused travel information assistant, evaluation might include the following:

- *Functional testing*—Verifying that the assistant provides correct, relevant, and complete answers across a wide set of test queries, such as "Best family-friendly beaches in Cornwall" or "Current weather forecast for St Ives." This ensures it stays within scope and retrieves accurate, up-to-date information.
- *Behavioral testing*—Confirming that the assistant follows policy and safety rules, avoids giving irrelevant or unsafe travel advice, and maintains a consistent, friendly tone suitable for tourism and customer service.
- *Performance testing*—Measuring latency and API costs under realistic user loads, such as peak summer tourist season when requests might surge.
- *Regression testing*—Ensuring the assistant's reliability is preserved when prompts are refined, tools are updated, or the underlying LLM is replaced. This is done by continuously testing the system against a predefined set of ground truths.

While this book doesn't cover evaluation frameworks and methodologies in detail, you should treat evaluation as a core part of your preproduction checklist and as an ongoing process after launch. Continuous evaluation helps you catch issues before users do, adapt to changes in local events or services, and maintain the trustworthiness of your travel assistant over time.

Equipping your agents with memory and guardrails is an important milestone, but it's only part of the journey. True production readiness requires a holistic approach that includes safety, compliance, reliability, and continuous evaluation. The good news is that the foundations you've built in this chapter will make it much easier to layer in these additional capabilities as your applications grow in scope and complexity.

14.3.5 Deployment on the LangGraph platform and Open Agent Platform

The final step in preparing agents for production is deployment. How you deploy depends heavily on your organization's infrastructure strategy—whether applications run on-premises or in the cloud, and on IT policies around privacy, security, and compliance. It also reflects organizational choices: some teams favor local DevOps and Site Reliability Engineering (SRE)-driven deployments, while others rely on software as a service (SaaS)-based hosting for simplicity and scalability.

Once you've developed your LangGraph-based multi-agent system, a natural path is to deploy it on the *LangGraph Platform*, which is LangChain's fully managed hosting solution for agentic applications. It provides built-in features such as horizontal scalability, persistent state management, and end-to-end monitoring through the familiar LangSmith dashboard. The platform abstracts away much of the operational overhead, letting you move from prototype to production with minimal friction while still retaining observability and fine-grained debugging tools.

Another powerful option is to integrate your agents into the *Open Agent Platform (OAP)*. OAP is a flexible runtime and orchestration layer for AI agents that comes with a set of prebuilt agent patterns, including the multi-tool agent and the supervisor agent. These can be customized and extended to fit enterprise use cases, whether by plugging into Model Context Protocol (MCP) servers, local vector stores, or other enterprise

data sources. OAP is designed to act as a bridge between custom LangGraph agents and a broader ecosystem of composable, interoperable agents, making it especially valuable for organizations planning to run multiple agents in coordination.

Both LangGraph Platform and OAP are available as fully managed SaaS offerings, but can also be deployed into a client's own cloud environment for teams that need to maintain tighter control over data residency and compliance. This dual deployment model means you can start quickly with managed hosting and later migrate to a private setup if regulatory or operational needs demand it. Together, these deployment paths provide a smooth continuum—from development on your laptop, to scalable production hosting, to enterprise-wide agent orchestration—allowing you to choose the right tradeoff between control, convenience, and operational complexity.

Summary

- Conversational memory stores message history across turns. A user asking "What's the capital?" then "What about population?" needs the system to remember "capital of France" from the first turn.
- LangGraph checkpoints save complete agent state after each node execution, enabling resumability and branching. Checkpoints store conversation history, tool outputs, and intermediate reasoning.
- You can pause long-running workflows, resume from breakpoints after errors, and branch conversation threads from any checkpoint. This supports scenarios such as "Show me what would happen if I chose option B instead."
- Input guardrails block or redirect queries outside the agent's scope. A customer service agent rejects requests to write creative fiction or provide medical advice. Implementation uses classification models or keyword filters to detect off-topic requests before processing. This prevents wasted LLM calls and maintains focused agent behavior.
- Layered guardrails apply checks at multiple stages. The routing layer rejects obvious violations (profanity, malicious prompts), the retrieval layer filters sensitive documents, and the output layer validates responses before delivery.
- Output guardrails validate agent responses before delivery to users. They check for hallucinated citations, biased language, leaked sensitive data, or formatting errors.
- Human-in-the-loop workflows route specific cases to human operators for approval before executing actions. High-value transactions, sensitive data access, or uncertain agent decisions trigger human-in-the-loop reviews.
- Approved and rejected cases become training data for improving automated decision thresholds. Log all human-in-the-loop decisions to refine classification models and reduce future escalations.
- Agent evaluation covers multiple dimensions. Functional testing verifies correct answers, behavioral testing checks for bias or unsafe outputs, and performance testing measures latency and cost per query.

- Evaluation datasets with labeled correct/incorrect responses enable automated scoring on new model versions. Track accuracy, precision, recall, and F1 scores across evaluation runs.
- Production deployment requires several capabilities. Persistent storage maintains conversation history across sessions, monitoring tracks errors and latency in real time, and staged rollout tests changes on subsets of traffic.
- Staged rollout with canary testing catches issues before full deployment.
- To build an evaluation dataset, collect 100+ query-answer pairs, label them as correct/incorrect, and include edge cases and adversarial examples (prompt injections, out-of-scope requests).
- To run evaluation with LangSmith, upload a dataset, run the agent against all examples, review failures, and iterate on prompts/tools to improve scores.
- To monitor production metrics, track the error rate, P95 latency, tokens per query, and tool success rate. Set up alerts for anomalies.
- To implement human-in-the-loop with approval queues, pause workflow at decision points, store state in checkpoint, notify the human reviewer, and resume after approval/rejection.

appendix A
Trying out LangChain

A.1 Trying out LangChain in a Jupyter Notebook environment

We'll begin with a straightforward example: completing a sentence using an OpenAI model and refining its output through prompt engineering. OpenAI's models are a convenient starting point because they are accessible through a public REST API and require no local infrastructure setup.

If you prefer, you can use an open source LLM inference engine such as Ollama. However, because many of you may not have run an open source LLM locally before, we'll cover that setup in detail in appendix E. For now, we'll keep things simple and use the OpenAI REST API.

Before you proceed, make sure you have Python 3.11 or later installed on your local machine and that the following prerequisites are met:

- You already own or generate an OpenAI key.
- You know how to set up a Python Jupyter Notebook environment.

If you haven't met these prerequisites and you're unfamiliar with creating an OpenAI key, I'll walk you through the steps next. In addition, if you need help setting up a Jupyter Notebook environment, see appendix B.

For those of you who need it, let's quickly create an OpenAI key. Assuming you've registered with OpenAI, which is necessary to explore the ChatGPT examples discussed at the beginning of the chapter, follow these steps:

1. Log in to your OpenAI account at https://platform.openai.com/, and navigate to the API section.
2. Access the API Keys by clicking your profile icon > Your profile > User API Key (tab).

3. Create a new secret key by clicking the Create Secret Key button, naming it (e.g., BuildingLLMApps), and confirming.
4. Safely save the key, for example, in a secure local file or a password vault tool, as it's impossible to retrieve the key later.
5. Set a spending limit (e.g., $10) to control costs. Configure this under Settings > Limits in the left-hand menu based on your usage preferences.

Because most readers use Windows, I'll be providing instructions specifically for Windows and a Python virtual environment. If you're using Linux or Anaconda, I assume you're advanced enough to adapt these instructions to your setup. If you prefer, you can also run these examples in an online notebook environment, such as Google Colab, as long as you install the required packages.

Now you can establish the virtual environment for the code in this appendix. First, open a new terminal in your operating system, create a folder such as `c:\Github\building-llm-applications\ch01`, navigate into it, and execute the following:

```
C:\>cd Github\building-llm-applications\ch01

C:\Github\building-llm-applications\ch01>python -m venv env_ch01

C:\Github\building-llm-applications\ch01>.\env_ch01\Scripts\activate

(env_ch01) C:\Github\building-llm-applications\ch01>
```

> **NOTE** I'm running these commands on a Windows cmd shell. If you're on a Linux computer, you might have to adapt things slightly. For example, you can activate a virtual environment with `./env_ch01/bin/activate` or `./env_ch01/Scripts/activate`. If you're using PowerShell, you should use `Activate.ps1`.

Now install the `notebook`, `langchain`, and—indirectly—`openai` packages. If you've cloned the GitHub repository for this book (https://mng.bz/V9DG) or downloaded the code zip file from the Manning website, you can install all the required packages with the following command. This ensures you're using the correct versions:

```
(env_ch01) C:\Github\building-llm-applications\ch01>
➥pip install -r requirements.txt
```

Once the installation is complete, start up a Jupyter Notebook:

```
(env_ch01) C:\Github\building-llm-applications\ch01>jupyter notebook
```

Create a notebook by choosing File > New > Notebook, and then rename it with File > Rename to `01-langchain_examples.ipynb`.

If you've cloned the GitHub repository and want to use the notebook directly, start it up as follows:

```
(env_ch01) C:\Github\building-llm-applications\ch01>
➥jupyter notebook 01-langchain_examples.ipynb
```

A.1.1 Sentence completion example

Now you're ready to execute LangChain code in the notebook. Start by importing the LangChain library and configuring an OpenAI LLM instance.

In the initial cell of your notebook, import the necessary libraries by running the cell:

```
from langchain_openai import ChatOpenAI
import getpass
```

> **NOTE** If you're unfamiliar with Jupyter Notebooks, you can run a cell by pressing Shift-Enter or clicking the Play button located next to the top menu.

Now add and execute a cell to grab your OpenAI API key (just press Enter after inserting the key):

```
OPENAI_API_KEY = getpass.getpass('Enter your OPENAI_API_KEY')
```

> **Setting the OpenAI API key with an environment variable**
>
> Another way to set the OpenAI key is by using an environment variable. For example, in Windows, you can set it in the command shell like this before launching the notebook:
>
> ```
> set OPENAI_API_KEY=your_openai_api_key
> ```
>
> Then, in your Python code, retrieve it with the following (make sure you use `import os`):
>
> ```
> api_key = os.getenv("OPENAI_API_KEY")
> ```
>
> Because setting environment variables varies across operating systems—and some of you may not be familiar with the process—I won't use this method for configuring the OpenAI key in this book. However, if you're comfortable using environment variables, feel free to use them.

Once you've entered your OpenAI key, add and execute this cell:

```
llm = ChatOpenAI(openai_api_key=OPENAI_API_KEY,
                 model_name="gpt-5-nano")
llm.invoke("It's a hot day, I would like to go to the…")
```

> **NOTE** To keep execution costs low, we use GPT-5-nano, the smallest and most affordable model in the GPT-5 family. You'll see it used throughout most of this book. However, you're welcome to switch to GPT-5-mini or GPT-5 if you prefer higher accuracy levels.

After executing the code, you'll see output similar to the following example, which represents a return message from the LLM:

```
AIMessage(content="…beach to cool off and relax in the refreshing water. The
sound of the waves crashing against the shore and the feeling of the warm sand
```

```
beneath my feet is exactly what I need to unwind and escape from the heat.",
response_metadata={'finish_reason': 'stop', 'logprobs': None})
```

As you can see in the `content` property, the completion generated by the LLM is `"…beach to cool off [. . .]"`.

A.1.2 Prompt engineering examples

I've mentioned prompts several times, but I haven't shown you any examples yet. A prompt is the instruction you give the LLM to complete a task and generate a response. It's such a key part of any LLM application that developing LLM applications often involves spending a lot of time designing and refining prompts through trial and error. Various patterns and techniques are already forming around prompts, leading to the emergence of a discipline called *prompt engineering*. This field focuses on crafting prompts that yield the best possible answers. I'll dedicate the next chapter to teaching you the fundamentals of prompt engineering. For now, let's start with a straightforward prompt.:

```
prompt_input = """Write a short message to remind users to be
vigilant about phishing attacks."""
response = llm.invoke(prompt_input)

print(response.content)
```

The output follows:

```
Just a friendly reminder to stay vigilant against phishing attacks. Be
cautious of any suspicious emails, messages, or requests for personal
information. Stay safe online!
```

PROMPT TEMPLATE

In chapter 2, you'll learn about prompt templates—structured prompts that allow you to run various versions of the same theme. LangChain offers a class called `PromptTemplate` that generates prompts from template structures and input parameters. The following listing is an example of how to create and execute a prompt from a template, which you can place in a single notebook cell.

Listing A.1 Creating a prompt from a `PromptTemplate`

```
from langchain_core.prompts import PromptTemplate

segovia_aqueduct_text = """The Aqueduct of Segovia
(Spanish: Acueducto de Segovia) is a Roman aqueduct in Segovia,
Spain. It was built around the first century AD to channel water
from springs in the mountains 17 kilometres (11 mi) away to the
city's fountains, public baths and private houses, and was in
use until 1973. Its elevated section, with its complete arcade
of 167 arches, is one of the best-preserved Roman aqueduct
bridges and the foremost symbol of Segovia, as evidenced by
its presence on the city's coat of arms. The Old Town of
```

```
Segovia and the aqueduct, were declared a UNESCO World
Heritage Site in 1985. As the aqueduct lacks a legible
inscription (one was apparently located in the structure's
attic, or top portion[citation needed]), the date of
construction cannot be definitively determined. The general
date of the Aqueduct's construction was long a mystery,
although it was thought to have been during the 1st century AD,
during the reigns of the Emperors Domitian, Nerva, and Trajan.
At the end of the 20th century, Géza Alföldy deciphered the
text on the dedication plaque by studying the anchors that held
the now missing bronze letters in place. He determined that Emperor
Domitian (AD 81–96) ordered its construction[1] and the year 98 AD
was proposed as the most likely date of completion.[2] However,
in 2016 archeological evidence was published which points to a
slightly later date, after 112 AD, during the government of
Trajan or in the beginning of the government of emperor Hadrian,
from 117 AD."""

prompt_template = PromptTemplate.from_template("""You are an
experienced copywriter. Write a {num_words} words summary of
the following text, using a {tone} tone: {text}""")

prompt_input = prompt_template.format(
    text=segovia_aqueduct_text,
    num_words=20,
    tone="knowledgeable and engaging")
response = llm.invoke(prompt_input)
print(response.content)
```

When executing the preceding code, you should get this output or something similar:

```
The Aqueduct of Segovia, a Roman marvel in Spain, dates back to the 1st century
AD, channeling water for centuries.
```

In chapter 2, I'll show you how to replicate the examples we've just implemented in LangChain using the plain OpenAI REST API so you can see the differences.

A.1.3 Creating chains and executing them with LCEL

One of the benefits of using LangChain is its processing technique built around the concept of a chain. A *chain* is a pipeline of components put together to achieve a particular outcome. For example, to illustrate, you could create a chain like this to scrape the latest news from a website, summarize it, and email it to someone (don't execute this code snippet):

```
chain = web_scraping | prompt | llm_model | email_text
```

This declarative, intuitive, and readable method of defining a chain showcases the LangChain Expression Language (LCEL), which is covered extensively in chapter 5. For now, let's walk through an example by reimplementing the previous summarization task using LCEL. First, set up the chain in a new notebook cell as follows:

```
prompt_template = PromptTemplate.from_template("You are an experienced
➥copywriter. Write a {num_words} words summary of the following text,
➥using a {tone} tone: {text}")
llm = ChatOpenAI(openai_api_key=OPENAI_API_KEY,
                 model_name="gpt-5-nano")

chain = prompt_template | llm
```

Now this chain is ready to accept any text, target number of words, and target tone. Execute it as shown here:

```
response = chain.invoke({"text": segovia_aqueduct_text,
              "num_words": 20,
              "tone": "knowledgeable and engaging"})
print(response.content)
```

You'll get this output or something similar:

```
The Aqueduct of Segovia: A Roman marvel channeling water to the city, adorned
with 167 arches, symbolizing Segovia's rich history.
```

As you can see, this way of setting up and executing the processing is somewhat simpler than the original imperative approach. The `chain` pipeline works because both `PromptTemplate` and `ChatOpenAI` objects implement a common interface (`Runnable`) that allows them to be linked. The `|` syntax is syntactic sugar for creating a `Chain` object behind the scenes.

appendix B
Setting up a Jupyter Notebook environment

If you're just starting with Python Jupyter notebooks, think of it as an interactive space where you can type and run code, see the results, and tweak the code to get the outcomes you want in real time. The code snippets I'll be sharing aren't exclusive to a particular Python version, but for the smoothest experience, run them on Python 3.11 or a newer version if you can.

B.1 Installing the Python interpreter or a Python distribution

If you haven't actively used Python lately, I suggest installing the latest version of the Python 3 interpreter. You have a few options:

- *Standalone Python interpreter*—Download and install Python 3 from python.org. Choose the installer appropriate for your operating system, and follow the setup instructions. You can find OS-specific guidance in the official documentation here: https://docs.python.org/3/using/index.html (you may need to scroll to locate your platform). This option is ideal if you prefer a lightweight Python installation without the additional libraries and tools bundled with distributions such as Anaconda or Miniconda.
- *Anaconda*—Download Anaconda from www.anaconda.com/download. It comes with Python and a wide range of data science libraries, including tools for visualization, data analysis, and numerical/statistical work. This option is perfect for data science and machine learning projects if you have enough disk space. Anaconda also makes it easy to create virtual environments for specific projects. It includes Anaconda Navigator, a user-friendly graphical interface, for those who prefer not to use command-line tools.

- *Miniconda*—This is a lighter alternative to Anaconda that allows you to manage virtual environments for specific applications without taking up much disk space. It only includes essential data science libraries. You can learn more at https://docs.anaconda.com/miniconda/.

For the remainder of this appendix, let's assume you have Python installed (as described in the first bullet option in the preceding list). If you've chosen Miniconda or Anaconda, I assume you're familiar with Python and can adapt my instructions as needed. Now you're ready to set up a virtual environment using `venv`, which is a tool for managing virtual environments, and to update pip, the Python package installer.

B.2 Creating a virtual environment with venv and upgrading pip

Open the operating system terminal shell (e.g., cmd on Windows), create a project folder, and navigate into it:

```
C:\Github\building-llm-applications\ch01>
```

Create a virtual environment with `venv`. A virtual environment serves as a self-contained Python installation, specifically for the `ch01` folder you've recently created. This ensures that package version conflicts with other projects on your machine are avoided. Create a virtual environment for `ch01` using the following command:

```
C:\Github\building-llm-applications\ch01>python -m venv env_ch01
```

You've successfully set up a virtual environment named `env_ch01`. Now activate it using the following command:

```
C:\Github\building-llm-applications\ch01>.\env_ch01\Scripts\activate
```

You should now see the updated operating system prompt, displaying the environment name in front as `(env_ch01)`:

```
(env_ch01) C:\Github\building-llm-applications\ch01>
```

Before proceeding further, it's useful to upgrade `pip`, the Python package management tool, as follows so you'll be able to install the necessary Python packages with no issues:

```
(env_ch01) C:\Github\building-llm-applications\ch01>
➥python -m pip install --upgrade pip
```

For additional details on the steps you've taken, refer to the following documentation on the python.org website: https://mng.bz/xZPX.

