

Generative AI Apps with LangChain and Python

A Project-Based Approach to Building
Real-World LLM Apps

—
Rabi Jay

Apress®

Generative AI Apps with LangChain and Python

**A Project-Based Approach
to Building Real-World
LLM Apps**

Rabi Jay

Apress®

Generative AI Apps with LangChain and Python: A Project-Based Approach to Building Real-World LLM Apps

Rabi Jay
Pennington, NJ, USA

ISBN-13 (pbk): 979-8-8688-0881-4
<https://doi.org/10.1007/979-8-8688-0882-1>

ISBN-13 (electronic): 979-8-8688-0882-1

Copyright © 2024 by Rabi Jay

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Melissa Duffy
Development Editor: James Markham
Coordinating Editor: Gryffin Winkler

Cover designed by eStudioCalamar

Cover image by Rawpixel.com on Freepik

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (<https://github.com/Apress>). For more detailed information, please visit <https://www.apress.com/gp/services/source-code>.

If disposing of this product, please recycle the paper

Table of Contents

- About the Authorxv**
- About the Technical Reviewersxvii**
- Chapter 1: Introduction to LangChain and LLMs1**
 - Understanding LangChain.....2
 - A Simple Generative App Using LangChain3
 - What Are LLMs, and Why Are They Important?6
 - Examples of LLMs7
 - Why Is LangChain Important?8
 - Simplifies LLM App Development and Boosts Productivity8
 - Modular and Scalable Architecture9
 - Open Source Support and Community Collaboration10
 - No Cost Barrier11
 - Real-World Examples of LangChain11
 - Personalized Responses Through Prompt Engineering11
 - Building Data-Aware Decision-Making Apps13
 - Building Context-Aware Applications14
 - Developing RAG-Based Apps15
 - Building Agentic Apps.....17
 - Why These Features Matter19
 - Integrating LLMs with LangChain20
 - Simplified Integration with Multiple LLMs22
 - Exploring Core Components of LangChain23

TABLE OF CONTENTS

LLM Application Development Workflow	26
Key Takeaways.....	31
Review Questions	32
Answers.....	34
Looking Ahead	35
Bring Your Ideas to Life with LangChain.....	35
Glossary of Technical Terms.....	36
Further Reading	38
Chapter 2: Integrating LLM APIs with LangChain	39
Understanding LLM APIs	40
Business Benefits of Using LLM APIs for Generative AI	40
Technical Benefits of Using LLM APIs.....	42
Addressing Challenges in LLM API Integration	43
Additional Things to Consider	46
Using Direct LLM API vs. LangChain	48
Development Complexity	48
Integration and Scalability Challenges	49
Generic Response Issue	49
Streamlining Data Integration	51
Choosing Between Direct LLM API Use and LangChain.....	51
Using LangChain.....	53
Preparing Your Dev Environment	55
Step 1: Obtaining OpenAI API Keys	55
Step 2: Setting Up the Python Development Environment.....	57
Congratulations!	61
Exercise 1: Calling an LLM API Directly	61
Exercise 2: Using LangChain for Enhanced Flexibility	65

Key Takeaways.....	70
Start Creating with LangChain.....	72
Glossary of Technical Terms.....	72
Further Reading	73
Chapter 3: Building Q&A and Chatbot Apps	75
LangChain Framework Components	75
Development	76
Production	82
Deployment Using LangServe	84
LangChain Ecosystem.....	84
Using LangChain Models with LLMs	89
Model IO: The Core Functionality of LangChain	89
Large Language Models (LLMs) with LangChain.....	90
Building a Simple Q&A Application	91
Full End-to-End Working Code.....	93
Building a Conversational App	94
Difference Between the Q&A and Chatbot Example.....	98
Error Handling and Troubleshooting.....	99
Understanding Common Errors	100
Implementing Error Handling in Code.....	100
Diagnosing and Resolving Common Issues.....	101
Development Playground	103
LangChain Playground.....	103
OpenAI API Playground	104
Hugging Face Spaces	104
Colab Notebooks.....	104
Kaggle Notebooks.....	105
Maximize Your Learning Through Experimenting.....	105

TABLE OF CONTENTS

Experiment Freely.....	105
Document Your Findings.....	105
Share and Collaborate.....	106
Review Questions	106
Answers.....	108
Additional Review.....	108
Key Takeaways.....	109
Glossary	110
Further Reading	111
Chapter 4: Exploring Large Language Models (LLMs)	113
OpenAI's Models	114
GPT: The Next-Generation Language Models.....	114
Getting Started with GPT (Code Snippet).....	114
Codex: Your AI Coding Assistant	115
DALL-E 2: The Image Generation Wizard	116
Google's AI Model Overview.....	120
Language and Chat Models	120
Codey Suite for Code-Related Tasks.....	131
Text and Image Processing Models.....	132
Multimodal and Security Models.....	132
Speech Models	133
Anthropic's Claude AI Models.....	133
Claude 3 Model Family: Claude 3 Opus, Sonnet, and Haiku	134
Key Features of Claude 3 Models	134
Getting Started with Code	135
Overview of Cohere AI Models	139
Practical Example: Using Cohere's Command Model (Code Snippet).....	140

Meta AI Models	143
Calling the LLaMA Model Using Hugging Face	144
PyTorch.....	152
Review Questions	152
Answers.....	155
Key Learnings	155
Glossary	156
Overview and Capabilities of Open Source Models.....	156
Definitions of Key Terms and Concepts	158
Further Reading	160
Chapter 5: Mastering Prompts for Creative Content.....	161
Importance of Prompt Engineering	161
Why Prompt Engineering?	162
Need for Scalability	163
Prompt Engineering Steps	164
What Is the Goal?.....	164
Composing Your Prompts	165
Selecting Your Examples	167
Testing Your Prompts.....	167
Reviewing the Outputs	167
Fine-Tuning Your Prompts	167
Embracing Iteration	169
Deploying Your Prompts	169
Monitoring and Maintaining	169
Components of a Prompt	169
Prompt Templates.....	170
Advantages of Using Prompt Templates.....	173
Example Selectors.....	174

TABLE OF CONTENTS

Few-Shot Prompt Template.....	181
Crafting a Few-Shot Prompt Template for Question Answering	182
Output Parsers	191
Types of Output Parsers	192
Practical Example: Using PydanticOutputParser for Movie Data	193
OutputFixingParser	198
ChatPrompt Templates.....	199
Building the Chat Prompt Template	199
Case Study: Streamlining Customer Service.....	203
Initial Design and Customization of Prompts.....	205
Initial Design.....	205
Advanced Engineering.....	206
Impact	206
Key Takeaways.....	207
Review Questions	207
Answers.....	209
Further Reading	210
Chapter 6: Building Intelligent Chatbots and Automated Analysis Systems Using Chains.....	211
Introduction to LangChain Chains	211
What Are LangChain Chains?	212
Why Are Chains Important in Generative AI?	213
Understanding the Components of Chains.....	214
Higher-Level Components of a Chain App	214
Internal Components in a Step	216
Types of Chains.....	216
LCEL Chains.....	217
Legacy Chains	217

Difference Between LCEL and Legacy Chains..... 218

- LCEL Chain Example 218**
- Legacy Chain Example..... 218**

When to Use Different Types of Chains 219

- LCEL Chains 219**
- Legacy Chains 220**

Building with LCEL Chains 220

- Constructing LCEL Chains 221**
- Customizing LCEL Chains 222**
- Executing LCEL Chains 224**

Types of LCEL Chains 226

- Command Generation Using Query Constructor Chain 227**

Building with Legacy Chains 228

- Constructing Legacy Chains 229**
- Executing Legacy Chains 229**

Types of Legacy Chains..... 230

Building Real-World Apps with Legacy Chains 231

- Document Chatbot App Using ConversationalRetrievalChain 232**
- Building Text Generation Apps Using LLMChain 233**
- Building Conversational Apps with ConversationChain 233**
- Building Q&A Apps Using RetrievalQA 234**
- Document Processing App with MapReduceChain..... 235**

More Complex Workflow Apps Using Chain Composition Strategies 236

- Data Summarization App with Sequential Chains 237**
- SequentialChain Use Case Example 1: Customer Support Chatbot App 238**
- SequentialChain Use Case Example 2: Content Generation Pipeline App 239**
- SequentialChain Use Case Example 3: Automated Fraud Detection in Finance 240**

TABLE OF CONTENTS

Task Allocation App Using Router Chains	240
Sentiment Analysis App Using Conditional Chains	243
Advanced Chain Techniques	245
Handling Large Datasets with Chains.....	245
Dealing with Errors and Exceptions in Chains.....	247
Optimizing Chain Performance.....	248
Testing and Debugging Chains	249
Key Takeaways.....	250
Recap of Key Concepts.....	250
Future Possibilities with LangChain Chains.....	251
Glossary	252
Review Questions	253
Answers.....	256
Further Reading	257
Chapter 7: Building Advanced Q&A and Search Applications Using Retrieval-Augmented Generation (RAG).....	259
Importance of RAG	260
How Does RAG Work?	262
RAG Use Case Example	264
Try It Yourself.....	265
LangChain Components.....	266
Document Loaders	266
Document Loaders in Action.....	267
Working with PDFs	269
Dealing with CSV Files.....	270
Working with JSON Files	273
Text Splitters	275

Fully Working Code Example for Text Splitting 276

Recursive Splitting 279

CodeTextSplitter 281

Splitting by Token 283

Vector Stores..... 285

Text Embedding Models..... 287

 Code Walk-Through for Text Embeddings..... 289

 Caching the Embeddings..... 292

 Building the Information Retrieval System 296

 Calling the Vector Store Asynchronously 299

Retrievers..... 300

 Code Walk-Through for Information Retrieval 302

Indexing 303

Key Takeaways..... 305

Review Questions 305

 Answers..... 308

Glossary 309

References 312

Chapter 8: Your First Agent App315

 Introduction..... 315

 What Are Agents? 315

 Example of an Agent’s Workflow 318

 Agent’s Thought Process 319

 Why Agents Matter..... 321

 Agents for Content Generation 322

 Agents As Task Managers..... 324

 How Do Chains Differ from Agents? 325

 Choosing Your Approach..... 327

TABLE OF CONTENTS

- Your First End-to-End Working Agent App..... 328
 - Code Explanation..... 332
 - Results After Running the Code..... 334
 - Interpreting the Results..... 336
- Key Takeaways..... 338
- Review Questions 339
 - Answers..... 342
- Further Reading 342
- Chapter 9: Building Different Types of Agents 345**
 - Learning Objectives 346
 - Designing and Implementing an Agent 347
 - Defining the Agent’s Objective..... 347
 - Concepts..... 350
 - Agent 352
 - Agent Inputs 353
 - Agent Outputs..... 353
 - AgentExecutor 355
 - Tools and Toolkits 356
 - Considerations..... 362
 - Building an Agent Using LangGraph for Enhanced Capabilities 362
 - Agent Types..... 366
 - Criteria for Choosing Agent Types..... 366
 - Types of LangChain Agents..... 367
 - Tool Calling Agent 373
 - OpenAI Tools 378
 - Structured Chat Agent 382
 - ReAct Agent 387
 - Self-Ask Agents 391

Autonomous Decision-Making Capability..... 393

Intelligent Agent Performing Tasks with Multiple Tools 397

Creating a Retriever Tool 399

Putting It All Together 400

Choosing the LLM 400

Selecting the Prompt..... 401

Initializing the Agent..... 401

Creating the AgentExecutor 402

Differences Between LangChain v0.1 and v0.2 Agents 406

Key Takeaways..... 409

Review Questions 410

 Answers..... 412

Further Reading 413

Chapter 10: Projects: Building Agent Apps for Common Use Cases.415

 Creating a Custom Agent 415

 Loading the Language Model 416

 Defining Tools 416

 Creating the Prompt 417

 Binding Tools to the Language Model..... 417

 Creating the Agent..... 418

 Testing Your Agent 419

 Adding Memory 419

 Practical Use Cases for Agents 421

 Customer Support Automation 421

 Personalized Recommendations 428

 Real-Time Data Analysis and Decision-Making..... 439

Key Takeaways..... 447

Review Questions 447

TABLE OF CONTENTS

Answers.....	448
Further Reading	449
Chapter 11: Building and Deploying a ChatGPT-like App Using Streamlit	451
Setting Up Your Development Environment	452
Installing Streamlit Library	452
Installing Python.....	452
Installing Required Dependencies.....	457
Building the Streamlit LangChain UI App	457
Components of the Streamlit App.....	458
Steps Involved in Building the App	458
Indentation Error in the Code.....	461
Run Your Streamlit Application	462
Testing the App.....	464
Deploying the LangChain Application.....	465
Installing Git on Your System	465
Setting Up Your Identity	468
Set Up the OpenAI Key As an Environmental Variable	469
Resolving Sensitive Information Issues in Your Repository	472
Preventing Email Privacy–Related Issues.....	474
Deploying the App in GitHub.....	476
Providing Access to GitHub.....	477
Deploying in the Streamlit Cloud	480
Other Cloud Deployment Options.....	482
Key Takeaways.....	484
Review Questions	485
Answers.....	488
Further Reading	489

TABLE OF CONTENTS

Setting Up Your Development Environment.....489

Running Streamlit in Your Desktop489

Installing Streamlit Library489

Creating and Using the Language Model.....490

Handling User Input490

Chat History Initialization.....490

Deploying the Application491

GitHub Integration.....491

Index.....493

About the Author



Rabi Jay has over 15 years of experience driving digital transformation with a unique “acumen. His background as a Java and SAP ABAP developer provides insights into the enterprise systems LLMs often needed to integrate with. As a leader in Deloitte’s Dig’ital/Cloud Native practice, he has gained cross-industry experience applying AI solutions, positioning him to identify where LLMs offer the greatest potential for business impact.

He is passionate about making complex technology accessible, leading him to authoring the books *SAP NetWeaver Portal Technology and Enterprise AI in the Cloud* along with regular contributions to industry publications. His role as a technical reviewer for *Large Language Model-Based Solutions* and as Vice President at HCL America, focused on digital transformation, demonstrates his active engagement in the LLM field. Additionally, he runs a LinkedIn newsletter (“Enterprise AI Transformation”) and free LinkedIn course (“Generative AI for Business Innovation”).

About the Technical Reviewers



Murali Krishnan is a hands-on software leader and startup founder with a passion for building efficient and revenue-generating businesses. He has 20+ years of experience at Microsoft (B2B, B2C) followed by an executive stint at Starbucks (B2C). As a cofounder, Murali led fundraising, product, customer success, marketing, and sales effort at RapL, a mobile-first and AI-driven adaptive

microlearning platform. With 30+ US patents, he has a proven ability to innovate in cutting-edge tech (cloud, AI/ML, mobile, big data). Murali also mentors, assists, and advises startup founders. He lives in Seattle with his family. He enjoys running, biking, reading, and cooking.



Shivakumar Gopalakrishnan has over 25 years of experience in software development, DevOps, SRE, and platform engineering. He has worked in various industries, from healthcare enterprises to consumer-facing web-scale companies. He founded a startup, was a key architect within a Fortune 1000 company, and is currently a Principal Architect at BD. He is a coauthor of *Hands-on Kubernetes*

on Azure and the author of *Kubernetes for Job Seekers and Modern Python Programming using ChatGPT*.

ABOUT THE TECHNICAL REVIEWERS



Keerthi Bharath is an accomplished thought leader in the AI industry. He has experience running numerous startups and leading AI projects in MNCs. He is also an investor and mentor to companies. He is an alumnus of Syracuse University, New York, and College of Engineering, Guindy, Chennai.

CHAPTER 1

Introduction to LangChain and LLMs

Welcome to the world of LangChain and LLMs, where you will learn how to build generative AI applications using one of the most popular generative AI application development frameworks, namely, LangChain. You will learn how to tap into the vast knowledge of these highly capable large language models, or LLMs, as we often call them. Together, we are going to explore how powerful LLMs like GPT-4, PaLM, and Gemini can be accessed with LangChain to develop some amazing, intelligent, and real-world applications that feel almost human-like.

The power of LangChain lies in its ability to make the power of large language models (LLMs) easily accessible to us to build real-world applications. Whether you are a veteran coder or just starting out, you are going to find LangChain refreshingly easy to use. It is that ease of coding that got me hooked onto LangChain. I hope you will be attracted to it as well once you start discovering how easy it is, as you start learning from practical examples throughout the book. The beauty of it is that you don't even need to be a machine learning guru or data science expert to leverage its capabilities.

By the end of this chapter, I am confident you will master the essentials of LangChain and start developing your own LLM-driven generative AI applications.

I hope you find this text to be both comprehensive and hands-on. My goal is for you to not only understand LangChain and LLM theory but also to apply this knowledge practically to bring your generative AI projects to life.

Understanding LangChain

LangChain is a powerful framework that will help you develop artificial intelligence applications based on LLMs easily. Let us take a closer look.

The official definition of LangChain on the LangChain.com website is like this:

***LangChain** is a framework for developing applications powered by language models. It enables applications that:*

***Are context-aware:** connect a language model to sources of context (prompt instructions, few shot examples, content to ground its response in, etc.)*

***Reason:** rely on a language model to reason (about how to answer based on provided context, what actions to take, etc.)*

LangChain is essentially a digital toolbox that you can use to build amazing, intelligent applications that can talk, understand, and even think like a human to some extent.

Here are some benefits:

- You can tap into the vast knowledge of advanced language models like GPT-4, PaLM, Gemini, or even open source models such as LLaMA. This opens up a world of possibilities for the types of applications you can develop.
- You can integrate these LLMs with your own specific, private data. This means you can tailor the LLM's output more closely to the unique needs and contexts of your business or project.
- And here is the good news. You are not limited to any specific LLM, and you can mix and match different models as needed. This allows for a level of customization in generative AI application development that can truly drive innovation.

LangChain provides the tools you need irrespective of whether you are looking to develop chatbots that enhance customer service, systems that generate creative content, or solutions that automate repetitive tasks. It is very inspiring to see the many ways we can apply this technology to solve real-world problems and drive innovation across various industries.

A Simple Generative App Using LangChain

Here is a simple, illustrative example of a basic application that uses GPT-4 to generate creative content based on user input. It may seem like a trivial example, but I am sharing this just to illustrate how easy it is to call an LLM. We will go deeper into these topics in future chapters. For now, just skim through this to get a rough idea of what is in store for us ahead.

Install the LangChain and OpenAI modules

```
!pip install openai==0.28.0  
!pip install LangChain==0.1.20
```

Import LangChain and OpenAI libraries

```
from langchain.llms import OpenAI  
from langchain.chains import LLMChain  
from langchain.prompts import PromptTemplate
```

Initialize the OpenAI model with LangChain

```
LLM_OPENAI_API_KEY="your_openai_api_key"  
llm = OpenAI(api_key=LLM_OPENAI_API_KEY)
```

Define a prompt template for generating story ideas

```
prompt_template = PromptTemplate(  
    input_variables=["user_input"],  
    template="Generate a creative product idea based on the  
    following industry input: {user_input}",  
)
```

User input for the theme

```
user_input = "eco-friendly home appliances"
```

Now, initialize the LLMChain with the prompt

```
chain = LLMChain(llm=llm, prompt=prompt_template)
```

Generate the creative content

```
response = chain.run(user_input)
```

Print the response

```
print("Creative Story Idea:", response)
```

Figure 1-1 shows the output I got after running this code when using the Google Colab. You can use the code attached to this chapter on GitHub to test this yourself.

```

+ Code + Text
[20] # Now, initialize the LLMChain with the prompt
chain = LLMChain(llm=llm, prompt=prompt_template)

[21] # Generate the creative content
response = chain.run(user_input)

[22] # Print the response
print("Creative Story Idea:", response)

Creative Story Idea:
Title: "Beyond the Stars"
In a distant future, humanity has expanded its reach beyond the confines of Earth and has discovered new worlds to explore. Among these new frontiers, lies an uncharted planet. A team of astronauts, led by the fearless Captain Alex, embark on a journey to this uncharted planet. Equipped with state-of-the-art technology and driven by the quest for discovery, they approach the planet, where they are met with a dazzling display of colors and an otherworldly aura. The surface of the planet is covered in lush vegetation and exotic creatures that seem to possess incredible abilities. They also discover ancient ruins and artifacts. As they begin to explore, the team encounters strange and exotic creatures that seem to possess incredible abilities. They also discover ancient ruins and artifacts. But their journey takes a dangerous turn when they come across a group of hostile beings who see the arrival of the humans as a threat to their way of life. The team must fight to survive as they delve deeper into the planet's mysteries.

```

Figure 1-1. Output After Running the Code

Explanation

Below is an explanation for the code:

- First, we install the required modules: OpenAI and LangChain. Please ensure that you download the exact versions used in this code; otherwise, it is quite likely that the code may not work.

Note To check what versions have been downloaded, use the commands below for checking versions of OpenAI and LangChain libraries, respectively.

```
!pip show openai
!pip show langchain
```

- Then, we import necessary classes from LangChain: OpenAI, LLMChain, and PromptTemplate.
- Initialize the OpenAI model with the API key. Please refer to the “Step 1: Obtaining OpenAI API Keys” section in Chapter 2 to obtain the OpenAI key.

- Define a prompt template for generating story ideas, using `user_input` as the input variable.
- Set the value of `user_input` to “eco-friendly home appliances.”
- Create an instance of `LLMChain` with the initialized OpenAI model and the prompt template.
- Generate the creative story idea by calling the `run` method of `LLMChain` with `user_input`.
- Print the generated story idea with the label “Creative Product Idea:”.

This example demonstrates how easy it is to integrate LLMs into applications using LangChain and hopefully gives you a glimpse into the framework’s ability to abstract and streamline the process.

Now that we have briefly looked at LangChain, let us explore what LLMs are and why they are important.

What Are LLMs, and Why Are They Important?

As mentioned in our learning objectives, you need to understand the capabilities of LLMs, like GPT-4, PaLM, and Gemini, well to develop powerful generative AI applications.

LLMs are like a robot that has read almost everything available on the Internet, such as books, articles, and websites. At the same time, they can write or chat about any topic under the sun. These “super-reading robots” allow computers to comprehend and generate text that is remarkably human-like.

The power of LLMs, though, extends far beyond simple conversations. They are transforming various fields by

Translating languages seamlessly, breaking down communication barriers

Writing stories and creative content and tapping into a well of artificial creativity

Summarizing large texts and condensing information quickly and efficiently

Generating code to help developers automate routine coding tasks

LLMs have become so powerful now that you can easily develop highly intelligent generative AI applications across multiple domains, starting from enhancing customer service automation to supporting creative endeavors like writing and design.

Examples of LLMs

Here are a few standout examples of LLMs that we will explore more deeply in Chapter 4.

GPT-4: GPT-4 is developed by OpenAI, and you can generate essays, create poems, and even produce code based on the prompts you provide. It is known for its robust performance across diverse benchmarks and is a powerhouse when it comes to complex reasoning tasks.

PaLM: PaLM was created by Google, and it excels in language understanding and generation. It is particularly adept at tackling intricate mathematical problems and explaining scientific concepts. It is known for impressive scalability and versatility across various tasks.

Gemini: Also from Google, Gemini is a multimodal LLM capable of processing and understanding different types of data, such as text, images, videos, audio, and code. Its general-purpose nature empowers you to build a vast array of applications that can push the boundaries of what LLMs can achieve.

Why Is LangChain Important?

Now that you know what LangChain and LLMs are, let us discuss why LangChain is important.

Simplifies LLM App Development and Boosts Productivity

When you are developing applications that use large language models (LLMs), you may feel overwhelmed due to the complexity involved in integrating data, training models, and implementing complex business logic. However, LangChain can simplify this entire process for you dramatically:

Real-World Example: For example, when building a conversational AI assistant that can answer questions about your company's products and

services, you would have to write complex code to integrate with various data sources, such as databases, APIs, and document repositories, and then meticulously implement prompts and business logic to ensure accurate and relevant responses.

LangChain Advantage: With LangChain, instead of developing code for complex data integrations, you can simply leverage LangChain's built-in tools to connect your generative AI application to virtually any data source, such as a SQL database or a cloud storage service provider, with just a few lines of code.

Modular and Scalable Architecture

LangChain stands out due to its modular, extensible architecture that allows you to easily customize and combine components to build complex applications quickly, much like assembling LEGO blocks. And because everything is standardized, you don't need to worry about whether one piece will work with another. This makes generative AI app development a breeze!

Development Efficiency: The modular approach improves your coding speed with relatively less effort because instead of programming complex business logic from scratch, you can simply leverage the reusable templates and libraries provided by LangChain. You can also easily combine different AI functionalities and features making your project scalable and also adaptable to future advancements in generative AI capability.

Open Source Support and Community Collaboration

LangChain is more than just a framework. It is also supported by a vibrant open source community. With LangChain, you have access to an incredible community of like-minded individuals like you who are passionate about generative AI development and committed to helping everyone succeed. Moreover, you also have the opportunity to contribute to open source projects by submitting bug reports, proposing new features, or even contributing code directly to the codebase. Let me illustrate here one way you can benefit from this open source community.

Example Consider you are developing an educational platform. While you could construct this platform directly using LLM APIs, you would need deep expertise in AI-driven educational best practices, model training, and user interaction design.

LangChain Advantage The advantage with LangChain is that instead of spending all your time and energy building the application from the ground up, you can simply leverage the already existing templates, contributed modules, and data connections. You can also choose to develop it as an open source project, thus leveraging the collective expertise and contributions of a world-class, global community of developers and educators.

You can also tap into the best practices from the LangChain ecosystem and collaborate with others to speed up the development of your education platform. More importantly, you can leverage tools specifically optimized for educational content, thus significantly enhancing its features and personalization capabilities.

No Cost Barrier

LangChain is completely free to use for all, including individuals, startups, or large enterprises. You don't need to worry about expensive licenses or restrictive terms, which lowers the entry barrier for many innovators and creators.

In summary, LangChain sets a new standard in the world of LLM application development. It caters to the growing interest in generative AI technologies and delivers substantial benefits such as simplicity, increased developer productivity and speed through reusable code, and rapid innovation through open source collaboration.

Real-World Examples of LangChain

With LangChain, you can solve real-world problems easily. You can build complex generative applications quickly using techniques such as prompt engineering, RAG (Retrieval-Augmented Generation), data-aware decision-making, context awareness, and agent tools.

Personalized Responses Through Prompt Engineering

When using LLMs, you may have realized that they provide generic responses, which is an issue, especially when dealing with domain-specific queries. For instance, even though an LLM might provide a rough estimate of the price of a computer, it won't be able to determine the cost of a particular model that your business sells unless it has access to that internal data.

This is where you can use prompt engineering combined with RAG (Retrieval-Augmented Generation). With prompt engineering, you can craft queries that not only guide the model to understand the task but

also provide it with the context needed to generate accurate responses. Using the RAG technique, you can provide the LLM with the necessary context and data for the task at hand, so that it can respond accurately and effectively to specific prompts or queries. Let me illustrate this with the example below.

Example Custom Chatbot Development

Scenario Consider a scenario where your company wants to develop a chatbot. You want this chatbot to go beyond the basic customer service functionalities and tackle complex queries specifically related to your company's product line and services. You have realized that when using traditional approaches like direct API integration with an LLM, responses are way too generic, given the fact that the model is only trained on generic Internet data.

LangChain Advantage To address this challenge, you decide to use the power of prompt engineering to construct highly specialized prompts. You designed these prompts to guide the LLM to understand the nuances of your company's unique market. They help the chatbot provide responses that are more precise and custom-tailored to your company, thus significantly improving the quality of customer interaction.

The chatbot could now go into more specifics, such as detailing the product specifications, checking availability, and recommending support options for different models of their products.

Building Data-Aware Decision-Making Apps

“Data aware” means your application can interact with various data sources and incorporate contextually relevant information into its responses. As you will see later in Figure 1-2, the integration of data connections enables your applications to pull in real-time data or reference-specific datasets to provide accurate outputs.

Case Study Let us consider a scenario where you are building a virtual assistant for the financial analytics domain. The assistant needs to understand user queries, access real-time market data, and provide insights.

Traditional Approach Drawbacks Imagine the amount of custom development you need to handle when using direct API integration with the LLMs. You would need to manually implement data fetching, processing, and integration logic within the assistant’s workflow, which is a time-consuming and error-prone process.

LangChain Advantage You can use the LangChain framework’s data connections component to seamlessly integrate external data sources. It takes care of the complexity of fetching and incorporating external data into the LLM’s context to provide more accurate, up-to-date financial insights.

Note The data connections component of LangChain not only reduces development time but also ensures your application can adapt to new data sources or changes in data structure with minimal adjustments.

Building Context-Aware Applications

Context-aware applications are those that can interpret the context in which they operate and then respond accordingly. LangChain achieves this goal by using data about the environment in which it operates or by using user interactions to offer personalized user experiences.

Why Does It Matter?

It matters because it allows your application to adapt to changing environmental conditions or user preferences, thus enabling highly personalized responses that are more relevant and effective.

Example Case Study: AI Tutoring System Develop a tutoring system that helps students with math by adjusting the difficulty level based on the student's interaction history and other contextual factors.

First, you will need to build contextual awareness. As the student interacts with the system, the application must capture the topics that were difficult for the student, bucket the student into a particular difficulty category, and then adjust the difficulty level of problems presented to the student in the future. It might also use the time of day, the accuracy of their responses, or the speed at which the student answers to adapt its teaching style.

LangChain then makes it easy to provide personalized questions to the student through the use of prompt templates by passing in the difficulty level and time of the day as variables to get the appropriate LLM response. You will be able to understand this better once you learn prompt templates in Chapter 5.

Below is a sample code to illustrate this. Don't worry, we will discuss this more in Chapter 5, but this is just to give you a feel of what is coming ahead.

Retrieve student's interaction history

```
difficulty_level = get_student_difficulty_level(student_id)
time_of_day = get_current_time_of_day()
accuracy = get_student_accuracy(student_id)
```

Generate personalized questions based on context

```
prompt_template = PromptTemplate(
    input_variables=["difficulty_level", "time_of_day",
                    "accuracy"],
    template="Generate a {difficulty_level} math problem
            suitable for {time_of_day} study, considering the student's
            accuracy of {accuracy}."
)
personalized_question = chain.run(prompt_template,
                                   difficulty_level=difficulty_level,
                                   time_of_day=time_of_day,
                                   accuracy=accuracy)
```

Developing RAG-Based Apps

With LangChain, you can help develop advanced applications through a feature known as Retrieval-Augmented Generation (RAG). This innovative capability allows you to introduce new, relevant information into the LLM's response process.

How RAG Works: Using RAG is like having the LLM consult a vast library to provide the most accurate answer. This feature is essential for decreasing errors (often referred to as “hallucinations”) and improving the accuracy of the data provided by your LLM applications.

Examples of RAG in Action

Research Assistant A research assistant tool can dig through extensive scientific literature to retrieve relevant information, which it then incorporates into the LLM’s responses. This capability can transform the way research is conducted by streamlining how knowledge is acquired during research.

Legal Analysis Tool A legal analysis tool can consult case law and legal precedents to provide well-informed legal opinions and recommendations. Such a tool could revolutionize the legal research process by making it more efficient and comprehensive.

Market Analysis Tool for Investment Firms Another example is a market analysis tool that can dynamically aggregate the latest market data and news articles to provide investors with summaries that reflect current market trends and actionable insights.

Building Agentic Apps

LangChain introduces the powerful concept of “Agentic” applications, which are applications that pretty much mimic human behavior. They can autonomously perform tasks and make decisions on their own to achieve specific goals. This capability allows applications to not only process information but also act independently based on AI-driven logic.

By combining LangChain Agents with the Chains component, you can build complex applications with multistep operations that are not only robust but also flexible. This capability allows your application to evolve based on user feedback and changing requirements without extensive redevelopment. This is a very exciting feature of LangChain and holds a lot of promise. We will be discussing this in greater detail in Chapters 8–10.

Let us review some interesting use cases.

Example: Transforming Customer Support Systems

A prime example of this innovation is how customer support systems have transformed. Traditionally, to create an AI-driven support system, you will have to build complex workflows and integrate multiple steps that involve handling inquiries, searching databases, and generating personalized responses. This is all made significantly easier with plug and play components from LangChain.

Example: Automated Content Scheduler

Let us take another example of a digital marketing company that wants to build a tool that aims to autonomously schedule and post marketing content across various platforms. It involves analyzing engagement data to determine the optimal posting times. And you want to achieve a significant boost in audience reach and interaction with minimal manual effort.

This is where the agent comes in. As you can see, a number of steps need to be taken to achieve these goals. The agent will not only streamline content management but also ensure content is delivered when it is most likely to make an impact.

Increased Development Productivity

As a developer, you can streamline your coding process and save precious time by avoiding repetitive tasks, including generating code snippets, or creating documentation.

Example: Generating Code for a Website App

Let us say you are building a web application, and you need to code for a bunch of API endpoints for CRUD operations. Normally, you would have to manually write the code for each endpoint, which can be time-consuming and repetitive.

Here is how LangChain can come to your aid:

- You can first start by creating a blueprint, which contains all your API endpoints and their corresponding HTTP methods in a YAML or JSON file. This would act as a road map for LangChain to follow.
- Then you can develop a prompt that guides LangChain with specific instructions on how to generate the code for each endpoint. You can generate functions or even entire modules based on high-level descriptions.
- You can create documentation, comments, and annotations automatically.
- You can automate code refactoring, optimization, and bug fixing by providing instructions in natural language.

- You can integrate with various programming languages, frameworks, and libraries.
- Finally, it is a matter of you taking the generated code snippets and integrating them seamlessly into your project structure.

LangChain Advantage

Whether you need to generate code, create documentation, refactor your codebase, or debug, LangChain will help you. It is a one-stop shop for all your coding needs.

But more importantly, you don't need to be a coding genius to use LangChain. The LLMs will also help you to adapt the code according to your unique coding style and project requirements. You can customize and fine-tune it to fit seamlessly into your development workflow.

Why These Features Matter

You can leverage these advanced features, such as RAG and agentic applications, to build generative AI applications that are more precise, intelligent, and autonomous. They allow you to reap the following benefits:

- **Enhance Accuracy and Reliability:** By integrating real-time data, your generative AI applications can offer more accurate and contextually relevant responses.
- **Automate Complex Workflows:** Agentic applications reduce the need for manual intervention in processes, allowing for more efficient resource allocation and better scalability.
- **Adapt to User Needs and Preferences:** Your applications can dynamically adjust based on user interactions and feedback, providing a personalized experience that improves over time.

LangChain’s robust framework, equipped with these advanced features, sets a new standard that you can leverage to build some amazing generative AI applications. It is an indispensable tool that you should take advantage of.

Integrating LLMs with LangChain

Now that we have taken a quick look at LLMs and the LangChain’s capabilities, it is time to take a closer look at how LangChain makes it all happen by acting as a crucial bridge between developers and large language models (LLMs).

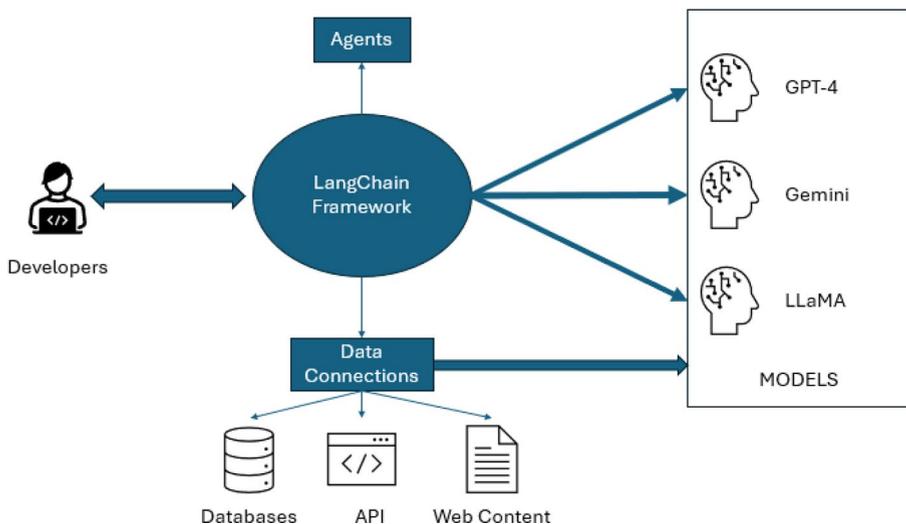


Figure 1-2. LangChain Framework Overview

Figure 1-2 visually represents the structure and workflow of the LangChain framework. It shows how the core components of the LangChain framework interact with each other during the creation of AI-driven applications:

LangChain Component: At the heart of the diagram is the LangChain component, which serves as the pivotal core in our framework. Using this component through its interface will simplify the complexity of interacting with various LLMs.

Developers: On the left side of the diagram, you will notice the representation of developers. You will interact with the LangChain component through a set of well-defined APIs or direct code integration. You will use this component to provide specific requirements, prompts, and instructions that guide the behavior of LLMs.

LLMs (GPT-4, PaLM, Gemini): Shown on the right are the LLMs that can be leveraged by LangChain to perform tasks ranging from text generation to complex reasoning.

Models: The models component abstracts the complexities of different LLMs through a unified API available within the LangChain library. This abstraction allows you to work with multiple LLMs without worrying about the underlying intricacies.

Data Connections: Data connections help LangChain fetch the latest or most relevant information available on the Internet or in databases to make the LLM responses more useful and accurate.

Agents: LangChain Agents represent innovative tools that enable your applications to perform tasks, make decisions, and interact with external systems autonomously based on AI-driven logic. Using these agents, you can develop applications that can operate independently in a variety of environments.

Each arrow in the diagram illustrates the flow of data and control, highlighting how you can leverage LangChain to build sophisticated, context-aware applications. This flow shows how every component, from the developer's input to the LLM's output, works in harmony to create responsive and intelligent generative AI applications.

Simplified Integration with Multiple LLMS

As can be seen in Figure 1-2, LangChain models are the abstracted versions of these powerful LLM models that help you easily manage the intricacies related to each model's API. The following example explains how it simplifies integration with multiple LLMS.

Example Scenario: Building a Content Generation Platform Let us consider a project where you are building a content generation platform using LLMS to produce articles, summaries, and reports. If you planned on integrating directly with each LLM API, such as GPT-3, GPT-4, and PaLM, you would have to solve for multiple integration points. Each model would require its own unique setup, which would complicate your codebase and significantly increase maintenance efforts.

LangChain's Streamlined Solution

LangChain comes to your rescue by abstracting these complexities through a unified interface. This interface provides several key advantages.

- It allows you to **easily switch between LLMs** or even use multiple models in tandem without making changes to the frontend or business logic. This is because the underlying API calls, data formatting, and response handling are managed by LangChain.
 - It enables your content generation platform to **leverage the strengths of different LLMs** for various content types. Your code quality and efficiency improve along the way, enabling you to accommodate more complex use cases easily. For example, you might use GPT-4 for its sophisticated narrative creation in articles and PaLM for its analytical abilities in generating detailed reports.
 - **Improving Code Quality and Efficiency:** The unified interface not only simplifies the development process but also enhances the maintainability and scalability of your application. It reduces the likelihood of errors and decreases the time spent on debugging and testing.
 - **Accommodating Complex Use Cases:** Your content platform can promote innovation by easily adapting to more complex scenarios.
-

Exploring Core Components of LangChain

Now that you have a clearer understanding of how LangChain makes it easy to integrate with multiple LLMs, it is a perfect time to explore the core components shown in Figure 1-3.

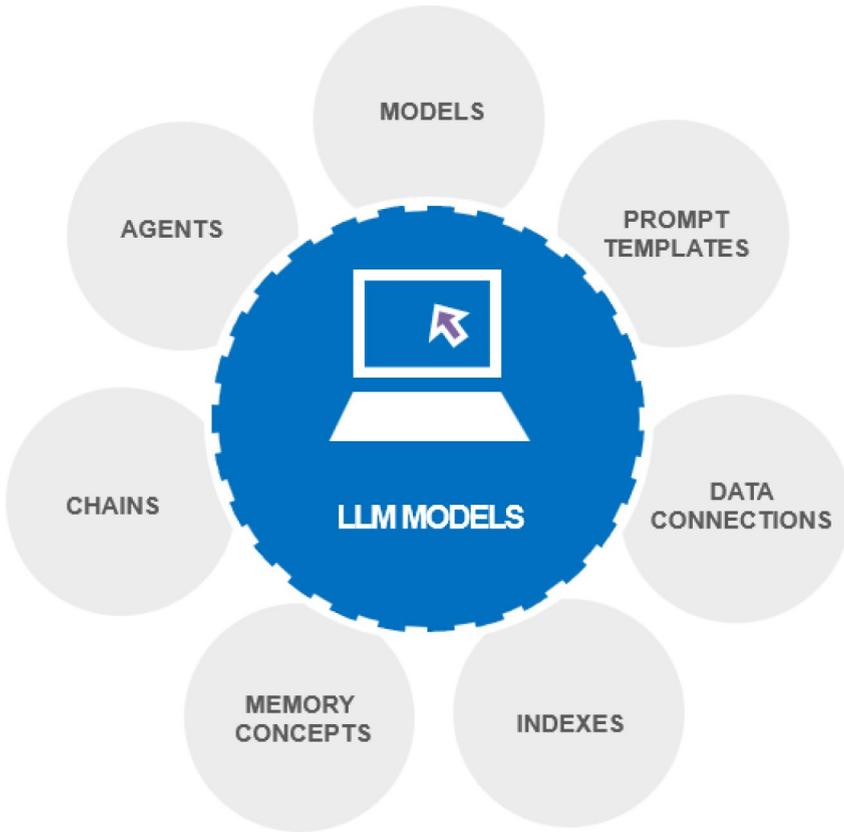


Figure 1-3. *Building Blocks of LangChain LLM Applications*

Models: At the heart of any LLM application are the models. You will learn how to connect with powerful language models like GPT-4 in your applications to create LLM apps. Sure, you could do this with the default LLM APIs, but like we discussed earlier, LangChain standardizes the process, making it easy to switch between different LLMs without rewriting your code.

Prompt Templates: I will teach you the art of creating dynamic prompts that make your language models understand and respond to queries effectively. They guide the language models by specifying the task at hand along with the context. By crafting effective prompts, you can get more accurate and relevant responses from the models, thus allowing you to tailor the output to your unique needs.

Data Connections: Data connections component allows you to feed your LLMs with the right information by connecting them to various data sources, like documents, PDFs, or even vector databases. We will explore techniques like indexing and embedding to make your data retrieval straightforward and efficient for language models.

Indexes: Indexes are all about organization. They transform large datasets into neatly arranged data libraries that your application can query effortlessly. This setup not only speeds up information retrieval but also enhances the overall performance of your LLM applications.

Memory Concepts: Understanding memory concepts is important when building applications that require ongoing interactions. This component helps maintain historical context across conversations, which allows the language models to remember previous exchanges. This continuity is key to providing a coherent and seamless user experience over time.

Chains: Chains are where things get even more interesting. You can use chains to link sequences of operations or models to execute complex, multistep tasks. This functionality is crucial for handling sophisticated processes within your applications and to make informed decisions based on a series of interactions.

Agents: Finally, we reach Agents. These are the advanced units within LangChain that bring together all the previous components. Agents are capable of executing tasks, making decisions, and interacting with external systems autonomously. They use APIs, databases, and custom scripts to reason through tasks and execute actions based on sophisticated logic.

Building Your Knowledge Step by Step

Given the power and capabilities of Agents, it is possible that you might be tempted to jump ahead with Agents, but I believe a foundational understanding of the other components is crucial. That is why I will take you through LangChain with a methodical and thorough approach to ensure that you grasp each component first before moving to the next. Along the way, we will be looking into code examples and explanations.

LLM Application Development Workflow

Now that we have seen why LangChain can be a game changer when developing LLM apps, it is time to take a quick look at the LLM application development workflow. Feel free to customize it to your needs.

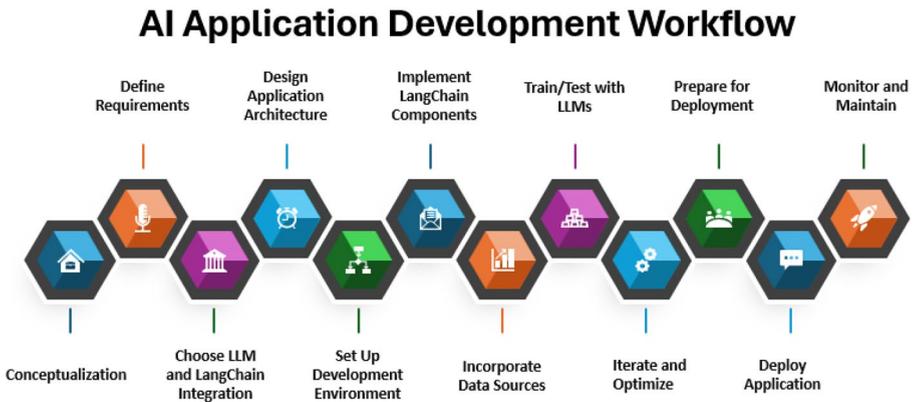


Figure 1-4. LLM App Development Workflow

This workflow displays the phases from ideation to deployment, ensuring you have a clear road map for creating LLM applications.

Conceptualization: Every great application begins with a concept. At this stage (see Figure 1-4), you need to identify a problem that your generative AI application will solve. This could range from automating mundane tasks to extracting and analyzing insights from complex datasets.

Just like in traditional SDLC (Software Development Lifecycle), what is important is to clearly define what success looks like for your project. However, the success metrics here are tailored toward LLM performance measures such as accuracy, relevance, and response time.

Mini-Exercise: Write down three innovative applications you believe could be enhanced or made possible with LLMs. Share your ideas in your development community forum or discuss them with peers to explore their feasibility.

Define Requirements: Once you have a clear idea of the goals for your generative AI application, the next step is to spell out the specific requirements of your application such as the type of LLMs needed, data sources, user interactions, and specific AI functionalities like sentiment analysis or entity recognition.

Choose LLM and LangChain Integration: This phase is similar to the technology stack decisions you would make in traditional SDLC, except that instead of choosing technology stack components such as databases, programming languages, and frameworks, you will be focused on selecting appropriate large language model(s) such as GPT-4 or PaLM based on your application's needs. You should also decide how LangChain will interact with these models to ensure scalability and easy model maintenance.

Design Application Architecture: Then we move on to the Design Application Architecture phase (see Figure 1-4), where you transition from abstract concepts to a detailed, structured design of your application. This is when you will start planning how LangChain components such as models, data connections, and agents will be combined to meet your application's objectives. Your architecture should address how to integrate LangChain components, focusing on how the data flows through LLMs, how models are orchestrated, and how AI responses are integrated back into the application logic.

How well you design the app will determine the scalability, maintainability, and performance of the application when done. This is quite similar to traditional SDLC except that in traditional SDLC, you would be focusing more on various aspects like client-server interaction, database design, and service-oriented architecture.

Set Up Development Environment: This is where the rubber meets the road, as you will start to prepare your development environment, including necessary software, tools, and access to LLM APIs.

Implement LangChain Components: Now, the exciting part – coding! You will start integrating LangChain’s components like models, data connections, and agents to build the functionality you have designed. This book is all about this step. I will be providing you with practical examples and guidance. This phase differs from traditional software programming in the sense that you are not coding for software functionalities but for LLM interactions.

Incorporate Data Sources: During this phase, you will integrate with external data, which will provide your application with the much-needed context and relevance, as depicted in the Incorporate Data Sources step of this workflow diagram (Figure 1-4). This stage emphasizes the need for applications to pull in data from various sources, be it databases, APIs, or live feeds. This phase differs from traditional SDLC in the sense that you will also need to focus on how data will be used to train and fine-tune the LLMs.

Train/Test with LLMs: If necessary, you may need to perform any training or fine-tuning of the LLMs using LangChain. Make sure you test the application thoroughly to ensure it meets your specified requirements. This differs from traditional testing in the sense that you will not only check for bugs but also ensure that the LLM is generating correct and contextually appropriate responses.

Iterate and Optimize: Use the feedback from your testing to iterate on your application's design and functionality. Focus on optimizing LLM performance, making LLM configuration changes, tweaking LangChain setups, enhancing usability, and even retraining models with new data.

Prepare for Deployment: It is time now to finalize your application for deployment. This may involve securing permissions, final testing, and preparing deployment scripts.

Deploy Application: The Deploy Application phase, highlighted in the workflow diagram (Figure 1-4), marks the important stage where you deploy the LLM application for real-world use. This is when you prepare the application for launch. You need to ensure the app is robust, secure, and ready to handle user interactions. You need to take care of things like selecting the right hosting environment, scalability to accommodate growth, and implementing monitoring tools for ongoing performance evaluation.

Monitor and Maintain: After deployment, ensure that you are continuously monitoring the application's performance and user interactions. Be proactive in making necessary updates and improvements to enhance functionality and user satisfaction. You may be required to make ongoing adjustments to the model given the dynamic nature of the data and the potential drift in LLM behavior.

Key Takeaways

Now that we have come to the end of the chapter on “this chapter” let us discuss some key takeaways.

Understanding LangChain: You learned that LangChain is a powerful framework that makes working with LLMs (like GPT-4, PaLM, and Gemini) much easier. Its modular, standardized design promotes scalability and adaptability throughout development.

The Power of LLMs: You also learned that large language models are sophisticated AI systems that can understand and generate human-like text. They bring an unprecedented level of sophistication and context awareness and have revolutionized how we interact with technology. LLMs can create content, summarize information, and even assist in coding.

LangChain's Building Blocks: You also learned the core components of LangChain, such as models, prompt templates, data connections, memory concepts, chains, and agents. These elements work together to build intelligent LLM applications, which is really the scope of this book.

The chapter sets the stage for us to explore deeper into LangChain and LLMs, and in the next chapter, we will be exploring the advantages of using the LangChain framework vs. traditional LLM API development approaches.

Review Questions

These questions are meant to help you assess and reinforce your understanding of key concepts.

1. What is the primary purpose of LangChain?
 - A. To replace existing LLMs with more advanced models
 - B. To facilitate the development of applications that use LLMs
 - C. To decrease the accuracy of LLM applications
 - D. To serve as an independent AI without LLM integration
2. Which component of LangChain allows it to integrate and use external data sources?
 - A. Prompt templates
 - B. Chains

- C. Data connections
 - D. Memory concepts
3. What advantage does LangChain offer when managing multiple LLMs?
 - A. Increased complexity in managing integrations
 - B. Limited flexibility with API integrations
 - C. Simplified switching and integration between different LLMs
 - D. Reduced ability to customize LLM interactions
 4. Which of the following best describes “prompt engineering” in the context of LangChain?
 - A. Disconnecting prompts from LLMs to enhance security
 - B. The process of creating ineffective prompts to test LLM robustness
 - C. Crafting inputs to guide LLMs to generate specific or desired outputs
 - D. Ignoring the context in prompts to simplify the development process
 5. How does LangChain enhance context-aware applications?
 - A. By completely automating the coding process
 - B. By ignoring user input to streamline interactions
 - C. By connecting language models to sources of context
 - D. By reducing the accuracy of language model responses

6. Which feature of LangChain helps manage complex workflows and decision-making processes?
 - A. Data connections
 - B. Agents
 - C. Memory concepts
 - D. Chains
7. What is a significant benefit of using LangChain for AI application development?
 - A. Increased development time and costs
 - B. Reduced flexibility in using LLMs
 - C. Simplified integration and enhanced scalability
 - D. Requirement for extensive technical knowledge in LLMs

Answers

1. B. The primary purpose of LangChain is to facilitate the development of applications that use LLMs.
2. C. Data connections allow LangChain to integrate and use external data sources.
3. C. LangChain offers the advantage of simplified switching and integration between different LLMs.
4. C. “Prompt engineering” in LangChain involves crafting inputs to guide LLMs to generate specific or desired outputs.
5. C. LangChain enhances context-aware applications by connecting language models to sources of context.

6. D. Chains help manage complex workflows and decision-making processes within LangChain.
7. C. A significant benefit of using LangChain for AI application development is simplified integration and enhanced scalability.

Looking Ahead

In the next chapter, you will learn about the challenges when calling the LLM API directly and how LangChain addresses those challenges to enhance your generative AI development experience and productivity. We will be

- Looking into the hurdles of direct LLM API interaction
- Exploring practical comparisons, case studies, and exercises demonstrating LangChain's benefits
- Understanding LangChain's architecture for seamless integration, scalability, and innovation

Bring Your Ideas to Life with LangChain

Now that you have a better idea of LangChain and LLMs and how they come together to build some amazing, powerful generative AI apps, the question I have for you is, what can you do to bring your ideas to life? Here are some suggestions:

Continue Your Learning: This chapter is just the beginning. In the next few chapters, we will go deeper into each component of LangChain, explore advanced features, and keep experimenting with new project ideas.

Take the First Step Today: Visit the LangChain GitHub repository for code samples for the latest updates and inspiration.

<https://github.com/langchain-ai/langchain>

Join the LangChain Community: Jump into the LangChain community forums or follow them on social media to connect with other LangChain enthusiasts. Share your ideas, ask questions, and discover new ways.

<https://discord.com/invite/6adMQxSpJS>

Glossary of Technical Terms

Below is a list of definitions of technical terms used in this chapter:

LangChain: A framework for developing applications powered by language models. It enables the creation of context-aware and reasoning applications by connecting language models to various sources of context.

Large Language Models (LLMs): Advanced AI systems capable of understanding, generating, and interacting with human language. They are trained on vast datasets to perform a wide range of language-related tasks.

GPT-4: A version of the Generative Pretrained Transformer, developed by OpenAI, known for its ability to generate coherent and contextually relevant text based on a given prompt.

PaLM: A language model developed by Google that excels in understanding and generating language, as well as solving complex problems across various domains.

Gemini: An example of a multimodal LLM model that processes and understands multiple data types, including text, images, and videos, for a broad range of applications.

Prompt Engineering: The practice of crafting input prompts to guide language models in generating specific or desired outputs. It involves structuring the prompts to provide the necessary context for accurate and effective responses.

Data Aware: The ability of an application to interact with and incorporate data from various sources, ensuring that responses or actions are informed by relevant and current information.

Agentic: Refers to applications that can perform actions or tasks autonomously, often mimicking human behaviors, such as searching the Web, interacting with databases, or completing forms.

Retrieval-Augmented Generation (RAG): A technique that introduces new information into the language model during the prompting process to reduce inaccuracies and improve response quality.

Tokenization: The process of breaking down text into smaller units (tokens), such as words or phrases, for easier processing by language models.

Further Reading

Below is a list of resources to help you solidify your understanding of the topics covered in this chapter and to further explore the world of LLMs and LangChain:

“Better Language Models and Their Implications”

(OpenAI Blog): While focused on GPT-2, this article provides valuable insights into how LLMs understand and generate text, serving as a foundation for prompt engineering. <https://openai.com/research/better-language-models>

“Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks” (Google Research Blog): Explores the concept of Retrieval-Augmented Generation (RAG) with detailed examples and applications in knowledge-intensive tasks. <https://arxiv.org/abs/2005.11401>

OpenAI API Documentation: The official documentation for OpenAI’s API, covering everything from authentication to request parameters. Essential reading for anyone building applications with LLMs.

<https://platform.openai.com/docs/introduction>

Google Gemini LLM Documentation: This contains some notebooks, tutorials, and other examples to help you get started with Gemini models.

<https://cloud.google.com/vertex-ai/generative-ai/docs/learn-resources>

CHAPTER 2

Integrating LLM APIs with LangChain

In this chapter, we will build upon the foundational knowledge from Chapter 1 to gain a more in-depth, practical understanding of working with LangChain and LLM APIs to build robust and scalable generative AI applications. While Chapter 1 provided a broad overview of LangChain and LLMs, this chapter will go deeper into the practical aspects of integrating LLM APIs with LangChain, including hands-on exercises and detailed technical setup instructions.

You will discover the numerous benefits these LLM APIs bring to the table, such as increased efficiency, scalability, and cost-effectiveness. These APIs make advanced AI features accessible to developers at all skill levels without requiring deep machine learning expertise. You will learn how to leverage these APIs to broaden the global reach of your applications and seamlessly include the latest LLM advancements to build practical generative AI solutions.

Furthermore, you will learn to tackle some common challenges you might face when integrating LLM APIs, such as addressing security issues, managing deprecated features, and handling rate limitations. For each of these challenges, I will provide practical strategies to help you overcome them effectively.

A key part of this chapter focuses on how LangChain can simplify your development process. By abstracting the complexities of direct API calls, LangChain allows you to manage multiple LLMs effortlessly focused on creating applications that impress users while maintaining a clean and manageable codebase.

Together, we will walk through the complete process of using LLM APIs within LangChain such as selecting the right models and preparing your development environment to deploying and maintaining your applications.

I have designed this chapter to be practical by providing you with the knowledge to start integrating LangChain and LLMs into your development workflow.

Understanding LLM APIs

In simple terms, an API is a way for different computer programs to talk to each other. In the world of generative AI development, LLM APIs can be very handy because they provide pre-built models and functionalities, saving you from the daunting task of building complex systems from the ground up.

Business Benefits of Using LLM APIs for Generative AI

What exactly are the benefits of using LLM APIs? Let us review them:

- **Advanced Natural Language Processing:** LLM APIs provide access to sophisticated natural language processing capabilities, which you can use to build generative AI applications. These applications can understand and generate human-like text, translate

languages, and comprehend context at a level that would be extremely difficult to achieve with traditional programming methods.

- **Rapid Prototyping of AI-Powered Features:** You can quickly prototype and implement advanced AI features like chatbots, content generation, and text summarization using LLM APIs without the need for extensive machine learning expertise or infrastructure setup.
- **Scalable Applications:** You can handle a wide range of language tasks from simple text completion to complex reasoning. You can use LLM APIs to scale your applications as your needs grow without making significant changes to your codebase.
- **Cost-Effective Access to Massive Language Models:** You do not have to invest in the substantial computational resources and expertise required to train and maintain large language models. Instead, you can leverage pretrained models through APIs which can significantly reduce your costs and time-to-market for GenAI features.
- **Continuous Improvement in Language Capabilities:** LLM API providers continuously update their models with the latest advancements in AI research. You can benefit from this improved language understanding and generation capabilities over time without having to update your core application.
- **Domain-Specific Knowledge Integration:** Many LLM APIs offer fine-tuned models for specific domains or tasks. You can easily integrate these specialized

language models for areas like legal, medical, or technical fields without having to conduct extensive domain-specific training, which can also be very expensive.

- **Multilingual and Cross-Cultural Capabilities:** LLM APIs are becoming increasingly capable of supporting multiple languages and even understanding cultural nuances. This capability enables you to build applications that serve a global audience with localized and culturally appropriate language processing.

Technical Benefits of Using LLM APIs

Here are some technical benefits specific to LLM APIs:

- **Simplified Complex NLP Tasks:** LLM APIs abstract away the complexities of tasks like sentiment analysis, entity recognition, and text classification. Using these LLM APIs, you can easily implement these complex features with just a few API calls rather than building complex NLP pipelines.
- **Flexible Integration of Language Models:** You can easily switch between different language models or combine multiple models for different tasks within your application which allows you to quickly offer more powerful language processing functionality to the end users.
- **Efficient Handling of Context and Memory:** Many LLM APIs provide built-in mechanisms for maintaining context across multiple interactions or processing long documents. This makes it easy for you to develop conversational AI or document analysis applications.

- **Advanced Prompt Engineering Capabilities:** LLM APIs often come with tools and best practices for prompt engineering. By using these capabilities, you can fine-tune the behavior of the language model for specific tasks without needing to understand the underlying model architecture.
- **Seamless Integration of Multimodal Inputs:** Some advanced LLM APIs can process not just text but also images, audio, and other data types. This capability allows you to develop more sophisticated generative AI applications that can understand and generate content across multiple modalities.

Addressing Challenges in LLM API Integration

Let us not forget that using APIs for LLM app development using LangChain, Pinecone, and models such as GPT and Gemini can be challenging. Below is a list of challenges that you need to stay on top of.

Security Concerns

When dealing with sensitive user data, security should be your top priority. Make sure to implement robust measures like authentication, input validation, and encryption.

Managing LLM API Deprecation

As APIs evolve over time, certain endpoints or features might become deprecated and eventually removed. This means you will need to stay on top of LLM API updates and be prepared to modify your code accordingly.