B.3 Setting up a Jupyter Notebook

With the virtual environment activated, you're ready to set up a Jupyter Notebook for running prompts with OpenAI models. Install Jupyter and the required LangChain packages using the following steps, referencing the `requirements.txt` file from the

book's GitHub repository (https://mng.bz/AG4x) or from the code zip file available on the Manning book page:

```
(env_ch01) C:\Github\building-llm-applications\ch01>
➥pip install -r requirements.txt
```

After about a minute, the installation of the notebook and LangChain packages should be finished. If you'd like, you can confirm it by using the following command:

```
(env_ch01) C:\Github\building-llm-applications\ch01>pip list
```

You can start the Jupyter Notebook by executing the following command:

```
(env_ch01) C:\Github\building-llm-applications\ch01>jupyter notebook
```

After a few seconds, you should see this output in the terminal:

```
[I 2023-10-23 22:43:26.870 ServerApp] Jupyter Server 2.8.0 is running at:
[I 2023-10-23 22:43:26.870 ServerApp]
➥http://localhost:8888/tree?
➥token=da9d38f7f0d9b2c4c3aba08c00c7ca2b5ae21f1ee1d42c30
[I 2023-10-23 22:43:26.871 ServerApp]
➥http://127.0.0.1:8888/tree?
➥token=da9d38f7f0d9b2c4c3aba08c00c7ca2b5ae21f1ee1d42c30
[I 2023-10-23 22:43:26.872 ServerApp] Use Control-C to stop
➥this server and shut down all kernels (twice to skip confirmation).
➥2023-10-23 22:43:26.978 ServerApp]
```

Subsequently, a browser window will open up at `http://localhost:8888/tree`, as shown in figure B.1. To create the notebook, choose File > New > Notebook. Then, select File > Rename, and name the file `langchain_examples.ipynb`. Now you're prepared to input code into the notebook cells.

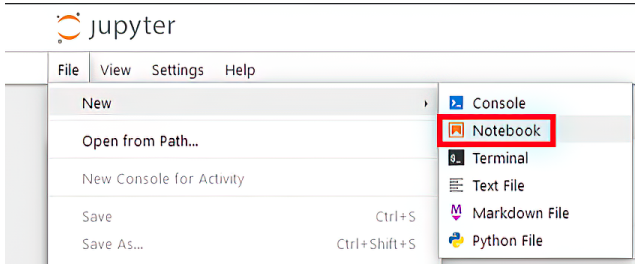


Figure B.1 Creating a new Jupyter Notebook

TIP If you're unfamiliar with Jupyter Notebook, remember to press Ctrl-Enter to execute the code in each cell.

appendix C
Choosing an LLM

This appendix outlines the key features of the most popular large language models (LLMs) available at the time of writing. It also highlights the criteria to consider when selecting the most suitable LLM for your project.

C.1 Popular large language models

Many LLMs are available today. Let's take a look at some of the most popular on the market.

C.1.1 OpenAI GPT series

OpenAI's GPT-4, released in March 2023, marked a turning point as the first widely recognized frontier model to demonstrate advanced reasoning capabilities. It also pioneered multimodality, initially handling text and images and later extending to audio and video, which opened the door to a wide range of applications. Although OpenAI hasn't disclosed its architecture, GPT-4 was widely reported to use a Mixture-of-Experts (MoE) design and possibly scale to trillions of parameters, a shift aimed at improving accuracy and efficiency across varied tasks.

The GPT-4 line evolved through successive versions—GPT-4o and GPT-4.1—before converging with OpenAI's separate reasoning-focused series (o1, o3, o4-mini) into the unified GPT-5 family, which also includes lighter mini and nano variants. GPT-5 represents a significant consolidation: instead of users choosing between instruction-tuned models and reasoning-tuned models, a single model now dynamically decides whether to emphasize fast instruction following or deeper reasoning based on the user's request. This shift reflects a broader trend in LLM development: reducing complexity for end users while expanding flexibility under the hood. LangChain integrates seamlessly with OpenAI models through the `langchain-openai` package and also supports open source models using inference engines compatible with the OpenAI REST API, as detailed in appendix E.

C.1.2 Gemini

In February 2024, Google introduced Gemini 1.5, a family of multimodal models capable of processing text, audio, images, and code. The lineup ranged from Gemini Ultra, designed for advanced tasks using an MoE architecture, to Gemini Nano, optimized for lightweight mobile applications. Gemini Ultra surpasses Pathways Language Model 1 (PaLM 2) in benchmarks, excelling in reasoning, math comprehension, and code generation.

Gemini Ultra's standout feature is its large context window, accommodating vast data inputs: up to 1 hour of video, 11 hours of audio, and codebases exceeding 30,000 lines, or more than 700,000 words. Research tests extended to 10M tokens, demonstrating its capacity for extensive workloads.

Gemini 2.0, introduced in December 2024, marks a significant advancement in AI, enabling sophisticated reasoning for tasks such as multimodal queries, coding, and complex math, powered by Trillium tensor processing units (TPUs). Its leading model, Gemini 2.0 Flash, offers fast, high-performance capabilities with support for multimodal input and output, including text, images, video, audio, and steerable multilingual text-to-speech (TTS). Enhancing its predecessor, Gemini 2.0 Flash operates at twice the speed of Gemini 1.5 Pro, integrates native tools, and sets new standards in AI performance.

Gemini 2.5, launched in March 2025, builds on the foundation of Gemini 2.0 with enhanced performance, efficiency, and broader multimodal capabilities. Designed for both advanced and real-time applications, Gemini 2.5 Pro offers improved reasoning, longer context handling, and faster response times across text, image, audio, and video inputs. It introduces better integration with Google's ecosystem, tighter tool use, and more reliable multilingual support, positioning it as a competitive alternative to other state-of-the-art models. LangChain offers seamless access to Gemini models through the `langchain-google-genai` package, enabling developers to integrate these powerful models into their applications effortlessly.

Gemini 3.0—introduced in November 2025—represents a major step forward in Google's multimodal AI lineup. It offers stronger reasoning abilities; native multimodal processing across text, images, video, audio, and code; and an extended context window of up to 1 million tokens, enabling extremely long inputs to be handled at once. Designed as a versatile builder and planner, Gemini 3.0 supports complex tasks such as coding, UI generation, and agent-style workflows. The model is deeply integrated into Google's ecosystem, powering features in Google Search, the Gemini app, AI Studio, and Vertex AI. It shows substantial improvements in reliability, speed, and coherence, especially for long-form and mixed-media queries.

C.1.3 Gemma

In February 2024, Google rolled out Gemma, an open source counterpart to Gemini, designed with a focus on lightweight functionality. Built on the same research and

tech stack as Gemini, Gemma offers model weight files at 2B and 7B parameters, freely accessible for use. The models are optimized to run on NVIDIA GPUs and Google Cloud TPUs. Additionally, Gemma is provided with a toolkit to facilitate effective use for both fine-tuning and inference tasks.

C.1.4 Claude

In March 2023, Anthropic and Google launched Claude, a language model designed with a strong emphasis on honesty and safety. Claude excels in summarization, coding, writing, and chatbot-based question answering. Although Anthropic hasn't shared details about its architecture or training, Claude set a new standard by supporting 100,000 tokens, which at the time was the biggest context window, enabling it to handle complex tasks. To improve speed, Anthropic released Claude Instant, optimized for faster performance.

Subsequent updates, including Claude 2 in July 2023, Claude 3 in March 2024, Claude 3.5 in October 2024, and Claude 3.7 in February 2025, enhanced accuracy, safety, and versatility across diverse language tasks. Claude 3.7 is available in the bigger, accurate version, called Sonnet, and the smaller, faster version, called Haiku. Claude 3.7 Sonnet is in the same segment as GPT-4.1 and Gemini 2.5 Pro. Haiku can be considered in the same group as GPT-4o-mini. Anthropic models, including the Claude family, are accessible by installing the `langchain-anthropic` package.

Claude Opus 4.5, released in late 2025, is the most advanced version of Claude, offering major improvements in reasoning, coding, and multi-step agentic tasks. It delivers stronger reliability, better long-context handling, and more robust performance on complex or messy prompts. With these upgrades, Claude 4.5 evolves from a general-purpose assistant into a capable tool for deep research, advanced coding, and sustained professional workflows.

C.1.5 Cohere

Cohere, backed by former Google Brain employees, including Aidan Gomez, coauthor of the influential "Attention Is All You Need" paper (https://arxiv.org/abs/1706.03762), focuses exclusively on enterprise needs. Known for precision and consistency, Cohere offers models ranging from 6B to 52B parameters, allowing organizations to tailor their approach.

Cohere has achieved the highest accuracy among LLMs at some point, making it a dependable choice for businesses. Major corporations such as Jasper and Spotify have adopted Cohere for advanced natural language processing (NLP), highlighting its practical applicability. However, it's worth noting that Cohere's technology comes at a higher cost compared to more widely recognized models from OpenAI. You can access Cohere models from LangChain by installing the `langchain-cohere` package.

C.1.6 Llama

Meta's Llama series of LLMs has played a major role in advancing open-access AI. First introduced in 2023 with a model with up to 65B parameters, Llama quickly evolved

into a flexible, open source platform with smaller variants to suit different computational needs. Built on transformer architecture and trained on diverse public datasets, Llama inspired derivative models such as Vicuna and Orca. Over 2023 and 2024, Meta released Llama 2 and Llama 3, including a 405B-parameter model and the introduction of vision capabilities. In April 2025, Meta launched Llama 4, featuring major architectural improvements. The Llama 4 lineup includes Scout (109B parameters, multimodal, 10M token context), Maverick (400B parameters, optimized for reasoning), and Behemoth (a 2T-parameter model still in training, aimed at outperforming current benchmarks). While open-weight, these models come with licensing constraints for commercial use. Llama's scalability, openness, and compatibility with tools such as LangChain—via GPT4All and Ollama wrappers, for example—make it a powerful resource for both researchers and developers in AI and NLP.

C.1.7 Falcon

Developed by the Technology Innovation Institute (TII), Falcon is a transformer-based, causal decoder-only model designed for NLP tasks. Falcon 3, released in December 2024, demonstrates strong performance compared to its peers. For instance, Falcon 3 10B achieves results comparable to Qwen 2.5 7B and Gemma 9B across benchmarks such as Multistep Soft Reasoning (MUSR) and Massive Multitask Language Understanding Professional (MMLU Pro), while outperforming them on the Mathematics Aptitude Test for Heuristics (MATH) benchmark. This positions Falcon 3 as a competitive option among contemporary open source LLMs.

C.1.8 Mistral

Founded in 2023, Mistral AI has rapidly become a leader in open source LLMs and is known for its efficient and high-performing designs. It introduced the Mistral 7B model, followed by the Mixtral 8x7B and 8x22B models—both using Sparse Mixture-of-Experts (SMoE) architectures that activate only a subset of parameters per token to improve cost-efficiency and performance. In May 2025, the company released Mistral Medium 3, a midsized model optimized for enterprise use, offering strong performance at lower cost and supporting hybrid deployments. Mistral's ongoing innovation and open source approach position it as a major competitor in the AI space, balancing scalability, affordability, and flexibility across a range of applications.

C.1.9 Qwen

Qwen, open sourced on GitHub in August 2023, is a family of LLMs developed by Alibaba Cloud. The models range from 1.8B to 72B parameters, trained on datasets between 2.2T and 3T tokens. While Qwen models are designed for general purposes, there are fine-tuned versions for specific tasks, such as Code-Qwen for coding and Math-Qwen for mathematics. Qwen-Chat has been fine-tuned using Reinforcement Learning from Human Feedback (RLHF), similar to ChatGPT. The models support a context length of around 30,000 tokens and perform particularly well in English and Chinese.

C.1.10 Grok

Grok, developed by xAI, has rapidly advanced since its debut in late 2023. Initially focused on conversational tasks and integrated with the X platform, Grok used an MoE architecture and supported up to 128,000 tokens. Subsequent versions—Grok-2 and Grok-2 Mini—added features such as image generation and improved performance and speed. In February 2025, xAI released Grok 3, significantly upgraded with 10x more compute, new reasoning modes (Think and Big Brain), real-time data retrieval via DeepSearch, and image editing capabilities. In May 2025, Grok 3.5 launched in beta with enhanced technical reasoning and a redesigned Retrieval-Augmented Generation (RAG) system. Microsoft also partnered with xAI to host Grok 3 and Grok 3 Mini on Azure AI Foundry, bringing the models to a broader enterprise audience.

C.1.11 Phi

The Phi-3 family comprises small language models (SLMs) designed to provide many capabilities of LLMs while being more resource-efficient. These models outperform others of the same and next size up in benchmarks for language, coding, and math tasks, thanks to Microsoft's innovative training methods. The Phi-3-mini (3.8B parameters) delivers exceptional performance, rivaling models twice its size, with future releases including Phi-3-small (7B parameters) and Phi-3-medium (14B parameters). The models are accessible through the Microsoft Azure AI Model Catalog, Hugging Face, Ollama for local deployment, and as NVIDIA Inference Microservices (NIM) with standard APIs.

Phi-3.5-MoE, the latest addition to the Phi family, is a lightweight, state-of-the-art open model optimized for reasoning-intensive tasks such as code, math, and logic. It supports multilingual applications with a 128K token context length. Developed using high-quality datasets, it incorporates advanced fine-tuning and optimization techniques for precise instruction adherence and robust safety. Designed for memory-constrained and latency-sensitive environments, Phi-3.5-MoE powers general-purpose AI systems and is accessible through Azure AI Studio and GitHub via a serverless API, offering scalable and cost-efficient deployment.

C.1.12 DeepSeek

DeepSeek, a Chinese AI company founded in 2023 by Liang Wenfeng, has developed open source LLMs that rival leading systems in performance and cost-efficiency. Its DeepSeek-V3 model, an MoE architecture with 671B parameters (activating 37B per token), was trained on 14.8T tokens using supervised fine-tuning and reinforcement learning. Despite its scale, training required only 2.788M GPU hours on NVIDIA H800 chips, costing less than $6M. It outperforms other open source models and matches top proprietary systems, excelling in mathematics, programming, reasoning, and multilingual tasks, with its AI assistant surpassing ChatGPT as the top free app on Apple's App Store.

DeepSeek-R1 also competes with high-end LLMs such as OpenAI's o1, specializing in complex reasoning and coding through chain-of-thought (CoT) reasoning. Trained with 2,000 NVIDIA H800 chips at a cost of $5.6M, DeepSeek-R1's efficiency sparked debates on sustainable AI training. While both models are open source, they avoid politically sensitive topics, raising privacy concerns and prompting global discussions about China's growing influence in AI and the shifting balance of technological power.

To wrap things up in this section, table C.1 provides a summary of the key characteristics of the LLMs discussed.

Table C.1 Comparison of LLMs

Model	Developer	Launch	No. of parameters	Max no. of tokens	Open source
GPT-4o	OpenAI	May 24	200B*	128K	No
GPT-4.1	OpenAI	Apr. 25	N/A	1M	No
o1	OpenAI	Oct. 24	400B*	128K	No
o3	OpenAI	Dec. 24	200B*	128K	No
Gemini 2.5 Pro	Google	May 25	N/A	1M	No
Gemma-3	Google	Mar 25	1B–27B	128K	Yes
Claude Sonnet-3.7	Anthropic	Feb. 25	N/A	200K	No
Command R+	Cohere	Aug. 24	100B	128K	No
Llama-4	Meta	Sep. 24	90B	128K	Yes
Falcon3-10B-Base	TII-UAE	Dec. 24	10B	32K	Yes
Mixtral-8x22B-Instruct-v0.1	Mistral AI	Mar. 24	141B	64K	Yes
Qwen2.5 72B	Alibaba Cloud	Sep. 24	72B	128K	Yes
Grok 3.5	xAI	Feb. 25	314B	1M	No
DeepSeek-V3	DeepSeek AI	Dec. 24	671B	128K	Yes
DeepSeek-R1	DeepSeek AI	Jan. 25	671B	132K	Yes
Phi-4.0	Microsoft	Dec. 24	14B	128K	Yes

*Estimated size

C.2 How to choose a model

Each use case has unique requirements. Selecting the ideal LLM for your specific needs can be a complex task that often involves testing multiple models to identify the most suitable one. However, you can employ certain criteria to streamline the

selection process. Following are several key factors to consider, which we'll explore in greater depth next to help you make well-informed decisions:

- Model purpose
- Proprietary versus open source
- Model size
- Context window size
- Supported languages
- Accuracy versus speed
- Cost and hardware requirements
- Task suitability
- Safety and bias

C.2.1 Model purpose

Some LLMs are flexible and can manage various tasks, while others are tailored for specific functions. For instance, OpenAI's GPT-4 or Gemini 1.5 Pro are versatile and adaptable models, suitable for a range of tasks, while Code Llama and Qwen-Coder, as their names suggest, are specialized for generating programming code.

Specialized models are usually smaller than their more general counterparts. If you have a clear use case in mind, it's a good idea to choose an LLM specifically created and trained for that purpose. There are common LLM specializations, ranging from simpler tasks such as text classification and sentiment analysis to more advanced functions such as text summarization, translation, code analysis, text generation, question-answering, and logic and reasoning. Many foundational LLMs can handle combinations of these functions, offering a wide range of possibilities to meet your specific needs.

C.2.2 Proprietary vs. open source

Many language models are proprietary, meaning their developers keep the internal details private. Information such as architecture, parameter count (discussed in the next section), or training specifics is often undisclosed. Proprietary models from providers such as OpenAI, Gemini, Cohere, Anthropic, and Stability AI are typically offered as cloud-based subscription services accessed through REST APIs. Pricing depends on the accuracy of the model used and the number of tokens processed. This approach provides convenience, allowing users to access ready-to-use services without managing hardware or infrastructure. However, data submitted to these services is retained and potentially processed by the provider, which can be a concern for sensitive data.

In contrast, open source models offer full transparency, providing access to their underlying implementation as well as detailed information about their architecture and training processes. This transparency provides two key advantages. First, it enables you to deploy a fully private solution, avoiding the need to send sensitive data to third-party vendors—an important consideration for privacy and security. Second, it allows for fine-tuning on domain-specific datasets, giving you the flexibility to adapt

the model to your unique needs. The tradeoff, however, is that you typically need to host and manage the infrastructure yourself, which involves provisioning GPUs and dedicating IT resources to maintain and scale the service. As an alternative, you can opt for managed LLM hosting platforms such as IBM watsonx, Amazon Bedrock, Azure AI Studio, or Google Vertex AI. These services provide a scalable, secure environment for running both open source and proprietary models—eliminating much of the operational complexity associated with self-hosting. LangChain supports both proprietary models (e.g., OpenAI, Gemini, and Cohere) and open source models through inference engine wrappers such as Ollama.

C.2.3 Model size (number of parameters)

A model's parameters represent its internal variables, specifically the weights in the artificial neural network. These weights are adjusted during training and enable the model to learn patterns from data. More parameters allow a model to capture greater complexity and nuance, potentially increasing accuracy. However, larger models with more parameters require substantial hardware resources—such as increased memory and high-performance GPUs—and often result in higher latency during inference. Compression techniques can reduce a model's size without major loss of accuracy, which we'll discuss in appendix E.