Debugging and Troubleshooting

Debugging can also be a bit trickier when working with APIs. If something goes wrong, it's not always clear whether the issue lies in your code or with the LLM API itself. By catching specific exceptions and providing informative error messages, you can more easily pinpoint where issues are occurring.

Overcoming Latency

When making LLM API calls, there's always some latency involved. To mitigate this, you can implement techniques like caching and asynchronous processing.

Remember, I am sharing these challenges not to discourage you, but to empower you to build robust AI-driven applications. With the right knowledge and guidance, you will be able to navigate these challenges like a pro.

Addressing Rate Limiting and Cost Management

Many APIs impose limits on the number of requests you can make within a specific time frame.

You will have to watch out for LLM API rate limits and ensure your code handles rate limiting properly in your code.

Some APIs charge based on the number of requests, amount of data processed, or specific features used. Similarly, training and running LLMs can be computationally expensive. Make sure to review the pricing models and consider the long-term costs when planning your application architecture.

You can leverage LangChain's strategies for efficient LLM API usage, including caching mechanisms and intelligent request batching, which help manage costs and mitigate the impact of rate limits. By optimizing how and when data is sent to LLMs with LangChain, you can ensure your application remains both performant and cost-effective.

Case Study 1: Real-Time Customer Service Chatbot

Scenario Let us say your company wants to deploy a real-time customer service chatbot capable of handling various inquiries about their product. Initially, you directly integrate GPT-4's API for generating responses. However, you are starting to encounter challenges as shown below.

Challenges with Direct LLM API Usage

Relevance: Inconsistencies in chatbot responses lead to customer dissatisfaction.

Rate Limiting: During peak times, the service hits the API's rate limit, causing delays and dropped interactions.

Cost: High query volumes significantly inflate costs.

LangChain Solution

Your company has decided to switch to using LangChain, which allows you to address these issues using LangChain.

Enhance Contextual Understanding: You leveraged LangChain's prompt engineering and memory concepts to ensure that the chatbot maintains context over the course of a conversation, significantly improving the relevance of the response.

Manage LLM API Calls: Using LangChain, you have successfully managed LLM API requests to optimize usage and stay within rate limits.

Reduce Costs: You reduced the number of necessary LLM API calls by using efficient caching and smarter prompt design techniques, thus lowering costs.

Additional Things to Consider

In addition to what we discussed earlier, here are some additional considerations that can significantly influence the success of your generative AI projects.

Choosing the Right Use Case

As shown in Figure 1-1, you need to choose the right use case. This initial step is crucial as it sets the direction for the entire project. Think about the specific problem you are trying to solve or the user experience you want to create. LLMs can be applied to a wide range of scenarios, such as chatbots, content generation, sentiment analysis, and more.

Question: Think about a project where generative AI wasn't used but could have made a difference. How could incorporating LangChain and LLMs have changed the outcome? For example, in customer service, using a chatbot for handling common inquiries could free up human resources for more complex issues, thereby enhancing overall efficiency.

Choosing the Right Model

You must select the right model for your use case and fine-tune it to improve performance. The model should align with your specific needs, whether it's answering questions, generating text, or performing other tasks.

When it comes to LLMs themselves, there are various options available, such as GPT-4, BERT, Gemini, and others. Each model has its own strengths and weaknesses, so you need to understand their capabilities and limitations.

You will have to experiment with different models and adjust their parameters to get the best results for your app. This process can involve tweaking things like the model's architecture, training data, and hyperparameters.

Prepare Your Data

You must properly prepare data to get more accurate and reliable outputs from your LLM. Before you feed your data into the LLM, you will want to make sure it is clean, formatted correctly, and ready to go. This might involve tasks like tokenization (splitting text into individual words or subwords), normalization (converting text to a consistent format), and handling any missing or inconsistent data.

Evaluate Your Model Performance

Once you have trained your LLM, you will want to assess its performance to see how well it is doing. This can involve metrics like perplexity (how well the model predicts the next word), BLEU score (measuring the similarity between generated text and reference text), or custom metrics specific to your use case.

Don't forget about the power of iteration and experimentation. Building an LLM app is rarely a one-and-done process. You will likely need to try different approaches, tweak your prompts, and fine-tune your models multiple times to get the results you want. Embrace the iterative nature of the development process and don't be afraid to experiment with new ideas.

Using Direct LLM API vs. LangChain

For beginners looking to build generative AI applications, LangChain offers a streamlined, powerful approach. While you can certainly work with LLM APIs directly, LangChain offers significant advantages in terms of simplicity, flexibility, and scalability. In this section, let us explore why LangChain is the recommended choice for most developers, especially for those just starting to work with LLMs. I will also briefly touch upon the challenges when using direct APIs just to provide context, but our focus throughout this book will be on leveraging LangChain's capabilities to build efficient, scalable AI applications.

Let us look at some of the challenges with LLM APIs and how LangChain can help address them.

Development Complexity

When using LLM APIs directly, you need to become deeply familiar with each model's intricacies and limitations, which is a steep learning curve.

LangChain Solution: Like we discussed earlier, LangChain abstracts the complexities involved in making direct API calls by offering you a unified interface to interact with multiple LLMs seamlessly. This abstraction layer means you spend less time wrestling with LLM API specifics and more on crafting the right prompts and integrating valuable data sources without worrying about the underlying model.

Integration and Scalability Challenges

When you are trying to scale generative AI applications, integrate new models, or change APIs, you may need to refactor your code significantly, which is quite cumbersome. LangChain can assist here.

LangChain Solution: With its modular architecture, LangChain makes it easy to switch between LLMs or upgrade to newer versions without overhauling your codebase. This flexibility ensures you can remain at the cutting edge of generative AI capabilities and incur minimal technical debt.

Generic Response Issue

LLMs provide generic responses unless provided with examples and task context. By adopting prompt engineering, you can extract desired responses from LLMs, though it may require considerable experimentation and expertise.

LangChain Solution: You can use LangChain's prompting tools and templates to simplify and optimize the prompt creation process, thus reducing the need for extensive trial and error to achieve better and reliable outcomes.

Case Study 1: Content Generation Platform

Let us consider a case study that involves building a platform to generate content.

Scenario Consider a platform that uses an LLM to generate articles based on user inputs. Direct integration with an LLM API presented multiple challenges, including inconsistent content quality and difficulties in handling diverse user requests.

Challenges with Direct LLM API Usage

Below are some issues with calling an LLM API directly:

- **Content Quality:** The platform struggles to consistently generate high-quality content that aligns with user inputs, often requiring multiple LLM API calls to get usable results.
- **User-Specific Customization:** Customizing a wide range of user requests for content on different topics and styles proves to be complex and resource-intensive.
- **Scalability:** Difficulty in managing and scaling the LLM API integration as the platform grows.

LangChain Solution

Here is how implementing LangChain can help address these issues:

- **Improve Content Customization:** You can use LangChain's model and prompt templates to tailor content more precisely to user needs, improving the quality and relevance of generated articles.

- **Streamline Process:** Through data connections and memory concepts, you can gain a deeper understanding of user preferences, reducing the need for excessive API calls and enabling more personalized content creation.
 - **Enhance Scalability:** LangChain’s architecture allows you to integrate additional LLMs and scale services easily to meet growing demand without overwhelming operational costs.
-

Streamlining Data Integration

Directly integrating diverse data sources with LLMs for context-rich applications involves significant engineering effort and complexity.

LangChain Solution: You can use LangChain’s data connections component to simplify the integration of diverse data sources, allowing applications to seamlessly incorporate relevant contextual information. It helps you to create sophisticated, context-aware LLM applications, reducing the engineering effort and complexity typically associated with such integrations.

Choosing Between Direct LLM API Use and LangChain

Like I mentioned earlier in the previous section, LangChain is the preferred approach for beginner developers. Direct LLM usage and LangChain approach come with their own benefits and trade-offs.

This section is meant for advanced programmers or those with specific project requirements, technical constraints, and development goals. Let us evaluate the two options.

Direct LLM API Use

Here, we discuss when you should prefer calling LLM APIs directly:

- **Simple Projects:** If your project is small and only involves a single LLM, then you may not need the complex abstractions provided by LangChain. Note that LangChain offers a more user-friendly experience even in these cases.
- **Maximum Control and Customization:** When working directly with the LLM provider's API, you will have more granular control over API requests. Granular control over the LLM API can come in handy if your app requires customizations or uses provider-specific features that LangChain doesn't support.
- **Learning and Educational Purposes:** You will also gain deep insights into the specifics of how LLM operates, handling requests and parsing responses. It is beneficial for educational or research-focused projects.

Trade-offs

However, it is not without trade-offs. Let us look at some of them:

- You will face challenges when integrating multiple LLM providers.
- Maintaining your code for LLM API changes or deprecations can be very maintenance heavy.

- You will miss out on the benefits of using boilerplate templates for common tasks like prompt engineering or response parsing available with LangChain.

Using LangChain

Let us discuss when to prefer LangChain:

- **Rapid Development and Prototyping:** If you want to quickly prototype applications without dealing with the intricacies of each LLM API, LangChain provides a set of high-level tools and templates that speed up development.
- **Flexibility and Scalability:** If your project is expected to scale or evolve over time and you will need to switch between LLM providers or experiment with different models, then LangChain's abstraction layer offers significant benefits. It will be easier to manage modifications and updates to your LLM without needing major code overhauls to integrate with the LLM.
- **Complex Applications Requiring Advanced Features:** You can use LangChain's advanced functionalities like prompt chaining, memory concepts, and data connections that can be very critical when developing sophisticated, enterprise-grade generative AI applications. These features are not available when making direct API calls.

- **Multi-LLM Integration:** LangChain is particularly useful when integrating with multiple LLM providers. You do not have to deal with the complexities involved when interacting with different LLM APIs because you will be working with LangChain's interface directly.
- **Cost Optimization:** For projects that do require multiple LLMs, LangChain allows easy switching between providers. This can be particularly useful for cost management because you could use low-cost or open source LLMs for development and testing and commercial providers for production.
- **Active Community and Ecosystem:** LangChain has a vibrant community and extensive ecosystem of tools, plug-ins, and resources. You can use this support network to significantly accelerate development and problem-solving.

Trade-offs

Let us explore some of the trade-offs when using LangChain:

- You may face potential limitations when trying to access all the features offered by specific LLM APIs.
- Debugging can be difficult due to the additional layers of abstraction introduced by LangChain which adds to the overhead or complexity.
- You will have to depend on LangChain updating their code to support new LLM features or models.

Ultimately, you will have to factor in your project's scale, complexity, and specific requirements to choose between making direct LLM API calls and LangChain.

Preparing Your Dev Environment

Alright, I think now is a good time to get our hands dirty. Like any other development project, it starts with you first setting up your development environment.

Step 1: Obtaining OpenAI API Keys

To start developing applications using OpenAI's LLMs, you first need to acquire an API key from OpenAI. Here's how to do it.

Create/Open an Account

First, you will need to create an account with OpenAI's API. This will give you access to their powerful language models, which we will be using throughout this book. To get started, head over to openai.com/api and click the "Sign Up" button. If you already have an account, you can simply log in.

Now, the sign-up process might require you to verify your account with a mobile phone number, depending on your location. Just follow the on-screen instructions, and you will be good to go.

Billing Information

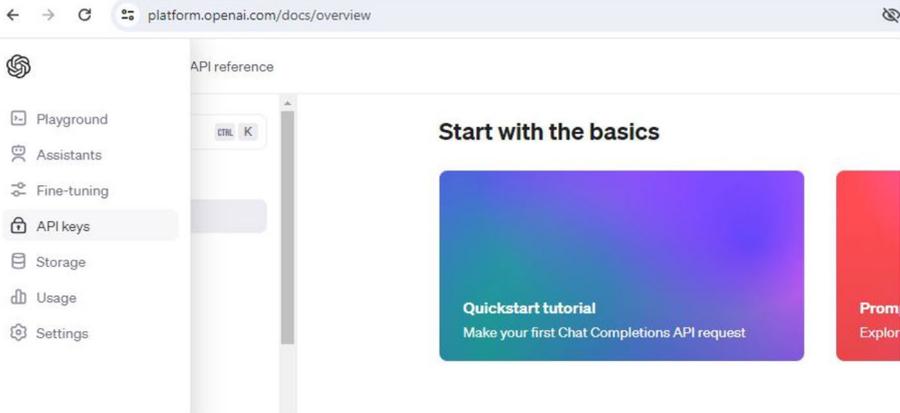
Once you have signed up or logged in, you will need to set up your billing information. As of this writing, OpenAI offers free credit for the first few months, so you can get some hands-on practice without spending a dime.

API Key

Note After you reach the free credit limit, you'll need to provide a credit card for any additional charges.

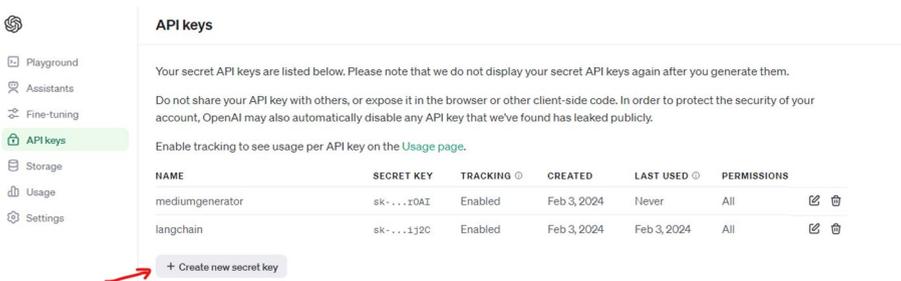
CHAPTER 2 INTEGRATING LLM APIS WITH LANGCHAIN

To get your API key, navigate to the “**View API Keys**” section under your account settings.



Click “**Create New Secret Key**,” copy the key that’s generated, and store it somewhere safe.

Note Treat this key like a precious item because if you lose it, you will have to create a new one, and you won’t be able to access the old one ever again.



Step 2: Setting Up the Python Development Environment

Now, let's move over to our Python development environment. I will assume you are using a Jupyter Notebook, but feel free to use your preferred setup, like Visual Studio Code or PyCharm.

Google Colaboratory

I will be using the Google Colaboratory here at <https://colab.research.google.com/>.

Create a new notebook by selecting File ► New Notebook and give a suitable name to the file.

Install OpenAI Library

First, we need to install the OpenAI Python library. You can do this by entering the command below and running it in your notebook by clicking the arrow icon:

```
!pip install openai
```

Note You can also use Poetry for managing dependencies. Poetry provides better dependency management, project isolation, and reproducibility compared to using pip directly.

Once successfully installed, you will see a success message as shown below:

```
Installing collected packages: h11, httpcore, httpx, openai
Successfully installed h11-0.14.0 httpcore-1.0.4 httpx-0.27.0 openai-1.14.1
```

Configure API Key

Once the installation is complete, you can import the library and set up your API key as an environment variable. This is a best practice to keep your sensitive credentials safe and secure:

```
import os
import openai

# Set the API key as an environment variable
os.environ["OPENAI_API_KEY"] = "your_api_key_here"

# Confirm that the API key is set correctly
openai.api_key = os.getenv("OPENAI_API_KEY")
```

Replace “your_api_key_here” with the actual API key you copied earlier. Don’t worry; we will keep this key safe and sound in our environment variable later.

Test the Setup

Now, test your setup by sending a simple request to the OpenAI API:

```
client = openai.Client() # Create a client
response = client.completions.create(
    model="davinci-002", # Or any currently available
    text-davinci model
    prompt=" What is machine learning model?.",
    max_tokens=300
)
print(response.choices[0].text)
```

This code will send a request to the OpenAI API, asking to explain what a machine learning model is. The model parameter specifies which language model we want to use (in this case, davinci-002), and the max_tokens parameter limits the length of the response.

Note Visit the OpenAI's model overview page to ensure that the model you are choosing is current and is not deprecated. Secondly, you may also run into issues due to not using the latest version of OpenAI.

If everything is set up correctly, you should see a response printed in your notebook, providing the definition for machine learning model.

Securing the API Key Using Environment Variables

You must always keep your API key confidential and follow security best practices to protect it. Below are some recommendations:

a) Setting up the environment variables

You can store your OpenAI API key as an environment variable on your development machine or server.

On macOS/Linux:

```
export OPENAI_API_KEY=your_api_key_here
```

On Windows:

```
set OPENAI_API_KEY=your_api_key_here
```

b) Retrieving the API key in your Python code:

```
import os
import openai
# Get the API key from the environment variable
api_key = os.getenv('OPENAI_API_KEY')
# Set up the OpenAI API client
openai.api_key = api_key
.....
```

In this code, you use the `os.getenv()` function to retrieve the API key from the environment variable named `OPENAI_API_KEY`. You then set the `openai.api_key` with the retrieved value before making API requests.

Securing the API Keys Using the Configuration Files

You can also use a configuration file to store the API key using the approach below:

- a) Create a file named `config.json` with the following content:

```
{ "OPENAI_API_KEY": "your_api_key_here" }
```
- b) Add `config.json` to your `.gitignore` file to prevent it from being tracked by version control.
- c) In your Python code:

```
import json
import openai

# Read the configuration file
with open('config.json') as f:
    config = json.load(f)

# Get the API key from the configuration
api_key = config['OPENAI_API_KEY']

# Set up the OpenAI API client
openai.api_key = api_key
```

In this code, you use the `Json` module to read the configuration from the `config.json` file. You then access the API key using the `config['OPENAI_API_KEY']` syntax.

There are also other methods to protect the API key, such as

- Using secret management tools like AWS Secrets Manager, HashiCorp Vault, or Azure Key Vault to store the key
- Storing the key as an environment variable on the server
- Setting up a proxy server in between the application and the OpenAI API to store the API key and then your application will call the proxy rather than the OpenAI API

Note Remember to never commit or publish your API key in public repositories or share it with unauthorized parties.

Congratulations!

You have successfully set up your development environment and connected it to the OpenAI API. You are now ready to explore the exciting world of LangChain and start building powerful applications with large language models.

Pat yourself on the back because you have taken the first and a very important step toward becoming a LangChain wizard!

Exercise 1: Calling an LLM API Directly

Before we uncover the awesomeness of LangChain, let us take a quick look at the difference between calling an OpenAI API directly and using the LangChain framework. This might seem like a trivial example, but you will start to see the benefits of abstracting the LLM API calls using LangChain's

APIs. Using this approach, you can adopt a more plug-and-play approach and avoid the complexities that come with making direct LLM API calls across multiple providers.

Below is the code to set up a basic text generation task using a direct LLM API vs. LangChain.

Direct LLM API Method

Task: Use the OpenAI API to generate a short story based on a given prompt.

Step 1: Choose an LLM API Provider and Obtain an API Key

Go to OpenAI's website and sign up for an account. Navigate to the API section and follow the instructions to obtain your API key. This key will allow your application to authenticate with OpenAI's servers and make requests to GPT-4.

Step 2: Install Necessary SDKs or Libraries

OpenAI provides an official Python library that simplifies interacting with GPT-4. You can install this library using pip, Python's package installer. Open your favorite code editor and create a new Python file. In this file, you are going to write a script that prompts the user for a story starter and then sends this prompt to GPT-4, which will generate a continuation.

Here's a simple example:

Install the openai package version 0.28

```
!pip install openai==0.28
```

```
import os
```

```
import openai
```

Import the new Chat Completion API from openai

```
import ChatCompletion
```

Set the OpenAI API key using an environment variable

```
os.environ["OPENAI_API_KEY"] = "sk-T"
```

```

# Confirm that the API key is set correctly
openai.api_key = os.getenv("OPENAI_API_KEY")

# Define a function to get the chat completion
def get_chat_completion(user_prompt):

# Use the Chat Completion API to generate a response
response = ChatCompletion.create(

# Specify the chat model engine to use
model="gpt-3.5-turbo",

# Provide the user prompt as a message
messages=[{"role": "user", "content": user_prompt}]
)

# Extract and return the generated response
return response.choices[0].message.content.strip()

# Prompt the user to enter a story prompt
user_prompt = input("Enter a story prompt: ")

# Generate the chat completion based on the user prompt
result = get_chat_completion(user_prompt)

# Print the generated result
print(result)

```

Note Remember to import the correct version of the OpenAI library, else it will not work.

Let us break down what is happening in the code:

- We install the OpenAI package with version 0.28.
- We import the necessary modules: os and openai.

- We import the new Chat Completion API from the OpenAI module.
- We set the OpenAI API key using an environment variable.
- We confirm that the API key is set correctly.
- We define the `get_chat_completion` function to generate a chat completion using the Chat Completion API. It takes a `user_prompt` as input and sends it as a message to the API. The model parameter specifies the chat model engine to use (e.g., “gpt-3.5-turbo”).
- The generated response is extracted from the API response and returned.

The code prompts the user to enter a story prompt as shown in Figure 2-1.

```
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->openai==0.28) (23.2.0)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from aiohttp->openai==0.28) (1.4.1)
Requirement already satisfied: multidict>=4.5 in /usr/local/lib/python3.10/dist-packages (from aiohttp->openai==0.28) (6.0.5)
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->openai==0.28) (1.9.4)
Requirement already satisfied: async-timeout<5.0,>=4.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->openai==0.28) (4.0.3)
Enter a story prompt: 
```

Figure 2-1. Enter a Story Prompt

Enter your input, in this case, I entered “Tell the story of the earth,” and press Enter. The `get_chat_completion` function is called with the user prompt to generate the chat completion.

The generated result is printed as shown in Figure 2-2.

```
Requirement already satisfied: async-timeout<5.0,>=4.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->openai==0.28) (4.0.3)
Enter a story prompt: Tell the story of the earth
In the beginning, billions of years ago, the Earth was formed from a cloud of dust and gas in a process known as accretion. As the newly formed planet continued
Over time, the Earth's surface cooled and solidified, forming a crust that was constantly reshaped by tectonic activity and volcanic eruptions. These processes cr
Life first emerged on Earth around 3.5 billion years ago in the form of simple single-celled organisms. Over millions of years, these early life forms evolved an
Throughout its long history, the Earth has experienced countless changes and upheavals, from mass extinctions to the rise and fall of civilizations. Despite these
Today, the Earth is a vibrant and diverse planet, home to millions of species and a wide variety of ecosystems. However, human activity has had a profound impact
As we move forward, it is more important than ever to protect and preserve the Earth's delicate balance and ensure a sustainable future for all life on this rema
```

Figure 2-2. LLM Response to the Prompt

Outcome

By completing this exercise, you will have hands-on experience with

- Obtaining and using an API key for authentication with an LLM API
- Installing and using a Python SDK to simplify API interactions
- Making requests to an LLM (GPT-4) and processing its responses

Exercise 2: Using LangChain for Enhanced Flexibility

Now that you have seen how to call an LLM API directly, let us explore how LangChain can make things easier by making the code more flexible.

Step 1: Set Up LangChain in Your Development Environment

To get started with LangChain, you will need to install it via pip. Open your terminal or command prompt or any other development tool, such as Google Colab, and execute the following command:

```
pip install langchain
```

This command installs LangChain and its dependencies, preparing your environment for development.

Step 2: Modify Your Script to Use LangChain

Open the Python script you created in Exercise 1. You are going to modify this script to use LangChain instead of directly interacting with the OpenAI API.

Import LangChain at the beginning of your script and configure it to use an LLM (e.g., OpenAI's GPT-4). Here's an example of how to modify the script:

Import the necessary modules

```
from langchain_openai import OpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
```

Note LangChain is actively developed, and its structure may change. Always refer to the official LangChain documentation for the most up-to-date import statements and usage patterns.

Your OpenAI API key

```
api_key = "your_api_key_here"
```

Confirm that the API key is set correctly

```
openai.api_key = os.getenv("OPENAI_API_KEY")
```

Initialize LangChain with OpenAI's GPT

```
llm = OpenAI(api_key=api_key)
```

Use LLMChain for easy interaction

```
chain = LLMChain(llm=llm)
```

Define a template for the prompt that will be used to generate responses.

```
prompt_template = PromptTemplate(input_variables=["user_
input"], template="You are a helpful chatbot. User: {user_
input} Response:")
```

Create an instance of OpenAI class and assign it to variable llm

```
llm = OpenAI()
```

Create an instance of the LLMChain class

```
chain = LLMChain(llm=llm, prompt=prompt_template)
```

Prompt the user for a story starter

```
user_prompt = input("Enter a story prompt: ")
```

Call the run method on the LLMChain instance

```
response = chain.run(user_prompt)
```

Print the generated response

```
print(response)
```

A couple of things to note from the above example. First is that the advantage with this approach is that it uses LangChain to abstract the API interaction, making it easier to switch between LLMs or adjust the prompt engineering approach without extensive modifications to your code.

Next, let's discuss the three modules we imported:

- **OpenAI from langchain.llms:** This is the class for interacting with the OpenAI language model.
- **PromptTemplate from langchain.prompts:** This class is used to define a template for the prompt that will be used to generate responses.

- **LLMChain from langchain.chains:** This class represents a chain that combines a language model (LLM) with a prompt template to generate responses.

Here's what is happening in the code:

- We set the **OpenAI API key** as an environment variable using `os.environ["OPENAI_API_KEY"]`. The value of the API key is assigned to the variable.
- The **API key** is confirmed to be set correctly by assigning it to `openai.api_key` using `os.getenv("OPENAI_API_KEY")`.
- We create a **PromptTemplate** object with the following parameters:
 - `input_variables=["user_input"]`: This specifies that the template expects a variable named "user_input".
 - `template="You are a helpful chatbot. User: {user_input} Response:"`: This defines the template string for the prompt. It includes the "user_input" variable within curly braces `{}` to indicate where the user's input will be inserted.
- We create an **instance of the OpenAI class** and assign it to the variable `llm`. This represents the OpenAI language model that will be used for generating responses.
- We create an **instance of the LLMChain class** with the following parameters:
 - `llm=llm`: This specifies the language model to be used in the chain, which is the OpenAI instance created in the previous step.

- `prompt=prompt_template`: This specifies the prompt template to be used in the chain, which is the `PromptTemplate` object created earlier.
- The user is prompted to enter their request using **user_prompt**; in my case, I asked, “Tell the story of earth.”
- The **run method** of the `LLMChain` instance is called with the argument variable `user_input`. This will generate a response from the language model based on the provided prompt and user input.
- The generated response is printed using `print(response)`.

In summary, you just learned how to call an OpenAI language model using `LangChain`, create a prompt template, and generate a response based on the user’s input. The response is then printed to the console.

The code demonstrates how to use `LangChain` to interact with the OpenAI API, define a prompt template, and generate responses using a language model in a structured and modular way.

Step 3: Experiment with Prompt Engineering within LangChain

`LangChain` facilitates advanced prompt engineering techniques. You can experiment with different prompt formats and parameters to see how they affect the story’s creativity, coherence, and relevance.

For example, you can adjust the temperature parameter to make the responses more or less predictable or use stop sequences to control the length and structure of the generated content. Below is an example of how to achieve that outcome:

```
chain = LLMChain(
    llm=llm, prompt=prompt_template,
```

```
llm_kwargs={
    "temperature": 0.7,
    "max_tokens": 100,
    "stop": ["\n"]
}
```

By experimenting with different prompt formats and parameters, you can see how changes in the prompt or parameters influence the output. This will help you understand how to get the best possible responses from the LLM.

Outcome

You can observe that not only does LangChain simplify the process of switching between different LLMs, but it also opens up possibilities for refining prompts to achieve higher-quality AI-generated content.

The LangChain code is also cleaner and simpler and allows plugging in multiple LLMs. Moreover, there is no tight coupling with the LLM's API. The benefits become even more apparent as we move on to more complex use cases, such as customizing responses using prompt engineering, integrating external data sources, and so on.

Key Takeaways

Let us discuss what we have learned so far:

- **Streamlined Development:** You learned that LangChain simplifies generative AI application development by abstracting the complexities of direct LLM API calls. You can focus more on application logic rather than the nuances of LLM APIs.

- **Enhanced Flexibility and Scalability:** You can easily integrate and switch between different LLMs without extensive code refactoring. This ensures applications can evolve alongside emerging AI technologies with minimal technical debt.
- **Optimized Prompt Engineering:** You can use LangChain tools and templates for prompt engineering which can reduce the time taken to experiment and the skills required to elicit desired responses from LLMs.
- **Cost and Efficiency:** By leveraging LangChain's approach to modular development and intelligent API request management, you can optimize costs and improve application performance, especially for complex and high-volume AI tasks.
- **Global Accessibility:** You can build and deploy AI-powered applications with global reach and consistent performance by leveraging the cloud-based LLM APIs.
- **Security and Data Processing:** You can maintain robust security standards and simplify the handling and processing of complex datasets while adhering to industry-standard security requirements.
- **Transition to Innovation:** Moving to LangChain from direct LLM API usage opens up new possibilities for innovation in generative AI application development.

In summary, you can use LangChain to streamline the development process, enhance the flexibility and scalability of applications, and open up innovation possibilities while addressing the inherent challenges of LLM APIs.

Start Creating with LangChain

Congratulations! You have learned about the power of LLM APIs and discovered the transformative potential of LangChain. Now, it's time to put your skills into action and bring your ideas to life.

In the next chapter, we will be going deeper into building an LLM app. In the meantime, I recommend you get started on your first LangChain project if you haven't already. For now, you can start with a simple content generator and later on build a chatbot or a data-driven application.

Continue your learning adventure by exploring LangChain's documentation, experimenting with different LLMs, and uncovering new use cases.

Glossary of Technical Terms

Below is a list of definitions of technical terms used in this chapter:

API (Application Programming Interface): A set of rules and definitions that allows software programs to communicate with each other for the integration of external services or data.

LLM (Large Language Model): Advanced LLM models for understanding and generating human-like text based on extensive training datasets.

SDK (Software Development Kit): A collection of software tools, libraries, and documentation to develop applications for a specific platform or technology.

Tokenization: The process of breaking down text into smaller units (tokens), such as words or phrases, for processing by machine learning models.

Normalization: The process of converting text into a consistent format (e.g., lowercasing, removing punctuation) to improve the performance of machine learning models.

Perplexity: A measure of how well a language model predicts a sample of text. Lower perplexity indicates better predictive accuracy.

Further Reading

Below is a list of resources to help you solidify your understanding of the topics covered in this chapter and to further explore the world of LLM APIs:

OpenAI API Guide: OpenAI offers extensive documentation on utilizing their API, including best practices for prompt engineering and model selection. <https://platform.openai.com/docs/introduction>

Google Cloud AI Services: Google Cloud provides a wide range of AI services. Their documentation is a treasure trove of information for developers looking to leverage Google's AI capabilities. <https://cloud.google.com/ai-platform/docs>

AWS AI Services: AWS offers a comprehensive guide to their AI services, ideal for developers who want to integrate AWS's machine learning tools into their applications. <https://docs.aws.amazon.com/machine-learning/>

“Design Patterns for Large Language Models”: This paper explores various strategies and design patterns for effectively utilizing LLMs in application development. <https://arxiv.org/abs/2004.13214>

CHAPTER 3

Building Q&A and Chatbot Apps

In this chapter, you will learn how to build powerful applications using large language models (LLMs) through LangChain and OpenAI. We will focus on building your practical skills and adopting straightforward strategies to improve your application development skills. I will walk you through the setup, implementation, and deployment of LLM-driven applications, such as Q&A and chatbots.

I have designed this chapter to help you with the skills to develop, deploy, and enhance LLM applications end to end so that you can go on to develop more impactful and powerful applications.

LangChain Framework Components

By now it must be clear to you that LangChain is a powerful framework that can help you develop LLM applications. You have already seen quite a few examples on how to call LLMs using the LangChain libraries. You also learned briefly about some of the components in Chapter 1. Let us explore the LangChain framework more. We have dedicated chapters for these components, so we will only discuss them at a high level now.

Development

We already learned in Chapter 1 that LangChain provides the following components:

- **Model I/O**

Model input/output (I/O) is all about how you can format and manage the data going into and coming out of the language models.

- **Prompts**

Prompts are specific formats that you can use to feed data into language models to guide how they generate responses. You can think of them as tools that help you create questions to steer the conversation in the direction you need. This approach lets you control the flow and ensures the responses are relevant to your specific requirements.

- **Chat Models**

Chat models are specialized interfaces to handle chat-like interactions. You can use these models to feed chat messages as inputs and get back responses in a conversational format. You can use them to build chatbots and similar interactive applications.

- **LLMs (Large Language Models)**

LLMs are sophisticated AI models primarily designed for understanding and generating human-like text. They excel at tasks like text generation, question answering, and information extraction. While their core functionality revolves around text, LLMs can be used for a wide range of applications, from creative writing to data analysis and problem-solving.

Note Recent advancements have expanded LLM capabilities beyond text processing. Through various extensions, LLMs can now

- Generate and manipulate images
 - Process and create audio content
 - Assist in code generation and analysis
 - Perform basic reasoning tasks
 - These extensions make LLMs powerful tools for diverse multimodal applications across industries.
-

- **Retrieval**

The retrieval component is a very significant advancement that allows you to bridge your application-specific data with the language model, which is perfect for applications like Retrieval-Augmented Generation (RAG). You can enable the model to pull relevant information from a vast dataset to develop its relevant and meaningful responses.

- **Document Loaders**

Document loaders are tools that import data from various sources and format it into “Documents” for your application to process later. You can load text files, extract data from databases, or pull content from the Web using this feature. I will be discussing this in [Chapter 7](#).

- **Text Splitters**

You can use text splitters to take large blocks of text and chop them into smaller, more manageable pieces. This not only makes processing easier but also improves the performance of your models by focusing on the most relevant sections of text.

- **Embedding Models**

Embedding models are a fascinating concept that allows you to convert chunks of text into vector representations. These vectors capture the essence of the text in a numerical format, which you can use to perform natural language searches through large datasets.

- **Data Storage and Retrieval**

LangChain supports various options for storing and retrieving data efficiently:

1. **Vectorstores:** These are specialized databases that manage vector representations for natural language search through unstructured data. You can use them particularly for building sophisticated search engines and recommendation systems for providing relevant results quickly.
2. **Graph Databases:** For advanced users, graph databases offer an alternative that can be more suitable for certain scenarios, such as when dealing with highly connected data. LangChain supports integration with graph databases which allows complex relationship queries.

3. **Traditional Databases:** For structured data and precise queries, traditional SQL databases remain a powerful option.

The choice between these depends on your specific use case, data structure, and query requirements. LangChain's flexibility allows you to integrate the most appropriate storage solution for your project.

Beyond Vectorstores While vectorstores are good with quick, relevant searches in large unstructured datasets, you should also consider graph databases for highly connected data scenarios. LangChain supports both, and hence you must choose the best fit based on your project's needs. Learn more about graph database integration in LangChain here: https://python.langchain.com/docs/use_cases/graph/.

- **Retrievers**

Retrievers are a bit more general than Vectorstores. They are interfaces that you can use to fetch documents based on an unstructured query. You can think of them as your personal data retriever that goes and fetches information based on your current needs.

- **Composition**

Composition components are like the architects of LangChain. You can use them to combine various systems and LangChain primitives to build complex functionalities. For instance, you may be linking a text splitter with an embedding model or creating a

full-fledged AI assistant. Using this composition module, you can tailor your application precisely to your needs and optimize its efficiency and effectiveness.

- **Tools**

You can use the interfaces provided by tools to allow a language model to interact with external systems. Using tools, you can expand the capabilities of your applications, such as connecting to a database or a third-party API.

- **Agents**

Agents are the decision-makers in LangChain. You can use them to analyze high-level directives and decide which tools to use to achieve the desired outcome in an automated fashion. They ensure your application is using the right resources at the right time on its own. Please refer to Chapters 8–10 for a detailed treatment of LangChain Agents.

- **Chains**

Chains are fundamental to LangChain’s modular design. They are compositions of various components that work together as building blocks. You can mix and match these to customize how your application behaves. Below is an example just to give you an idea of how chains can be used:

```
!pip install langchain==0.2.7 langchain_openai==0.1.16
import os
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
```

```

from langchain.chains import LLMChain

# Set your OpenAI API key
os.environ["OPENAI_API_KEY"] = "your open ai key"

# Initialize your LLM model
llm = ChatOpenAI()

# Create a prompt template
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a world class technical
    documentation writer."),
    ("user", "{input}")
])

# Create a chain that defines how your application
interacts with the LLM
chain = LLMChain(llm=llm, prompt=prompt)

# Use the chain
result = chain.run("Explain the concept of recursion in
programming.")

print(result)

# Now you are ready to start adding more functionality
to your app!

```

- **Memory**

You can use this component to persist application state between runs of a chain and maintain continuity and context across sessions.

- **Callbacks**

You can use callbacks to log and stream intermediate steps of any chain, thus providing transparency and traceability during execution.

As you can see, each of these components plays a vital role in the functionality and efficiency of LangChain-based applications. You will be learning how to use them to build LLM applications in the next few chapters.

Production

During the production phase, your focus is on making sure your application runs smoothly and efficiently. You can use LangSmith, a platform within the LangChain ecosystem, to inspect, monitor, and evaluate your application's performance.

Here is a little insight into how you might use LangSmith to check your application:

```
import os
!pip install langchain==0.2.7 langchain_openai==0.1.16
from langchain_openai import ChatOpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
from langchain.smith import RunEvalConfig, run_on_dataset

# Set your API keys
os.environ["OPENAI_API_KEY"] = "your_openai_api_key"
os.environ["LANGCHAIN_API_KEY"] = "your_langsmith_api_key"

# Initialize your model and chain
llm = ChatOpenAI(temperature=0)
prompt = PromptTemplate.from_template("Tell me a short joke
about {topic}")
chain = LLMChain(llm=llm, prompt=prompt)
```

```

# Define your evaluation configuration
eval_config = RunEvalConfig(
    evaluators=[
        "criteria",
        "embedding_distance",
    ],
    custom_evaluators=[],
)

# Run evaluation on a dataset
results = run_on_dataset(
    client=client,
    dataset_name="my-dataset",
    llm_or_chain_factory=chain,
    evaluation=eval_config,
)

# Print results
print(results)

```

In this example, we use LangSmith, which is LangChain’s evaluation and monitoring platform. It allows you to run your chain on a dataset and evaluate its performance using various metrics. The **RunEvalConfig** lets you specify which evaluators to use, and **run_on_dataset** executes the evaluation.

Note To use LangSmith, you need to sign up for an account and obtain an API key. Visit <https://smith.langchain.com/> to get started. Remember to keep your API keys confidential.

Deployment Using LangServe

During the deployment phase, you will be using LangServe to deploy by converting any chain into a REST API effortlessly to allow your LLM app to be accessed over the Web by other services.

To deploy your LangChain application as an API, LangServe provides a simple command-line interface. After setting up your application, you can serve it with a single command:

```
poetry run langchain serve --port=8100
```

This command will start a server on port 8100, making your LangChain application.

This command uses LangServe, a tool designed to easily deploy LangChain applications. It automatically sets up the necessary API endpoints for your chains and agents, making them accessible over HTTP.

Before running this command, ensure you have LangServe installed in your project:

```
poetry add langserve
```

Also, make sure your project structure follows LangServe conventions. Typically, this means having a `server.py` file that defines your chains and how they should be served.

LangChain Ecosystem

LangChain started as a modest Python package but has grown into a robust framework thanks to input and collaboration from the active developer community. As it evolved, the LangChain team realized the need to streamline the architecture for better usability and scalability. When developing applications, you are going to need to understand these components to use them effectively and avoid potential confusion. Here is how they organized the pieces.

LangChain-Core: LangChain-Core is the foundation of the framework.

- It provides core abstractions that have become standard building blocks for LangChain components.
- You can use the LangChain Expression Language to compose these components smoothly.
- Now at version 0.1, LangChain-Core ensures that any significant updates come with a minor version bump to keep things stable for you.

Here is an example of using LangChain core:

```
# # Example of using LangChain Core
from langchain_core.language_models import BaseLLM
from langchain_openai import ChatOpenAI

# Initialize your language model
llm = ChatOpenAI(model="gpt-3.5-turbo")

# Use the model
response = llm.invoke("Tell me a joke about programming.")
print(response.content)
```

This example demonstrates how to use LangChain Core to interact with an OpenAI chat model. The `ChatOpenAI` class provides a high-level interface for working with OpenAI's GPT models. The `BaseLLM` class from `langchain_core.language_models` defines the abstract interface that all language models in LangChain should implement.

LangChain's structure and best practices are continuously evolving. Always refer to the latest documentation at <https://python.langchain.com/> for the most up-to-date information on classes and usage patterns.

LangChain-Community: LangChain-Community is a package that encompasses third-party integrations.

- It simplifies the process for you to connect with external data sources and tools.
- You can expect this package to expand as more partnerships and collaborations are formed.

Example Usage

Below is an example of how it is used:

```
# Example of leveraging a third-party integration
from langchain_community.document_loaders import
TwitterTweetLoader

# Initialize the Twitter loader
loader = TwitterTweetLoader(
    query="LangChain",
    bearer_token="your_twitter_bearer_token",
    num_tweets=100
)

# Load the tweets
documents = loader.load()")
```

This example demonstrates how to use LangChain’s community-contributed Twitter loader to fetch tweets. The `TwitterTweetLoader` allows you to search for tweets based on a query and load them as documents for further processing in your LangChain pipeline.

Note that you’ll need a Twitter Developer account and a bearer token to use this loader. Also, be aware of Twitter’s API usage limits and terms of service when using this loader in your applications.

Always refer to the latest LangChain documentation for the most up-to-date information on available loaders and their usage: https://python.langchain.com/docs/integrations/document_loaders/.

High-Level Components

LangChain: The LangChain package includes high-level, use case-specific chains, agents, and retrieval algorithms that form the backbone of your generative AI application's architecture. Aiming for a stable 0.1 release soon, this package will bring you sophisticated functionalities to build complex AI-driven solutions.

Below is an illustrative example:

```
# An Example of using LangChain for a retrieval task
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain.chains import RetrievalQA
from langchain.document_loaders import TextLoader

# Load and prepare your data
loader = TextLoader('your_data.txt')
documents = loader.load()

# Create a vector store
embeddings = OpenAIEmbeddings()
vectorstore = FAISS.from_documents(documents, embeddings)

# Initialize the language model
llm = ChatOpenAI(model_name="gpt-3.5-turbo")

# Create the retrieval chain
qa_chain = RetrievalQA.from_chain_type(
```

```
llm=llm,  
chain_type="stuff",  
retriever=vectorstore.as_retriever()  
)  
  
# Use the chain  
query = "Your question here"  
result = qa_chain.invoke({"query": query})  
print(result['result'])
```

This simplified example shows you how to create a basic retrieval-based question-answering system using LangChain.

1. First, you load documents from a text file.
2. Then you create a vector store (FAISS) to efficiently store and retrieve document embeddings.
3. You initialize a language model (ChatOpenAI).
4. You create a RetrievalQA chain that combines the retriever and the language model.
5. Finally, you use the chain to answer a question based on the loaded documents.

This setup allows you to ask questions about the content of your documents, and the system will retrieve relevant information to generate an answer.

Remember to replace “your_data.txt” with the path to your actual data file, and ensure you have the necessary API keys set up for OpenAI.

Ecosystem and Integration

You should realize that the LangChain ecosystem isn’t just about these packages. It also provides you with a variety of tools and integrations that you can use to deploy, monitor, and manage your applications:

LangTemplates: These are ready-to-use templates that you can use to get started quickly with common use cases. LangTemplates provide a solid starting point for use cases such as responding to emails, analyzing text, and a lot more.

LangServe: You will be using LangServe to deploy your LangChain applications as REST APIs. You can easily convert your chains into widely accessible web services.

LangSmith: LangSmith offers debugging and monitoring tools that seamlessly integrate with LangChain. You can use them to gain analytical insights and use them for further optimization of your gen AI app.

By splitting the package into core and community components, LangChain maintains a lightweight and flexible structure which allows you to build applications for a wide range of use cases.

Using LangChain Models with LLMs

In this section, we will explore models, their input, and outputs with LangChain.

Model IO: The Core Functionality of LangChain

The heart of any language model application is the model itself. Model IO is the mechanism by which you can communicate with large language models (LLMs) seamlessly. You will use it to connect to the vast knowledge and capabilities of these incredible LLM models.

One of the benefits of LangChain's Model IO is that it allows you to easily switch between providers. This is a huge benefit because even though you may have started your project using OpenAI, you can choose to explore the capabilities of other models from Google Cloud or Anthropic. You can make that switch effortlessly without having to rewrite your entire codebase.

Large Language Models (LLMs) with LangChain

In this section, we will discuss the different types of language models and explore how LangChain makes it easy to interact with them.

Types of LangChain Models

You can use two types of models in LangChain, namely, the general LLM and the Chat model:

LLM Models: The general LLM model is also known as a text completion model and is like a word wizard that can predict the most likely continuation of your input text. You can send a string prompt as input and generate a string completion as output. It is a straightforward approach, like having a smart text generator at your fingertips. Imagine you start with a prompt like “To kill two birds with a” and the model completes it with “stone.”

ChatModels: Chat models are built on top of LLMs but are specifically tuned for engaging in back-and-forth conversations using messages. You send a human message, and the AI responds with its own message, and the conversation keeps going back and forth. It is like having someone who is always there ready to chat!

You can also use the system-level prompt in chat models to give a personality to the AI model. These system-level prompts help you to set a specific tone or role for the AI, such as a friendly and patient tutor, and it will engage in the conversation accordingly.

One thing you should keep in mind when selecting the models is that different models have their own characteristics and preferences, and you should choose the one that suits your application. We will discuss LLM models in greater detail in the upcoming chapter.

Note For the full list of LLMs supported by LangChain, check the LangChain documentation: <https://python.langchain.com/v0.2/docs/integrations/llms/>.

Building a Simple Q&A Application

Alright, let us get our hands dirty with some code to understand this better.

Step 1: Import the Libraries

```
from langchain_openai import OpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
```

First, you must import necessary classes from LangChain libraries. OpenAI is the LLM, ChatPromptTemplate is for structuring prompts, and StrOutputParser is for parsing the output.

Step 2: Output Parser

Then you must create an instance of StrOutputParser, which will convert the LLM's output into a string:

```
output_parser = StrOutputParser()
```

Step 3: Create a Prompt Template

You must then create a chat prompt template with two messages:

- A system message that sets the context for the AI (acting as a wellness expert)
- A user message with a placeholder {input} for the actual query

```
prompt = ChatPromptTemplate.from_messages([
("system", "You are a world recognized wellness expert
especially in cardio activities."),
("user", "{input}")
])
```

Step 4: Initialize the LLM

You then initialize the OpenAI language model. Remember to replace “openai_api_key” with your actual OpenAI API key:

```
llm = OpenAI(api_key="openai_api_key")
```

Step 5: Create the Chain

Here, you create a processing chain using the “|” operator. It connects the prompt, the LLM, and the output parser in sequence. We will discuss chains in greater detail in the chapter on chains.

```
chain = prompt | llm | output_parser
```

Step 6: Invoke the Chain

Finally, you invoke the chain with a specific input. The chain will

1. Format the prompt with the given input.
2. Send the formatted prompt to the OpenAI LLM.
3. Parse the LLM’s response using the StrOutputParser.

Step 7: Output

The final output will be a string containing the LLM's response about the benefits of walking a mile a day, framed as if it is coming from a wellness expert specializing in cardio activities.

```
output = chain.invoke({"input": "What are the benefits of walking a mile a day?"})
```

Congratulations, you just adopted a simple yet powerful way to interact with an LLM using LangChain and used structured inputs and outputs in a chain-like processing flow.

Full End-to-End Working Code

Below is the full end-to-end working code:

```
!pip install langchain_core==0.2.17 langchain_openai==0.1.16
from langchain_openai import OpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

output_parser = StrOutputParser()

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a world recognized wellness expert especially in cardio activities."),
    ("user", "{input}")
])

llm = OpenAI(api_key="your open AI key")

# Create a chain that defines how your application interacts with the LLM
chain = prompt | llm | output_parser
```

```
output = chain.invoke({"input": "What are the benefits of
walking a mile a day?"})

print(output)
```

Note When running this code, if you happen to run into error, check with Gemini or ChatGPT by pasting the error you are facing, and they should point you to the correct syntax based on the different versions of OpenAI and LangChain you may be using.

Building a Conversational App

Chatting with chat models is quite straightforward once you get the hang of it. First off, you will need to import the ChatOpenAI class from langchain_openai. This will let you interface with OpenAI’s chat models like GPT-3.5 Turbo and GPT-4.

Here is how you do it.

Step 1: Importing Necessary Modules

First, you must import the required modules for interacting with OpenAI’s chat models, formatting messages, and handling environment variables:

```
# Import the necessary modules
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage
import os
```

Step 2: Setting the API Key

You must set the OpenAI API key as an environment variable. This is a more secure practice than hardcoding the key in your script. You must remember to replace “Your OpenAI Key” with your actual API key:

```
# Set the API key as an environment variable (more secure)
os.environ["OPENAI_API_KEY"] = "Your OpenAI Key"
```

Step 3: Creating a ChatOpenAI Object

You must create an instance of the ChatOpenAI class, which represents the chat-based model. The model parameter specifies the specific model to use, in this case, "gpt-3.5-turbo-0125." The openai_api_key parameter is set to your OpenAI API key, which is required to authenticate and access the OpenAI API. The temperature parameter is set to zero, which means the model will generate deterministic responses:

```
# Create a ChatOpenAI object
chat_model = ChatOpenAI(model="gpt-3.5-turbo-0125", openai_api_key="Your OpenAI Key", temperature=0)
```

Step 4: Defining a Function to Generate Responses

You must define a function called generate_response that takes a single parameter text to generate a response from the chat model based on the input text:

```
# Create a function to generate responses
def generate_response(text):
    messages = [HumanMessage(content=text)]
    response = chat_model.invoke(messages)
    return response.content
```

Let us discuss the above code further:

- **messages = [HumanMessage(content=text)]**: Inside the generate_response function, you create a list called messages that contains a single HumanMessage object. The HumanMessage class represents a message sent by a human user, and it takes the input text as its content.

- **response = chat_model.invoke(messages):** You invoke the chat model using the invoke method of the ChatOpenAI object. You pass the messages list as an argument, which contains the human message. The chat model processes the message and generates a response.
- **return response.content:** You return the content of the generated response. The response object returned by the chat model has a content attribute that contains the actual text of the response.

Step 5: Creating the Main Interaction Loop

Below, you start an infinite loop that allows the user to interact with the chatbot repeatedly:

```
# Create a loop to interact with the chatbot
while True:
    # Get user input
    user_input = input("Enter a message: ")

    # Generate a response
    response = generate_response(user_input)

    # Print the response
    print(response)
```

Here is the full end-to-end working code for a chatbot:

```
!pip install openai==1.35.13 langchain==0.2.7 langchain_
openai==0.1.16
!pip show openai langchain langchain_openai
# Import the necessary modules
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage
import os
```

```

# Set the API key as an environment variable (more secure)
os.environ["OPENAI_API_KEY"] = "your open ai key"

# Create a ChatOpenAI object
chat_model = ChatOpenAI(model="gpt-3.5-turbo-0125",
temperature=0)

# Create a function to generate responses
def generate_response(text):
    messages = [HumanMessage(content=text)]
    response = chat_model.invoke(messages)
    return response.content

# Create a loop to interact with the chatbot
while True:
    # Get user input
    user_input = input("Enter a message (or 'quit' to exit): ")

    if user_input.lower() == 'quit':
        break

    # Generate a response
    response = generate_response(user_input)

    # Print the response
    print("AI:", response)

```

Excellent. You just created a chatbot. In summary, you have used the `langchain_openai` module to interact with the OpenAI chat-based model, specifically GPT-3.5-turbo. You created an instance of the `ChatOpenAI` class, defined a function called `generate_response` that takes user input and generates a response from the chat model, and started an infinite loop that allows the user to interact with the chatbot repeatedly. The user enters a message, the code generates a response using the chat model, and then it prints the response to the console.

Difference Between the Q&A and Chatbot Example

In this section, I want to walk you through the differences between the code in the Q&A and chatbot examples.

Key differences

1. Model Used
 - Q&A App: Uses the OpenAI class
 - Chatbot: Uses the ChatOpenAI class, specifically for chat models
2. Prompt Handling
 - Q&A App: Uses ChatPromptTemplate to structure the prompt
 - Chatbot: Directly uses HumanMessage for each input
3. Chain vs. Direct Invocation
 - Q&A App: Creates a chain (prompt | llm | output_parser)
 - Chatbot: Directly invokes the model for each input
4. Interaction
 - Q&A App: Single invocation
 - Chatbot: Continuous loop for multiple interactions
5. API Key Handling
 - Q&A App: Passes the API key directly to the model
 - Chatbot: Uses an environment variable for the API key (more secure)

6. Output Parsing

- Q&A App: Uses `StrOutputParser`
- Chatbot: Directly accesses response content

7. Use Case

- Q&A App: Set up for a specific wellness expert scenario
- Chatbot: Generic chatbot without a specific role

8. User Input

- Q&A App: Hardcoded input
- Chatbot: Takes input from the user in real time

The Q&A App code is more focused on a single-turn question and answer interaction with a specific prompt structure that sets up a context (wellness expert). It is designed to take a single input and provide a single output. This structure is typical of Q&A applications where you have a specific domain or context and want to get an answer to a single query.

Chatbot is set up for continuous interaction. It doesn't have a predefined context or role for the AI. It can handle multiple back-and-forth exchanges. This structure is characteristic of chatbots, where the conversation can flow naturally and cover various topics.

Error Handling and Troubleshooting

When working with large language models (LLMs) and their APIs, you may face various issues that can impact your application's performance and reliability. Let us explore a few strategies and best practices for diagnosing and resolving common problems you might face.

Understanding Common Errors

It is important for you to be familiar with the types of errors you may encounter that I have listed below:

API Connectivity Issues: These occur when there are problems connecting to the API. These can be due to network issues, incorrect API endpoints, or even downtime on the API provider's side.

Authentication Errors: These happen when your API key is missing, expired, or incorrect.

Rate Limiting: This occurs when you exceed the number of requests allowed within a specific timeframe.

Invalid Request Errors: These occur when the API cannot process an incoming request. It could be due to incorrect parameters or unsupported operations.

Model-Specific Limitations: You may also face errors specific to an LLM, for example, due to token limits for inputs and outputs.

Implementing Error Handling in Code

To implement error handling gracefully, wrap your API calls in try-except blocks to catch exceptions. Here is an example using Python with OpenAI and LangChain:

```
from openai import openai
from langchain_openai import ChatOpenAI
from openai import APIError, AuthenticationError,
RateLimitError
```

```

# Initialize the OpenAI client
client = OpenAI()
# Initialize LangChain's ChatOpenAI
llm = ChatOpenAI()
try:
    # Code that might raise an error
    # For example:
    response = llm.invoke("Hello, how are you?")
print(response)
except AuthenticationError as e:
    print(f"Authentication problem: {e}")
except RateLimitError as e:
    print("Rate limit exceeded. Try again later.")
except APIError as e:
    print(f"API Error: {e}")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
finally:
    # Code that always executes, regardless of whether an
    # error occurs
    print("Operation completed.")

```

Diagnosing and Resolving Common Issues

Here are some tips for diagnosing and resolving common issues:

API Connectivity Issues: Check your Internet connection, verify the API endpoint, and look for any announcements from the API provider regarding downtime or maintenance.

Authentication Errors: Ensure your API key is correctly set in your environment variables or passed to your API client. If required, regenerate your API key from the provider's dashboard.

Rate Limiting: Implement retry logic with exponential backoff. This means if a request fails due to rate limiting, wait for a predetermined period, then try again, increasing the wait time with each attempt.

Invalid Request Errors: You should check the API documentation to ensure you are using the correct parameters and data formats. In your code, add input validation to catch errors early.

Model-Specific Limitations: Ensure you are aware of the specific limitations of the LLMs you are using, such as token limits. One solution is to break down larger requests into smaller ones if necessary.

Logging and Monitoring: Implement logging to capture detailed information about errors and exceptions. This data is invaluable for troubleshooting issues. Consider using monitoring tools to track the performance and health of your API interactions over time.

Hope this helps you to build resilient AI applications using LLMs. Remember, your goal must be to not just handle errors when they occur but also to prevent them through careful planning and testing.

Development Playground

I have added a few suggestions below for you to practice experimenting with LangChain and LLMs in a controlled, interactive environment.

LangChain Playground

LangChain has a playground where you can freely experiment with LangChain’s capabilities without any setup required on your end. As shown in Figure 3-1, it is a web-based interface where you can write, execute, and test prompts directly in your browser.

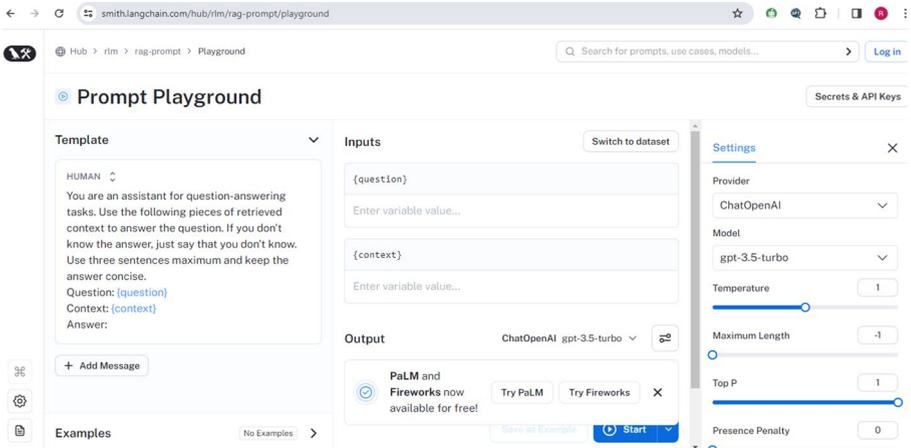


Figure 3-1. LangChain Playground

You can also explore LangChain’s GitHub repository and similar resources to practice.

OpenAI API Playground

Link: <https://platform.openai.com/playground>

OpenAI's Playground provides an intuitive interface for interacting with OpenAI's models directly through your web browser. You can experiment with different prompts, settings, and models, including the latest versions of GPT.

I recommend you use the Playground to understand how different prompts and parameters affect the output of GPT models. It will help you learn how to structure your interactions when implementing LangChain in your projects.

Hugging Face Spaces

Link: <https://huggingface.co/spaces>

Hugging Face Spaces hosts a variety of machine learning models, including LLMs. You can interact with pre-built applications or deploy your own and start experimenting.

Search for Spaces that use LangChain and start interacting with these applications to gain a sense of how you can combine LLMs with web applications. You can also consider deploying your own LangChain-based project on Spaces to get feedback from the community.

Colab Notebooks

Link: <https://colab.research.google.com>

Google Colab offers a free Jupyter notebook environment that requires no setup and runs entirely in the cloud. You can use it to write and execute Python code, and it integrates with GitHub and other external datasets.

Use it to create or search for existing notebooks that demonstrate LangChain and LLM interactions.

Kaggle Notebooks

Link: <https://www.kaggle.com/code>

Kaggle provides a cloud-based Jupyter notebook environment similar to Google Colab. It is integrated with Kaggle's competitions and datasets, but you can use it for any data science or machine learning project.

Use Kaggle to explore notebooks that feature LLM experiments. Its vast dataset repository can also be a valuable resource for feeding real-world data into your LangChain experiments.

Maximize Your Learning Through Experimenting

I recommend you continue your learning through continuous practice and experimentation. Here are some suggestions.

Experiment Freely

- Don't be afraid to try out different models, prompts, and configurations.
- Remember that learning what doesn't work is just as important as finding what does.

Document Your Findings

- Keep detailed notes on your experiments, especially on the prompt structures and parameters that lead to interesting results.

- Use a structured format to document your findings, such as a spreadsheet or a dedicated experimentation tool.
- Include information on the model used, the prompt, the parameters, and the outcome of each experiment.

Share and Collaborate

- Leverage platforms like GitHub, Kaggle, and Hugging Face to share your work and collaborate with others.
- Create clear and well-documented repositories or notebooks to showcase your experiments and findings.
- Engage with the community by participating in discussions, asking for feedback, and offering insights on others' work.

Review Questions

Here are some questions to help you synthesize the knowledge gained from the chapter and apply it to broader contexts and future projects.

1. What is the primary advantage of using LangChain's model I/O for LLM integration?
 - A. It simplifies the computational requirements.
 - B. It allows for direct access to the Internet.
 - C. It simplifies the integration of LLMs into applications.
 - D. It increases the cost-efficiency of server maintenance.