Language models vary widely in parameter count, from trillions in GPT-4 and Gemini 1.5 Ultra to a few billion in Mistral and Gemma. Parameters form the foundation of a model's ability to process text tasks.

C.2.4 Context window size

The number of input tokens allowed in LLMs (LLMs) directly impacts their functionality and suitability for different tasks. Token limits vary across models, influencing the complexity of prompts they can handle.

Models with smaller token limits are well-suited for concise prompts and straightforward interactions. These limitations often stem from design choices or computational constraints, and such models are typically specialized for a narrow set of tasks. In contrast, LLMs with higher token capacities—often designed as generalists—can handle more detailed, context-rich inputs, making them better suited for complex, multi-step tasks.

Choosing the right token limit depends on the task. For short, straightforward inputs, smaller token allowances may suffice. For applications requiring detailed interactions or extended context, models with larger token limits are more appropriate.

In summary, token capacity is a key consideration when selecting an LLM. Aligning the token limit with your project's requirements ensures effective use of the model's capabilities.

C.2.5 Multilingual support

When considering an LLM for multilingual support, it's essential to research which one aligns with your requirements. Some LLMs are proficient in multiple languages,

including ancient Latin, Greek, and even Phoenician, while others are primarily English-focused. Typically, LLMs excel with languages that have extensive training data, such as English, Chinese, and Spanish. If your needs involve a less common language, you might need to seek a specialized LLM or even undertake fine-tuning yourself. It's crucial to match your language requirements with the capabilities of the chosen LLM for optimal results.

C.2.6 Accuracy vs. speed

Selecting an LLM often requires balancing accuracy and processing speed. Larger models with more parameters deliver higher precision and nuanced responses, especially for complex language tasks. However, this accuracy comes at the cost of slower processing and significant computational requirements.

Smaller models, with fewer parameters, are faster and better suited for real-time applications. While they sacrifice some accuracy, they excel in tasks requiring quick responses. The choice between a large or small model depends on the specific needs of the application—detailed comprehension favors larger models, while speed-sensitive tasks benefit from smaller ones.

Advances in compression techniques, discussed in appendix E on open source models, have bridged this gap. Compact models with lower parameter counts now achieve accuracy comparable to much larger models, making the tradeoff between speed and precision less of a limitation.

C.2.7 Cost and hardware requirements

Cost and hardware requirements are key factors when deploying LLMs. Organizations must carefully weigh financial and technical considerations to ensure effective use of these models.

Proprietary LLMs are typically priced based on their accuracy and the number of tokens processed. While they deliver high precision, enhanced capabilities result in higher costs, requiring organizations to balance accuracy against budget constraints.

Open source LLMs, though more affordable in terms of licensing, shift the cost burden to hardware and infrastructure. Deploying these models demands powerful GPUs, sufficient RAM, and reliable virtual machines. Additionally, organizations need skilled IT staff to manage deployment, maintenance, and support.

The choice between proprietary and open source LLMs depends on organizational goals and resources. Proprietary models offer convenience and performance for a fee, while open source models provide flexibility and cost savings at the expense of hardware and IT investment. Careful evaluation ensures that the solution aligns with both objectives and available resources.

C.2.8 Task suitability (standard benchmarks)

The effectiveness of a language model for specific tasks depends on its architecture, size, training data, and fine-tuning. Different models are optimized for different use cases, and standardized benchmarks help evaluate their performance across various tasks.

Leaderboards such as the Hugging Face Open LLM Leaderboard and LMSYS Chatbot Arena Leaderboard offer a centralized way to compare models across benchmarks, streamlining the evaluation process. Following are some widely recognized benchmarks, with descriptions drawn from the corresponding research publications:

- *Instruction-Following Evaluation (IFEval)*—IFEval is an evaluation benchmark for LLMs that is designed to assess their ability to follow natural language instructions. It uses 25 types of verifiable instructions, such as word count or keyword mentions, and includes 500 prompts. IFEval provides a simple, reproducible alternative to both human evaluations, which are costly and inconsistent, and LLM-based evaluations, which may be biased. See https://arxiv.org/abs/2311.07911.
- *BIG-Bench Hard (BBH)*—The BBH benchmark is a subset of 23 challenging tasks from the BBH evaluation suite, designed to test areas where language models have historically underperformed compared to average human raters. These tasks often require multi-step reasoning and highlight the limitations of language models without advanced prompting techniques. Using CoT prompting, models such as PaLM and Codex demonstrated significant improvements, surpassing human performance on 10 and 17 tasks, respectively. The benchmark emphasizes that few-shot prompting alone underestimates the potential of language models, and CoT prompting reveals their advanced reasoning capabilities, particularly as model scale increases. See https://arxiv.org/abs/2210.09261.
- *Mathematics Aptitude Test for Heuristics, Level 5 (MATH, L5)*—The MATH benchmark is a dataset of 12,500 challenging mathematics problems designed to evaluate the mathematical problem-solving abilities of machine learning models. Each problem includes a detailed step-by-step solution, enabling models to learn answer derivations and explanations. Alongside MATH, an auxiliary pretraining dataset is provided to help models grasp fundamental mathematical concepts. Despite improvements, model accuracy on MATH remains low, indicating that scaling transformer models alone is insufficient for strong mathematical reasoning. Advancing this capability will likely require novel algorithms and research breakthroughs. See https://arxiv.org/abs/2103.03874.
- *Graduate-Level Google-Proof Q&A (GPQA)*—GPQA is a dataset of 448 multiple-choice questions in biology, physics, and chemistry, created by domain experts to be exceptionally difficult. Expert participants with advanced degrees achieve 65% accuracy (74% excluding identified errors), while nonexperts with unrestricted web access average only 34%, making the questions effectively "Google-proof." GPQA is designed to aid in developing scalable oversight methods, enabling humans to supervise and extract reliable, truthful information from AI systems, even those surpassing human capabilities. See https://arxiv.org/abs/2311.12022.
- *Multi-step Soft Reasoning (MuSR)*—MuSR is a benchmark designed to evaluate LLMs on multi-step reasoning tasks framed within natural language narratives.

It features complex reasoning scenarios, such as 1,000-word murder mysteries, generated using a neurosymbolic synthetic-to-natural algorithm. These tasks are challenging for advanced models and can scale to match future LLM advancements. MuSR emphasizes real-world reasoning, offering narratives that are more realistic and difficult than typical synthetic benchmarks while remaining accessible for human annotation. The benchmark highlights limitations in current reasoning techniques, such as CoT prompting, and helps identify areas for improvement in robust reasoning capabilities. See https://arxiv.org/abs/2310.16049.

- *Massive Multitask Language Understanding Professional (MMLU-Pro)*—MMLU-Pro is an advanced benchmark that builds on the MMLU dataset by introducing more challenging, reasoning-focused questions and expanding answer options from 4 to 10. It removes trivial and noisy questions, resulting in a dataset that better evaluates complex reasoning capabilities. Compared to MMLU, MMLU-Pro decreases model accuracy by 16% to 33% and reduces sensitivity to prompt variations, improving stability. Models using CoT reasoning perform better on MMLU-Pro, highlighting its emphasis on complex reasoning over direct answering. This benchmark provides a more effective tool for tracking advancements in AI reasoning and comprehension. See https://arxiv.org/abs/2406.01574.

Table C.2 provides a snapshot of how leading LLMs perform across key benchmarks at the time of publication. For the most accurate insights, check for updated scores on the latest versions of the models you plan to use.

Table C.2 Benchmark scores (%) of the most popular LLMs

Model	IFEval	BBH	MATH-L5	GPQA	MuSR	MMLU-Pro
GPT-4o	85.60	83.10	76.60	53.60	N/A	74.68
Gemini 2.0 Flash	N/A	86.80	89.70	62.10	N/A	76.40
Claude 3.5 Sonnet	88.00	N/A	71.1	59.40	N/A	78.00
Command R+	N/A	N/A	44.0	N/A	N/A	N/A
Grok-2	75.50	N/A	76.1	N/A	N/A	N/A
DeepSeek-V3	86.10	N/A	N/A	59.10	N/A	75.90
Qwen2.5-72B-Instruct	86.38	61.87	1.21	16.67	11.74	51.40
Qwen2.5-Coder-32B	43.63	48.51	30.59	12.86	15.87	47.81
Qwen2.5-Math-7B	24.60	22.01	30.51	5.82	5.00	19.09
Llama-3.3-70B-Instruct	89.98	56.56	0.23	10.51	15.57	48.13
Mixtral-8x22B-Instruct-v0.1	71.84	44.11	18.73	16.44	13.49	38.70

Table C.2 Benchmark scores (%) of the most popular LLMs *(continued)*

Model	IFEval	BBH	MATH-L5	GPQA	MuSR	MMLU-Pro
Mistral-7B-v0.3	22.66	23.95	3.02	5.59	8.36	21.70
Gemma-2-27B-it	79.78	49.27	0.76	16.67	9.11	38.35
Falcon 3-10B-Base	36.48	41.38	24.77	12.75	14.17	36.00
Phi-3.5-MoE-instruct	69.25	48.77	22.66	14.09	17.33	40.64

C.2.9 Safety and Bias

Ensuring the safety and fairness of LLMs is essential for ethical and responsible AI use. The focus is on preventing harmful, offensive, or biased outputs that could negatively impact individuals or reinforce stereotypes.

To improve safety, LLMs must avoid generating content that is sexist, racist, or hateful. Because these models learn from extensive internet text, which can contain harmful material, implementing robust filtering and monitoring is critical to align outputs with ethical standards.

Reducing bias is equally important. Without careful oversight, LLMs can unintentionally perpetuate stereotypes. For example, assuming doctors are male and nurses are female reinforces outdated gender biases, undermining inclusivity and fairness.

This challenge is especially important in fields such as healthcare, customer service, and education, where unbiased and accurate responses are vital. Regularly refining training data and fine-tuning models are necessary steps to ensure fairness and inclusivity.

In summary, promoting safety and reducing bias in LLMs requires continuous effort and a commitment to ethical AI practices. As you explore and implement AI solutions, prioritize testing and refinement to address potential issues, as covered in the next section.

C.2.10 A practical example

Let's integrate all the key factors and decisions involved in selecting an LLM for a specific application, using a practical example. Imagine implementing a code assistant chatbot to generate Python snippets for stock trading strategies. The target users are software developers supporting traders in a financial organization.

The code generator must use proprietary in-house trading libraries, making confidentiality a critical concern. Company policy strictly prohibits submitting any code referencing proprietary trading libraries to external systems. The assistant must generate code quickly while maintaining high accuracy to minimize bugs and avoid delays, especially during volatile trading periods. As this is for a financial company, cost isn't a limiting factor. The flowchart shown in figure C.1 illustrates the decision-making workflow for selecting an LLM for this use case.

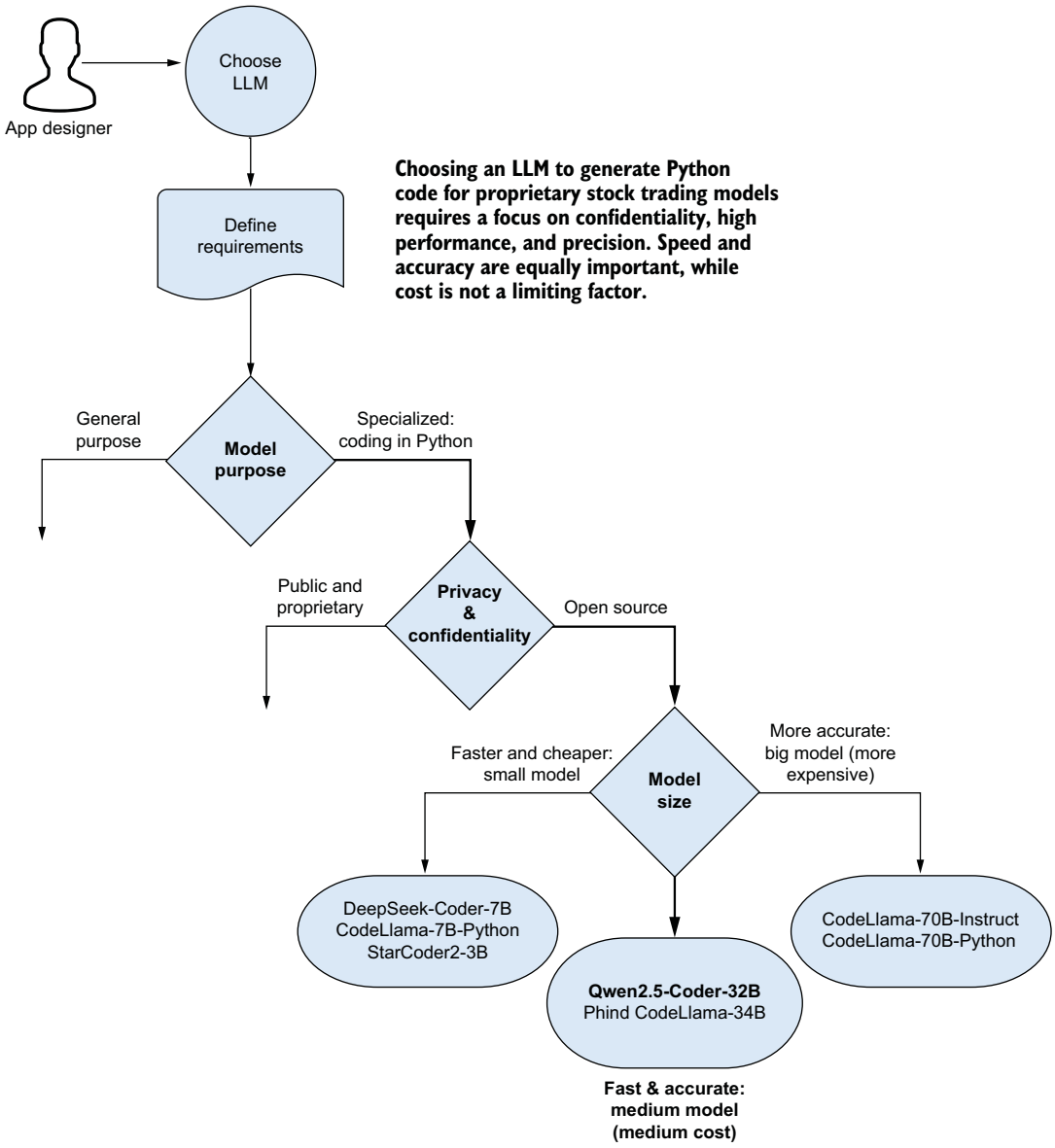


Figure C.1 Flowchart for selecting an LLM for a Python coding assistant designed to generate code for trading strategies. The process begins with choosing between a general-purpose and a specialized model, with a preference for LLMs tailored for code generation, particularly in Python. The next step focuses on selecting open source models to comply with strict IT policies requiring all proprietary code to remain confidential. The final decision balances speed and accuracy, leading to the selection of Qwen2.5-Coder-32B based on its strong performance, including a high Python HumanEval score for accurate Python code generation.

The process begins with defining requirements and progresses through the following decision points:

1. *General-purpose versus specialized models*—The focus is on specialized LLMs optimized for code generation, particularly for Python. These models are better suited to the task than general-purpose models.
2. *Privacy and confidentiality*—Given the sensitivity of the proprietary trading libraries, company policy mandates that the code can't be exposed externally. This requires selecting an open source model to ensure that data remains within the organization.
3. *Model size*—While cost isn't a concern, the model must balance accuracy and speed. A medium-sized model is ideal to achieve this tradeoff. The selection process narrows the options to a shortlist of seven LLMs, including general-purpose code generators and Python-specific models.
4. *Accuracy evaluation*—The shortlisted models are evaluated using the Python HumanEval score (refer to table C.3, sourced from the Hugging Face leaderboard at https://mng.bz/Z95A), which measures Python code generation accuracy. Qwen2.5-Coder-32B is chosen for its superior balance of accuracy and speed compared to other options.

Table C.3 Python HumanEval rankings

Model	Python HumanEval
Qwen2.5-Coder-32B-Instruct	83.20
DeepSeek-Coder-7B-Instruct	80.22
CodeLlama-70B-Instruct	75.60
Phind-CodeLlama-34B-v2	71.95
CodeLlama-70B-Python	55.49
CodeLlama-7B-Python	40.48
StarCoder2-7B	34.09

Source: Hugging Face leaderboard

Although Qwen2.5-Coder-32B has been selected as the top candidate, it's valuable to evaluate other strong alternatives, including Phind-CodeLlama-34B, DeepSeek-Coder-7B, and CodeLlama-7B-Python.

C.3 A word of caution

LLMs have revolutionized NLP, but they come with limitations and risks you need to understand. Preparing for these challenges ensures that you deploy LLMs responsibly and avoid unintended risks:

- *Bias*—LLMs can inherit and reproduce biases present in their training data, which may result in harmful stereotypes or unfair outcomes. Addressing these issues is crucial for building ethical and trustworthy AI systems. If your application involves sensitive user profiles, it's important to implement guardrails—such as content filtering, human oversight, or prompt engineering—and to evaluate multiple LLMs to identify the one that exhibits the least bias in your specific use case.
- *Privacy and security*—Proprietary LLMs may use prompt data for model improvement, which can raise privacy and confidentiality concerns. To mitigate this risk, consider deploying open source models in a private environment where you retain full control over data handling. Some providers, such as OpenAI, offer enterprise plans that guarantee prompt data won't be used for training. However, it ultimately comes down to whether you trust the provider to uphold their data privacy commitments.
- *Hallucinations*—LLMs sometimes produce incorrect or fabricated answers, known as hallucinations. Transparency and explainability are key when users question an output. Tools such as LangChain's evaluation framework help reduce hallucinations by improving response accuracy.
- *Responsibility and liability*—The legal landscape for LLMs remains unclear. Determining who is responsible for errors or harm caused by LLM outputs is complex. Providers often include disclaimers in their terms and chatbot interfaces to limit liability. When deploying an LLM, consider accountability standards and define clear usage guidelines.
- *Intellectual property (IP) rights*—LLMs trained on unrestricted data can inadvertently generate content that violates IP laws. Some models address this by training exclusively on public domain material to avoid legal issues. If your application requires strict IP compliance, verify that your chosen LLM avoids proprietary or copyrighted content.

appendix D Installing SQLite on Windows

SQLite doesn't require a full installation. Simply unzip the package, place it in a folder, and add the folder to your system's Path environment variable.

D.1 Installing SQLite

Follow these steps for Windows setup (for other operating systems, refer to the SQLite documentation):

1. Download SQLite:
   - Go to the SQLite download page at www.sqlite.org/download.html.
   - Download the latest zipped tools package, e.g., `sqlite-tools-win-x64-3460100.zip`.
2. Extract files:
   - Unzip the downloaded file to a folder, for instance, `C:\sqlite`.
   - After unzipping, you should see the files, including the SQLite executable, in `C:\sqlite\sqlite-tools-win-x64-3460100`.
3. Add SQLite to the system path:
   - Open the Start menu, go to Control Panel, and search for "edit system environment variables."
   - In System Properties, click the Environment Variables button.
   - In the System Variables section, select Path and click Edit.
   - Add `C:\sqlite\sqlite-tools-win-x64-3460100` at the end of the list, and then click OK to close all dialog boxes.