2. Which types of applications did you learn to build in this chapter?
 - A. Database management systems
 - B. Q&A and conversational apps
 - C. Financial forecasting tools
 - D. Ecommerce platforms
3. Which tool is recommended for setting up your development environment for LangChain and OpenAI?
 - A. Microsoft Excel
 - B. Google Colaboratory
 - C. Adobe Photoshop
 - D. AutoCAD
4. What is a key strategy for effective error handling in application development?
 - A. Ignoring minor errors
 - B. Hardcoding all responses
 - C. Using try and except blocks
 - D. Limiting user input
5. What does continuous learning in the context of LLM application development involve?
 - A. Regularly updating your personal blog
 - B. Engaging with interactive platforms and resources
 - C. Attending annual tech conferences only
 - D. Reading fiction to boost creativity

6. Which module do you import from LangChain to interact with OpenAI's language models?
 - A. langchain.graphics
 - B. langchain.llms
 - C. langchain.audio
 - D. langchain.visuals

Answers

1. C. It simplifies the integration of LLMs into applications.
2. B. Q&A and conversational apps.
3. B. Google Colaboratory.
4. C. Using try and except blocks.
5. B. Engaging with interactive platforms and resources.
6. B. langchain.llms

Additional Review

- Describe how LangChain's model I/O benefits the integration of LLMs into your applications. What specific features make it advantageous over direct API calls?
- Explain the process of setting up a development environment for working with LangChain and OpenAI. What are the key components you need to configure?

- Discuss the types of applications you learned to build in this chapter. How do Q&A apps differ from conversational apps in terms of their development and functionality?
- What are some effective error handling strategies you learned in this chapter? How do these strategies improve the robustness of LLM applications?
- Reflect on the importance of continuous learning in the field of LLM application development. How can engaging with interactive platforms and resources enhance your skills and keep you updated with the latest developments?

Key Takeaways

The journey through this chapter is just the beginning. As you move forward, remember that the skills and concepts you have acquired here will serve as the foundation for more complex and diverse applications.

Familiarity with Framework: You learned how LangChain's model I/O simplifies the integration of LLMs like OpenAI into your applications, enhancing both functionality and user experience.

Diversity of LLM: You explored the variety of LLMs available within LangChain, including specialized models for general queries and conversational interactions.

Development Setup: You set up a functional development environment, prepared all necessary tools, and gained confidence in configuring LangChain and OpenAI APIs.

Building Applications: You built and deployed two key types of applications – a Q&A app and a conversational app using Python demonstrating practical application of your skills.

Mastering Troubleshooting: You learned effective error handling and troubleshooting strategies, ensuring your applications run smoothly and reliably.

Continuous Learning: I hope by now you have also engaged with interactive learning resources to continuously improve your skills and adapt to new developments in LLM technology and application building.

LangChain’s modular and standardized approach offers a flexible and scalable framework that will continue to help you in your AI development efforts. In the chapters to come, I will show you more exciting possibilities that LangChain brings to the table.

Glossary

API Key: A unique identifier used to authenticate a user, developer, or calling program to an API. It is essential for tracking API usage and ensuring secure access to the service.

Chat Model: A type of LLM designed for generating responses in a conversational format, simulating a dialogue between humans and AI.

Completion: In the context of LLMs, a completion refers to the text generated by the model in response to a prompt or input provided by the user.

Hugging Face Spaces: An online platform for hosting and sharing machine learning projects, including those involving LLMs. It allows developers to create and share interactive ML demos.

Model IO (Input/Output): Refers to the processes of sending inputs to and receiving outputs from a machine learning model. In the context of LangChain, it relates to how prompts are sent to LLMs and how their responses are handled.

Prompt Templates: Predefined structures or formats for creating prompts to send to LLMs, facilitating consistent and effective communication with the models.

Streamlit: An open source Python library for creating and sharing beautiful, custom web apps for machine learning and data science projects.

This glossary serves as a quick reference to navigate the concepts and techniques involved in developing applications with LLMs and LangChain.

Further Reading

Below resources can help deepen your understanding of the topics covered in this chapter:

“Design Patterns for Large Language Models” – Arxiv Paper: Explores various strategies and patterns for effectively utilizing LLMs in application development. <https://arxiv.org/abs/2004.13214>

OpenAI API Documentation: The official documentation for the OpenAI API, provides detailed information on using different models, including GPT-3 and Codex. <https://platform.openai.com/docs>

LangChain GitHub Repository: The source for LangChain code, documentation, and examples, is invaluable for developers looking to integrate LLMs into their projects. <https://github.com/LangChain/langchain>

“Prompt Engineering for GPT-3” on OpenAI Blog: Discusses strategies for designing effective prompts to elicit desired responses from GPT-3, a crucial skill for developing applications with LLMs. <https://www.openai.com/blog/prompt-engineering/>

Hugging Face Forums: A community forum for discussions on machine learning, with a strong focus on NLP and transformer models. Great for staying updated on the latest research and practical applications. <https://discuss.huggingface.co/>

r/MachineLearning on Reddit: A subreddit dedicated to machine learning, where practitioners and researchers share news, discuss projects, and ask for advice on a wide range of topics related to AI and NLP.

These resources will help you expand your knowledge and skills in LLMs and application development, from theoretical foundations to practical implementation strategies.

<https://medium.com/@swethag04/build-a-q-a-app-for-a-webpage-431b7b8220e6>

https://colab.research.google.com/drive/19a_v8tBN4fwELxTn04zN1NHQWb0BtMB1?usp=sharing

CHAPTER 4

Exploring Large Language Models (LLMs)

In this chapter, we will quickly review a number of large language models (LLMs) from major providers such as OpenAI, Google, Anthropic's Claude, Cohere, and Meta. To make it interesting, I have taken a hypothetical case study where you have been asked by your company to create an intelligent, highly responsive customer service chatbot that not only answers simple queries but also thrills your customers by anticipating their unmet needs and responding accordingly. I will take you through the process of how you could possibly go about exploring various large language models (LLMs), like GPT-4, Codex, DALL-E 2, and more from leading providers like OpenAI, Google, Anthropic, Cohere, and Meta.

This chapter should equip you with practical skills and knowledge to understand, integrate, and apply these models in your own generative AI applications.

OpenAI's Models

Let us discuss the OpenAI models first.

GPT: The Next-Generation Language Models

To begin with, you decide to start your model exploration effort by looking at OpenAI as it offers a wide range of models with their own unique capabilities and strengths. You are particularly impressed with GPT-4 and GPT-4o, the most popular among the large language models both within and outside of OpenAI. Your research has shown that you can build incredible apps like chatbots, content creation tools, and AI writing assistants. You are also impressed by their ability to understand and generate human-like text with amazing accuracy as they have been trained on a massive amount of data. Now you want to research more by calling a GPT model.

Getting Started with GPT (Code Snippet)

To get started with GPT, you install the OpenAI Python package. You then provide a prompt to the GPT-4 model and specify the maximum number of tokens (words or subwords) you want it to generate. You also use the temperature parameter to control the creativity of the generated text.

The higher the temperature, the more creative and unpredictable the output will be.

Below is a sample code you can use to call a GPT model:

```
!pip install openai==1.3.7
from openai import OpenAI
import os

# You need to set your API key as an environment variable for
security
os.environ["OPENAI_API_KEY"] = "Your OpenAI key"
```

```

# Create an OpenAI client
client = OpenAI()

# Here's our business-oriented prompt
prompt = "Provide a brief market analysis for a new
eco-friendly, reusable water bottle."

# Now we'll set up our request to the AI
response = client.chat.completions.create(
    model="gpt-4",
    messages=[
        {"role": "system", "content": "You are a market
research analyst with expertise in consumer goods."},
        {"role": "user", "content": prompt}
    ],
    max_tokens=150,
    n=1,
    temperature=0.6,
)

# Extract and print the generated analysis
generated_analysis = response.choices[0].message.content
print("Market Analysis:")
print(generated_analysis)

```

Codex: Your AI Coding Assistant

Next, you decide to explore OpenAI's Codex that helps you write code.

To start using Codex, you integrate it into your favorite code editor. Go to <https://openai.com/index/openai-codex/> and click "Start Using Codex" and then "Try GPT-4o." Please note that these features are constantly evolving, and these links may not work by the time you read this book, but these features would be still present and have evolved.

Here is a quick example of sample Python code that was generated by Codex based on a prompt to generate a function that adds two numbers:

```
# Ask Codex to generate a function that adds two numbers
def add_numbers(a, b):
    """
    Add two numbers together
    """
    return a + b
```

The above example shows how you can speed up your coding workflow and get instant suggestions for completing your code using Codex. It can assist you with tasks like

- Completing function implementations based on comments or docstrings
- Generating boilerplate code for common programming patterns
- Providing you with suggestions for optimizing or refactoring your code

DALL-E 2: The Image Generation Wizard

You then decide to bring your ideas to life visually with DALL-E 2 from OpenAI. You just need to provide a textual description, and DALL-E 2 will generate an image based on it. Imagine the possibilities, for example, you can create unique illustrations, design elements, or visual concepts simply by describing them in words.

Here is a simple example that shows how you can use the OpenAI API to create stunning images based on your prompts:

```
# Install the openai package
!pip install openai==0.28
```

```
# code begins here
import openai
openai.api_key = "YOUR_API_KEY"
def generate_image(prompt, num_images=1, size="1024x1024"):
    try:
        response = openai.Image.create(
            prompt=prompt,
            n=num_images,
            size=size
        )
        image_urls = [data['url'] for data in response['data']]
        return image_urls
    except Exception as e:
        print(f"Error generating image: {e}")
        return None

# Example usage
prompt = "A mobile phone displaying a customer support chatbot
app interface"
image_urls = generate_image(prompt)

if image_urls:
    for url in image_urls:
        print(url)
else:
    print("Failed to generate image.")
```

- In this example, you provide a textual description of the image you want to generate. You can define the `generate_image` function to take the following parameters:
 - **prompt:** The text description of the image you want to generate.
 - **num_images:** The number of images to generate (default is 1).
 - **size:** The size of the generated image (default is “1024x1024”).
- Inside the `generate_image` function, you will make a request to the OpenAI API using `openai.Image.create()`, passing the `prompt`, `num_images`, and `size` parameters.
- If the request is successful, you extract the image URLs from the response data and return them as a list.
- If an exception occurs during the API request, you print an error message and return `None`.
- If the `image_urls` list is not empty, you can iterate over each URL and print it. Otherwise, you can print a failure message.

Figure 4-1 shows an image that was generated.

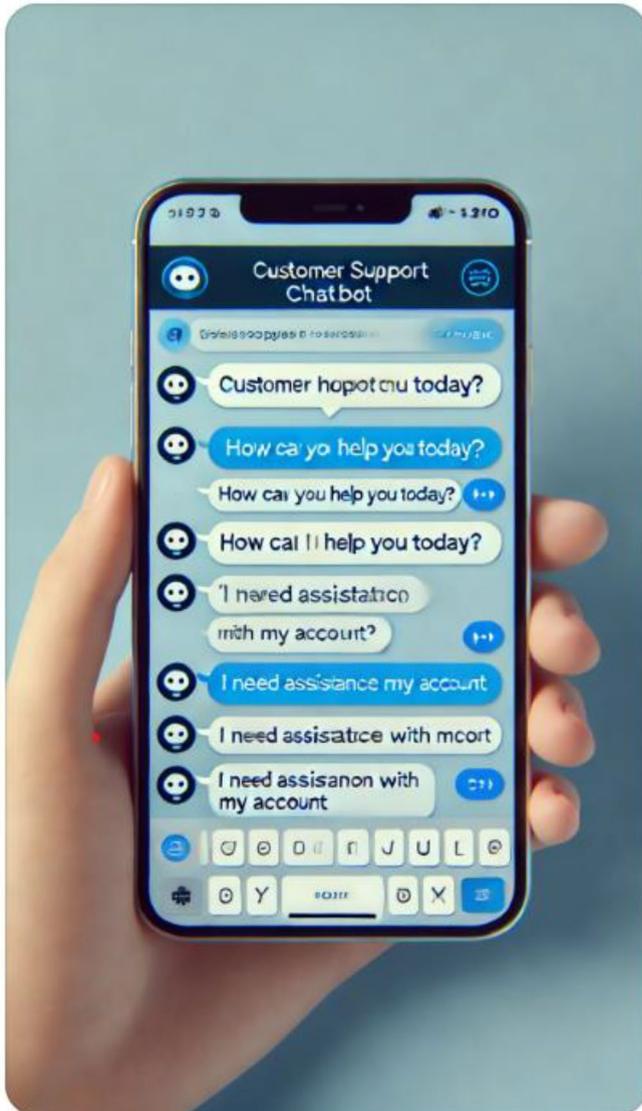


Figure 4-1. Example of a DALL-E Generated Image

OpenAI also offers models for speech recognition (**Whisper**), text-to-speech (**TTS**), and even text moderation models that you can use to build responsible generative AI applications.

Google's AI Model Overview

Having explored the OpenAI models, now let us explore the equally impressive list of LLM models from Google. You will realize that Google has you covered whether you want to tackle natural language processing tasks or solve computer vision problems.

Language and Chat Models

You have decided to first start with the language and chat models.

Gemini 1.0 Pro

Your research has shown that Gemini 1.0 Pro too is good at understanding and generating human-like text. It could be your go-to model for tackling text completion, summarization, question answering, and language translation tasks with great accuracy. You decide to keep Gemini 1.0 Pro as a suitable candidate for building sophisticated text-based applications given its deep understanding of complex language patterns.

Some practical use cases are content creation and curation such as generating high-quality articles, reports, and creative writing pieces, customer support bots for personalized customer service and recommendations, language translation services for global communication, and interactive educational learning tools.

Code Example: Calling Google Cloud Natural Language API for Text Generation

Let us switch gears and create our first Google generative AI app.

Go to the Google Cloud Console (see Figure 4-2) to create your Google project.

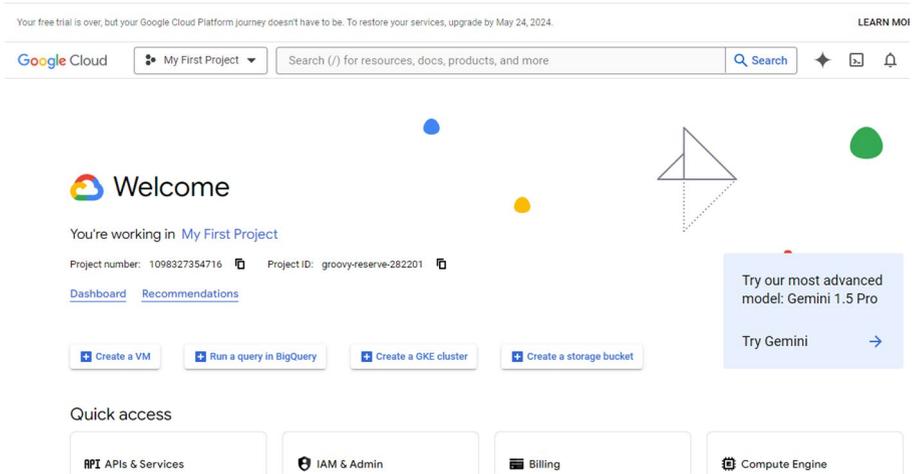


Figure 4-2. Google Console Page

Step 0: Creating Your API Keys

Go to **APIs and Services** and click **Credentials** on the left menu and **Create Credentials** ► **API Key** (see Figure 4-3).

CHAPTER 4 EXPLORING LARGE LANGUAGE MODELS (LLMS)

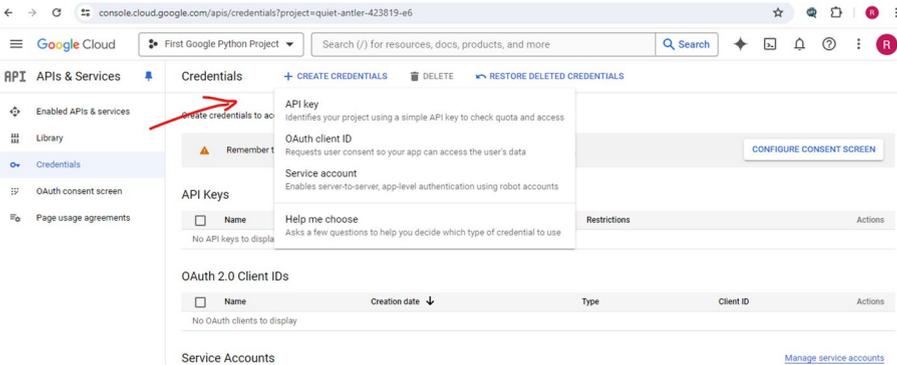


Figure 4-3. *Creating Your API Key*

Step 1: Create a Google Cloud Project

Go to the Google Cloud Console. The URL now is <https://console.cloud.google.com/welcome?pli=1>. As always, these links can change.

Create a new project.

Click the project drop-down at the top of the page.

Click “New Project.”

Enter the project name and other details.

Click “Create.” See Figure 4-4.

☰ Google Cloud

New Project

▲ You have 21 projects remaining in your quota. Request an increase or delete projects. [Learn more](#)

[MANAGE QUOTAS](#)

Project name *
 ?

Project ID: quiet-antler-423819-e6. It cannot be changed later. [EDIT](#)

Location *
 [BROWSE](#)

Parent organization or folder

[CREATE](#) [CANCEL](#)

Figure 4-4. *Creating Your First Google Project*

Step 2: Enable the Google Cloud Natural Language API

Navigate to the APIs & Services Dashboard.

From the left-hand menu, select “APIs & Services” ► “Dashboard.”

Enable the API.

Click “+ ENABLE APIS AND SERVICES.”

Search for “Natural Language API” (see Figure 4-5).

CHAPTER 4 EXPLORING LARGE LANGUAGE MODELS (LLMS)

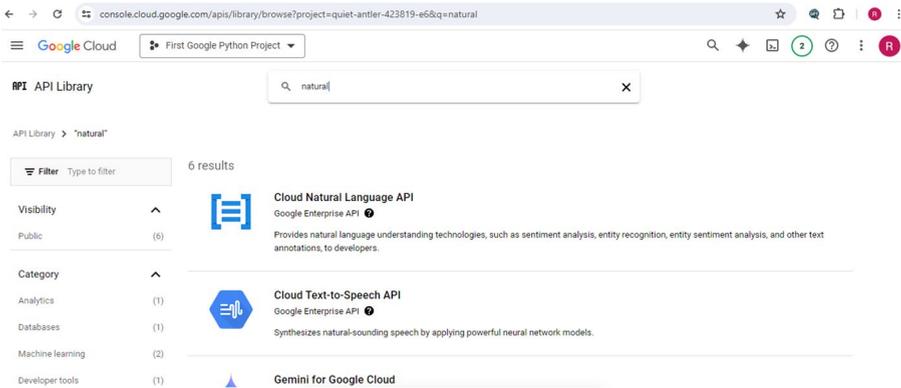


Figure 4-5. Enter Search to Find APIs

Click the “Cloud Natural Language API” result.
Click “Enable.” See Figure 4-6.

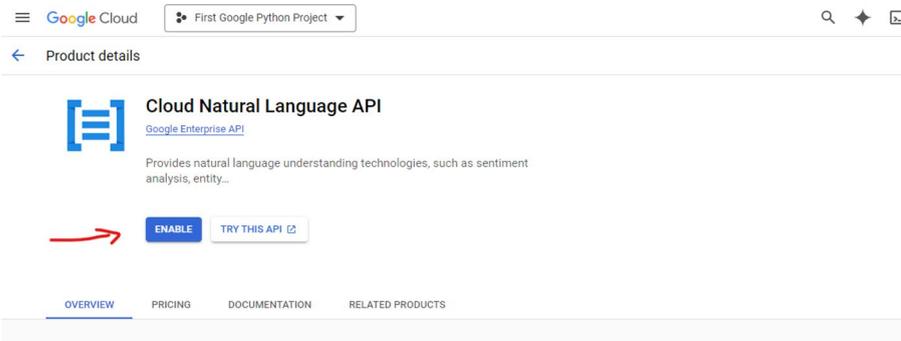


Figure 4-6. Enable Cloud Natural Language API

Note that you may have to create a billing account to use the API. Proceed with the steps when presented with a pop-up like shown in Figure 4-7.

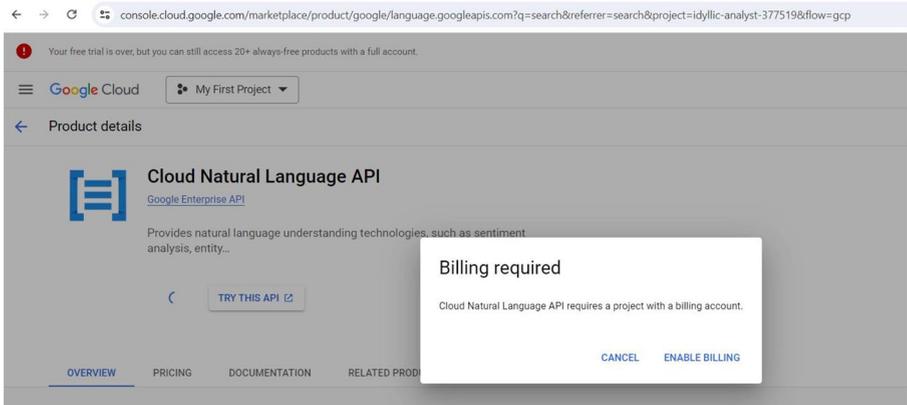


Figure 4-7. Enable Billing

Step 3: Create a Service Account

Navigate to the Service Accounts Page. See Figure 4-8.

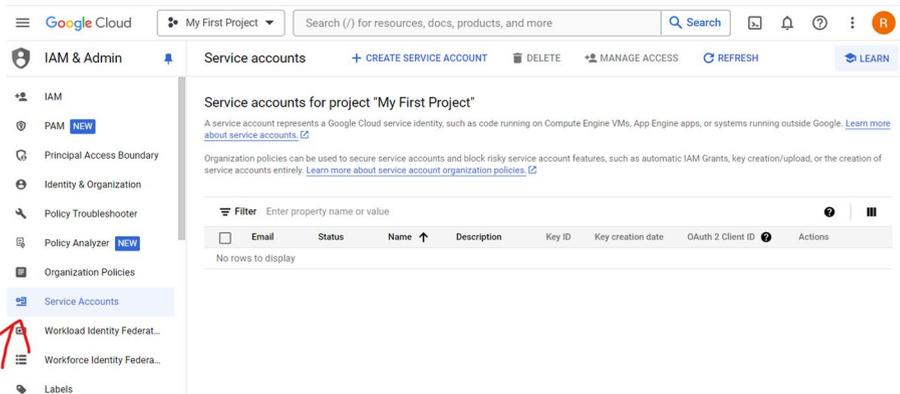


Figure 4-8. Creating a Service Account

From the left-hand menu, go to “IAM & Admin” ► “Service Accounts.”

Create a new Service Account (see Figure 4-9).

Click “Create Service Account” at the top.

Enter a name and ID for the service account and an optional description.

Click “Create and Continue.”

Grant this service account access to the project.

Select a role for the service account. For full access, you can choose “Owner,” but for more restricted access, you can choose “Cloud Natural Language API User.”

Click “Continue.”

Optional: Grant users access to this service account.

You can skip this step and click “Done.”

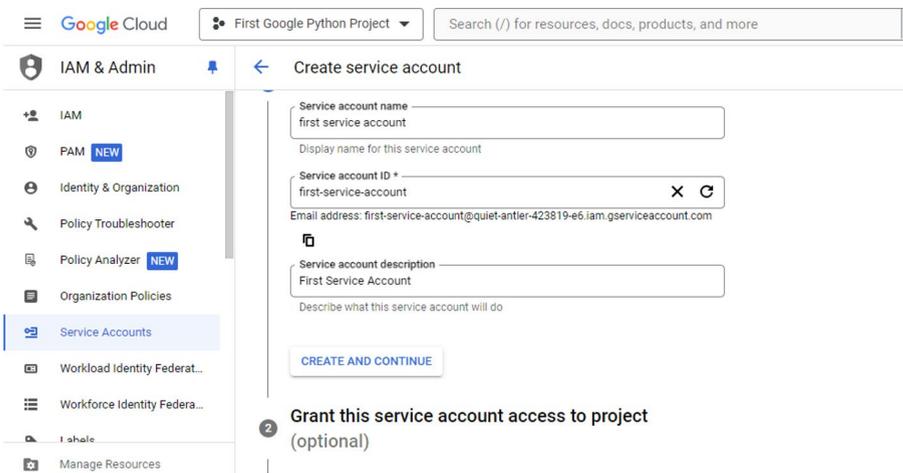


Figure 4-9. Granting Access to the Project

Step 4: Create and Download the JSON Key File

Navigate to the Service Accounts Page.

From the left-hand menu, go to “IAM & Admin” ► “Service Accounts.”

Select the Service Account.

Find the service account you created and click it.

Click the “Keys” tab.

Click “Add Key” ► “Create New Key.”

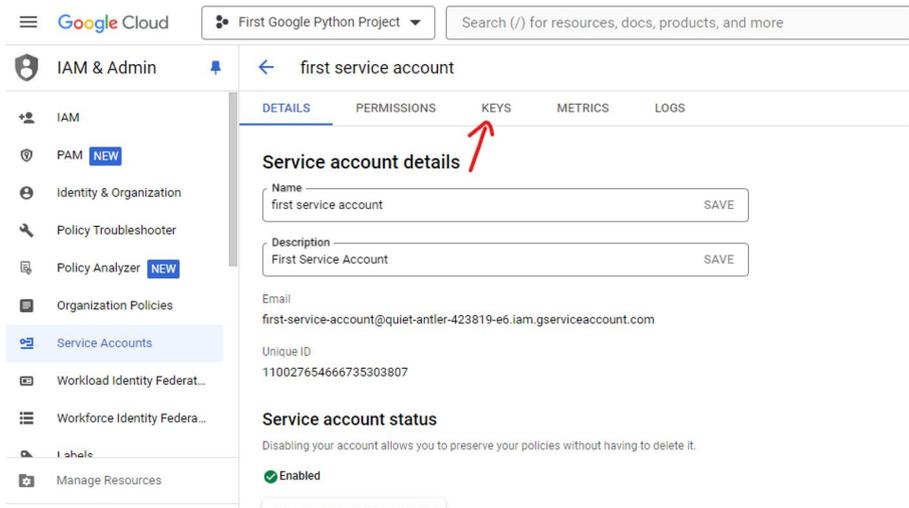


Figure 4-10. Downloading the JSON Key File

Select “JSON” and click “Create.”

The JSON key file will be downloaded to your computer. Save this file securely as it contains sensitive information.

Step 5: Set the Environment Variable

Set the `GOOGLE_APPLICATION_CREDENTIALS` Environment Variable by pointing to the path of the JSON key file you downloaded.

You can set this environment variable in your terminal or in your Python script as shown below.

Example (Terminal)

```
sh
export GOOGLE_APPLICATION_CREDENTIALS="path/to/your/service-account-file.json"
```

Example (Python Script)

```
import os
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "path/to/your/service-account-file.json"
```

Full Example Python Script

Here is the complete Python script to analyze text using the Google Cloud Natural Language API.

First, let us install the necessary libraries:

```
!pip install google-cloud-language==2.13.3
!pip install google-cloud-vision==3.7.2
!pip install google-cloud-translate==3.11.3

#import the necessary classes
from google.cloud import language_v1
import os

# Set up authentication
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "path/to/your/
service-account-file.json"

def analyze_text(text):
    # Initialize the LanguageServiceClient
    client = language_v1.LanguageServiceClient()

    # Create a document object with the text to be analyzed
    document = language_v1.Document(content=text,
    type_=language_v1.Document.Type.PLAIN_TEXT)

    # Use the client to analyze the sentiment of the document
    response = client.analyze_sentiment(document=document)

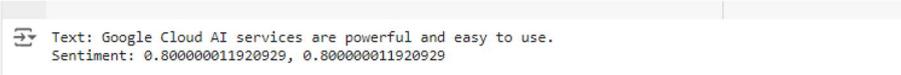
    # Extract the sentiment score and magnitude from the
    response
    sentiment = response.document_sentiment

    # Print the results
    print(f"Text: {text}")
    print(f"Sentiment score: {sentiment.score}")
    print(f"Sentiment magnitude: {sentiment.magnitude}")
```

```
# Example text to analyze
text = "Google Cloud AI services are powerful and easy to use."

# Call the function to analyze the text
analyze_text(text)
```

Here is the output I got:



```
Text: Google Cloud AI services are powerful and easy to use.
Sentiment: 0.800000011920929, 0.800000011920929
```

Interpreting the Score

Let us interpret the score.

Sentiment

Sentiment: 0.800000011920929, 0.800000011920929

This line shows the sentiment analysis results, which consist of two values: sentiment score and sentiment magnitude.

Sentiment Score

The first value 0.800000011920929 represents the sentiment score.

The sentiment score ranges from -1.0 (very negative) to 1.0 (very positive).

A score of 0.800000011920929 indicates that the text is strongly positive.

Sentiment Magnitude

The second value 0.800000011920929 represents the sentiment magnitude.

The sentiment magnitude indicates the overall strength of emotion (positive or negative) expressed in the text.

It is a non-negative number that increases with the amount of emotional content. In this case, 0.800000011920929 suggests a moderate amount of positive sentiment.

Interpretation

The analyzed text, “Google Cloud AI services are powerful and easy to use,” is identified as having a strongly positive sentiment, as indicated by a high sentiment score close to 1.0.

The sentiment magnitude is also relatively high, reflecting a moderate level of emotional intensity in the positive sentiment.

Conclusion

The Google Cloud Natural Language API has correctly identified the input text as positive. It also gives you a sentiment score and magnitude that quantitatively represent this positive sentiment. This is useful in applications where you want to understand the sentiment of user feedback, reviews, or any text content that is crucial for insights and decision-making.

By following these steps, you now have a Google Cloud project with the Natural Language API enabled, a service account with the necessary permissions, and a JSON key file for authentication. You have learned a lot, and now you know how to use the Google Cloud Natural Language API in your Python applications securely. And it opens up more avenues for you to leverage many other models from Google for vision, image, audio, and video.