4 Verify the installation by opening a new command shell and entering `sqlite3`. If everything is set up correctly, you'll enter the SQLite prompt where you can start creating and managing databases.

With SQLite installed and configured, you can now create the database for the examples in chapters 10 and 12.

appendix E
Open source LLMs

In earlier chapters, you worked with OpenAI's public REST API. It's a straightforward way to build large language model (LLM) applications because you don't need to set up a local LLM host. After signing up with OpenAI and generating an API key, you can send requests to their endpoints and access LLM capabilities. This quick setup lets you work with state-of-the-art models efficiently. The main drawback is cost—running examples such as summarization might cost a few cents or even dollars. If you're working on projects for your company, privacy might also be a concern. Some employers block OpenAI entirely to avoid the risk of leaking sensitive or proprietary data.

This appendix introduces open source LLMs, a practical solution for reducing costs and addressing privacy concerns. These models are especially appealing to individuals and organizations that prioritize data confidentiality or are new to AI. I'll guide you through the most popular open source LLM families, their features, and the advantages they offer. The focus will be on running these models, ranging from high-performance, advanced setups to user-friendly tools that are ideal for learning and experimentation.

Finally, I'll show you how to transition the summarization and QA systems you built earlier to a local open source LLM. By the end of this appendix, you'll understand open source LLMs well and feel confident using them when they're the right choice.

E.1 Benefits of open source LLMs

Open source LLMs offer clear advantages in cost, privacy, and flexibility. They provide control over data, lower costs by avoiding licensing fees, and allow customization. Community-driven development fuels innovation, making these models competitive with proprietary ones. This section explores the key benefits of open source LLMs.

E.1.1 Transparency

Closed source models often function as black boxes, making them difficult to understand and potentially problematic for compliance, especially in industries such as healthcare and finance. Their lack of transparency limits customizability and creates challenges for auditing and accountability.

Open source LLMs, by contrast, are transparent in their architecture, training data, and methods. Organizations can inspect, validate, and modify the code to suit specific needs, ensuring compliance and fostering trust. Developers gain flexibility to extend and adapt models for unique applications, improving control and oversight.

Transparent origins enhance trust in the model's integrity, offering verifiable assurances instead of relying on blind faith. This clarity also helps address potential privacy risks. However, with transparency comes responsibility—users bear accountability for flaws and financial impacts. Open access can pose cybersecurity risks if not adequately protected.

E.1.2 Privacy

Privacy and security are critical, particularly in regulated industries (e.g., finance and healthcare). Leaks of sensitive data or Personally Identifiable Information (PII) can harm trust and reputation. Using proprietary public-cloud LLMs may raise concerns about security and intellectual property risks, especially for companies processing sensitive data.

Open source LLMs mitigate these concerns by enabling on-premises or private cloud deployment. Data stays within the corporate network, reducing exposure risks. These models can be customized to meet privacy needs, including implementing security protocols, content filtering, and data anonymization. They also support compliance with privacy laws and industry standards. As privacy regulations grow stricter, open source LLMs become increasingly appealing to organizations seeking control over their data.

E.1.3 Community driven

Proprietary LLMs are typically developed with a fixed vision by a single organization. Open source LLMs, however, benefit from contributions by a diverse community of developers, ranging from individuals to enterprises. These contributions improve features, provide flexibility, and foster growth through public forks and private customizations.

The open source community produces a wide variety of models and training methods, advancing the field rapidly. While prominent contributors influence direction, the open nature allows anyone to contribute, driving collaboration and innovation. This dynamic reduces the performance gap with proprietary models as advancements in architecture, datasets, and training methods evolve.

However, community projects can face challenges, including disputes or mismanagement, which may slow progress. Despite these risks, open source models empower smaller organizations to compete with industry leaders, leveling the playing field.

Choosing between proprietary and open source LLMs depends on your priorities. If control, customization, and community engagement are essential, open source models are ideal. If strict SLAs or turnkey solutions are more critical, proprietary models might be better suited. The decision depends on your project's requirements and goals.

E.1.4 Cost savings

Open source LLMs are typically more cost-effective than proprietary models due to the lack of licensing fees. However, deploying them on-premises may involve significant up-front capital costs, and running them in the cloud can incur ongoing operational expenses. Even with these factors, the total cost of ownership (TCO) for open source models is usually lower over the medium to long term compared to the recurring fees of proprietary LLMs. The right choice depends on your use case, expected text-processing volume, and readiness to handle initial deployment costs.

For low initial usage, a pay-as-you-go proprietary model might be more practical. As usage grows and the client base justifies investment in infrastructure, transitioning to an open source model can save money. If you already have the skills to deploy and manage an open source LLM, starting locally might be cost-effective. However, you must consider hidden costs, such as time spent by staff on setup and maintenance.

The cost-effectiveness of an LLM also depends on its application. Proprietary vendors charge based on the number of tokens processed, which can become expensive for tasks such as summarizing large amounts of text. In such cases, an open source model may reduce costs. On the other hand, for applications using efficient strategies such as Retrieval-Augmented Generation (RAG) with minimal text processing, proprietary models might be more economical.

An additional advantage of open source LLMs is the ability to fine-tune them through specialized training to create a custom model. However, this involves expenses such as consultancy fees, staff time, computational resources, and extended timelines. In cases where no proprietary model suits your specific domain, building and fine-tuning an open source LLM may be the only option, albeit at your own expense.

E.2 Popular open source LLMs

The world of open source LLMs is moving fast, making it tricky to figure out which ones stand out. To help, I've pulled together key details about popular open source LLMs into a series of tables, starting with table E.1. This table gives a straightforward overview of some of the most interesting open source LLMs available just before this book was published. Models are grouped by their type, listed under the Model Type column:

- *Foundation*—General-purpose models trained on raw data. Great for research, experimenting, or fine-tuning, but not usually ready for direct use with end users.
- *Instruct*—Models fine-tuned to follow instructions or handle question answering.
- *Code*—Models built to generate, explain, or debug code.
- *Domain-specific*—Models designed for specialized industries or business needs.

This setup makes it easier to see what each type of model is good at and how it fits your needs.

Table E.1 Most popular open source LLMs at the time of publication

Model	Developer	Hugging Face URL	Model type	Size (billion parameters)	Context window (thousand tokens)	License
DeepSeek-V3	Deep-Seek AI	https://huggingface.co/deepseek-ai/DeepSeek-V3	Foundation	671	128	MIT License
Qwen2.5-72B-Instruct	Qwen	https://huggingface.co/Qwen/Qwen2.5-72B-Instruct	Instruct	72	128	Qwen License
Qwen2.5-Coder-32B	Qwen	https://huggingface.co/Qwen/Qwen2.5-Coder-32B	Code	32	32	Apache 2.0
Qwen2.5-Math-7B	Qwen	https://huggingface.co/Qwen/Qwen2.5-Math-7B	Domain-specific (Math)	7	32	Apache 2.0
Llama-3.3-70B-Instruct	Meta Llama	https://mng.bz/642p	Instruct	70	128	Llama 3.3 Community License
Mixtral-8x22B-Instruct-v0.1	Mistral AI	https://mng.bz/oZEy	Instruct	141	64	Apache 2.0
Mistral-7B-v0.3	Mistral AI	https://huggingface.co/mistralai/Mistral-7B-v0.3	Foundation	7	32	Apache 2.0
Gemma-2-27b-it	Google	https://huggingface.co/google/gemma-2-27b-it	Foundation	27	8	Gemma License
Falcon3-10B-Base	TII-UAE	https://huggingface.co/tiiuae/Falcon3-10B-Base	Foundation	10	32	Falcon LLM License
Phi-3.5-MoE-instruct	Microsoft	https://mng.bz/nZ8V	Instruct	41.9	128	MIT License

Table E.2 shows the performance of these models based on standard benchmarks, which are fully defined in appendix C.

Table E.2 Standard performance benchmarks on the most popular open source LLMs at the time of publication

Model	IfEval (%)	BBH (%)	MATH-L5 (%)	GPQA (%)	MuSR (%)	MMLU-PRO (%)
DeepSeek-V3	86.10	N/A	N/A	59.10	N/A	75.90
Qwen2.5-72B-Instruct	86.38	61.87	1.21	16.67	11.74	51.40
Qwen2.5-Coder-32B	43.63	48.51	30.59	12.86	15.87	47.81
Qwen2.5-Math-7B	24.60	22.01	30.51	5.82	5.00	19.09
Llama-3.3-70B-Instruct	89.98	56.56	0.23	11.51	15.57	48.13
Mixtral-8x22B-Instruct-v0.1	71.84	44.11	18.73	16.44	13.49	38.70
Mistral-7B-v0.3	22.66	23.95	3.02	5.59	8.36	21.70
Gemma-2-27b-it	79.78	49.27	0.76	16.67	9.11	38.35
Falcon3-10B-Base	36.48	41.38	24.77	12.75	14.17	36.00
Phi-3.5-MoE-instruct	69.25	48.77	22.66	14.09	17.33	40.64

Source: Extract from the Hugging Face Open LLM Leaderboard (https://mng.bz/RwWv)

As shown in the table E.2, models from the same family can vary in suitability depending on the use case. For instance, if you need an LLM for an instruction-based chatbot, you should look for one with a high IFEval score. In that case, Qwen2.5-72B-Instruct is a strong option with an impressive IFEval score of 86.38. On the other hand, if your focus is solving math problems, Qwen2.5-Math-7B might be a better choice due to its high MATH score of 30.51, combined with the advantage of being smaller in size.

Now that you know about popular open source LLMs and their features, I'll show you how to run these models on your own computer. This will let you experiment and explore their capabilities firsthand.

E.3 Considerations on running open source LLMs locally

LLMs operate in two main phases: training and inference. During training, the model learns patterns from the training set, adjusting its weights (parameters). These weights are then used during inference to make predictions or respond to new inputs. Open source LLM weights are usually easy to access, often available on platforms such as Hugging Face or through tools such as LM Studio or Ollama. Running the inference phase locally requires suitable hardware. This section covers hardware requirements and how modern techniques such as quantization make it feasible to run

models even on consumer-grade machines. An important consideration when hosting an LLM locally is whether it provides an OpenAI-compatible API, such as a REST API. Using OpenAI during the proof of concept (PoC) phase and transitioning to an open source LLM later can reduce costs or address privacy concerns. If the inference engine supports an OpenAI-compatible API, you can switch seamlessly to production without rewriting or retesting code.

E.3.1 Limitations of consumer hardware

Hosting a large LLM in production demands powerful hardware. For example, Llama-70B requires 140 GB of RAM and disk space when loaded in single-precision (float16). High-end GPUs, such as NVIDIA's A100 (priced at $10,000 or more), are essential for efficient inference. Cloud options are available, but they are expensive, costing about $1.50 per hour for A100 rentals.

For local experimentation, however, consumer hardware can still be viable. Techniques such as quantization and specialized inference engines reduce the hardware burden, enabling LLMs to run on laptops or desktops with as little as 16 GB or 32 GB of RAM and no dedicated GPU. These methods make it possible to explore LLM functionality on modest machines.

E.3.2 Quantization

Quantization reduces LLM size by lowering the precision of weights, trading some accuracy for faster and more efficient inference. Large models typically use 16-bit or 32-bit floating-point precision (2–4 bytes per parameter). Quantization compresses this to 4-bit integer precision (0.5 bytes per parameter). Quantization offers a balance between size, speed, and accuracy. For example, Llama 2 7B, originally 13.5 GB with 16-bit precision, can be reduced to 3.9 GB with 4-bit quantization, allowing it to run on modest laptops. The most common quantization techniques used are as follows:

- *Post-Training Quantization (PTQ)*—Applied after training; simpler but may reduce accuracy.
- *Quantization-Aware Training (QAT)*—Integrated during training for better accuracy but more complex.
- *Generalized Post-Training Quantization (GPTQ)*—Combines GPT with PTQ techniques.
- *Normal Float 4 (NF4)*—Uses 4-bit normal float precision, often outperforming standard 4-bit integer quantization.
- *Georgi Gerganov Machine Learning (GGML) and GPT-Generated Unified Format (GGUF)*—Tools by Georgi Gerganov for running models on CPUs. GGUF, an enhanced version, supports a wider range of open source models.

Some quantization tools, such as `bitsandbytes` (by Tim Dettmers), are available through Python's `pip install` and are well-documented on Hugging Face. Quantized models can be found on Hugging Face by searching terms like "GGML" or "GGUF," or directly through inference engines such as Ollama or LM Studio.

TIP For a deeper dive into quantization, check out "How Is llama.cpp Possible?" by Finbarr Timbers (https://finbarr.ca/how-is-llama-cpp-possible).

Modern advancements in quantization and optimized engines have significantly lowered the barrier to running open source LLMs locally, making them accessible even for users with limited hardware, especially for learning and experimentation.

E.3.3 OpenAI REST API compatibility

Many inference engines include an embedded HTTP server that accepts requests through endpoints compatible with the OpenAI REST API. This compatibility allows immediate use with standard OpenAI libraries, simplifying the engine's codebase and supporting quick adoption. For users, the benefits include minimal learning requirements compared to learning proprietary APIs and the ability to swap engines seamlessly. With this standardization, you can efficiently test multiple engines using the same client code or replace an engine with one better suited to your needs.

In this section, I'll provide code snippets that you can run against each inference engine I'll introduce later. These snippets require minimal modifications—often just adjusting the port number. They demonstrate how to make requests using raw HTTP (e.g., `curl`), the OpenAI Python library, and LangChain. Let's start with direct HTTP requests.

Direct curl invocation

The easiest way to test an OpenAI-compatible REST API endpoint is by using the `/chat/completions` endpoint. Following is an example with OpenAI's public service. Open a shell (on Windows you can open a command-line window or a Git Bash shell), and replace `YOUR-OPENAI-KEY` with your actual OpenAI API key:

```
$ curl https://api.openai.com/v1/chat/completions
➥-H "Content-Type: application/json"
➥-H "Authorization: Bearer YOUR-OPENAI-KEY"
➥-d '{
    "model": "gpt-4o-mini",
    "messages": [
    { "role": "system", "content": "You are a helpful AI assistant." },
    { "role": "user", "content": "How many Greek temples are in Paestum?" }
    ],
    "temperature": 0.7
  }'
```

The response might look like this:

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
➥Current
                                 Dload  Upload   Total   Spent    Left
➥Speed
100   810  100   578  100   232    209     83  0:00:02  0:00:02 --:--:--
➥293
{
    "id": "chatcmpl-AjANT94Lel2oZIOz8kmF0PKWfRuGa",
```

```
    "object": "chat.completion",
    "created": 1735328015,
    "model": "gpt-4o-mini-2024-07-18",
    "choices": [
        {
            "index": 0,
            "message": {
                "role": "assistant",
                "content": "Paestum, an ancient Greek city located in
➥southern Italy, is known for its well-preserved Greek temples. There
➥are three major temples in Paestum:\n\n1. **Temple of Hera (Basilica)**
➥- This is one of the oldest temples in the area, dating back to around
➥550 BC.\n2. **Temple of Neptune (or Poseidon)** - This temple was built
➥around 460 BC and is notable for its impressive Doric architecture.\n3.
➥**Temple of Ceres** - This temple, which dates to around 500 BC, is
➥smaller and dedicated to the goddess of agriculture.\n\nIn addition to
➥these temples, there are also remnants of other structures and smaller
➥sanctuaries, but the three mentioned are the most significant and well-
➥preserved Greek temples in Paestum.",
                "refusal": null
            },
            "logprobs": null,
            "finish_reason": "stop"
        }
    ],
    "usage": {
        "prompt_tokens": 28,
        "completion_tokens": 163,
        "total_tokens": 191,
        "prompt_tokens_details": {
            "cached_tokens": 0,
            "audio_tokens": 0
        },
        "completion_tokens_details": {
            "reasoning_tokens": 0,
            "audio_tokens": 0,
            "accepted_prediction_tokens": 0,
            "rejected_prediction_tokens": 0
        }
    },
    "system_fingerprint": "fp_0aa8d3e20b"
}
```

To make the same request to a local open source LLM, adjust the command as follows:

```
curl https://localhost:8000/v1/chat/completions
➥-H "Content-Type: application/json"
➥-d '{
    "model": "mistralai/Mistral-7B-v0.3",
    "messages": [
    { "role": "system", "content": "You are a helpful AI assistant." },
    { "role": "user", "content": "How many Greek temples are in Paestum?" }
```

```
    ],
    "temperature": 0.7
  }'
```

For local inference engines, you don't need to include an OpenAI key in the request header, so it has been omitted in the example. However, you need to adjust two key details in the `curl` request based on the inference engine and LLM you're using:

- *Port number*—Each inference engine's local HTTP server is set to a default port, which is often different from 8000. Refer to the engine's documentation, and update the port number as needed.
- *Model name*—Inference engines use specific naming conventions for models. Some follow Hugging Face conventions (e.g., mistralai/Mistral-7B-v0.3), while others may not. Check the engine's documentation to ensure you provide the correct model name.

PYTHON OPENAI LIBRARY

To use the OpenAI library for requests, create a virtual environment, install the library with `pip install openai`, create a Jupyter Notebook, as you've seen in the previous chapters, and use the code from the following listing (after replacing `YOUR-OPENAI-KEY` with your actual OpenAI key).

Listing E.1 Calling the OpenAI completions endpoint

```
import getpass
from openai import OpenAI

OPENAI_API_KEY = getpass.getpass('Enter your OPENAI_API_KEY')
client = OpenAI(
    api_key = OPENAI_API_KEY
)

completion = client.chat.completions.create(
  model="gpt-4o-mini",
  messages=[
    { "role": "system",
      "content": "You are a helpful AI assistant." },
    { "role": "user",
      "content": "How many Greek temples are there in Paestum?" }
  ],
  temperature=0.7
)

print(completion.choices[0].message.content)
```

You'll get output similar to this:

```
Paestum, an ancient Greek city located in present-day Italy, is renowned
➥for its well-preserved Greek temples. There are three major temples in
➥Paestum […SHORTENED…]:
```

To run the preceding code against a local open source LLM, you need to adapt it as follows (note the extra `base_url` parameter when instantiating the OpenAI client):

```
from openai import OpenAI

port_number = '8080'

client = OpenAI(
    base_url=f'http://localhost:{port_number}/v1',
    api_key = "NO_KEY_NEEDED"
)

completion = client.chat.completions.create(
  model="mistral",
  messages=[
    { "role": "system",
      "content": "You are a helpful AI assistant." },
    { "role": "user", "content":
      "How many Greek temples are there in Paestum?" }
  ],
  temperature=0.7
)

print(completion.choices[0].message.content)
```

Run this code after starting up a local inference engine (e.g., Ollama or LM Studio) on port 8080.

To adapt the example for your inference engine and local open source LLM, update the port number to match the engine's local HTTP server, and set the model name according to the engine's specific naming convention. For example, if using LM Studio, use port 8080.

LangChain's OpenAI wrapper

For LangChain, direct its OpenAI wrapper to the local inference engine by updating the base URL. Ensure the port matches the local engine configuration:

```
from langchain_openai import ChatOpenAI

port_number = '8080'

llm = ChatOpenAI(openai_api_base=f'http://localhost:{port_number}/v1')
response = llm.invoke("How many Greek temples are there in Paestum?")

print(response.content)
```

Now let's explore the essential component required to execute open source models on regular consumer hardware: local inference engines.

E.4 Local inference engines

Running an open source LLM on consumer hardware is most practical with an inference engine. These engines host the local model and handle requests from client applications, often through native bindings for languages such as Python, JavaScript,

Java, C++, Go, or Rust. Many inference engines also include a local HTTP server with OpenAI-compatible REST API endpoints, allowing you to use familiar OpenAI libraries or frameworks such as LangChain without significant changes. This flexibility lets you switch between local open source LLMs and OpenAI's public service with minimal effort.

In the following sections, I'll introduce several inference engines, beginning with foundational tools such as llama.cpp, optimized for high performance, and moving to user-friendly options such as Ollama and production-grade solutions such as vLLM. I'll also cover consumer-focused alternatives, including LocalAI (a simplified wrapper for engines such as llama.cpp or vLLM), GPT4All, and LM Studio, which are notable for their intuitive user interfaces. These options will help you select the best engine based on your hardware, experience, and project requirements.

As shown in figure E.1, which serves as a guide for this section, llama.cpp and vLLM are foundational backends for many inference engines. Higher-level tools such as Ollama, llamafile, and LocalAI, along with user-friendly engines such as GPT4All and LM Studio, are built on llama.cpp. Meanwhile, vLLM functions independently, with LocalAI being the only tool currently building on it. Let's start our tour with llama.cpp.

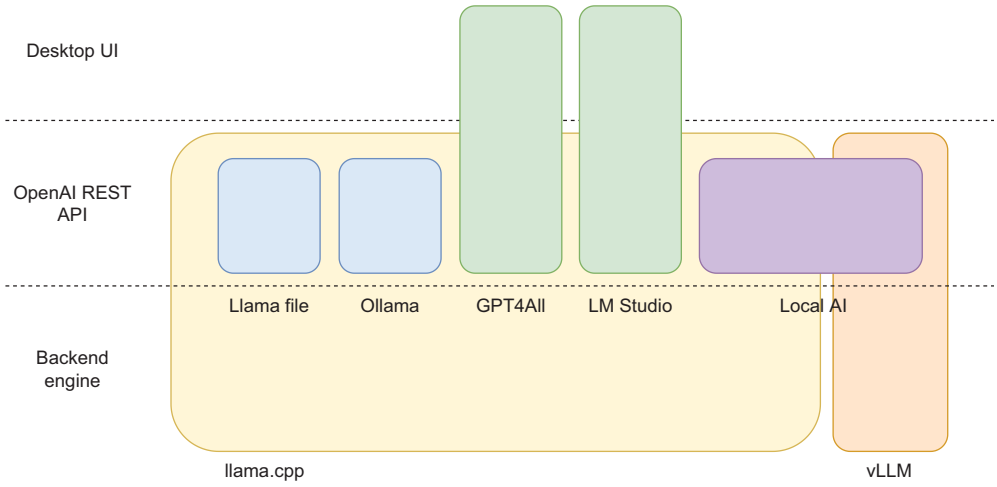


Figure E.1 Lineage and functionality of local inference engines

E.4.1 llama.cpp

The *llama.cpp* engine was one of the first designed to run open source models efficiently on consumer hardware. Initially developed to support the Llama model using 4-bit integer quantization for Apple silicon GPUs, it has since expanded to support Linux, Windows (x86 processors), and even Raspberry Pi. It now handles a wide range

of quantized models, including Mistral, Gemma, Phi, and Falcon. It supports 2-bit to 8-bit integer quantization and offers bindings for Python, Java, C#, Go, Scala, and Ruby.

SETTING UP LLAMA.CPP

To use llama.cpp, follow these steps:

1. *Build the executable.* Download the source code from GitHub, and build it using your preferred strategy, such as `make`, `CMake`, `Zig`, or `gmake`. Advanced build options include Metal, MPI, and BLAS for enhanced performance.
2. *Prepare the model weights.* Obtain a quantized version of the model (e.g., `Mistral-7B-Instruct-v0.2-GGUF`) from Hugging Face, or generate it using GitHub instructions. Ensure the model fits within your system's disk and RAM capacity.
3. *Run inference.* Execute the inference command, pointing to the quantized model file:

```
./main -m ./models/mistral-7b-instruct-v0.2.Q2_K.gguf
➥-p "How many Greek temples are there in Paestum?" -n 512
```

NOTE For detailed instructions, refer to the official GitHub page: https://github.com/ggerganov/llama.cpp.

PYTHON BINDINGS

If you prefer a higher-level API, links to relevant bindings are available on the llama.cpp GitHub page. Before installation, review both the llama.cpp documentation and the selected bindings' documentation to prevent redundant setup.

To install the Python bindings from PyPI, use the following command:

```
pip install llama-cpp-python
```

This command not only downloads the library but also tries to build llama.cpp from source using CMake and your system's C compiler. For GPU support, follow additional instructions and configurations.

After setup, you can use Python to interact with your local quantized LLM instance. The following listing shows an adapted example from the official `llama-cpp-python` bindings documentation.

Listing E.2 Running prompts on a local quantized Mistral model

```
### Adapted from official documentation at
### https://github.com/abetlen/llama-cpp-python
from llama_cpp import Llama
llm = Llama(
      model_path="./models/mistral-7b-instruct-v0.2.Q2_K.gguf"
)
output = llm(
      """Q: What are the planets
      in the solar system? A: """,   <-- Prompt
      max_tokens=32,                 <-- Generates up to 32 tokens, set to None to generate up to the end of the context window
```

```
      stop=["Q:", "\n"],
      echo=True
)
print(output)
```

Echoes the prompt back in the output

Stops generating just before the model would generate a new question

You get the following output:

```
{
  "id": "cmpl-xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
  "object": "text_completion",
  "created": 1679561337,
  "model": "./models/mistral-7b-instruct-v0.2.Q2_K.gguf",
  "choices": [
    {
      "text": "Q: Name the planets in the solar system. A: Mercury, Venus,
      ➥Earth, Mars, Jupiter, Saturn, Uranus, Neptune and Pluto.",
      "index": 0,
      "logprobs": None,
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 14,
    "completion_tokens": 28,
    "total_tokens": 42
  }
}
```

LangChain integration

LangChain's `LlamaCpp` client can connect directly to a quantized local instance of Llama-based models, provided you've installed the `llama-cpp-python` library (https://github.com/abetlen/llama-cpp-python):

```
from langchain_community.llms import LlamaCpp
llm = LlamaCpp(model_path="./models/llama-2-7b-chat.ggmlv3.q2_K.bin")
```

OpenAI-compatible REST API endpoints

llama.cpp includes a local HTTP server that provides OpenAI-compatible REST API endpoints. To try it out, you can use the `curl` script or Python examples from section E.3.3, ensuring the port number and model name are correctly configured.

While detailed installation and setup instructions for llama.cpp aren't included here, it's important to understand its basics. Many of the local inference engines discussed later are built on llama.cpp or draw inspiration from its design. Next, I'll introduce Ollama.

E.4.2 Ollama

Ollama is a hosting environment for open source LLMs, available for Windows, macOS, and Linux. Unlike llama.cpp, Ollama doesn't require you to build executables from source; installation packages are readily available.

INTERACTIVE MODE

After downloading and installing Ollama from the Ollama.ai homepage (https://ollama.ai), launch the Ollama client (in Windows, launch it from the Start menu). This opens a terminal (or PowerShell window on Windows) where you can immediately run an LLM using the following command:

```
ollama run mistral
```

This command downloads a quantized version of the selected model (Mistral in this example) from the remote Ollama library. The terminal will display the download progress, as shown in figure E.2.

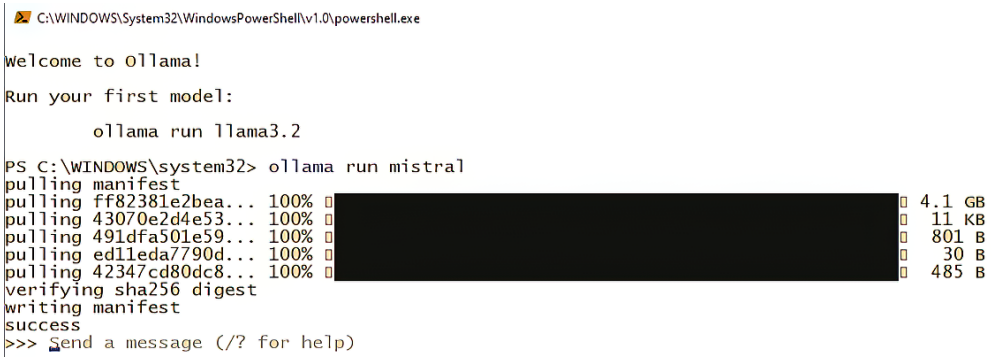


Figure E.2 The Ollama terminal while installing the Mistral LLM

At this stage, you can send a prompt and receive a response, as shown here (note the response may be slow):

```
>>> How many Greek temples are in Paestum?
 There are three Greek temples in Paestum, which is a town on the coast of
➥southern Italy. The three temples are:

1. Temple of Hera I (also known as the Basilica) - This temple was built
➥around 550 BC and is dedicated to the goddess Hera. It is the largest
➥and most impressive of the three temples in Paestum.
2. Temple of Neptune - This temple was also built around 550 BC and is
➥dedicated to the god Poseidon (known as Neptune in Roman mythology). It
➥is smaller than the Temple of Hera I, but it is still an impressive
➥structure.
3. Temple of Athena - This temple was built around 460 BC and is dedicated
➥to the goddess Athena. It is the smallest of the three temples and is
➥incomplete due to damage sustained during an earthquake.

All three temples are well-preserved and are a popular tourist destination.
➥They are also a UNESCO World Heritage
Site.
```

Ollama natively supports several popular open source LLMs, including Llama, Phi, and Gemma. You can browse the available models on the library page: https://ollama.com/library. To run a GGUF model from another source, such as Hugging Face, copy the model to a local folder, and import it following the instructions in the Ollama documentation.

SERVER MODE

Once the server is running, it exposes a REST API on port 11434, ready to accept requests for LLM interactions.

> **NOTE** Ollama's REST API endpoints are proprietary and not compatible with OpenAI's specifications. Code examples from earlier sections can't be used directly.

Following is an example using Ollama's API from `curl` (or Postman, if you prefer):

```
curl http://localhost:11434/api/generate -d '{
  "model": "mistral",
  "prompt":"How many Greek temples are in Paestum?"
}'
```

By default, the output streams one word at a time, each included in a separate JSON object, as shown here:

```
{
    "model": "mistral",
    "created_at": "2024-12-28T11:52:26.1375731Z",
    "response": " There",
    "done": false
}
{
    "model": "mistral",
    "created_at": "2024-12-28T11:52:26.5276658Z",
    "response": " are",
    "done": false
}
{
    "model": "mistral",
    "created_at": "2024-12-28T11:52:26.9259504Z",
    "response": " three",
    "done": false
}
…
```

If you prefer to wait for the full response, you must set the `stream` attribute to `false`:

```
curl http://localhost:11434/api/generate -d '{
  "model": "mistral",
  "prompt":"How many Greek temples are in Paestum?",
  "stream":false
}'
```

You'll get a single response object:

```
{
    "model": "mistral",
    "created_at": "2024-12-28T13:07:03.1862002Z",
    "response": " There are three Doric temples in Paestum, located in the
    ➥modern region of Campania, Southern Italy. These well-preserved
    ➥temples, dating back to the 6th century BC, were built by the ancient
    ➥Greeks and are among the best-known examples of Magna Graecia (Great
    ➥Greece) architecture. The three temples are:\n\n1. Temple of Hera I
    ➥ (Basilica Paestana or Temple A)\n2. Temple of Neptune (Temple B)\n3.
    ➥Temple of Ceres (Temple C)",
    "done": true,
    "done_reason": "stop",
    "context": [
        3,
        29473,
       … SHORTENED,
        1102,
        29499
    ],
    "total_duration": 40777816100,
    "load_duration": 13459239600,
    "prompt_eval_count": 16,
    "prompt_eval_duration": 1454000000,
    "eval_count": 123,
    "eval_duration": 25791000000
}
```

This is a request to the `/chat` endpoint, using a payload format compatible with OpenAI's corresponding request structure:

```
curl http://localhost:11434/api/chat -d '{
  "model": "mistral",
  "messages": [
    { "role": "system", "content": "You are a helpful AI assistant" },
    { "role": "user", "content": "How many Greek temples are in Paestum?" }
  ],
  "temperature":0.7
}'
```

These API calls allow interaction with the LLM using any programming language. However, for Python or JavaScript, you can use libraries provided by Ollama that wrap the low-level REST API, simplifying client development.

NATIVE PYTHON LIBRARY

To get started, install the Ollama Python library:

```
pip install ollama
```

You can then interact with the LLM's `/chat` endpoint as shown here:

```
import ollama
response = ollama.chat(model='mistral', messages=[
    { "role": "system", "content": "You are a helpful AI assistant" },
```

```
    { "role": "user", "content": "How many Greek temples are in Paestum?" }
])
print(response['message']['content'])
```

Alternatively, you can send instructions to the `/generate` endpoint:

```
ollama.generate(model='mistral',
    prompt='How many Greek temples are in Paestum?')
```

To enable streaming responses, set `stream=True` in the `chat`, or `generate` call. If you need to specify a custom `host` or `timeout`, create a `Client` object:

```
from ollama import Client
client = Client(host='http://mylocalserver:8000')
response = client.chat(model='mistral', messages=[
    { "role": "system", "content": "You are a helpful AI assistant" },
    { "role": "user", "content": "How many Greek temples are in Paestum?" }
])
```

For asynchronous execution, use `AsyncClient` with `asyncio`:

```
import asyncio
from ollama import AsyncClient

async def chat():
    message = {'role': 'user',
        'content': 'How many Greek temples are in Paestum?'}
    response = await AsyncClient().chat(model='mistral', messages=[message])
    print(response['message']['content'])

asyncio.run(chat())
```

LangChain integration

LangChain provides a wrapper for the Ollama Python library, integrating it with LangChain interfaces such as the `Runnable` interface. Following is an example of re-implementing a basic synchronous LLM invocation using LangChain:

```
from langchain_community.llms import Ollama
ollama = Ollama(model='mistral')

query = 'How many Greek temples are in Paestum?'
response  = llm.invoke(query)
print(response)
```

To handle streaming responses, you can use the following approach:

```
for chunks in llm.stream(query):
    print(chunks)
```

For more information, I recommend consulting the Ollama documentation on GitHub. It includes community-driven integrations, such as web and desktop UIs, terminal plugins for Emacs and Vim, and other useful extensions.

Ollama is designed to help you explore open source models locally. If you decide to move beyond a proof of concept on a Linux system and build a production-grade

solution using nonquantized LLMs on powerful GPU hardware, potentially in the cloud, consider exploring vLLM. While vLLM isn't specifically for consumer-grade hardware, it demonstrates the capabilities of inference engines designed to host open source models of any size on various types of hardware.

E.4.3 vLLM

vLLM is a high-performance Python library for LLM inference, built on a C++ core and CUDA binaries. It targets Linux systems and high-grade GPUs, including V100, T4, RTX20xx, A100, L4, and H100. According to the official web page (https://docs.vllm.ai), vLLM offers the following advanced performance features:

- *High-throughput serving*—State-of-the-art inference speed
- *PagedAttention*—Efficient memory management for attention key and value data.
- *Continuous batching*—Dynamically batches incoming requests for better efficiency
- *CUDA/HIP graph execution*—Speeds up model execution
- *Quantization support*—Includes GPTQ, Activation-aware Weight Quantization (AWQ), and SqueezeLLM methods

vLLM is designed with flexibility and usability in mind. Key features include the following:

- *Hugging Face integration*—Seamless support for popular Hugging Face models
- *Advanced decoding algorithms*—Supports parallel sampling, beam search, and other methods
- *Tensor parallelism*—Enables distributed inference across multiple GPUs
- *Streaming outputs*—Provides real-time response generation
- *OpenAI-compatible API*—Allows easy integration with existing applications
- *GPU support*—Works with both NVIDIA and AMD GPUs

vLLM supports a variety of architectures, including BLOOM, Falcon, GPT-J, GPT-NeoX, Vicuna, Llama, and Mistral. While it can handle quantized models, vLLM is optimized for larger model versions, such as LLama-2-70B-hf and Falcon-180B. For a detailed list of supported models, refer to the Supported Models section of the vLLM documentation.

INSTALLATION

To install vLLM, ensure you're using Linux with one of the recommended high-grade GPUs. It's best to use a dedicated virtual environment created with `venv`. Install the library with

```
pip install vllm
```

Once installed, you can activate the server mode. Start a local OpenAI-compatible HTTP server with the following command in a separate terminal:

```
$ python -m vllm.entrypoints.openai.api_server
➥--model mistralai/Mistral-7B-v0.1
```

This will download the model from Hugging Face (if not already downloaded) and activate an HTTP server on port 8000, exposing OpenAI-compatible endpoints.

OpenAI-compatible REST API endpoints

Refer to section E.3.3 for examples of `curl`, Python, and LangChain code that you can use with vLLM. Ensure you set the port to 8000 and use the correct model name. Consult the vLLM documentation for further details.

Offline batched inference

vLLM offers a unique capability for high-performance offline batched inference, allowing you to process multiple requests in a single operation. This feature sets it apart from other local inference engines. To use it, import the `LLM` and `SamplingParams` modules, create a list of prompts, and process them using an LLM instance. The results can be iterated and reviewed, as shown in the following listing.

Listing E.3 vLLM offline batched inference

```
from vllm import LLM, SamplingParams

prompts = [                                          <-- Defines prompts
    "How many temples are there in Paestum?",
    "Who built the aqueduct in Segovia?",
    "Is LeBron James better than Michael Jordan?",
    "Summarize the Lord of The Rings in three sentences.",
]
sampling_params = SamplingParams(                    <-- Configures sampling
    temperature=0.7, top_p=0.95)                         parameters

llm = LLM(model="mistralai/Mistral-7B-v0.1")         <-- Initializes the LLM

outputs = llm.generate(prompts, sampling_params)     <-- Generates responses

for output in outputs:                               <-- Prints the outputs
    prompt = output.prompt
    llm_response = output.outputs[0].text
    print(f"Prompt: {prompt!r}, LLM response: {llm_response!r}")
```

After presenting the advanced capabilities of vLLM, designed for deploying open source LLMs in professional production environments, let's return to exploring local inference engines built for consumer-grade hardware.

E.4.4 llamafile

One of the simplest ways to run an LLM locally is with *llamafile*. According to its GitHub description, llamafile allows you to "distribute and run an LLM with a single file." This executable combines the quantized weights of an LLM in GGUF format with the llama.cpp C++ executable, packaged using `Cosmopolitan-Libc`. `Cosmopolitan-Libc`, described as "making C a build-once, run-anywhere language," enables the llamafile to run on macOS, Linux, and Windows (with a `.exe` extension) without requiring any installation. This approach sets an incredibly low barrier for experimenting

with local LLMs. The llamafile GitHub page offers a variety of prebuilt LLMs, including recent versions of Llama, Gemma, and Phi.

SERVER MODE

Running an LLM with llamafile is straightforward. Follow these steps (refer to the GitHub project for detailed instructions):

1. *Download a llamafile.* Obtain a prebuilt file, such as `mistral-7b-instruct-v0.2.Q5_K_M.llamafile`, from the GitHub project page (https://github.com/Mozilla-Ocho/llamafile).
2. *Place the file in a folder.* Copy the llamafile to a directory on your system.
3. *Set permissions as follows*:
   - On macOS or Linux, grant execute permissions using

     ```
     chmod +x mistral-7b-instruct-v0.2.Q5_K_M.llamafile
     ```

   - On Windows, rename the file to include the `.exe` extension:

     ```
     mistral-7b-instruct-v0.2.Q5_K_M.llamafile.exe
     ```

4. *Run the file.* Launch the llamafile. For example, on Windows, use the following:

   ```
   c:\temp>mistral-7b-instruct-v0.2.Q5_K_M.llamafile.exe -ngl 9999
   ```

The llamafile will start a web server at `http://127.0.0.1:8080` and open a browser pointing to a chat web interface at the same address. You can begin interacting with the LLM immediately, as shown in figure E.3. For example, you can enter the same prompt you entered into Ollama previously:

```
>>> How many Greek temples are in Paestum?
```

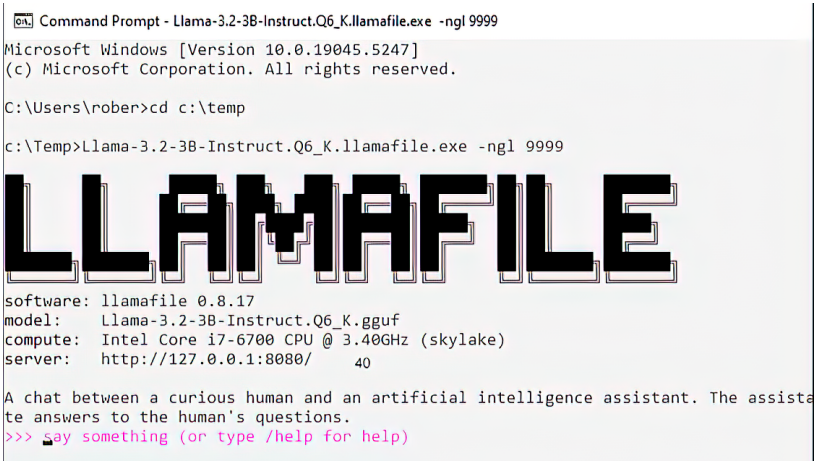


Figure E.3 The llamafile chat web UI pointing to the local web server communicating with the local open source LLM

OpenAI API-compatible endpoints

The web server also includes an OpenAI API-compatible `/chat/completions` endpoint. Use the examples from section E.3.3 for `curl`, Python, or LangChain, ensuring you set the port to `8080`.

> **TIP** To learn more about llamafile, a project by Justine Tunney in collaboration with Mozilla, visit the GitHub page. For additional insights, check out Justine's blog post "Bash One-Liners for LLMs" (https://justine.lol/oneliners/).

E.4.5 LM Studio

LM Studio is a user-focused local inference engine and a direct competitor to GPT4All. Executables for macOS, Windows, and Linux can be downloaded from the LM Studio website (https://lmstudio.ai). Once launched, LM Studio provides a seamless experience for searching, selecting, and downloading GGUF models, thanks to its integration with Hugging Face.

Figure E.4 shows the model search interface. Here, (1) search for a model using the search box (in our case, Mistral), (2) select a model variant from the left panel,

Figure E.4
GGUF search screen

and download a specific quantized version from the right panel (in this case, we select a 2-bit quantization, which has very low fidelity) by clicking the Download button (not shown). After downloading a model, you can either chat directly through the chat screen or activate the backend server for programmatic interaction.

CHAT SCREEN

To send interactive prompts, open the Chat menu on the left bar, select the downloaded model from the dropdown at the top, and type your prompt in the text box at the bottom, as shown in figure E.5.

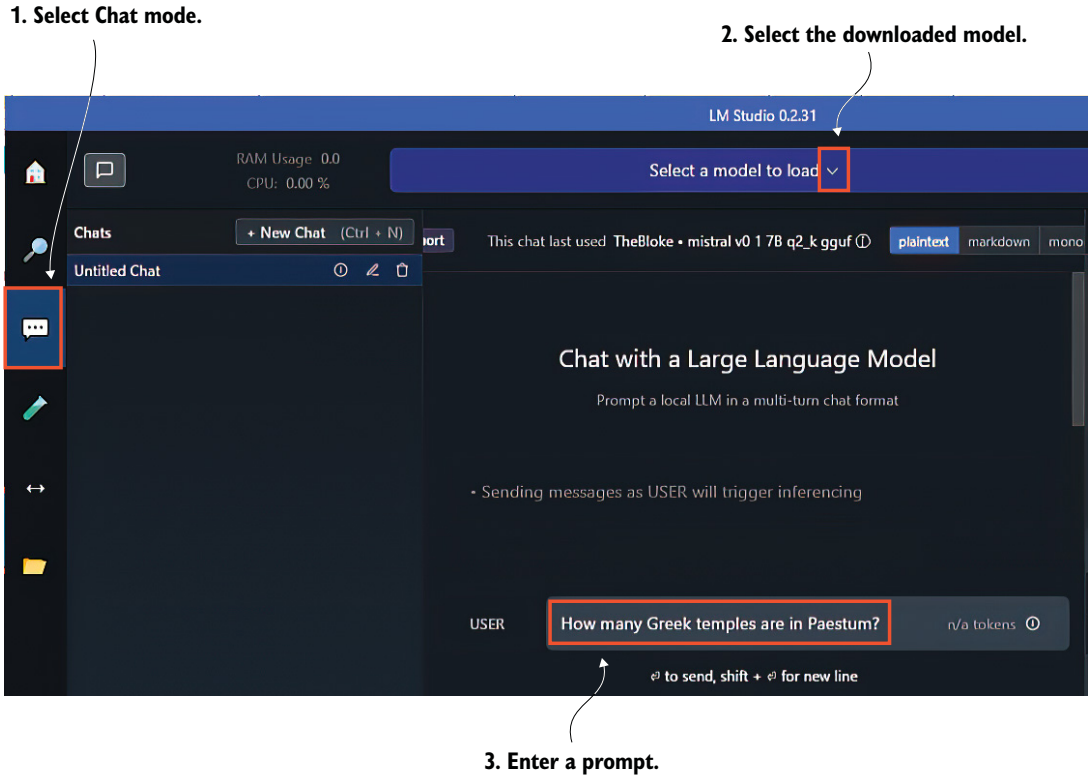


Figure E.5 Chat screen

The response time will vary based on the model, its quantization level, and your computer's hardware. Expect a delay of a few seconds before the LLM provides an answer, as shown in figure E.6.

Once you've explored the UI, it's time to examine the server API mode. We'll do that next.

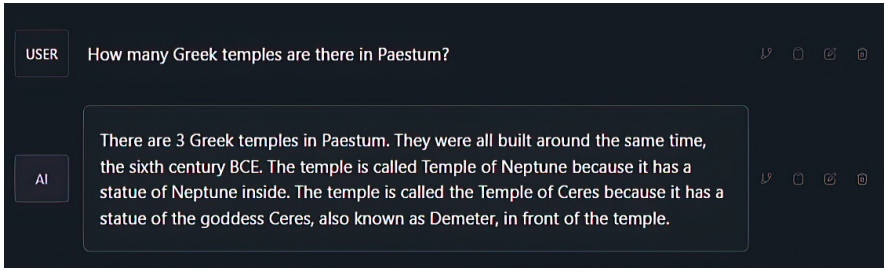


Figure E.6 The speed of the prompt response depends on the model, its quantization, and your computer's hardware.

SERVER MODE

To activate the local HTTP server and expose OpenAI-compatible REST API endpoints, follow these steps:

1. Click the Server icon in the left-hand menu.
2. Set the desired port number (e.g., 8080).
3. Click the Start Server button, as shown in figure E.7.

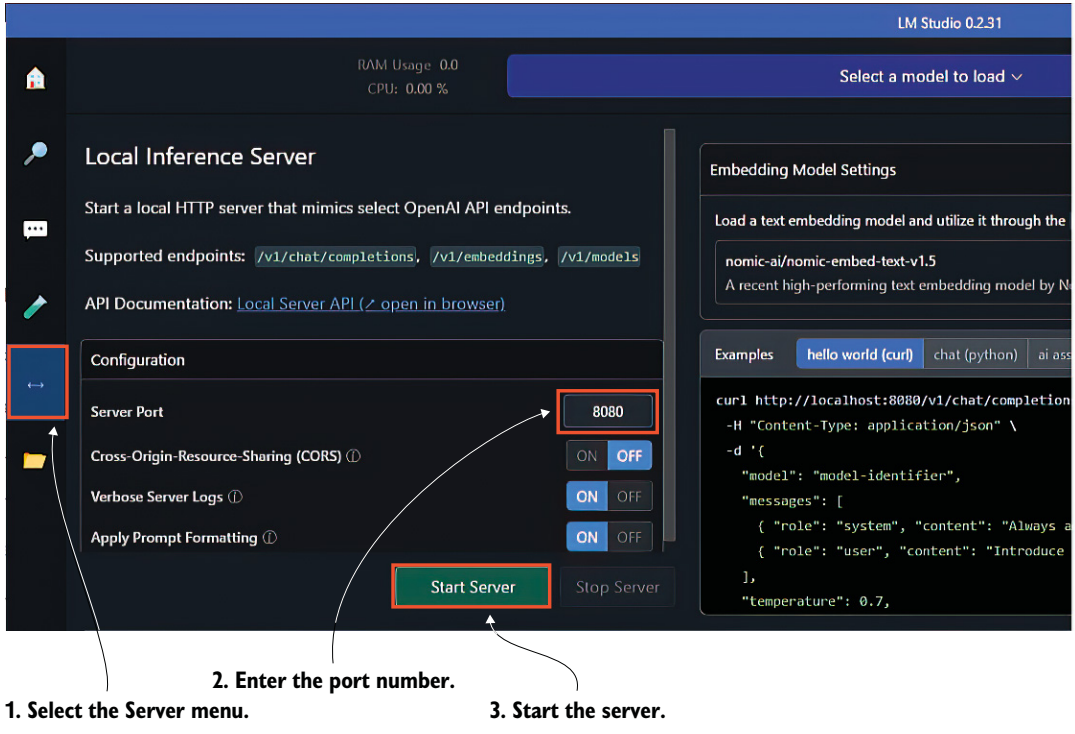


Figure E.7 Activating the local HTTP server

After starting the server, the logs will appear in the terminal panel, confirming the server is running. Example logs include the following:

```
[2024-12-30 13:42:41.709] [INFO] [LM STUDIO SERVER] Verbose server logs are
➥ ENABLED
[2024-12-30 13:42:41.721] [INFO] [LM STUDIO SERVER] Success! HTTP server
➥listening on port 8080
[2024-12-30 13:42:41.721] [INFO] [LM STUDIO SERVER] Supported endpoints:
[2024-12-30 13:42:41.721] [INFO] [LM STUDIO SERVER] ->    GET
➥http://localhost:8080/v1/models
[2024-12-30 13:42:41.722] [INFO] [LM STUDIO SERVER] ->    POST
➥http://localhost:8080/v1/chat/completions
[2024-12-30 13:42:41.722] [INFO] [LM STUDIO SERVER] ->    POST
➥http://localhost:8080/v1/completions
[2024-12-30 13:42:41.722] [INFO] [LM STUDIO SERVER] ->    POST
➥http://localhost:8080/v1/embeddings     <------------ NEW!
[2024-12-30 13:42:41.723] [INFO] [LM STUDIO SERVER] Model loaded:
➥ TheBloke/Mistral-7B-v0.1-GGUF/mistral-7b-v0.1.Q2_K.gguf
[2024-12-30 13:42:41.723] [INFO] [LM STUDIO SERVER] Logs are saved into
➥ C:\tmp\lmstudio-server-log.txt
```

To test the server, copy the sample `curl` request provided in the left-hand panel, and paste it into a terminal (on Windows, use Git Bash for easier handling of escaping), or alternatively run on Postman against the specified URL. As the terminal begins processing the request, the LM Studio logs panel will display activity, providing real-time updates during the request's execution:

```
[2024-12-30 13:46:12.782] [INFO] Received POST request to
➥/v1/chat/completions with body: {
  "model": "TheBloke/Mistral-7B-v0.1-GGUF",
  "messages": [
    {
      "role": "system",
      "content": "Always answer in rhymes."
    },
    {
      "role": "user",
      "content": "Introduce yourself."
    }
  ],
  "temperature": 0.7,
  "max_tokens": -1,
  "stream": true
}
[2024-12-30 13:46:12.783] [INFO] [LM STUDIO SERVER] Context Overflow Policy
➥ is: Rolling Window
[2024-12-30 13:46:12.783] [INFO] [LM STUDIO SERVER] Streaming response...
[2024-12-30 13:46:20.630] [INFO] [LM STUDIO SERVER] First token generated.
➥Continuing to stream response..
[2024-12-30 13:46:52.409] [INFO] Finished streaming response
```

You'll now be able to interact with the OpenAI-compatible REST API endpoints. As usual, refer to section E.3.3 for code examples, and make sure you use the correct port number and model name.

E.4.6 LocalAI

LocalAI is a free inference engine designed to run OpenAI-compatible REST API LLMs on consumer hardware, including systems with plain CPUs or low-grade GPUs. It supports various quantized open source LLMs and can also handle audio-to-text, text-to-audio, and multimodal models. Here, the focus is on its text LLM capabilities.

Written in Go, LocalAI serves as a higher-level inference engine that routes OpenAI-like REST API calls to back-end engines such as llama.cpp or vLLM. Figure E.8, adapted from the LocalAI documentation, illustrates this architecture.

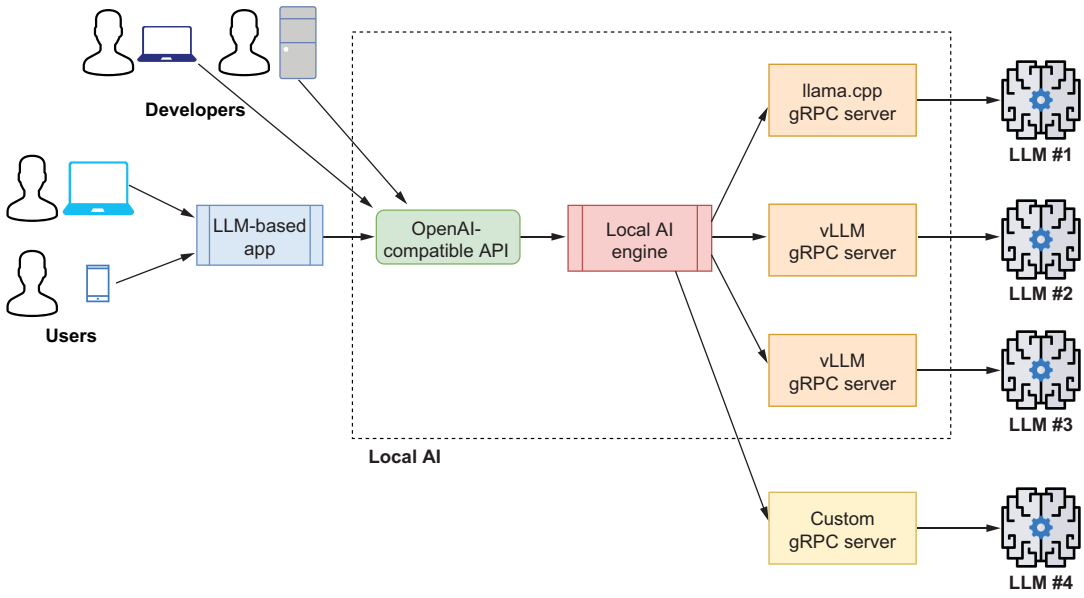


Figure E.8 LocalAI architecture. LocalAI routes OpenAI-like REST API calls to inference engines such as llama.cpp, vLLM, or other custom backends.

SERVER MODE

The primary distribution method for LocalAI is through container images, which can be deployed using Docker, Podman, or Kubernetes. Popular models are automatically downloaded when starting the container. For example, to run `Mistral-OpenOrca` on CPU, use the following:

```
docker run -ti -p 8080:8080
➥localai/localai:v2.7.0-ffmpeg-core mistral-openorca
```

If you have CUDA-12 GPU, you can run the related image with the `-gpu` option:

```
docker run -ti -p 8080:8080 --gpus all
➥localai/localai:v2.7.0-cublas-cuda12-core mistral-openorca
```

Container images for CPU, CUDA-11, and CUDA-12 are available on Docker Hub and Quay.io. The LocalAI documentation (https://localai.io/docs/overview/) provides the appropriate Docker commands for each model and hardware configuration.

If you have a custom quantized model in GGUF format, you can place it in a local folder (e.g., `local_models`) and reference it when starting the container:

```
docker run -p 8080:8080 -v $PWD/local_models:/local_models
➥-ti --rm localai/localai:v2.7.0-ffmpeg-core
➥--models-path /local_models --context-size 700 --threads 4
```

OpenAI-compatible REST API endpoints

When a model is started using Docker, LocalAI launches an HTTP server on port 8080 with OpenAI-compatible endpoints. Refer to section E.3.3 for examples of `curl`, Python, and LangChain code you can use with LocalAI. Ensure the port is set to 8080, and verify the model name in the LocalAI documentation for accurate configuration.

E.4.7 GPT4All

GPT4All is an inference engine built on llama.cpp that allows you to run GGUF quantized LLM models on consumer hardware, including CPUs and low-grade GPUs. It improves on llamafile with a more user-friendly graphical interface, making it accessible to nontechnical users.

Installation packages for Windows, macOS, and Ubuntu Linux are available on the GPT4All website (https://gpt4all.io). After installation, you'll see a desktop client, as shown in figure E.9, with an interface that allows you to download and chat with a model, as well as upload your documents to enable out-of-the-box RAG Q&A.

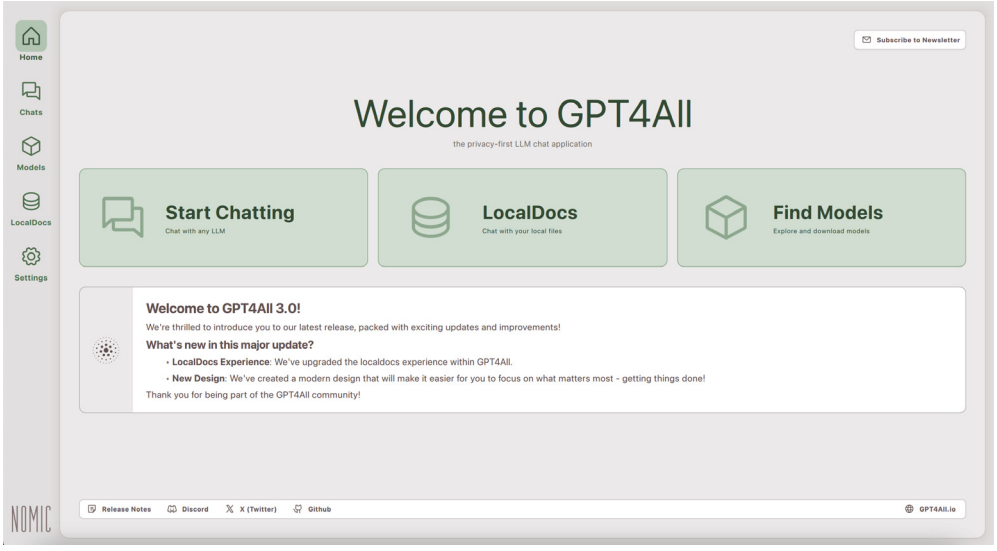


Figure E.9 GPT4All desktop application home screen

GPT4All includes the following components, visible and hidden:

- *Backend inference engine*—Built on llama.cpp, the engine supports GGUF quantized LLM models (typically under 8 GB) for architectures such as Falcon, Llama, MPT, GPT-J, and Mistral.
- *Language-specific bindings*—High-level API libraries are available for C++, Python, Go, Node.js, and more, enabling programmatic access to the inference engine.
- *Local web server*—The desktop application can start a local web server that exposes chat completions through OpenAI-compatible REST API endpoints.
- *Desktop GUI*—The graphical client lets you interact with an LLM through a user-friendly interface. Models can be selected from a dropdown or added from the Model Explorer on the GPT4All homepage or Hugging Face.
- *LocalDocs plugin (optional)*—This plugin allows you to import files containing data or unstructured text and chat with the content. It implements a local RAG architecture using SBERT and an embedded vector database, enabling basic Q&A functionality with zero coding.

SERVER MODE

To enable the local HTTP server for REST API access (port 4891), navigate to GPT4All > Settings, and then enable the web server option.

OPENAI-COMPATIBLE REST API ENDPOINTS

The server exposes OpenAI-compatible endpoints. Use the examples from section E.3.3 to send requests with `curl`, Python, or LangChain. Ensure the port is set to 4891, and then verify the correct model name in the GPT4All documentation.

GPT4ALL PYTHON BINDINGS

You can create a Python client using the GPT4All Python generation API. Install the package from PyPI:

```
pip install gpt4all
```

Then, you can invoke the `generate()` function as follows:

```
from gpt4all import GPT4All
model = GPT4All('mistral-7b-instruct-v0.1.Q4_0.gguf')
output = model.generate(
    'How many Greek temples are in Paestum?',
    max_tokens=10)
print(output)
```

When you instantiate the model, the GGUF file will download automatically to a local directory if it's not already available. The Python API also supports streaming responses.

LANGCHAIN GPT4ALL PYTHON LIBRARY

LangChain integrates with the native GPT4All Python bindings, providing an advantage by implementing standard LangChain Python interfaces, such as `Runnable`. This

allows you to build LangChain Expression Language (LCEL)-based solutions, as demonstrated in previous chapters. First, install the `gpt4all` package:

```
pip install gpt4all
```

Next, test the GPT4All LangChain integration by running the code in listing E.4. Use a dedicated environment created with `venv` for best practice.

Before running the code, however, ensure you download a GGUF quantized model of your choice into the specified local directory: open the GPT4All application, and navigate to the Models section in the left-hand menu. Click on + Add Model to browse the available models, and then search for those labeled with the `.gguf` extension. Once you find the model you want, simply click Download to save it to your device.

Listing E.4 Using LangChain's GPT4All wrapper

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_community.llms import GPT4All

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful AI assistant."),
    ("user", "{input}")
])

model_path =
➥ ('./models/mistral-7b-instruct-v0.1.Q4_0.gguf')
llm = GPT4All(model=model_path)

chain = prompt | llm

response = chain.invoke(
   {"input": "How many Greek temples are there in Paestum?"})
print(response)
```

Defines the prompt template

Specifies the path to the model

Chains the prompt with the model

Generates a response

This code demonstrates how to access a local open source model through GPT4All using LangChain's wrapper. With this integration, you can seamlessly implement complex workflows.

E.4.8 Comparing local inference engines

Before closing this section, I want to provide a comparison of the local inference engines discussed. Some engines are optimized for consumer-grade hardware, while others support advanced configurations, including high-end GPUs. Most engines include an OpenAI-compatible HTTP server, but only a few provide dedicated bindings for specific programming languages. A summary of the main characteristics of each engine is shown in table E.3.

Table E.3 Characteristics of local LLM inference engines

Inference engine	Backend engine	Installer available	Supported operating system	OpenAI REST API compatibility	Native bindings	Desktop user interface
llama.cpp	llama.cpp	No (must build from source)	MacOS, Linux, Windows	Yes	Python, Java, C#, Go, Scala, Ruby, Rust, Scala, JavaScript, Node.js	No
Ollama	llama.cpp	Yes	MacOS, Linux, Windows	Yes	Python, JavaScript	No
vLLM	vLLM	Yes (pip)	Linux	Yes	Python	No
llamafile	llama.cpp	Yes	MacOS, Linux, Windows	Yes	N/A	No
LocalAI	Llama.cpp and vLLM	Docker image	Any hardware running Docker	Yes	N/A	No
GPT4All	Llama.cpp	Yes	MacOS, Linux Ubuntu, Windows	Yes	Python	Yes
LM Studio	Llama.cpp	Yes	MacOS, Linux Ubuntu, Windows	Yes	Python	Yes

By now, you should have a clear understanding of how to use local inference engines to run open source LLMs. However, inference engines aren't the only option for serving local LLMs. In the next section, I'll briefly explain how to perform inference using the Hugging Face Transformers library.

E.4.9 Choosing a local inference engine

If you're new to local LLM inference engines, start with a simple option such as llamafile. It offers an easy way to experiment with lightweight models. Once you're comfortable, consider moving to Ollama, which simplifies downloading, configuring, and testing additional models. If you prefer a graphical user interface, consider upgrading to LM Studio or GPT4All instead.

For production-grade solutions, vLLM is a strong choice. If you need maximum control over all aspects of hosting an LLM, consider using llama.cpp. It provides advanced customization options but requires a deeper understanding of the hosting process.

E.5 Inference via the Hugging Face Transformers library

For full control over a model's architecture and the ability to modify it, you can run a pretrained model using the Hugging Face Transformers library. This library is not only designed for experimentation and configuration but also for fine-tuning models, enabling you to share your improvements with the Hugging Face community. Built on deep learning frameworks such as JAX, PyTorch, and TensorFlow, it allows you to use one framework for training and another for inference.

E.5.1 Hugging Face Transformers library

Before installing the `transformers` package, set up a virtual environment with `venv`, and install one or more of the following backends: Flax, PyTorch, or TensorFlow. Backend installation can vary depending on your hardware, so it's not covered here. Assuming the backends are installed, you can install `transformers` via PyPI:

```
pip install transformers
```

Once installed, you can interact with pretrained models stored locally. For example, with the PyTorch backend, you can run inference on a quantized 4-bit version of Mistral Instruct as shown in the following listing, which I've adapted from the Hugging Face documentation.

Listing E.5 Inference via the Hugging Face Transformers library

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

model_id = "mistralai/Mixtral-8x7B-Instruct-v0.1"
tokenizer = AutoTokenizer.from_pretrained(model_id)

model = AutoModelForCausalLM.from_pretrained(model_id, load_in_4bit=True)

text = "Hello my name is"
inputs = tokenizer(text, return_tensors="pt").to(0)

outputs = model.generate(**inputs,
    max_new_tokens=20)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

Specifies the model ID

Loads the tokenizer and model

Defines input text and tokenizes

Generates output

When using the Transformers library, you must explicitly handle tokenization. While this approach provides significant flexibility, it requires a deeper understanding of transformer architecture, making it well-suited for advanced use cases, research, and experimentation. Next, let's see how you'd write the same code with LangChain.

E.5.2 LangChain's Hugging Face pipeline

LangChain offers a wrapper for the Hugging Face Transformers pipeline, simplifying integration into LangChain applications. If you want to use Hugging Face with LangChain, you can implement the code shown in the following listing.

Listing E.6 Hugging Face transformers via LangChain

```
from langchain_community.llms.huggingface_pipeline
➥import HuggingFacePipeline
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline
from langchain.prompts import PromptTemplate

model_id = "mistralai/Mixtral-8x7B-Instruct-v0.1"

tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(model_id)

pipe = pipeline("text-generation",
    model=model,
    tokenizer=tokenizer,
    max_new_tokens=50)
hf_pipeline = HuggingFacePipeline(pipeline=pipe)

prompt_template = """Question: {question}

Answer: Let's think step by step."""
prompt = PromptTemplate.from_template(prompt_template)

llm_chain = prompt | hf

question = "How many Greek temples are there in Paestum?"

print(llm_chain.invoke({"question": question}))
```

Defines the model ID

Loads the tokenizer and model

Creates the Hugging Face pipeline

Defines a prompt template

Creates the LangChain pipeline

Generates a response

Now that I've covered how to run an open source LLM locally, the next step is to explore practical applications. In the following section, I'll show how to redirect the research summarization engine we built in chapter 4 from public OpenAI endpoints to a local open source LLM such as Mistral.

E.6 Building a local summarization engine

Revisit the research summarization engine from chapter 4 to compare the original OpenAI-based solution with a local open source LLM. To do this, duplicate the Visual Studio Code project folder by copying the `ch04` folder and renaming it `apE`. This setup allows you to test both versions side by side, making it easier to evaluate accuracy and performance.

Keep in mind that processing with a local LLM will take significantly longer compared to OpenAI's API. However, this exercise will provide valuable insights into how hardware affects inference speed. It will also give you hands-on experience in evaluating whether and how to implement a project using a local open source LLM.

E.6.1 Choosing the inference engine

First, decide which inference engine to use. Options such as GPT4All and LM Studio are user-friendly and suitable for those with basic LLM experience.

To minimize code changes, avoid rewriting the application with native Python bindings like those offered by GPT4All. Instead, use the OpenAI-compatible REST API endpoints provided by both GPT4All and LM Studio. Choosing between them is a matter of preference because both meet the requirements of usability and OpenAI compatibility. For this example, I'll use LM Studio.

E.6.2 Starting up the OpenAI-compatible server

If LM Studio isn't already installed, follow the installation steps covered earlier. Open the Server menu, select a port (e.g., 8080), and start the server. LM Studio will now accept OpenAI-compatible requests.

E.6.3 Modifying the original solution

Because requests will be routed through an OpenAI-compatible endpoint, you only need to make a small change in the `llm_models.py` file, as shown in the following listing.

Listing E.7 Original `llm_models.py` code

```
from langchain.llms.openai import OpenAI

openai_api_key = 'YOUR_OPENAI_API_KEY'

def get_llm():
    return OpenAI(openai_api_key=openai_api_key, model="gpt-4o-mini")
```

You simply need to replace the implementation of `get_llm()` as shown next.

Listing E.8 Modified `llm_models.py` using a local open source LLM

```
def get_llm():
   port_number = '8080'

   client = OpenAI(
       base_url=f'http://localhost:{port_number}/v1',
       api_key = 'NO-KEY-NEEDED',
       model='mistral'
   )

   return client
```

This approach avoids changes to the rest of the code. However, for flexibility, you could make `get_llm()` configurable, as shown in the next listing.

Listing E.9 Configurable `get_llm()`

```
def get_llm(is_llm_local=False):
    port_number = '8080'
```

```
if is_llm_local:
    client = OpenAI(
        base_url=f'http://localhost:{port_number}/v1',
        api_key='NO-KEY-NEEDED',
        model='mistral'
    )
else:
    client = OpenAI(openai_api_key=openai_api_key, model="gpt-4o-mini")

return client
```

This version allows you to switch between OpenAI and local LLMs. Keep in mind that some inference engines have fixed port numbers or specific model naming conventions, which may require you to further generalize the code and make it more configurable.

E.6.4 Running the summarization engine through the local LLM

To test the summarization engine, execute `chain_try_5_1.py`. The LM Studio server logs panel should display activity as the request is processed. When the output is generated (this may take some time, as noted earlier), it should closely match the results from the original OpenAI-based summarization.

E.6.5 Comparison between OpenAI and local LLM

When running the summarization engine with a local LLM, you'll notice the following:

- *Accuracy*—The output is similar for this use case (web scraping and summarization).
- *Performance*—Local LLM inference is slower, especially on CPUs. Without a GPU, processing times can be significantly longer.

This suggests that while a quantized local open source LLM is sufficient for development, it may not meet production requirements. Consider deploying LM Studio on hardware with an NVIDIA GPU or using a more advanced inference engine such as vLLM. vLLM supports unquantized models and high-end GPUs, offering better performance for demanding use cases.

index

A

accuracy 23
  vs. speed 368
agent graph structure 278
agentic workflows 103–107
agents 15, 104–107
  agent development frameworks 106
  agent state 275
  building tool-based with LangGraph 268–271
  executing tool calls 275
  function calling to tool calling 271
  integrating Weather MCP tool into 322–325
  overview of 106
  tool-based, running agent chatbot 279
AgentState 289
AI (artificial intelligence), agents and applications
  LLM terminology 24
  open source LLMs, running locally 382–386
AI agents
  deployment 348
  evaluation of 347
  guardrails 337–345
  human-in-the-loop 347
  memory, long-term user and application memory 346
  productionizing 327
  text summarization with LangChain 56–61
AI agents and applications 3
  overview of 24
Anaconda 357
ANN (Approximate Nearest Neighbor) 131
architecture, enhancing with query rewriting 76
Assistant Instructions chain 90–91
autonomous reasoning 19
AZLyricsLoader 64

B

B&B booking tool 295
backend inference engine 403
BBH (BIG-Bench Hard) 369
Beautiful Soup 74
behavioral testing 348
bias 371, 374
bitsandbytes 382
BM25 229
BnBBookingService 295
bs4 package 73
BSON (Binary JSON) 230

C

calculate_my_embedding() function 246
chains, defined 15
CharacterTextSplitter 179
chat messages 158
chatbots
  launching loops 280
  Q&A chatbots 157–163
  running agent chatbot 279
ChatOpenAI client 34
ChatOpenAI wrapper 32
checkpoints, defined 329

ChromaDB 133–136
  building filter statements from structured queries 237
  setting up collection 232
chunk expansion 197–201
chunks, defined 14
Claude 362
CMake 388
cmd (Command Prompt) 294
code generation 19
Cohere 362
conditional edges 109
content database query generation 229–230
context window 14, 24, 56–57, 367
context, defined 122
Cosmopolitan-Libc 396
cost 23
  hardware requirements 368
CoT (Chain of Thought) 28, 46–48, 365

D

databases, setting up and connecting to 240
ddgs package 73
DeepSeek 364
dense retrieval 229
dense vector search 131
dependencies, installing 268
deployment, defined 348
desktop GUI 403
DirectoryLoader 65, 151–152
doc_summary_chain 66
docs object 181
document, defined 122
Document class 144, 146
document databases 230
document indexing, advanced 177
Document list, creating 65
document loaders 63–65
Document objects 12, 14–15, 57, 62–65
document store 185
document type 179
Domain-specific model type 380
DuckDuckGo search engine 73–74

E

edges 109–110
embedding 5, 129
embedding document summaries 189–192
embedding hypothetical questions 193–197
embedding model 5, 14
embedding strategy 177, 183
Embeddings class 144, 146, 148
end conditions 110
END node 329
entry points 110
environment variables, loading 269
executing prompts programmatically 28–31
  minimal prompt execution 30–31
  prompt templates 32–34
  reasoning 40–48
  setting up environment 28
executing tool calls 275
explicit metadata tags 230

F

FAISS (Facebook AI Similarity Search) 131
Falcon 363
fastmcp package 316
few-shot learning 42
  implementing with LangChain 44
few-shot prompting 20
file-based content 64
fine-tuning 22–23
FK (foreign key) 240
Foundation model type 380
function calling 271
functional testing 348

G

Gemini 361
Gemma 362
generating, semantic SQL queries 244–248
generative AI 36
GGML (Georgi Gerganov Machine Learning) 382
GGUF (GPT-Generated Unified Format) 382
glob pattern 152
gmake 388
GoogleSearchAPIWrapper wrapper 74
GPQA (Graduate-Level Google-Proof Q&A) 369
GPT-5 Thinking 42
gpt-5-mini chat model 275
GPT4All 402–404
GPTQ (Generalized Post-Training Quantization) 382
granular chunk expansion 197–201
graph databases, query generation for 248–250
GraphRAG 230

Grok 364
grounding, defined 21
GuardrailDecision model 339
guardrails 337–345
  rejecting nontravel-related questions 338–342
  restrictive guardrails at agent level 343–345

H

hallucinations 21, 240, 374
hardware requirements, cost and 368
HTMLHeaderTextSplitter 178–179
Hugging Face Transformers library, inference via 406
human-in-the-loop 347
  workflows 329
HumanMessage 279
HyDE (Hypothetical Document Embeddings) 222–224

I

IFEval (Instruction-Following Evaluation) 369
IMSDb (Internet Movie Script Database) 64
in-context learning 20, 47
indexing
  advanced 173, 177–183
  embedding child chunks with MultiVectorRetriever 187–189
  embedding child chunks with ParentDocumentRetriever 184–186
  embedding document summaries 189–192
  embedding hypothetical questions 193–197
  embedding strategy 183
  granular chunk expansion 197–201
  improving RAG accuracy 174–177
  semi-structured content 201
  structured and semi-structured data 177
inference engines 404–405
ingestion, metadata enrichment 232–233
InMemoryByteStore 187
InMemorySaver 333
InMemoryStore 185
installing, SQLite on Windows 375
Instruct model type 380
instruction versus reasoning models 24
IP (intellectual property) rights 374

J

JSON, to Python object converter 76
JsonOutputFunctionsParser 89
Jupyter Notebook environment 357–359

K

key–value
  databases 230
keyword
  extraction 230
  postprocessors 257
  suggestions 230
KG-RAG (Knowledge Graph RAG) 249
KNN (k-Nearest Neighbors) 131
knowledge graph
  databases 14, 230
  embeddings 250

L

LangChain 11–17
  agentic workflows, LangChain to 107
  architecture 12–15
  asking question through chain 153
  chains to LangGraph 107
  core object model 15
  implementing few-shot learning with 44
  in Jupyter Notebook environment 351–356
  LangChain's Hugging Face pipeline 406
  llama.cpp integration 389
  object model for Q&A chatbots 144–148
  original implementation overview 111
  PromptTemplate 33
  Python library 404
  Q&A chatbots with, vector store content ingestion 148–152
  registering tools in 275
  running prompts with 31
  text summarization with 55–68
  trying out 351
langchain packages 32, 73, 152, 360, 362
LangGraph
  agent graph structure 278
  agentic workflows, LangChain to 107
  assembling agent graph 277
  basics 107
  building tool-based agents with
    adding weather forecast tool 283
    enabling agents to call tools 271–277
    executing requests 279–283
    running agent chatbot 279
    travel information agents 268–271

LangGraph *(continued)*
checkpoints in 329
core components 108–110
tool-based agents 267
turning web research assistant into AI agent 110–117
using prebuilt components for rapid development 288–291
LangGraph platform 348
LangGraph ReAct agent 288–289
LangSmith
enabling tracing 289
observing and debugging with 289
Q&A chatbots with, vector store content ingestion 148–152
setting up API key 164–167
tracing execution with 163–167
language-specific bindings 403
LanguageModel class 146, 148
LCEL (LangChain Expression Language) 12, 55, 69, 160, 247, 404
creating chains and executing them with 355
reimplementing research summarization engine in 89–101
Assistant Instructions chain 91
Search and Summarization chain 95–99
Web Research chain 99
Web Searches chain 93
lexical retrieval 229
lexical search 131
liability, defined 374
Llama 363
llama.cpp 388–389
llamafile 396–397
LLM cache 15
LLM-based applications and agents 4–11
AI agents 8–11
LLM-based applications 4–7
LLM-based chatbots 7, 15, 122
LLMChain block 58
LLMs (large language models) 27, 49, 75, 121, 229, 328, 360–365
activating nodes 280
adapting to needs 20–23
building local summarization engine 407–409
choosing 23, 360, 366–373
choosing local inference engine 405
Claude 362
Cohere 362
DeepSeek 364
Falcon 363
Gemini 361
Gemma 362
Grok 364
inference via Hugging Face Transformers library 406
Llama 363
LLM node 277
LM Studio 397–399
local inference engines 404
Mistral 363
open source, running locally 382–386
OpenAI GPT series 360
Phi-3 family 364
Qwen 363
reasoning 40–48
registering tools with 273–275
terminology 24
tool calling with 273, 281
use cases for 19
local inference engines 387
local running of open source LLMs 382–386
local web server 403
LocalAI 401–402
LocalDocs plugin 403
LoRA (Low-Rank Adaptation) 22

M

map operation 59–62, 99–100
MapReduce 60–61
MarkdownHeaderTextSplitter 178
MATH (Mathematics Aptitude Test for Heuristics) 363, 369
MCP (Model Context Protocol) 11, 309, 349
context integration at scale 309
ecosystem 311
Model Context Protocol 311
MCP (Model Context Protocol) servers and consuming
building 308, 312–322
MCP (multi-channel processing) servers, integrating Weather MCP tool into agent 322–325
MCP Inspector
connecting to Weather MCP server 317
exploring and testing weather tool 318
installing 316
memory 328–337
adding short-term memory to travel assistant 331–333
chatbot memory of message history 157–163

checkpoints in LangGraph 329
executing checkpointer-enabled assistant 334
long-term user and application memory 346
need for short-term 328
rewinding state to past checkpoint 334–337
types of 328
metadata, enrichment 232–238
Miniconda 358
Mistral 363
Mixture-of-Experts (MoE) 360–361
MMLU Pro (Massive Multitask Language Understanding Professional) 363, 370
Moby Dick book 57–58
model
name 385
purpose 24
size (number of parameters) 367
MoE (Mixture-of-Experts) 360–361
multi-agent graph 300
multi-agent systems 293
building accommodation booking agent 294–298
handling multi-agent requests with Supervisor component 302–305
router-based travel assistant 298–302
multi-query generation 214–218
setting up chain for 215
setting up custom multi-query retriever 216
using standard MultiQueryRetriever instance 218
multi-step decomposition 224–227
multi-tool agents, executing 285–288
multi-turn conversational context 329
multilingual support 24, 368
multimodal RAG (retrieval-augmented generation) 202
MultiQueryRetriever 215, 258
MultiVectorRetriever 183, 187, 191, 201–203
comparing with direct semantic search on child chunks 189
ingesting content into document and vector stores 188
performing search on granular information 189
performing search using 196, 200
setting up 187, 194
setting up for chunk expansion 198
MuSR (Multi-step Soft Reasoning) 363, 370

N

natural language generation 19
natural language understanding 19
NF4 (Normal Float 4) 382
NIM (NVIDIA Inference Microservices) 364
NLP (natural language processing) 362
nodes 108–110

O

OAP (Open Agent Platform) 349
Ollama 389
interactive mode 390
LangChain integration 393
native Python library 392
server mode 391
one-shot learning 40
open source LLMs
benefits of 377–379
GPT4All 402–404
llama.cpp 388–389
llamafile 396–397
local inference engines 387
LocalAI 401–402
popular 379–381
running locally 382–386
vLLM 394–395
open source versus proprietary 24, 366
OpenAI API
adding API key 269
compatible endpoints 397
llama.cpp compatible endpoints 389
manual tool registration with 274
OpenAI GPT series 360
OpenAI REST API 383–386, 403
direct curl invocation 383
LangChain's openai wrapper 386
Python openai library 385
OpenAIEmbeddings wrapper 246
OpenWeatherMap 288
output parser 15

P

PaLM 2 (Pathways Language Model 2) 361
ParentDocumentRetriever 184, 186–187, 189
performance testing 348
personalized education 19
PII (Personally Identifiable Information) 378
pip 358, 385

PK (primary key) 240
plugins, defined 15
PoC (proof of concept) phase 382
PostgresSaver 333
PostgresSaverAsync 333
privacy 374, 378
prompt engineering 20, 27, 78–82, 354–355
  prompt template 354
  research report prompt 81
  summarization prompts 81
  web search prompts 79–81
prompt templates 14, 32–34
  implementing with Python functions 32
  using LangChain's PromptTemplate 33
prompt types 34–40
  question answering 38
  reasoning 39
  sentiment analysis 35
  text classification 34
  text composition 36–38
  text summarization 36
prompt, defined 122
prompts 27
  executing programmatically 28–31
  programmatic execution of, prompt structure 48–49
PromptTemplate 56, 89
proprietary models vs. open source 366
PTQ (Post-Training Quantization) 382
Python
  bindings 388, 403
  implementing prompt templates with functions 32
  installing interpreter or distribution 357
  JSON to Python object converter 76
  LangChain GPT4All library 404
  openai library 385

Q

Q&A (Question & Answer) engine 5
Q&A chatbots 143
  across stored documents 153–157
  chatbot memory of message history 157–163
  LangChain object model for 144–148
  tracing execution with LangSmith 163–167
  vector store content ingestion 148–152
QAT (Quantization-Aware Training) 382
query generation 229–230
  for graph databases 248–250
  generating semantic SQL queries 244–248
  generating structured SQL queries 240–244
  metadata-enriched collections 234–238
  self-querying 230
query generation, routing, and retrieval postprocessing 228
  chain routing 251–256
query rewriting 76
question answering 38
question transformations 205
  generating multiple queries 214–218
  HyDE (Hypothetical Document Embeddings) 222–224
  Rewrite-Retrieve-Read 206–213
  single-step and multi-step decomposition 224–227
  step-back question 218–221
Qwen 363

R

RAG (Retrieval-Augmented Generation) 6, 20, 76, 121, 205, 228, 268, 364, 379, 402
  implementing from scratch 136–141
  improving accuracy of 174–177
  incorporating step-back question generation into 220
  integrating HyDE chain into RAG chain 223
  multimodal 202
  retrieval postprocessing 256–263
  vector stores 130
RAG agents, advanced indexing
  splitting by HTML header 179–183
  splitting strategy 177–179
RAG chain
  completing setup 154
  feeding chat history to 160
RAG design pattern, semantic search 122–130
  content ingestion stage (indexing) 128
  Q&A chatbot over knowledge base 126
  Q&A chatbot over single document 122–126
  question-answering stage (retrieval and generation) 129
  RAG design pattern 127–130
RAG fusion (Reciprocal Rank Fusion) 258–263
  generating multiple queries 259
  incorporating into RAG chain 261
  overview of 258
  RRF algorithm 260
  setting up retrieval chain 261
RateLimitError 61
RDF (Resource Description Framework) 248

ReAct (Reasoning and Acting) design pattern 272
  accommodation booking agent 297
reasoning 39–48
  Chain of Thought 46–48
  few-shot learning 42, 44
  one-shot learning 40
  providing steps 41
  two-shot learning 41
RecursiveCharacterTextSplitter 181
reduce chain 60
reduce operation 62
refine technique 62, 66
registering tools 273–275
regression testing, defined 348
relational (SQL) database, overview 229
RemainingSteps utility 288
REPL (Read-Eval-Print Loop) 279
requests package 73
Research Report chain 90
research report prompt 81
research reports, generating 87–89
research summarization engine
  enhancing architecture with query rewriting 76
  implementing core functionality 73–76
  initial implementation 82–89
  overview of 70
  prompt engineering 78–82
  reimplementing in LCEL 89–101
  setting up project 71
ResearchState 108–109
responsibility, overview 374
result_text_summary_list 86
retrieval chunks, defined 183
retrieval postprocessing 256–263
  keyword postprocessors 257
  RAG fusion (Reciprocal Rank Fusion) 258–263
  similarity postprocessors 257
  time weighting 257
retrievers, overview 14
Rewrite-Retrieve-Read 206–213
  combining everything into single RAG chain 212
  retrieving content using original user question 209
  retrieving content with rewritten query 211
  setting up query rewriter chain 210
RLHF (Reinforcement Learning from Human Feedback) 22, 363
router agent
  designing 298
  trying out 301
router-based travel assistant 298–302
  building multi-agent graph 300
  designing router agent 298
  routing logic 298
  trying out router agent 301
routing logic 298
routing, chain routing 251–256
  integrating chain router into full RAG chain 254
  setting up data retrievers 252
  setting up query router 252
  setting up retriever chooser 254
  testing router chain 253
RRF (Reciprocal Rank Fusion) algorithm 259–260
Runnable interface 12, 14, 17, 59, 154, 393
Runnable protocol 89
RunnableConfig 332
RunnableLambda 59, 94, 97
RunnableParallel 59, 97

S

SaaS (software as a service) 348
score threshold similarity retriever 257
Search and Summarization chain 90, 95–99
  assembling 98
  Search Result Text and Summary chain 96
  Search Result URLs chain 95
search type 179
searching, using MultiVectorRetriever 192
security, overview 374
self-querying 230–231
SelfQueryRetriever 232
  generating metadata filters with 235
semantic retrieval 229
semantic search 19, 122–130
  direct 192
  Q&A chatbot over knowledge base 126
  Q&A chatbot over single document 122–126
  RAG design pattern 127–130
semantic SQL queries, generating 244–248
sentence
  completion example 353
  expansion 177
sentiment analysis 19, 35
server mode 396, 403
similarity
  postprocessors 257
  searches 131

single-step decomposition 224–227
SLMs (small language models) 364
SMoE (Sparse Mixture-of-Experts) 363
sparse
  retrieval 229
  search 231
  vectors 131
speed (latency) 23
speed, accuracy vs. 368
splitting strategy 177–178
  choosing right strategy 179
  factors to consider 179
  splitting by HTML header 179–183
SQL (Structured Query Language)
  generating semantic SQL queries 244–248
  generating structured SQL queries 240–244
SQLite, installing 240, 375
SqliteSaver 333
SRE (Site Reliability Engineering) 348
Standalone Python Interpreter 357
standard benchmarks 368–370
START node 329
state management 109
state rehydration after failure 329
StateSnapshot entries 336
STDIO (standard input/output) 311
step-back question 218–221
StrOutputParser 92–93, 216
structured data extraction 19
summarization chain 191–192
summarization engine, building local 407–409
summarization prompts 59–60, 81
summarization, flowchart 68
summarizing text, across documents 61–67
  creating Document list 65
  creating list of Document objects 63
  file-based content 64
  progressively refining final summary 65
  Wikipedia content 64
Supervisor component 302–305
synthesis chunks, defined 183
synthesize, defined 122

T

TavilySearchResults wrapper 74
TCO (total cost of ownership) 379
templates, prompt templates 32–34
text classification 19, 34
text composition 36–38
text summarization 36, 55
  with LangChain 56–61
TextLoader 65
TextSplitter class 144, 146
TF-IDF (Term Frequency-Inverse Document Frequency) 131, 229
threads, defined 331
tiktoken package 57
time weighting 257
TokenTextSplitter 57
tool-based agents 267
  adding weather forecast tool 283–284
  agent graph structure 278
  assembling agent graph 277
  enabling agents to call tools 271–277
  executing multi-tool agents 285–288
  executing requests 279–283
  LangGraph 288–291
  running agent chatbot 279
toolkit, defined 15
ToolsExecutionNode 276, 281
TPUs (Trillium tensor processing units) 361
transformers package 406
transparency, overview 378
travel assistant
  adding short-term memory to 331–333
  building accommodation booking agent 294–298
  executing checkpointer-enabled assistant 334
  rewinding state to past checkpoint 334–337
  router-based 298–302
travel information agents 268–271
  loading environment variables 269
  preparing travel information vector store 270
  project setup 268–269
TTS (text-to-speech) 361
tutoring 19
two-shot learning 41
TypedDict 108

U

UkBooking database 240–242
unstructured package 152
UnstructuredLoader 65
UUID (Universally Unique Identifier) 331

V

vector databases 131
  retrieving content from 137
vector libraries 131

vector stores 5, 14, 130, 185, 229
content ingestion 148–152
creating 280
how they work 131
most popular 132
overview of 131
querying directly 153
storing text and performing semantic search using Chroma 133–135
vector libraries vs. vector databases 131
virtual environments 268
vLLM 394–395
VS Code (Visual Studio Code) 71, 295

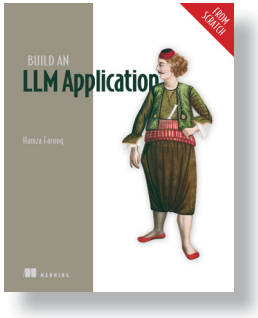
W

weather forecast tool 283–284
weather MCP (Model Context Protocol) servers, building 314–322
Weather MCP tool 323–325
web research assistant, turning into AI agent 110–117
Web Research chain 99
web results 83–86
web search prompts 79–81
web searches chain 90, 93
web searching 73
WebBaseLoader 64
Wikipedia content 64
WikipediaLoader 64, 150
workflows 105
agentic, LangChain to LangGraph 107
execution 19

Z

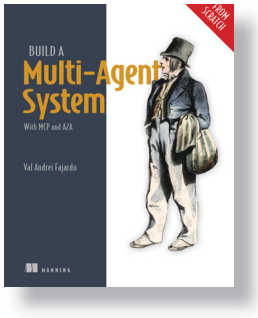
zero-shot learning 40
Zig 388

RELATED MANNING TITLES



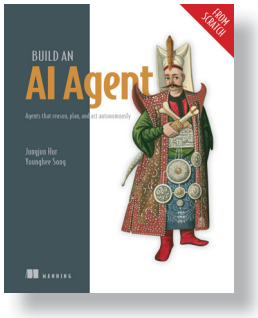
Build an LLM Application (from Scratch)
By Hamza Farooq

ISBN 9781633436527
325 pages (estimated), $59.99
Summer 2026 (estimated



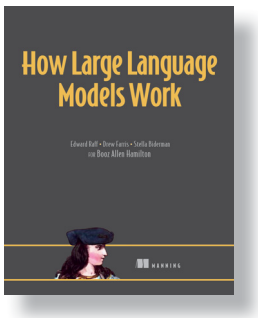
Build a Multi-Agent System (from Scratch)
By Val Andrei Fajardo

ISBN 9781633434660
325 pages (estimated), $59.99
Summer 2026 (estimated



Build an AI Agent (From Scratch)
By Jungjun Hur and Younghee Song

ISBN 9781633434615
375 pages (estimated), $59.99
Summer 2026 (estimated)

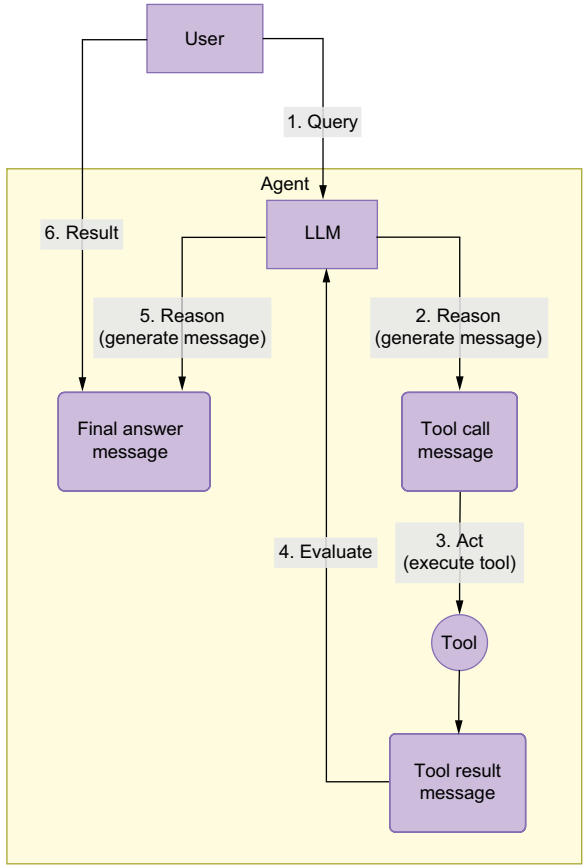


How Large Language Models Work
By Edward Raff, Drew Farris, and Stella Biderman for Booz Allen Hamilton

ISBN 9781633437081
200 pages, $49.99
June 2025

For ordering information, go to www.manning.com

ReAct Pattern



The ReAct pattern alternates between reasoning and action, enabling the agent to process a user question by thinking, calling tools as needed, and following up with further reasoning before delivering the final answer.

DATA SCIENCE

AI Agents and Applications

Roberto Infante

This book teaches you to design reliable LLM-powered systems by focusing on the concepts, architectures, and design patterns that will stay stable even as models and APIs change. You'll learn to structure prompts, compose modular chains, and build RAG pipelines that ingest documents, split them into chunks, embed them, retrieve the right context, and ground answers to elliminate (or vastly reduce) hallucinations.

Along the way you'll build concrete applications—summarization and Q&A engines, context-aware chatbots with memory, and tool-using AI agents that orchestrate multi-step workflows with branching logic. For the examples, the book uses Python, LangChain, LangGraph, and LangSmith, but you'll be able to generalize to other frameworks. You'll understand with clarity and confidence how to keep integrations maintainable, manage context limits and cost/latency tradeoffs, and evaluate, debug, and monitor behavior so your systems work in production.

Roberto Infante is an AI innovator with deep FinTech experience, working for a London-based hedge fund. He specializes in building agentic systems for both plain vanilla and exotic quantitative analysis.

For print book owners, all digital formats are free:
https://www.manning.com/freebook

"The playbook for building AI systems that actually work at scale."
—Muntazir Abidi, Auquan

"A perfect bridge between AI theory and real-world implementation."
—Abdullah Al Imran, Aon PLC

"The practical, modern guide the industry has been waiting for."
—Ajay Solanki
Innovation Hacks AI Inc.

"A must-read for those looking to understand the 'why' and the 'how' of using LLMs in practice."
—Octavian Brînzea, EROM.io

"The hands-on guide I wish existed when I started building production AI agents."
—Tirso Lopez
Multiverse Computing

ISBN-13: 978-1-63343-654-1

MANNING