Gemini 1.0 Pro Vision

You continue to explore Google 1.0 Pro Vision, which allows you to include images and videos in your prompts, thanks to its multimodal input capabilities. This means you will get text or code responses that correspond to the visual content you provide. You realize that Gemini 1.0 Pro Vision opens up a world of possibilities for you to create generative AI applications that deeply comprehend multimodal data by combining visual and textual understanding.

Based on your research, you have documented a few practical use cases, such as image captioning and analysis, creating art or design concepts based on textual descriptions, video content analysis to generate summaries, captions, and tags for videos. Even in the field of augmented reality (AR), you can dynamically generate text or narratives based on the visual surroundings.

PaLM 2

You then move on to PaLM 2 for Text, which stands out for its ability to follow complex, multistep instructions and excels in zero-shot and few-shot learning scenarios. You discover that you can get impressive results with minimal or no training data, saving you time and resources. You can generate long, coherent responses that stay on track and maintain a clear and logical flow.

Your research further leads to PaLM 2 for Chat, which turns out to be your secret weapon to create engaging conversational AI. It is tailored specifically for chat-based interactions, and it makes conversations feel more natural and engaging. You decide to include it in your list of candidates to build customer support bots, interactive games, or conversational interfaces for IoT devices. You decide to test it out because it has the potential to amaze your users with lifelike and contextually relevant responses.

Codey Suite for Code-Related Tasks

You also want to explore Google's models for coding, so you decide to explore Google's Codey Suite. Your research has led you to the following observations:

Codey for Code Completion: You can speed up your coding with real-time suggestions and auto-completions from Codey.

Codey for Code Generation: You can turn natural language descriptions into syntactically correct and logically sound code.

Codey for Code Chat: You can use this virtual chat assistant for debugging, explaining APIs, and getting coding advice.

Text and Image Processing Models

You then learn about the potential of text and image data using Google models:

- **Embeddings for Text** converts textual data into numerical vectors. You can use them to process and analyze text data for advanced NLP capabilities.
- **Imagen for Image Generation** lets you create or edit high-quality images using text prompts. You can use it to generate visually appealing content at scale.
- **Imagen for Captioning and VQA (Visual Question Answering)** generates relevant descriptions for images, enhancing image understanding and retrieval.

Multimodal and Security Models

You also find out that there are some powerful multimodal and security models from Google to work with:

- **Embeddings for Multimodal Vision** generates vectors based on images. You can use it to build intelligent vision-based applications for tasks like image classification and search.

- **Sec-PaLM2** is your go-to model for security-specific tasks, such as threat intelligence, security operations, and malware analysis. You can use it to develop secure and resilient systems.

Speech Models

You discover Google's Chirp Speech, which is a universal speech model that can transcribe over 100 languages using just one model. You realize that you can use this to develop voice-enabled applications that work globally without needing separate models for different languages.

Getting Started with Google AI Models

Note that to get started with these models, Google provides detailed model cards and quickstarts that offer guidance and code samples. I recommend you explore the documentation, experiment with the provided examples, and adapt them to your specific use cases. Check this link: <https://cloud.google.com/vertex-ai/generative-ai/docs/model-garden/explore-models>.

Anthropic's Claude AI Models

You then move on to Anthropic's Claude models that provide a family of state-of-the-art large language models. Claude 3 family models are the best among the Claude models. These models represent the cutting edge of AI technology with unparalleled performance, versatility, and ease of use.

Claude 3 Model Family: Claude 3 Opus, Sonnet, and Haiku

You go on to compare the Claude 3 family's three models: Opus, Sonnet, and Haiku.

- **Claude 3 Opus**

Claude 3 Opus is the most powerful model that demonstrates fluency and human-like understanding.

- **Claude 3 Sonnet**

Next, you have Claude 3 Sonnet, which strikes the perfect balance between intelligence and speed. You can use it for enterprise workloads and scaled AI deployments. Sonnet is a dependable and balanced model that can handle a wide range of tasks efficiently.

- **Claude 3 Haiku**

Claude 3 Haiku is the fastest and most compact model in the family. You can use it for near-instant responsiveness, and if speed is your top priority, Haiku is your best bet.

Key Features of Claude 3 Models

Here are some of the key features of these models that you have documented:

- **Multilingual Capabilities**

Claude 3 models stand out for their multilingual capabilities. These models have improved fluency in non-English languages like Spanish and Japanese, and, therefore, you can use them for translation and creating global content.

- **Vision and Image Processing**

Another awesome feature is vision and image processing. All Claude 3 models can process and analyze visual input. You can extract insights from documents, process web UI, generate image catalog metadata, and more.

- **Ease of Use and Following Direction**

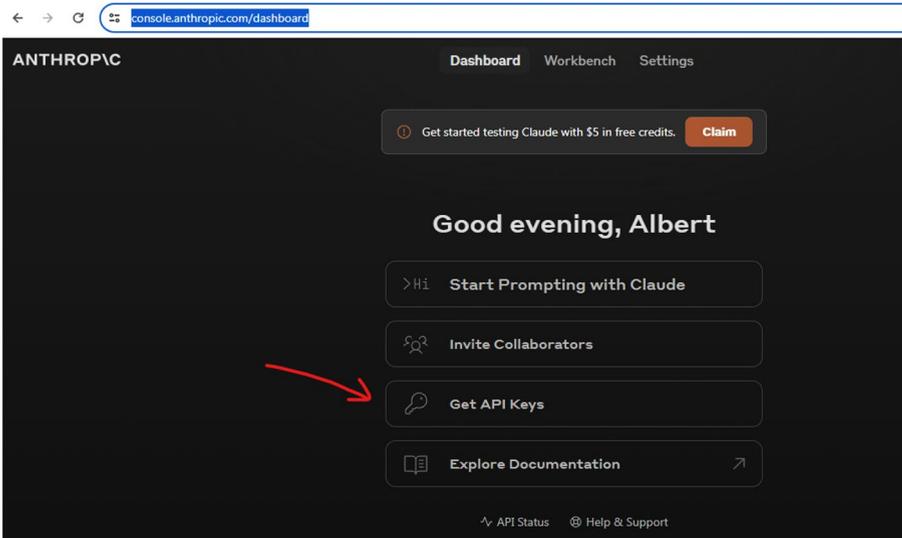
Claude 3 models are easy to use; I have also found them to be better at following directions which allows you to have more control over their behavior and ensure predictable and higher-quality responses.

- **Legacy Models and Transitioning**

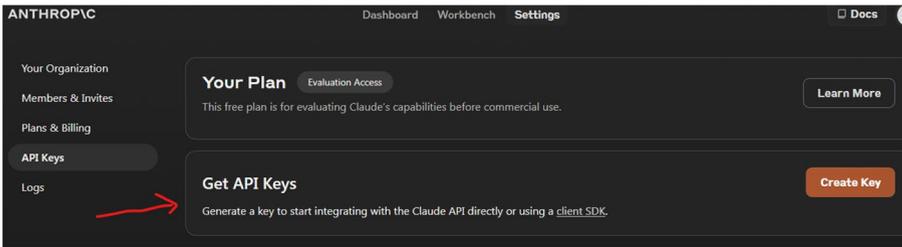
As of this writing, Claude still offers legacy models like Claude 2.0, Claude 2.1, and Claude Instant 1.2 for users who may need time to transition. However, I highly recommend you use the Claude 3 models whenever possible, as they outperform the legacy models in terms of speed, performance, and capabilities.

Getting Started with Code

Now, let us get our hands dirty with some code! To start using Claude, you will need to sign up for an API key by going to <https://console.anthropic.com/dashboard>.

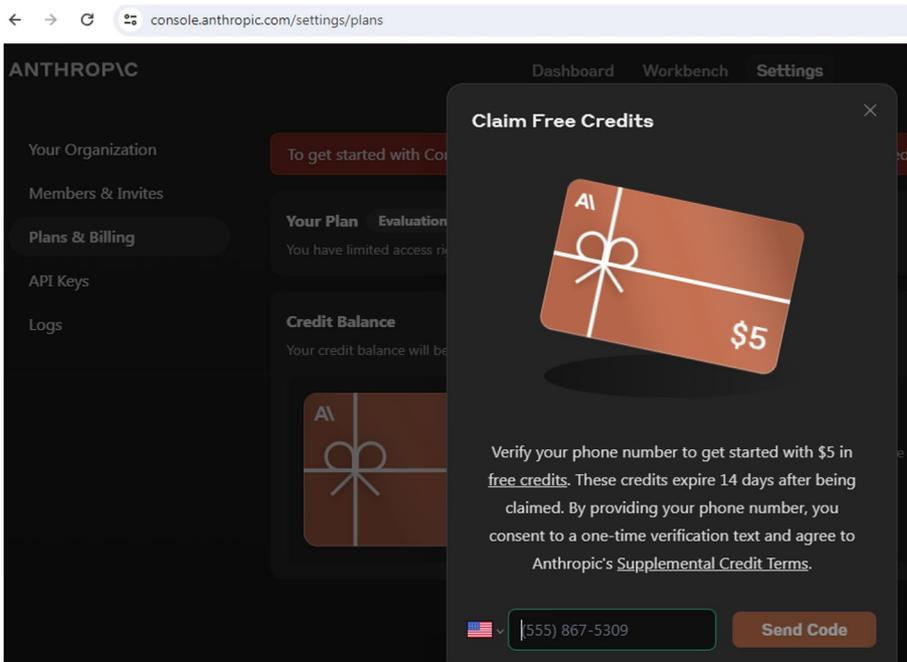


Visit “Get API Keys” and click Create Key.



Enter a name for your key and click “Create Key.” Store the key in a safe place for future use because you will not be able to view it again. Of course, do not share it with anyone.

Once you have your key, you can easily integrate Claude into your Python projects using the Anthropic SDK. Moreover, your code will not work unless you have enough credits, so you can avail their \$5 credit by activating as shown below.



Here is a simple example of how to generate text using Claude 3 Opus:

```

from langchain.chains import LLMChain
from langchain_core.prompts import PromptTemplate,
ChatPromptTemplate
from langchain_anthropic import ChatAnthropic
prompt_template = """
You are an AI assistant created by Anthropic to be helpful,
harmless, and honest.
Please provide a fun fact about AI.
"""
system = ( prompt_template )

```

CHAPTER 4 EXPLORING LARGE LANGUAGE MODELS (LLMS)

```
claude = ChatAnthropic(temperature=0, api_key=ANTHROPIC_API_KEY, model_name="claude-3-sonnet-20240229")
# claude = AnthropicLLM(model="claude-2.1", anthropic_api_key="ANTHROPIC_API_KEY")
# llm_chain = prompt | claude; # LLMChain(prompt=prompt, llm=claude)
if __name__ == "__main__":
    human = "{text}"
    prompt = ChatPromptTemplate.from_messages([("system", system), ("human", human)])
    chain = prompt | claude;
    response = chain.invoke(
    {
    "text": "Give me fun fact about Claude",
    }
    )
    print(response)
----
```

Let us walk through the code:

1. First, you will install and import the necessary modules: `os` for accessing environment variables, `Anthropic` from `langchain.llms` for interacting with the Claude model, and `PromptTemplate` and `LLMChain` from `langchain` for constructing the prompt and chain.
2. You retrieve the Anthropic API key from the environment variable `ANTHROPIC_API_KEY`. Make sure to set this environment variable with your actual API key before running the code.

3. You define the `prompt_template`, which is a string that serves as the template for the prompt. In this example, the prompt asks the AI assistant to provide a fun fact about AI.
4. You create a `PromptTemplate` instance called `prompt` using the `prompt_template`. Since there are no input variables in this example, we pass an empty list `[]` to `input_variables`.
5. You create an instance of the `Anthropic` class called `Claude`, specifying the model name as “`claude-v1`” and providing the `anthropic_api_key`.
6. You create an `LLMChain` instance called `llm_chain`, passing the `prompt` and `Claude` instances as parameters. This chain connects the `prompt` with the `Claude` model.
7. In the `if __name__ == “__main__”:` block, you run the `llm_chain` using the `run()` method, which generates a response from the `Claude` model based on the provided prompt.
8. Finally, you print the generated response.

As you can see, with just a few lines of code, you can use the power of Claude 3 models to generate amazing AI-powered content.

Overview of Cohere AI Models

You then move on to Cohere’s AI model offerings. I have found them to be a little different from other models in the way they have been classified. They have two types of models, namely, generative models and representative models.

You observe that Cohere's generative models, namely, **Command**, **Command-R**, and **Command-light**, have shown impressive capabilities in generating text based on input prompts. The flagship Command model is known to be good from content creation to chatbot development. Command-R model is focused on conversational interaction and long context tasks and has shown promise in applications like educational resources and code documentation. Command-light model is another model that is optimized for speed and efficiency, and you identify it as a good fit for tasks like generating product descriptions or quick customer service responses.

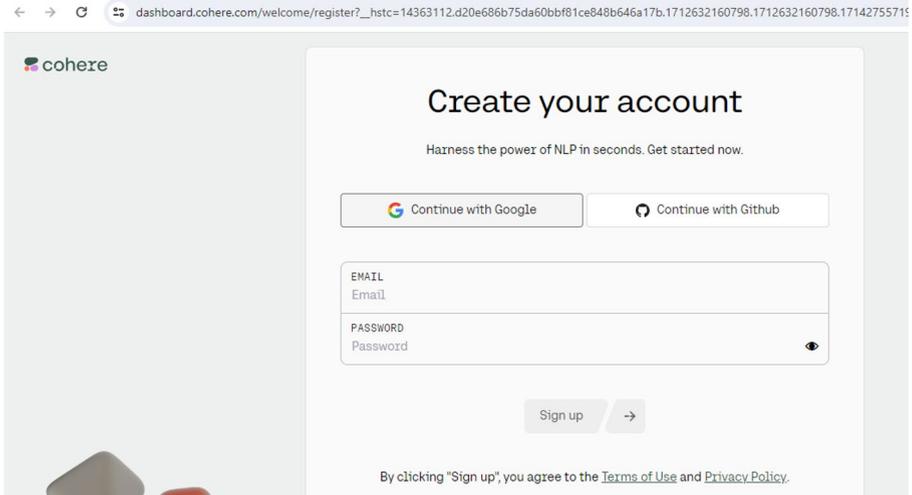
Then you move on to Cohere's representative models and start investigating **Embed-english**, **Embed-multilingual**, **Rerank-english**, and **Rerank-multilingual**. You discover that these models excel at transforming text into numerical representations, or embeddings, which capture the contextual meaning of the input. Embed-english and Embed-multilingual have shown potential in powering semantic search engines, content recommendation systems, and text classification tasks. Rerank-english and Rerank-multilingual are known to improve search relevance through second stage ranking, and you identify it as useful for applications like customer support ticket routing or multilingual knowledge base search.

While Cohere's models are certainly impressive, you realize that it is just one piece of the puzzle, and so you decide to move on with your research after trying out its model in the next section.

Practical Example: Using Cohere's Command Model (Code Snippet)

First, you need to get a Cohere API key, which you can do by going to their website and registering at <https://dashboard.cohere.com/welcome/register>.

Fill up the form asking for your profile detail and then go the “API Keys” page as shown below. Cohere provides two types of keys: one is the production key, and the other is the trial key. Go ahead and copy the trial key.



In this example, you import the Cohere library and create a Client instance with your API key.

You then provide a prompt to the Command model and specify the maximum number of tokens to generate and the temperature (which controls the creativity of the generated text).

The generated text is stored in the generations attribute of the response, which you can access and print.

As you can see, with just a few lines of code, you could unleash the power of Cohere’s generative models and start creating amazing AI-powered applications.

Here is a simple example of how you can use Cohere's Command model to generate text:

```
!pip install langchain==0.2.0
!pip install langchain_community==0.2.0
!pip install cohere==5.5.0

import os
from langchain_core.prompts import PromptTemplate,
ChatPromptTemplate
from langchain_cohere import ChatCohere
# Set up your Cohere API key
os.environ["COHERE_API_KEY"] = "YOUR_COHERE_API_KEY "

prompt_template = """ You are an AI assistant created to
generate creative business pitches. Please continue the
following pitch:
Pitch: Introducing a revolutionary new product that will
transform the way people work and collaborate. Our innovative
solution combines cutting-edge technology with intuitive
design to create a seamless experience for teams of all sizes.
With features like real-time collaboration, intelligent
task automation, and integrated analytics, our product
empowers teams to achieve more, faster. Imagine a world where
communication is effortless, productivity is maximized, and
success is inevitable. That is the world we are building
with """

if __name__ == "__main__":
# Initialize the Cohere model
cohere = ChatCohere(temperature=0, api_key=COHERE_API_KEY,
model_name="command-r")
system = ( prompt_template )
human = "{text}"
```

```
# Create a prompt template
prompt = ChatPromptTemplate.from_messages([("system", system),
("human", human)])
chain = prompt | cohere;
response = chain.invoke(
{
"text": "Give me fun fact about Cohere",
}
)
print("Cohere Generated Pitch:")
print(response)
```

Meta AI Models

Next, your model discovery leads you to the set of Meta AI models. You first start with **LLaMA**, or **Large Language Model Meta AI**, which is a foundational model that comes in various sizes (7B to 65B parameters) and excels at tasks like language understanding, text generation, and question answering.

You also discover **OPT**, the **Open Pretrained Transformer**, which is another series of open source models that excels in language modeling and generation. And then there is **NLLB**, or **No Language Left Behind**, a multilingual model that can translate between over 200 languages.

But you realize that Meta AI's offerings don't stop with language models. They have also developed **RoBERTa**, an optimized version of BERT that delivers improved performance on a range of natural language understanding tasks from text classification to sentiment analysis. **DPR**, or **Dense Passage Retrieval**, is a retrieval-based model that efficiently tackles document retrieval and question answering. And **M2M-100** is a true polyglot capable of translating between any pair of 100 languages.

You realize that the potential use cases for these models are vast and exciting, and you have documented some clear use cases such as automated content creation with LLaMA, global ecommerce platforms powered by M2M-100, and personalized learning experiences enhanced by RoBERTa.

You then move on to the world of audio and efficient text classification with **WaVE** and **FastText**. WaVE is the short form for Waveform-to-Vector and converts audio waveforms into fixed-dimensional vector representations. It opens up opportunities for audio classification, retrieval, and similarity analysis. FastText, on the other hand, is a lightweight library which will be helpful for speedy text classification and representation learning.

You document use cases such as voice-activated systems, music recommendation engines, and real-time sentiment analysis on social media for future reference.

Calling the LLaMA Model Using Hugging Face

Before calling the models in the Hugging Face repository, note that most of these models are private or gated. It means you need to provide your Hugging Face with the access token to authenticate and gain access to the model.

Passing the Access Token to Hugging Face

Here is how you can pass your access token when loading the tokenizer and model:

```
!pip install langchain==0.2.0
!pip install transformers
!pip install torch
!pip install fasttext
```

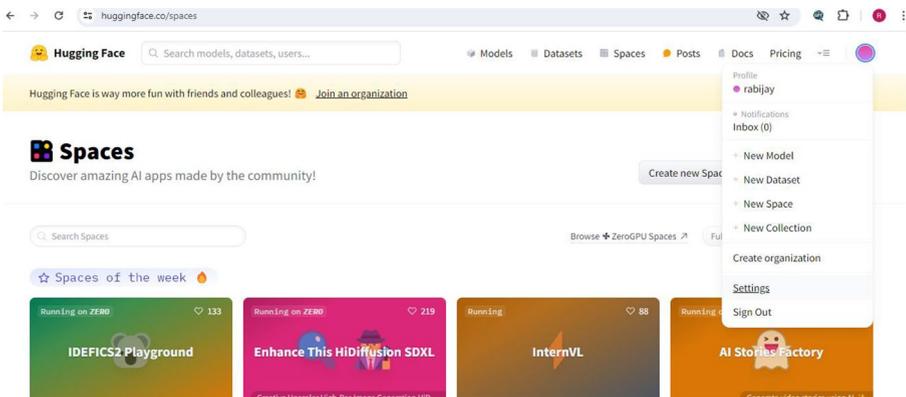
```
from transformers import LlamaTokenizer, LlamaForCausalLM
```

```
model_name = "MetaAI/llama-7b"
access_token = "your_access_token"
```

```
tokenizer = LlamaTokenizer.from_pretrained(model_name, use_auth_token=access_token)
model = LlamaForCausalLM.from_pretrained(model_name, use_auth_token=access_token)
```

Make sure to replace “your_access_token” with your actual Hugging Face access token. You can obtain an access token by following these steps:

1. Sign up or log in to your Hugging Face account at <https://huggingface.co/>.
2. Go to your profile settings by clicking your profile picture in the top-right corner and selecting “Settings.”



Remember to keep your access token secure and avoid sharing it publicly, as it grants access to your Hugging Face account and repositories.

By providing the access token or logging in using the `huggingface-cli`, you should be able to load the private or gated model repository successfully.

Code Explanation

Here is the explanation for the code.

First, you must import the necessary libraries and modules, such as PyTorch, LangChain, and Hugging Face’s Transformers. Next, you need to authenticate with Hugging Face using an access token. Like we discussed earlier, this authentication step is required if you want to access certain models such as LLaMA from the Hugging Face platform. Note that you will also have to sign an agreement to use these models.

Moving on, you load the pretrained model and tokenizer. In this case, you are using the “Meta-Llama-3-8B” model. The tokenizer helps in breaking down the text into tokens that the model can understand. You configure the model to use float16 data type for memory efficiency and automatically utilize available GPUs.

Now, let us talk about the custom stopping criteria. This optional step allows you to define when the model should stop generating text. Here, you create a `StopOnTokens` class that checks if the generated text reaches the end-of-sequence token. This helps in preventing the model from generating an indefinite amount of text.

With the model and stopping criteria ready, you create a Hugging Face pipeline for text generation. You specify parameters like the maximum number of new tokens to generate, the stopping criteria, sampling method, and temperature for controlling the randomness of the output.

To integrate the pipeline seamlessly with LangChain, you wrap it using the `HuggingFacePipeline` class. This allows you to use the pipeline as a language model (LLM) within the LangChain framework.

Finally, you are ready to interact with the model! You provide a prompt asking about the potential benefits and risks of artificial intelligence. The LLM processes the prompt and generates a response, which is then printed.

Note This code may take a long time to complete.

Code to Use LLaMA Model from Hugging Face Platform

Here is the code:

```
!pip install langchain==0.2.0
!pip install transformers==4.40.2
!pip install accelerate==0.30.1
!pip install langchain-community==0.2.0

import torch
from langchain import HuggingFacePipeline
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    pipeline,
    StoppingCriteria,
    StoppingCriteriaList,
    logging
)
from huggingface_hub import login

logging.set_verbosity_error() # Suppress warnings

# 1. Authenticate with Hugging Face (if required)
login(token="Your access token")
```

2. Load the Model and Tokenizer

```

model_id = "meta-llama/Meta-Llama-3-8B"
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    torch_dtype=torch.float16, # Use float16 for memory
                               # efficiency
    device_map="auto",        # Automatically use
                               # available GPU(s)
)

```

3. Custom Stopping Criteria (optional, but recommended)

```

class StopOnTokens(StoppingCriteria):
    def __call__(self, input_ids: torch.LongTensor, scores:
        torch.FloatTensor, **kwargs) -> bool:
        for stop_id in [tokenizer.eos_token_id]:
            if input_ids[0][-1] == stop_id:
                return True
        return False

```

```

stopping_criteria = StoppingCriteriaList([StopOnTokens()])

```

4. Create the Hugging Face Pipeline

```

pipe = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    max_new_tokens=256,
    stopping_criteria=stopping_criteria,
    do_sample=True,
    temperature=0.7,
    top_p=0.95,
)

```

5. Wrap the Pipeline in LangChain

```
llm = HuggingFacePipeline(pipeline=pipe)
```

6. Interact with the Model

```
prompt = "What are the potential benefits and risks of  
artificial intelligence?"  
output = llm(prompt)  
print(output)
```

Large Running Time

When running models from the Hugging Face platform, you are going to face large execution times. I want us to go through some of the reasons for this.

One of the main factors is the size and complexity of the language model being used. Note that in this case, you are working with the “Meta-Llama-3-8B” model, which is a large model with billions of parameters. Obviously, when you are loading and initializing such a massive model, you are going to use significant computational resources and time.

When you load the model using `AutoModelForCausalLM.from_pretrained()`, the model needs to download and process the pretrained weights and parameters. This process can take a while, especially if you have a slower Internet connection or limited bandwidth.

Moreover, in this example, we are loading the model with float16 data type (`torch_dtype=torch.float16`) for memory efficiency reasons. While this helps in reducing the memory footprint, it may slightly increase the computation time compared to using the default float32 data type.

Another factor that can impact the execution time is the hardware resources available. If you are running the code on a machine with limited CPU or GPU capabilities, it may take longer to process the model and generate the output. The code automatically uses available GPUs (`device_map="auto"`), but the performance still depends on the specifications of your hardware.

Additionally, the generated text length can influence the execution time. In this code, you set the `max_new_tokens` parameter to 256, which means the model can generate up to 256 new tokens (roughly equivalent to 256 words) as output. If you wanted to generate longer text, then more computational effort and time will be needed.

Here are some useful tips to improve the execution speed:

1. You can run the code multiple times and save the loaded model and tokenizer to disk and reload them instead of downloading and initializing them each time.
2. You can experiment with different hardware configurations, such as using a more powerful GPU or distributing the workload across multiple GPUs if available.
3. You can adjust the `max_new_tokens` parameter to generate shorter text if the full 256 tokens are not necessary for your specific use case.
4. If the model size is not a strict requirement, you can explore using smaller pretrained models that offer a good balance between performance and execution time.

Note Working with large language models often involves a trade-off between model size, performance, and execution time. You need to find the right balance based on your specific needs and available resources.

PyTorch

To complete your analysis of Meta AI models, you review PyTorch as an open source deep learning framework. While not a model itself, PyTorch provides the foundation for building and training various AI models and neural networks for a wide range of AI applications.

Review Questions

Let us refresh our knowledge gained from this chapter through these review questions.

1. Which of the following is a key feature of GPT-4?
 - A. Image generation
 - B. Code completion
 - C. Text generation and understanding
 - D. Speech recognition
2. What is the primary use of Codex from OpenAI?
 - A. Generating images from text
 - B. Completing and generating code
 - C. Translating languages
 - D. Processing speech to text
3. Which model is known for generating high-quality images from textual descriptions?
 - A. GPT-4
 - B. Codex

- C. DALL-E 2
 - D. PaLM 2
4. Which of the following is a feature of Anthropic's Claude 3 models?
- A. Multimodal input capabilities
 - B. Improved fluency in non-English languages
 - C. Specialization in image generation
 - D. Focus on real-time speech transcription
5. What is a practical use case for Google's Codey Suite?
- A. Image recognition
 - B. Natural language translation
 - C. Real-time code suggestions and completions
 - D. Speech-to-text conversion
6. Which of the following models excels in transforming text into numerical vectors for semantic search?
- A. GPT-4
 - B. Embed-english
 - C. Codex
 - D. DALL-E 2

7. Meta's LLaMA model is primarily used for
 - A. Text generation and language understanding
 - B. Speech recognition
 - C. Image processing
 - D. Multimodal data analysis
8. What does the "temperature" parameter control in text generation models?
 - A. The length of the generated text
 - B. The creativity and randomness of the output
 - C. The accuracy of the generated text
 - D. The speed of text generation
9. Which model from Google is known for its ability to transcribe over 100 languages?
 - A. PaLM 2
 - B. Codey
 - C. Chirp Speech
 - D. Gemini 1.0 Pro
10. Which of the following is NOT a benefit of using LangChain?
 - A. Simplified AI application development
 - B. Modular and scalable architecture
 - C. Direct integration with hardware components
 - D. Enhanced flexibility and scalability

Answers

1. C. Text generation and understanding
2. B. Completing and generating code
3. C. DALL-E 2
4. B. Improved fluency in non-English languages
5. C. Real-time code suggestions and completions
6. B. Embed-english
7. A. Text generation and language understanding
8. B. The creativity and randomness of the output
9. C. Chirp Speech
10. C. Direct integration with hardware components

Key Learnings

This chapter provided a comprehensive exploration of large language models (LLMs) from leading entities like OpenAI, Google, Anthropic, Cohere, and Meta. Each platform offers unique models tailored to specific tasks, from text generation and coding assistance to image creation and language translation. Key takeaways include

1. **Diversity of LLM Applications:** LLMs are versatile and can enhance or automate various tasks across different domains and industries.
2. **Understanding Model Specificities:** It is crucial to understand the strengths, weaknesses, and ideal use cases for each model to choose the right tool for the job.

3. **Getting Started and Integration:** The chapter provided practical insights into integrating LLMs into projects, including acquiring API keys, installing packages, and crafting prompts or inputs.
4. **Evolving Landscape of LLMs:** The field of LLMs is rapidly advancing, requiring continuous learning and adaptation to stay current with new improvements and models.

In conclusion, LLMs offer unprecedented opportunities for innovation across various fields. While the journey from understanding the basics to integrating these models into real-world applications may seem daunting, the insights provided in this chapter equip developers to embark on this exciting path. The key is to start with a clear understanding of the project's needs, choose the right model, and proceed with thoughtful implementation and continuous exploration. The potential of LLMs is vast and largely untapped, waiting for creative minds to harness their power for transformative solutions.

Glossary

In addition to these models discussed so far, you can explore other open source models.

Overview and Capabilities of Open Source Models

Here is an overview of several notable models:

XLNet: Perfect for tasks like text classification and question answering as it excels in understanding context and nuance in text

ALBERT: A lighter version of BERT that reduces memory consumption and increases training speed without sacrificing performance. Ideal for efficient processing in sentiment analysis and document classification

Transformer-XL: Introduces a recurrence mechanism to learn dependencies beyond a fixed length, making it great for tasks involving long documents, such as book summarization or document classification

CTRL: Allows for controllable text generation using control codes to signal the desired content, style, task, or domain. Perfect for generating diverse content like news articles, stories, and code

BART: Combines the benefits of auto-encoding and auto-regressive approaches for denoising text. Shines in text generation, summarization, and translation tasks

GPT-Neo: An open source alternative to GPT-3 that provides similar capabilities for text generation and understanding. Suitable for writing assistance, chatbots, and creative content generation

MegatronLM: A large, powerful transformer model that emphasizes scalability and efficiency in training. Used in research and applications requiring massive-scale language models

ELECTRA: Trains smaller models to distinguish between “real” and “fake” input tokens, improving efficiency and performance on tasks like text classification, entity recognition, and question answering

Longformer: Addresses the limitation of processing long texts by using an attention mechanism that scales linearly with sequence length. Ideal for analyzing large texts, document summarization, and question answering over extensive documents

ConvBERT: Integrates the strengths of convolutional neural networks (CNNs) with BERT architecture, enhancing model efficiency and understanding of local context in text. Effective for tasks requiring nuanced understanding of text structure, such as sentiment analysis and named entity recognition

Definitions of Key Terms and Concepts

Definitions of key terms and concepts introduced in this chapter:

Codex: An AI model developed by OpenAI, designed to understand and generate code in multiple programming languages, aiding in coding tasks and automation

DALL-E 2: Another innovative model from OpenAI, which generates images from textual descriptions

Gemini 1.0 Pro: A powerful language model by Google designed for a variety of tasks, including natural language processing, text generation, chat, and code generation

PaLM 2: A model from Google specialized in understanding and following natural language instructions, making it suitable for chatbots and instructional applications

Codey: A suite of models by Google tailored for code-related tasks such as code completion, generation, and conversational coding assistance

Embeddings: Numerical vector representations of text that capture its contextual meaning, used in tasks like semantic search, text classification, and topic modeling

Imagen: A model by Google for generating high-quality images from textual prompts and for providing relevant descriptions and answers to visual questions

Sec-PaLM2: A model pretrained on security-specific tasks, aiding in threat intelligence, security operations, and malware analysis

Chirp Speech: A universal speech model by Google capable of transcribing speech in over 100 languages

Further Reading

These resources offer a range of perspectives and levels of technical depth, from introductory overviews to detailed research papers:

OpenAI Blog and Research Papers: OpenAI frequently publishes blog posts and research papers. <https://openai.com/research>

Google AI Blog: For insights into Google's AI models, including BERT, PaLM, and others, the Google AI Blog is an excellent resource. <https://blog.google/technology/ai/>

Anthropic Research and Blog Posts: Anthropic, the creator of Claude, shares insights and developments on their models through their research publications and blog. <https://www.anthropic.com/research>

Cohere's Documentation and Blog: Cohere offers extensive documentation and insightful blog posts that explain how to use their models. <https://txt.cohere.com/tag/research/>

ArXiv.org: For those interested in the latest research papers in the field of AI and LLMs, ArXiv.org is an invaluable resource. <https://ArXiv.org>

CHAPTER 5

Mastering Prompts for Creative Content

In this chapter, I will be showing you powerful tools and techniques to use prompt engineering within the LangChain framework. You will learn how to design and optimize prompts that effectively communicate with LLMs to achieve precise and reliable outputs tailored to your specific application needs.

Importance of Prompt Engineering

Prompt engineering is becoming a game changer as it helps you unlock the full potential of LLM models. As a prompt engineer, your job is to carefully create instructions that guide the LLM model to produce the outputs you want. This is a dynamic back-and-forth process between you and the LLM. You set the direction with clear, concise prompts, and the LLM model responds by using its extensive knowledge to generate content that matches your exact vision.

Why Prompt Engineering?

Here are some reasons why prompt engineering is crucial:

- **Access to Knowledge:** These LLM models have been trained on vast datasets with a lot of information in them, but to extract the knowledge that meets your exact needs, you need precise prompts.
- **Career Opportunities:** Developers skilled in prompt engineering are highly sought after. Companies are actively looking for individuals who can use the power of language models to create innovative solutions.
- **Cost Efficiency:** With prompt engineering, you don't need extensive infrastructure or resources. You can innovate using existing models without the need to develop them from scratch, thus saving both time and money.
- **Handling Multiple Models:** In the past, you would have used different models for different tasks. If you wanted to summarize text, you would use one model, and if you needed to do a Q&A task, you would use another. But with the rise of large language models (LLMs), a single model can handle various tasks. The catch? You need to tell the model how to behave for each specific task.
- **Handling Different Formats:** You can leverage prompt engineering to exercise precise control over LLM outputs across various formats. You can guide models to generate responses in desired structures such as JSON, Python functions, or structured lists.

Need for Scalability

First, let us make a call using LLMs without prompt templates. Imagine you want to get a fact about a specific history topic, like the World War. You would do something like this:

```
result = LLM("What were the major causes of tech stock rally
in 2021")
print(result)
```

And you would get a fascinating fact about tech stock rally, like “The tech stock rally in 2021 was driven by a combination of low interest rates, economic recovery, strong earnings growth, increased digitization, investor enthusiasm, government stimulus, and technological advancements.”

But what if you want to switch things up and get facts about the retail sector growth in 2021? You could use an f-string literal and define the World War variable outside, like this:

```
topic = "retail sector growth in 2021"

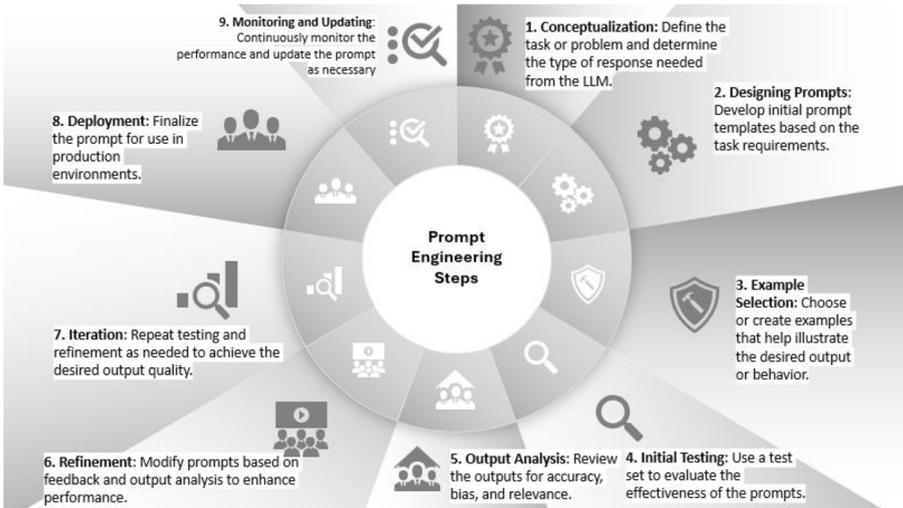
prompt = f"""
Provide a concise summary of the major factors influencing
{topic}.
Include at least three key points in your response.
Format your answer as a bulleted list.
"""

result = LLM(prompt)
print(result)
```

While this works, it doesn’t scale well when you start building more complex applications or chains. That is where prompt templates come to the rescue!

Prompt Engineering Steps

Below is a flowchart that shows one way you can implement prompt engineering in your projects with various steps starting from conceptualization to execution.



What Is the Goal?

First, you should have a clear idea of what you want to achieve. Ask yourself, “What do I need the AI to accomplish?” Whether you are generating jokes, summarizing complex articles, or anything in between, having a well-defined goal is the starting point.

Things to Consider When Creating Prompts

First of all, apart from providing information to the LLM, you should also focus on potential issues that may come up and guide the LLM in the right direction accordingly. For example, when

generating a fictional story, you might include instructions to avoid overused plots and to maintain a consistent tone and style throughout.

Your prompts can make or break your language model's performance. Imagine a prompt template as a piece of text with blank spaces waiting to be filled with the user's input. Another way is to think of them as a means to convert strings into function parameters that you can pass in values for.

Don't be discouraged if your first few attempts don't yield the exact results you are looking for. It takes practice and patience to master it like any other skill. With each iteration, you will learn valuable lessons and refine your approach.

You should also watch out for biases and try to mitigate them. Language models can inherit biases from the data they were trained on. One way to reduce bias is to carefully craft prompts and provide diverse and inclusive examples.

Composing Your Prompts

Next, you need to sketch an outline. You will have to first start with an initial draft prompt. Keep it simple and straightforward. For instance, if you need a summary, your prompt might start with, "Summarize the following article for a high school student."

When crafting prompts, you must consider both system and user prompts:

System Prompt: You should start by defining the AI's role and overall expected behavior using a system prompt. This sets the context for all subsequent interactions. For example:

System: You are an expert educator skilled at explaining complex topics to high school students.

User Prompt: You should then follow it with a user prompt that specifies the actual task. Keep it simple and straightforward. For instance:

User: Summarize the following article for a high school student: [Article text]

By combining system and user prompts, you create a more controlled and context-aware interaction:

```
messages = [ {"role": "system", "content": "You are an expert educator skilled at explaining complex topics to high school students."}, {"role": "user", "content": "Summarize the following article for a high school student: [Article text]"}] result = LLM(messages) print(result)
```

This approach helps to provide a clear structure for the LLM to follow.

Selecting Your Examples

Here is where you help the AI understand the style, tone, and format you are looking for with examples. It is essential to provide high-quality examples because the AI will learn from the examples you provide, so choose them wisely. If you are after brevity, choose examples that are short and sweet.

Testing Your Prompts

With your prompts and examples ready, it is time to test them. Feed your prompts to the AI and analyze how well it responds. Pay close attention to the quality, relevance, and coherence of the generated outputs. Are they hitting the mark, or do they need some fine-tuning?

Reviewing the Outputs

Assess not only the accuracy of the LLM's responses but also their overall quality. Consider factors like fairness, coherence, and adherence to the provided examples. Ask yourself, "Would I be proud to showcase these results, and do they resemble the examples I provided?" This evaluation will help you identify areas for improvement.

Fine-Tuning Your Prompts

Based on the insights gained from your evaluation, you need to fine-tune your prompts. If the LLM's outputs lack the desired tone or clarity, adjust accordingly. Perhaps you need to provide more specific instructions or tweak the examples to better align with your goals.

Optimizing Your Approach: Prompt Engineering vs. Model Fine-Tuning

It is important to make the distinction between fine-tuning a prompt, a.k.a. prompt engineering, and model fine-tuning. You should first start by optimizing your prompts, which involves refining your instructions, adjusting examples, and experimenting with different phrases. It is recommended because it is a quick approach that doesn't require changing the model. For example, instead of using this prompt - "Summarize this article" - you could use a more refined prompt such as "Provide a 3-bullet summary of the key points in this article that can be suitable for a high school student."

Few-Shot Learning: If simple prompts are not enough, you could try few-shot learning by including two to three examples in your prompt. This is more helpful because it guides the model more explicitly.

Model Fine-Tuning: However, if both prompt engineering and few-shot learning are not helpful, you should consider model fine-tuning. This involves retraining the model on a specific dataset to adapt its capabilities for your task. It is useful when you have a large and high-quality dataset. And if you are looking for consistent performance on a specific type of task where prompt engineering can't help, then model fine-tuning can be your solution.

Embracing Iteration

Don't expect perfection on the first try. Instead, be prepared to iterate, test, and refine your prompts multiple times.

Deploying Your Prompts

Once you are satisfied with the LLM's responses and confident in the quality of your prompts, you should deploy them into the real world.

Monitoring and Maintaining

It is crucial to keep a watchful eye on their performance over time. If you notice any drifts or inconsistencies, be proactive in updating and refining your prompts.

Components of a Prompt

The prompts module consists of three main components:

- Prompt templates
- Example selectors
- Output parsers

Let us take a closer look at each of these components and see how they can help you create awesome prompts.

Prompt Templates

You can leverage two types of prompt templates when working with language models:

1. **Regular Prompt Templates:** These are used for straightforward text generation use cases, such as answering questions, completing sentences, or any other text generation task.
2. **Chat Prompt Templates:** These are specifically designed for a more interactive and conversational experience and have components like system messages and user messages to guide the interaction.

You should note that most of the latest models, including those with chat completion enabled by default, support both types of templates. This means you can use the same model for general tasks like answering questions as well as for creating conversational interactions. You can therefore choose the appropriate template based on your specific needs without switching models.

Let us discuss what goes into a prompt template:

1. **Instructions:** You will use instructions to provide clear directions to the LLM and to communicate your expectations to the LLM.
2. **Few-Shot Examples:** Including a few examples of desired outputs gives the LLM a concrete reference point. Through these examples, you can communicate the style, tone, and format you are looking for, enabling the model to grasp the nuances of your requirements.

3. **Context and Questions:** You can help the LLM understand the setting and focus of the conversation by providing relevant context and specific questions. It helps the LLM model to provide responses that are very specific to your task.

Creating a Multi-string Prompt Template

Let us see how you can create and use a prompt template that works with any type of LLM. First, you need to import the `PromptTemplate` class from the `langchain.prompts` library. Hopefully, you have already imported the `LangChain` library:

```
from langchain.prompts import PromptTemplate
from langchain.chat_models import ChatOpenAI
```

Create a multiline string called `template`. It is similar to an f-string, but instead of using `{variable}`, you use `{variable}` (without the “f” at the beginning).

```
template = """
You are a seasoned software engineer.
Explain the following algorithm: {algorithm} in {language}.
Describe its purpose, time complexity, and a common use case.
"""
```

In this example, you replace `{algorithm}` with “machine learning” and `{language}` with “French.” The resulting prompt will look like this:

```
You are a seasoned software engineer.
Explain the following algorithm: machine learning in French.
Describe its purpose, time complexity, and a common use case.
```

Calling the LLM Using the Prompt Template

Time to create the LLM object. Here is an example when using LangChain with OpenAI. You should note that the process of initializing and calling an LLM can vary depending on the specific model, API, or framework used:

```
llm = OpenAI(openai_api_key="Your OpenAI key", temperature=0.7)
```

Finally, you invoke the LLM with the prompt string as an argument and print the response:

```
# Create a very simple LLM chain (note: some output control
is lost)
chain = LLMChain(llm=llm, prompt=prompt_template)
```

In the above code, you are creating an instance of the LLMChain class, which represents a simple chain in a language model. The LLMChain constructor takes two arguments, namely, the llm and the prompt, to which you have assigned the prompt template that you created previously.

Note Remember to import the necessary libraries for this to work. For example, you would need to import the following:

```
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
```

You then generate a response by calling the chain with the run method, which takes in two keyword arguments through which you pass in the two keywords, namely, the type of algorithm and the language that you used to build the prompt template earlier:

```
# Run the chain to get the response
response = chain.run(algorithm="machine learning",
language="french")
print(response)
```

You will get a short response about machine learning in French.

See how easy it is to use prompt templates? They allow you to create dynamic prompts effortlessly, making your code more readable and maintainable.

Advantages of Using Prompt Templates

Before moving on to the next section on example selectors, let us review some of the advantages of using prompt templates in the previous example:

- **Consistency:** You ensure that the input to the language model follows a consistent structure and format. It also allows the language model to understand and process the input in a more predictable and structured manner.
- **Reusability:** The prompt template is reusable because it is separate from the code that calls the model. Therefore, the same template can be used across multiple models as well as other places in the code.
- **Customization:** You can customize the input to the language model based by using placeholders or variables within the prompt template. You can dynamically fill with relevant information during runtime.

- **Improved Fine-Tuning:** You can guide the language model toward generating responses that align with your desired format, style, or content. In fact, the prompt template acts as a form of fine-tuning, allowing you to control the language model's output based on the structure and content of the template.
- **Maintainability:** Separating the prompt template from the code improves the readability and maintainability of your codebase.

Example Selectors

Sometimes, you need to give LLMs a little guidance to get the specific output you are looking for. That is where example selectors come in!

You can use example selectors to provide examples to your LLM, helping it understand the desired output format and patterns. By showing your model relevant examples, you can guide it toward generating more accurate and appropriate responses. This technique is called **few-shot learning**. Example selectors help you to choose which ones will best instruct the language model for your current task.

Choosing the Right Examples for Your Prompts

An example selector takes the input variables and selects a subset of examples that are most suitable for the given input.

Imagine you are building a system that generates personalized workout plans based on user preferences. Assume you have a dataset of example workout plans for different user preferences such as fitness levels and goals. When a user provides their preferences, you want to select the most relevant examples to include in the prompt.

Here is a simplified implementation of an example selector.

Defining the BaseExampleSelector Class

First, you must define the example selector interface:

```
class BaseExampleSelector(ABC):
    """Interface for selecting examples to include in
    prompts."""

    @abstractmethod
    def select_examples(self, input_variables: Dict[str, str])
    -> List[dict]:
        """Select which examples to use based on the inputs."""

    @abstractmethod
    def add_example(self, example: Dict[str, str]) -> Any:
        """Add new example to store."""
```

This is an abstract base class that defines the interface for our example selector.

You need to implement the abstract method **select_examples** in a class inheriting from **BaseExampleSelector**. It takes a dictionary of input variables and returns a list of selected examples.

You can add new examples to the example selector’s store by implementing the **add_example** method.

Note that you need to import `ABC` and `abstractmethod` from the `abc` module. You should also import necessary types from the `typing` module for type hinting.

Creating a List of Examples

Below is a list of example inputs and outputs for translating English words to Italian. Each example is represented as a dictionary with “input” and “output” keys:

```
examples = [
    {"input": "hi", "output": "ciao"},
    {"input": "bye", "output": "arrivaderci"},
    {"input": "soccer", "output": "calcio"},
]
```

Implementing a Custom Example Selector

Next, you develop a custom implementation of the `BaseExampleSelector` class.

Use the `__init__` method to initialize the example selector with a list of examples.

Using the `add_example` method, you can add a new example to the example selector's list of examples.

The `select_examples` method selects the example that is closest in length to the input word. It iterates through each example, calculates the length difference between the input word and the example's input, and keeps track of the example with the smallest length difference.

```
class CustomExampleSelector(BaseExampleSelector):
    def __init__(self, examples):
        self.examples = examples

    def add_example(self, example):
        self.examples.append(example)

    def select_examples(self, input_variables):
        new_word = input_variables["input"]
        new_word_length = len(new_word)

        best_match = None
        smallest_diff = float("inf")
```

```

for example in self.examples:
    current_diff = abs(len(example["input"]) - new_
word_length)

    if current_diff < smallest_diff:
        smallest_diff = current_diff
        best_match = example

return [best_match]

```

Using the Custom Example Selector

Here, you create an instance of the CustomExampleSelector with the list of examples:

```
example_selector = CustomExampleSelector(examples)
```

Then, you call the select_examples method with the input word “okay” to get the closest matching example:

```
example_selector.select_examples({"input": "okay"})
```

The output is

```
[{'input': 'bye', 'output': 'arrivaderci'}]
```

Then, you add a new example using the add_example method:

```
example_selector.add_example({"input": "hand", "output": "mano"})
```

You call select_examples again with the same input word to see the updated result:

```
example_selector.select_examples({"input": "okay"})
```

Now, the revised output is

```
[{'input': 'hand', 'output': 'mano'}]
```

Using the Example Selector in a Prompt

Below is the code to use the example selector in a prompt. Here, you import the necessary classes from LangChain to create a prompt template:

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

You define an `example_prompt` using the `PromptTemplate`. `from_template` method, which specifies the format of each example in the prompt:

```
example_prompt = PromptTemplate.from_template("Input: {input}
-> Output: {output}")
```

You create a `FewShotPromptTemplate` using the custom example selector, the example prompt, a suffix, a prefix, and the input variables:

```
prompt = FewShotPromptTemplate(
    example_selector=example_selector,
    example_prompt=example_prompt,
    suffix="Input: {input} -> Output:",
    prefix="Translate the following words from English to
    Italian:",
    input_variables=["input"],
)
```

Finally, you use the `format` method of the prompt template to generate a prompt with the input word “word”:

```
print(prompt.format(input="word"))
```

Now, you have seen how to create a custom example selector that selects examples based on their similarity in length to the input word. The example selector is then used in a prompt template to generate prompts for translating English words to Italian. You should be able to use the same approach to create custom example selectors for similar user cases.

Choosing the Right Example Selector

There are different types of example selectors you can choose from:

1. **Similarity Example Selector**

The Similarity example selector uses semantic similarity between the input and the available examples and chooses the one with the highest similarity scores in the prompt. This is useful when you want to provide the language model with examples that are semantically close to the input to generate more relevant outputs.

2. **MMR (Maximal Marginal Relevance) Example Selector**

The MMR example selector aims to balance the relevance and diversity of the selected examples. It uses the concept of Maximal Marginal Relevance, which considers both the similarity between the input and the examples and the dissimilarity among the selected examples. This approach ensures that the selected examples are not only relevant to the input but also diverse, reducing redundancy and increasing the coverage of different aspects related to the input.

3. **Length Example Selector**

The Length example selector focuses on selecting examples based on how many can fit within a specified length constraint. This example selector is handy when you have a limited context window or want to control the size of the prompt. This helps in providing the language model with a diverse set of examples while staying within the length constraints.

4. **Ngram Example Selector**

The Ngram example selector uses ngram overlap between the input and the examples to determine which examples to choose. An ngram is a contiguous sequence of n items (words or characters) from a given text. By considering ngram overlap, this selector favors examples that share common phrases or word sequences with the input, potentially improving the model's ability to generate relevant outputs.

Factors to Choose When Selecting an Example Selector

Consider the following factors when selecting an example selector:

- **Relevance:** If generating highly relevant outputs is crucial, the Similarity or MMR example selectors might be suitable choices.
- **Diversity:** If you want to cover diverse aspects related to the input, the MMR example selector can help balance relevance and diversity.

- **Length Constraints:** If you have strict length limitations for your prompts, the Length example selector can assist in selecting examples that fit within those constraints.
- **Phrase Matching:** If you want to emphasize examples that share common phrases or word sequences with the input, the Ngram example selector can be effective.

Few-Shot Prompt Template

The few-shot prompt template is a technique where a model learns to perform a task from a very limited amount of training data. It involves providing a few examples (shots) to help the model understand the task context and desired output format. Figure 5-1 shows how few-shot learning works.

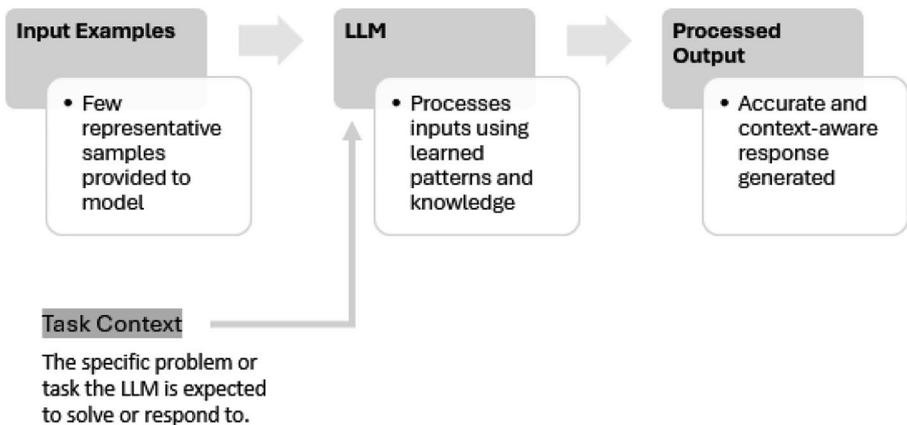


Figure 5-1. How Few-Shot Learning Works

LangChain provides a handy `FewShotPromptTemplate` class that makes working with examples even easier.

Crafting a Few-Shot Prompt Template for Question Answering

In this section, you will learn how to create a few-shot prompt template that teaches a language model to generate self-asking questions and search for answers based on a set of examples.

Step 1: Prepare Example Set

To begin, create a list of few-shot examples. Each example should be a dictionary containing a question and its corresponding answer:

```
examples = [  
    {  
        "question": "What is the largest planet in our solar  
system?",  
        "answer": "Jupiter is the largest planet in our solar  
system."  
    },  
    {  
        "question": "Who painted the Mona Lisa?",  
        "answer": "The Mona Lisa was painted by Leonardo  
da Vinci."  
    },  
    {  
        "question": "What is the currency of Japan?",  
        "answer": "The currency of Japan is the Japanese yen."  
    }  
]
```

Step 2: Format Few-Shot Examples

Next, use a `PromptTemplate` to format the examples into a string that will be presented to the language model. It takes the question and answer as input variables and formats them into a string:

```
from langchain.prompts import PromptTemplate
example_prompt = PromptTemplate(
    input_variables=["question", "answer"],
    template="Question: {question}\nAnswer: {answer}"
)
# Test the formatting by printing the first example
print(example_prompt.format(**examples[0]))
```

Output:

```
Question: What is the largest planet in our solar system?
Answer: Jupiter is the largest planet in our solar system.
```

Step 3: Create Few-Shot Prompt Template

Now, you will create a `FewShotPromptTemplate` that will serve as the framework for the language model to learn from the provided examples. This code creates a `FewShotPromptTemplate` using the examples and the formatted example prompt. It also specifies a suffix that will be appended to the examples and the input variables:

```
from langchain.prompts import FewShotPromptTemplate

prompt = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    suffix="Question: {input}",
    input_variables=["input"]
)
```

```
# Preview the formatted template
print(prompt.format(input="What is the capital of Australia?"))
```

Output:

```
Question: What is the largest planet in our solar system?
Answer: Jupiter is the largest planet in our solar system.
```

```
Question: Who painted the Mona Lisa?
Answer: The Mona Lisa was painted by Leonardo da Vinci.
```

```
Question: What is the currency of Japan?
Answer: The currency of Japan is the Japanese yen.
```

```
Question: What is the capital of Australia?
```

Step 4 (Optional): Select Examples with Example Selector

If you have a large set of examples, you can use an example selector. This code initializes a `LengthBasedExampleSelector` that selects examples based on their length. It takes the examples, the example prompt, and the maximum length as parameters. You can choose a selector of your choice.

```
# Initialize the selector with your examples
example_selector = LengthBasedExampleSelector(
    examples=examples,
    example_prompt=example_prompt,
    max_length=50
)

# Find the most relevant example for a new question
selected_examples = example_selector.select_examples(
    {"question": "Who sculpted the Statue of David?"}
)
```

Remember to import the following:

```
from langchain.prompts import FewShotPromptTemplate,
PromptTemplate
from langchain.prompts.example_selector import
LengthBasedExampleSelector
```

The above line selects the most relevant examples based on the question “Who sculpted the Statue of David?” using the `LengthBasedExampleSelector`. The selected examples are then used to create a new `FewShotPromptTemplate` that incorporates the example selector below.

Let us discuss how the `LengthBasedExampleSelector` works.

When initializing the `LengthBasedExampleSelector`, you provide the list of examples (`examples`), the example prompt template (`example_prompt`), and the maximum length (`max_length`) for the selected examples.

The `select_examples` method is called with an input dictionary containing the question “Who sculpted the Statue of David?” However, in the case of the `LengthBasedExampleSelector`, the input question is not used for selecting examples based on relevance.

The `LengthBasedExampleSelector` iterates through the list of examples and selects examples based on their length. It tries to include as many examples as possible while ensuring that the total length of the selected examples does not exceed the specified `max_length`. Its goal is to maximize the number of examples to include within the specified length limit.

Step 5 (Optional): Integrate Example Selector into Prompt Template

By integrating the example selector into the prompt template, the selected examples are dynamically included in the prompt based on the relevance to the input question.

This approach allows for a more targeted and efficient use of examples, improving the quality and relevance of the generated responses.

```
prompt = FewShotPromptTemplate(
    example_selector=example_selector,
    example_prompt=example_prompt,
    suffix="Question: {input}",
    input_variables=["input"]
)

# Load an OpenAI language model (you'll need your API key)
llm = OpenAI(
    model_name="gpt-3.5-turbo",
    openai_api_key="your openai key",
    temperature=0.7,
)
```

Testing the Prompt Templates

Below, you test the prompt templates with and without the example selector. You format the prompts based on the given input and generate answers using the loaded language model:

```
print("Prompt Template without Example Selector:")
print(prompt.format(input="What is the capital of Australia?"))

print("\nGenerated Answer:")
print(llm(prompt.format(input="What is the capital of
Australia?")))
```

```
print("\nPrompt Template with Example Selector:")
print(prompt_with_selector.format(input="Who sculpted the
Statue of David?"))

print("\nGenerated Answer:")
print(llm(prompt_with_selector.format(input="Who sculpted the
Statue of David?")))
```

Below is the answer I got:

Prompt Template without Example Selector:

Question: What is the largest planet in our solar system?

Answer: Jupiter is the largest planet in our solar system.

Question: Who painted the Mona Lisa?

Answer: The Mona Lisa was painted by Leonardo da Vinci.

Question: What is the currency of Japan?

Answer: The currency of Japan is the Japanese yen.

Question: What is the capital of Australia?

Answer:

Generated Answer:

The capital of Australia is Canberra.

Prompt Template with Example Selector:

Question: What is the largest planet in our solar system?

Answer: Jupiter is the largest planet in our solar system.

Question: Who painted the Mona Lisa?

Answer: The Mona Lisa was painted by Leonardo da Vinci.

Question: Who sculpted the Statue of David?

Answer:

Generated Answer:

The Statue of David was sculpted by Michelangelo.

Reviewing the Output

The above answers clearly demonstrate the functionality of the few-shot prompt templates with and without the example selector. Let us break down each part:

1. Prompt Template Without Example Selector

Question: What is the largest planet in our solar system? Answer: Jupiter is the largest planet in our solar system.

Question: Who painted the Mona Lisa? Answer: The Mona Lisa was painted by Leonardo da Vinci.

Question: What is the currency of Japan? Answer: The currency of Japan is the Japanese yen.

Question: What is the capital of Australia? Answer:

In this case, the prompt template without the example selector includes all the examples provided in the examples list. The examples are formatted according to the `example_prompt` template, which includes the question and answer for each example. After the examples, the prompt template appends the input question “What is the capital of Australia?” using the `suffix` parameter. The generated answer is then provided by the language model.

2. Generated Answer (Without Example Selector)

The capital of Australia is Canberra.

This is the answer generated by the language model for the question “What is the capital of Australia?” based on the prompt template without the example selector. The model uses the provided examples and its preexisting knowledge to generate the correct answer.

3. Prompt Template with Example Selector

Question: What is the largest planet in our solar system? Answer: Jupiter is the largest planet in our solar system.

Question: Who painted the Mona Lisa? Answer: The Mona Lisa was painted by Leonardo da Vinci.

Question: Who sculpted the Statue of David? Answer:

In this case, the prompt template with the example selector (`LengthBasedExampleSelector`) selects examples based on their length. The selected examples are formatted according to the `example_` prompt template. The input question “Who sculpted the Statue of David?” is appended to the selected examples using the `suffix` parameter. The generated answer is then provided by the language model.

4. Generated Answer (with Example Selector)

The Statue of David was sculpted by Michelangelo.

This is the answer generated by the language model for the question “Who sculpted the Statue of David?” based on the prompt template with the example selector. The model uses the selected examples and its preexisting knowledge to generate the correct answer.

The main difference between the two prompt templates is that the one without the example selector includes all the examples, while the one with the example selector (`LengthBasedExampleSelector`) selects examples based on their length.

In both cases, the language model uses the provided examples and its preexisting knowledge to generate appropriate answers to the input questions. Thus, the examples serve as a guide or context for the model to understand the type of information being asked and the expected format of the answer.

Conclusion

You have now created a few-shot prompt template that selects relevant examples and presents them in a structured format to the language model. You have enabled the model to learn from the examples and generate appropriate answers to new questions.

Output Parsers

Finally, we have output parsers. Sometimes, you want your LLM to generate output in a specific format, like JSON or a question-answer pattern. You can use output parsers to extract the relevant information from the model's response and structure it according to your needs.

Output parsers are helpful when building an application that needs to extract specific data types from the output of an LLM. Maybe you want a Python datetime object or a nicely formatted JSON object. You can use the output parsers to effortlessly convert the string outputs from LLMs into the exact data types you need or even your own custom class instances using `pydantic`.

Output parsers typically perform two primary functions:

1. **Get Format Instructions:** This method provides guidelines to the LLM on how the information should be structured and presented.
2. **Parse:** Once the LLM generates a response, this method takes that text and structures it according to the provided instructions.

In cases where the LLM's response doesn't perfectly match the desired format, output parsers offer an additional method:

1. **Parse with Prompt:** This method serves as a second attempt to structure the data correctly. It takes into account the LLM's response and the original prompt and provides more context to refine the output.

Types of Output Parsers

LangChain has a diverse range of output parsers. Whether you need JSON objects or you are working with CSV files, LangChain likely has a parser for you. And a big plus is that many of these parsers support streaming, meaning they can handle continuous data feeds:

- **OpenAITools Parser:** This is handy when dealing with the latest OpenAI functions. It structures the output based on the given arguments, tools, and tool choice.
- **OpenAIFunctions Parser:** This one is useful when using legacy OpenAI function calling arguments like functions and function_call. It is reliable and streams the output as JSON objects.
- **JSON Parser:** It is one of the most reliable parsers out there, and it returns a JSON object, which can be defined by a Pydantic model if you like your data dressed in a particular schema.
- **XML Parser:** Use this parser when your output needs to be in XML format.
- **CSV Parser:** This parser turns outputs into a neat list, perfect for spreadsheets.
- **OutputFixing Parser:** Sometimes, the first draft isn't perfect. This parser wraps another parser and steps in if there is an error, asking an LLM to fix the output.
- **RetryWithError Parser:** This is similar to OutputFixing but more comprehensive, as it also considers the original inputs and instructions and asks the LLM for a redo if there is an error.

- **Pydantic Parser:** Pydantic helps you define the model output structure and ensure the output fits right into the structure you want.
- **YAML Parser:** For outputs that need to be in YAML format, particularly useful when your data schema is defined in YAML.

And that is just a glimpse! There are parsers for DataFrames, Enums, datetimes, and even basic structured dictionaries for simpler tasks. The key takeaway is that LangChain's output parsers are all about giving you control over how you receive and use the data generated by LLMs.

Example Use Case: Suppose you are building an application to determine the dates of scientific discoveries, and you want these dates to be datetime objects in Python. You might face two main challenges:

1. The LLM's output is always a string, regardless of how you interact with it or what instructions you provide.
2. The date format can vary. The LLM might respond with "January 1st, 2020," "Jan 1st, 2020," or "2020-01-01."

This is where parsers can help. They use format instructions to ensure the output is in the correct format (e.g., "2020-01-01" for datetime) and provide a parsing method to convert the string into the desired Python object type.

Practical Example: Using PydanticOutputParser for Movie Data

In this example, you will create a system to store and organize information about movies.

Importing Required Libraries

You import the following libraries:

- `pydantic`: Provides the `BaseModel`, `Field`, `ValidationError`, and `field_validator` classes for defining the data model and validation
- `langchain.output_parsers`: Provides the `PydanticOutputParser` class for parsing the language model's output
- `langchain.prompts`: Provides the `PromptTemplate` class for creating the prompt template
- `langchain.chat_models`: Provides the `ChatOpenAI` class for interacting with the OpenAI chat model
- `langchain.schema`: Provides the `HumanMessage` class for representing human messages in conversation

```
from pydantic import BaseModel, Field, ValidationError, field_validator
from langchain.output_parsers import PydanticOutputParser
from langchain.prompts import PromptTemplate
from langchain.chat_models import ChatOpenAI
from langchain.schema import HumanMessage
```

Defining the Movie Data Model

Here, you define the movie data model:

- You define a `Movie` class that inherits from `BaseModel` to represent the structure of a movie.
- The class has three fields: `title`, `director`, and `year`, each with a description.

- You use the `@field_validator` decorator to ensure that the movie title is capitalized. If the title is not in title case, a `ValidationError` is raised.

```
class Movie(BaseModel):
    title: str = Field(description="The title of the movie")
    director: str = Field(description="The director of the movie")
    year: int = Field(description="The release year of the movie")

    # Quality control: Ensure the movie title is capitalized
    @field_validator('title')
    def title_must_be_capitalized(cls, value):
        if not value.istitle():
            raise ValueError("Movie title must be
                               capitalized.")
        return value
```

Initializing the Output Parser

You create an instance of `PydanticOutputParser` called `parser` and pass the `Movie` class as the `pydantic_object` argument to specify the desired output structure:

```
# Initialize the parser with our data structure
parser = PydanticOutputParser(pydantic_object=Movie)
```

Creating a Prompt Template

You define a `PromptTemplate` called `prompt` that provides instructions to the language model on how to format the movie information:

- The template includes placeholders for the format instructions (`{{format_instructions}}`) and the query (`{{query}}`).

- You specify the input variable “query” and provide the format instructions obtained from the parser using `parser.get_format_instructions()`.

```
# Create a prompt template with instructions for the LLM
prompt = PromptTemplate(
    template="Please provide information about a movie in
    the following format:\n{format_instructions}\nQuestion:
    {query}",
    input_variables=["query"],
    partial_variables={"format_instructions": parser.get_
    format_instructions()},
)
```

Setting Up the OpenAI API Key

You retrieve the OpenAI API key from the environment variable using `os.environ.get()`. If the environment variable is not set, you use a default value (replace “your_api_key_here” with your actual API key):

```
# Set up the OpenAI API key
openai_api_key = os.environ.get("OPENAI_API_KEY")
if openai_api_key is None:
    openai_api_key = "your_api_key_here" # Replace with your
                                        actual API key
```

Choosing the Language Model and Its Settings

You create an instance of `ChatOpenAI` called `llm` and specify the model name (“gpt-3.5-turbo”), temperature (0), and the OpenAI API key:

```
# Choose the LLM and its settings
llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0,
openai_api_key=openai_api_key)
```

Generating Movie Information

- You define a query variable with the movie title you want to get information about (“Inception”).
- You create a `HumanMessage` instance called `human_message` with the content of the formatted prompt using `prompt.format(query=query)`.
- You pass a list containing `human_message` to the `llm` instance to generate a response.

```
# Generate movie information using the prompt
query = "Tell me about the movie 'Inception'."
human_message = HumanMessage(content=prompt.
format(query=query))
response = llm([human_message])
```

Parsing the Language Model’s Response

You use a `try-except` block to handle any validation errors that may occur during parsing.

- You call `parser.parse(response.content)` to parse the content of the language model’s response using the initialized `PydanticOutputParser`.
- If the parsing is successful, you print the structured movie data stored in `parsed_movie`.
- If a `ValidationError` occurs during parsing, you print the error message.

```
# Parse the LLM's response
try:
    parsed_movie = parser.parse(response.content)
    # Print the structured movie data
    print(parsed_movie)
except ValidationError as e:
    print(f"Validation Error: {e}")
```

By using the `PydanticOutputParser`, you have effectively transformed the LLM's text response into a well-structured movie object.

This example demonstrates how output parsers can be applied to various domains, such as organizing information about movies, books, products, or any other structured data you need to extract from LLM responses.

OutputFixingParser

`LangChain` has a handy tool called the `OutputFixingParser`. This parser takes the original output, sends it back to the model, and asks it to fix any formatting issues. It is like having a helpful tool that double-checks the model's work and makes sure it is just right.

I want to point out that as models continue to improve, they are getting better at following instructions. Plus, there is a bit of randomness involved in working with LLMs. So, if you are coding along with this, you might find that the model generates the correct output right off the bat, and everything works smoothly. That is great news! But don't worry if you can't reproduce the error – you can still learn from this.

In the next section, you'll explore the other type of prompt called "chat prompt template," which is perfect for building chatbots and conversational agents.

ChatPrompt Templates

Chat prompt templates are a useful variant of the prompt templates that can make your life easy when working with conversational tasks using large language models (LLMs).

As you learned before, the Chat Completions API uses a list of messages, including system messages, human messages (prompts), and AI messages. LangChain provides some useful classes that make working with these messages easy.

Building the Chat Prompt Template

Let us see how to create such a template using the `ChatPromptTemplate` class.

Importing Required Libraries

First, take care of the basics such as importing the libraries shown below:

- `langchain.prompts`: Provides the `ChatPromptTemplate`, `SystemMessagePromptTemplate`, and `HumanMessagePromptTemplate` classes for creating chat prompts
- `langchain.chat_models`: Provides the `ChatOpenAI` class for interacting with the OpenAI chat model

```
from langchain.prompts import ChatPromptTemplate,  
SystemMessagePromptTemplate, HumanMessagePromptTemplate  
from langchain.chat_models import ChatOpenAI  
import os
```

Setting Up the OpenAI API Key

Do whatever is needed to set up the API key:

```
openai_api_key = os.environ.get("OPENAI_API_KEY")
if openai_api_key is None:
    openai_api_key = "your_api_key_here" # Replace with your
    actual API key
```

Creating a Chat Model Instance

You create an instance of ChatOpenAI called chat and specify the desired temperature (0) and the OpenAI API key:

```
chat = ChatOpenAI(temperature=0, openai_api_key=openai_api_key)
```

Defining the Chat Template

You define a chat template as a multiline string using triple quotes (""").

- The template includes a system message that describes the assistant's role and behavior.
- It also includes a placeholder for the user's input ({text}).

```
template = """
```

```
You are an enthusiastic assistant that rewrites the user's text
to sound more exciting.
```

```
User: {text}
```

```
Assistant: """
```

Creating a Prompt Template

You create a `ChatPromptTemplate` instance called `prompt` using the `from_messages()` method.

You pass a list of message prompt templates to `from_messages()`:

- `SystemMessagePromptTemplate.from_template()`:
Creates a system message prompt template with the specified content
- `HumanMessagePromptTemplate.from_template()`:
Creates a human message prompt template with the `{text}` placeholder for user input

```
prompt = ChatPromptTemplate.from_messages([
    SystemMessagePromptTemplate.from_template(
        "You are an enthusiastic assistant that rewrites the
        user's text to sound more exciting."
    ),
    HumanMessagePromptTemplate.from_template("{text}"),
])
```

Getting User Input

You prompt the user to enter some text using `input()` and store it in the `user_input` variable:

```
user_input = input("Enter some text: ")
```

Formatting the Prompt with User Input

You use the `format_prompt()` method of the `prompt` instance to format the prompt with the user's input.

- The `text` parameter is passed as a keyword argument to replace the `{text}` placeholder in the prompt template.
- The formatted prompt is stored in the `formatted_prompt` variable.

```
formatted_prompt = prompt.format_prompt(text=user_input)
```

Printing the Formatted Prompt

You print a separator line and the heading “Formatted Prompt:”. You convert the formatted prompt to a list of messages using the `to_messages()` method and print it:

```
print("\nFormatted Prompt:")
print(formatted_prompt.to_messages())
```

Generating the Chat Completion

You pass the formatted prompt messages to the `chat` instance to generate the chat completion. The generated response is stored in the `response` variable:

```
response = chat(formatted_prompt.to_messages())
```

Printing the Assistant's Response

You print a separator line and the heading “Assistant's Response” and access the generated response content using `response.content` and print it:

```
print("\nAssistant's Response:")
print(response.content)
```

Congratulations! You just learned how to use the LangChain library to create a chat prompt template, format it with user input, and generate a chat completion using the OpenAI chat model. The chat prompt template includes a system message that defines the assistant's role and behavior and a human message that represents the user's input.

The code creates a chat model instance, prompts the user for input, formats the prompt with the user's input, and generates a chat completion based on the formatted prompt. Finally, it prints the assistant's response.

Case Study: Streamlining Customer Service

Background and Context: Your company is a prominent online retailer specializing in consumer electronics and is having challenges with high customer inquiry volumes leading to prolonged wait times and decreased customer satisfaction. Your company wants to enhance response accuracy and speed by using large language models (LLMs) in their customer service operations.

Implementation Details

Here is how you would go about implementing it:

Prompt Design: You would create initial prompts based on frequent customer inquiries, including order tracking and product issues. You would further design them to recognize various customer emotions and intents for personalization.

Integration with LLMs: You would choose any of the GPT models and integrate it into the existing CRM system to handle queries effectively.

Feedback and Iteration: During the pilot phase, track the automated responses and use customer and agent feedback, refine prompts, and adjust strategies continually.

Outcomes and Benefits

Here are the benefits one can expect:

Quantitative Results: Reduced average response times from 15 to 2 minutes. First contact resolution rates increased by 40%.

Qualitative Feedback: Customer satisfaction surveys showed a 30% improvement in interaction quality. Agents reported less workload and higher job satisfaction due to decreased manual query handling.

Lessons Learned and Best Practices

Here are the lessons learned:

Prompt Flexibility: You found out that the ability to modify prompts dynamically based on real-time data was crucial for addressing customer needs.

Scalability: The system was designed to easily handle increased loads during peak periods.

Continuous Improvement: Regular prompt reviews and ongoing training sessions were implemented to keep the system and team performance at peak levels.

Now let us dig a little deeper into the design process itself.

Initial Design and Customization of Prompts

Objective: Your company wants to develop an initial set of prompts that could efficiently handle the most common customer inquiries and improve responsiveness to customer needs through customization.

Initial Design

Here is a walk-through of the process you could take:

Development Process: Your team analyzed historical customer service data to identify frequent inquiries related to order status, product issues, and returns.

Prompt Construction: You then created specific prompts for each category.

Order Tracking: “Can you provide me with an update on my order status? My order number is {order_number}.”

Product Issues: “I’m having trouble with my {product_name}, could you help me troubleshoot it?”

Returns: “I would like to return my purchase. What is the process for returns?”

Testing: You tested these prompts in a controlled environment to monitor and refine the LLM’s performance based on its responses.

Advanced Engineering

Here are some advanced techniques you can use:

Sentiment Analysis: You can integrate tools to assess the emotional tone of inquiries and tailor the LLM to respond accordingly, for example, by detecting frustration in delayed order messages.

Intent Recognition: You can use NLP techniques to differentiate customer intents behind similar questions and make responses more contextually appropriate.

Example-Driven Customization: Use examples such as the ones below to train your LLM for personalized responses.

Frustrated Customer: “I have been waiting for over a week, where is my order?”

Anxious Customer: “I’m worried about my order status. Can you update me as soon as possible?”

Impact

If you follow the above best practices, results are sure to follow:

Enhanced Interaction: The well-crafted prompts enabled quicker, more informative, and empathetic responses, leading to improved customer satisfaction.

Efficiency in Resolution: The precise design reduced the need for multiple interactions to solve the customer needs.

This approach showcases the critical role of meticulous prompt design and ongoing refinement in leveraging LLMs to improve customer service, demonstrating both operational efficiency and enhanced customer experience.

Key Takeaways

Let us review the key takeaways from this chapter.

You have learned how to create detailed prompts that guide LLMs to generate responses that are not only relevant but also contextually aligned with your specific needs. This will help you to tap into the full potential of these LLMs and ensure your LLM interactions are both meaningful and productive.

You also explored how to refine prompts, conduct iterative testing, and use parameters such as temperature and max tokens. You also learned how to fine-tune the model's outputs to make them more precise and customized to specific tasks. You can now continuously improve your prompts and adapt to new requirements as they arise.

I hope the knowledge and techniques that you learned in this chapter will help you to tweak your prompting strategies as LLM technologies continue to evolve.

Review Questions

Let us test your understanding of this chapter's content.

- 1) What is the primary purpose of prompt engineering in the context of using LLMs?
 - A. To increase the computational speed of language models
 - B. To guide the model to generate specific and relevant outputs

- C. To reduce the cost of data storage for language models
 - D. To enhance the graphical interface of AI applications
- 2) Which component is NOT typically involved in crafting an effective prompt?
- A. Instruction
 - B. Context
 - C. User interface design
 - D. Expected output format
- 3) What is an example of few-shot learning as used in prompt engineering?
- A. Using many examples to perform a task at one time
 - B. Providing a few key examples to help the model learn a task
 - C. Continuously training the model with examples until it performs a task
 - D. Using a single example in multiple different tasks
- 4) How can the clarity of a prompt affect an LLM's response?
- A. It does not affect the LLM as it relies on underlying algorithms.
 - B. Clear prompts can confuse the LLM, leading to irrelevant responses.

- C. Clarity in prompts can lead to more accurate and relevant responses.
 - D. Clear prompts reduce the processing time of responses.
- 5) Which of the following is a true statement about prompt templates?
- A. They limit the LLM's ability to generate creative content.
 - B. They are mainly used for database management systems.
 - C. They provide a structured way to consistently invoke LLMs.
 - D. They prevent the LLM from accessing its trained data.

Answers

1. B. The primary purpose of prompt engineering is to guide the model to generate specific and relevant outputs.
2. C. User interface design is not typically involved in crafting an effective prompt for language models.
3. B. Few-shot learning in prompt engineering involves providing a few key examples to help the model learn how to perform a specific task.
4. C. Clarity in prompts can lead to more accurate and relevant responses from a language model.

5. C. Prompt templates provide a structured way to consistently invoke LLMs, ensuring that inputs are formatted in a way that maximizes the model's ability to understand and respond appropriately.

Further Reading

***You Look Like a Thing and I Love You* by Janelle Shane:** This lighthearted yet informative book shows how LLM works in weird and wonderful ways, including how prompts can go hilariously wrong.

OpenAI Blog (<https://openai.com/blog/teaching-with-ai>): A fantastic resource for articles on the latest advancements in AI technology, including detailed discussions on prompt engineering and LLM capabilities.

Prompt engineering and you: How to prepare for the work of the future (<https://cloud.google.com/blog/transform/how-to-be-a-better-prompt-engineer>): Stay updated with Google's latest research and insights in AI, which often cover topics related to the development and ethical use of LLMs.

***Designing Data-Intensive Applications* by Martin Kleppmann:** While not specifically about LLMs, this book is crucial for understanding how to manage the data that feeds into AI systems, which is vital for prompt engineering.

LangChain Documentation (https://python.langchain.com/docs/modules/model_io/prompts/): For hands-on guidance and to explore more about prompt templates and other features, the LangChain documentation is the go-to resource.

CHAPTER 6

Building Intelligent Chatbots and Automated Analysis Systems Using Chains

In this chapter, we will explore the concept of “Chains” within the LangChain framework. Chains are a powerful feature that helps you to orchestrate sequences of actions to process data, make decisions, and interact with external systems effectively. You can develop complex, context-aware applications using large language models. In this chapter, you will learn about the components of a Chain, how they are structured, and how to build complex and robust gen AI applications using them.

Introduction to LangChain Chains

LangChain chains are an exciting foundational concept that helps you build robust generative AI applications.

What Are LangChain Chains?

LangChain chains are the building blocks of generative AI applications. They help you to build applications with a series of interconnected steps that work together to complete a specific task. For example, consider a use case where you need to gather data from various sources, analyze this data to extract meaningful insights, and then generate a human-like response based on these insights. Using LangChain chains, you can combine these steps into an efficient workflow where each step is well defined and integrated. To actually implement these steps, you will be leveraging a combination of large language models (LLMs), prompts, and tools.

Let us look at a simple example to understand them:

```
# Import LangChain and OpenAI libraries
from dotenv import load_dotenv
import os
load_dotenv()
MY_OPENAI_API_KEY = os.environ.get("MY_OPENAI_API_KEY")
print(f'OPEN AI KEY is: {MY_OPENAI_API_KEY}');
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
from langchain.llms import OpenAI

# Define the prompt template
template = "What is the capital of {country}?"
prompt = PromptTemplate(template=template, input_variables=["country"])

# Initialize the LLM
llm = OpenAI(temperature=0.9)

# Create the LLMChain
chain = LLMChain(llm=llm, prompt=prompt)

# Run the chain
result = chain.run("France")
print(result)
```

In this example, you are creating an LLMChain and passing in the country name as input, and the chain returns its capital city as output. The chain itself is made of a prompt template and an OpenAI language model. When you run the chain with the input “France,” you get the output “Paris.”

Why Are Chains Important in Generative AI?

Now that you have a basic understanding of what chains are, let us discuss why they are important when developing generative AI apps.

Using chains, you can adopt a structured and modular approach when building AI applications. You can break down complex tasks into smaller and manageable steps. This modularity makes your code more organized, readable, and easy to maintain. You can easily swap out components, modify existing chains, or create new ones with minimal effort and the least impact to the overall app functionality.

Secondly, you can leverage the full power of language models while at the same time not having to worry about the internal workings of the LLMs and the complexities involved in interacting with them. By leveraging the high-level interface of Chains, you can just focus on the task at hand.

Thirdly, you can easily integrate various tools and services into your AI applications. You can easily retrieve data from an API, query a database, or process text using external libraries and include them as part of your application workflow.

When your application grows, its complexity can also grow. By using chains, you can easily scale your operations by simply modifying existing chains or adding new ones without having to overhaul your entire application. You can also easily maintain the app and troubleshoot individual components independently.

You can reuse a chain across multiple parts of your application or even in different projects, which not only saves time but also ensures consistency in how these tasks are performed across your applications.

As you progress through this chapter, you will learn more about the different types of chains available in LangChain, their use cases, and learn how to use them to build powerful generative AI applications.

Understanding the Components of Chains

Now that you have a basic idea of what chains are and why they are important, let us explore what these chains are made of.

When talking about the components of a chain, there are two ways to look at it. First, let us look at it a higher level how you can break down these components, and then we will take a deeper look at the internal workings of the chain, in terms of how it brings together various LangChain modules such as LLMs, prompts, and tools to execute a step in the workflow.

Higher-Level Components of a Chain App

We already discussed that a Chain in LangChain can be thought of as a series of interconnected steps, each designed to execute a specific function with the objective of ultimately achieving a desired outcome. But then, we need to discuss how to structure these components:

1. **Trigger:** First is the trigger. This is what starts the chain. It could be a user request, a scheduled task, or an event within your application.
2. **Steps (or Nodes):** These are the individual tasks that the chain executes. Each step might involve processing data, making a decision, or interacting with an external system. You will use the steps to execute tasks in a predefined order, and each step's output can serve as input for the next. In LangChain, you will leverage an LLM, a prompt, and possibly

CHAPTER 6 BUILDING INTELLIGENT CHATBOTS AND AUTOMATED ANALYSIS SYSTEMS USING CHAINS

some tools to implement these steps. You will use the LLM to process the input, the prompt to guide the LLM's response, and the tools to interact with external resources.

3. **Decision Points:** As the chain progresses, you may encounter decision points where you need to make choices based on the data the chain has processed so far. These decision points will act as gateways to direct the chain's flow based on predefined conditions. You will implement these decision points using conditional logic or router chains.
4. **End Point:** This is where the chain will conclude its execution and return the final output. The end point represents the completion of the chain's task, whether it is generating a response, updating a database, or triggering another action.

Here is an example of a customer support chatbot implemented using LangChain:

- **Trigger:** A user sends a message to the chatbot.
- **Steps**
 - Step 1: You process the user's message and pass it as input to an LLM.
 - Step 2: The LLM generates a response based on the input and a predefined prompt.
 - Step 3: You then process the response using tools (e.g., sentiment analysis, entity recognition).

- **Decision Points**
 - If the user's message is classified as a complaint, you may route the conversation to a human agent.
 - If the user's message is a frequently asked question, you can provide a predefined answer.
- **End Point:** You then complete the conversation flow by sending the final response back to the user.

Internal Components in a Step

In this example, the internal components (LLM, prompts, tools) work together within each step, while the high-level components (trigger, steps, decision points, end point) define the overall structure and flow of the chain.

The language model (LLM) is at the heart of every chain and is responsible for understanding and generating human-like text. Prompts guide the LLM to generate relevant and coherent responses by providing context and instructions.

Tools allow chains to interact with the outside world, such as fetching data from APIs, querying databases, or performing specific tasks. Some chains also have a memory component to retain context across multiple interactions.

Types of Chains

Now that you know the basic components, let us explore the two main types of chains you will see in LangChain: LCEL (LangChain Execution Language) chains and legacy chains. Think of LCEL chains as more modern and flexible, while legacy chains are more straightforward and suitable for less complex tasks.

LCEL Chains

LCEL stands for “LangChain Execution Language” and is the modern way of creating chains. LCELs allow you to define chains with granular control over each step using a domain-specific language designed specifically for this purpose:

- **Flexibility:** You can define complex logic and integrate various operations seamlessly.
- **Scalability:** Due to their modular nature, you can easily scale and modify them, making them ideal for growing applications.
- **Use Case:** Suppose you need to develop a system that fetches user data, analyzes it, and then dynamically generates a personalized report. You can use LCEL Chains to build this workflow with precision and handle each aspect of the task effectively.

Legacy Chains

Legacy Chains are the original method of building chains in LangChain. While they are less flexible than LCEL Chains, you will find them easier to use because they have been pre-built to handle specific tasks.

- **Simplicity:** These chains are straightforward to implement with less setup and configuration required as compared to LCEL Chains.
- **Direct Application:** You could choose them for applications where the workflow is stable with less frequent changes or customization.

- **Use Case:** If your application needs to perform a standard task, like sending a formatted email response based on user queries, a legacy chain might be the perfect fit.

Difference Between LCEL and Legacy Chains

It is better to look at a few examples of how to create LCEL and legacy chains to understand the difference between them.

LCEL Chain Example

Here is a simple example of an LCEL chain:

```
# LCEL Chain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import load_chain

template = "What is the capital of {country}?"
prompt = PromptTemplate(template=template, input_variables=["country"])
chain = load_chain("llm_chain", llm=OpenAI(), prompt=prompt)
result = chain.run("France")
```

Legacy Chain Example

Here is a simple example of a legacy chain:

```
# Legacy Chain
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
```

```
template = "What is the capital of {country}?"  
prompt = PromptTemplate(template=template, input_  
variables=["country"])  
chain = LLMChain(llm=llm, prompt=prompt)  
result = chain.run("France")
```

In case your code doesn't work due to version issues, you may also try

```
chain = prompt | llm  
chain.invoke("France") will work
```

As you can see, the legacy chain is constructed using the LLMChain class, while the LCEL chain is loaded using the load_chain function. The LCEL chain offers a more streamlined and flexible approach. The chain consists of an OpenAI LLM and a prompt template. LCEL chains offer advanced features like streaming, asynchronous execution, and automatic observability, making them the go-to for most modern applications.

So, which one should you choose? Let's discuss it next.

When to Use Different Types of Chains

Let us discuss the common scenarios where each type of chain can be used.

LCEL Chains

Here are some use cases where LCEL is better suited:

1. Use LCEL when you have to define complex logic and need greater control over the flow of data between steps.

CHAPTER 6 BUILDING INTELLIGENT CHATBOTS AND AUTOMATED ANALYSIS SYSTEMS USING CHAINS

2. Use it if you are working with large datasets or need to process data in real time, because LCEL supports streaming, asynchronous execution, and parallelization.
3. When you need to integrate with external APIs, databases, or services, use LCEL chains to include custom tools and plug-ins.

Legacy Chains

Use legacy chains for the following situations:

- Use legacy chains when you have a straightforward use case that doesn't require complex logic or custom integrations. They are ideal for quick prototyping and experimentation.
- If you are working with a well-defined and stable dataset.
- When you are just starting out with LangChain and want to get a feel for how chains work, legacy chains are a great place to begin.

Building with LCEL Chains

Now that we have discussed when to use LCEL chains, let's discuss how to create them. There are two main aspects to consider when building LCEL chains, namely, constructing the chain and then customizing it to your requirements.

Constructing LCEL Chains

To construct an LCEL chain, you will typically use the `load_chain` function, which takes care of initializing the chain with the necessary components, such as language models, prompts, and tools. Here is a simple example:



The `"llm_chain"` template in the provided example could lead to confusion since it might not directly exist in the user's installation of LangChain. This might give the impression that the template is predefined, but in reality, the `load_chain` function might require custom chain definitions rather than pre-built ones like `"llm_chain."` Let's adjust the explanation to make it clearer and address the comment.

Updated Explanation:

To construct a chain in LangChain, you often use the `load_chain` function, which initializes the chain by connecting necessary components like language models, prompts, and tools. However, the specific chain template, such as `"llm_chain"` in this example, needs to be either created beforehand or loaded from a custom setup that you've defined in your project. LangChain doesn't come with predefined chains like `"llm_chain"` out of the box.

Here is how you can create a simple custom chain manually without relying on `load_chain`:

```
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
```

CHAPTER 6 BUILDING INTELLIGENT CHATBOTS AND AUTOMATED ANALYSIS SYSTEMS USING CHAINS

```
# Step 1: Initialize your language model
```

```
llm = OpenAI(temperature=0.9)
```

```
# Step 2: Define a prompt template
```

```
prompt = PromptTemplate(  
    input_variables=["product"],  
    template="What is a good name for a company that makes  
    {product}?",  
)
```

```
# Step 3: Construct the chain manually using the LLMChain class
```

```
chain = LLMChain(llm=llm, prompt=prompt)
```

In this example, you are constructing an LCEL chain using the `llm_chain` template. You provide an OpenAI language model (`llm`) and a prompt template (`prompt`) as parameters to customize the behavior of the chain.

Customizing LCEL Chains

Once you have constructed the LCEL chain, the next step is to customize it. The ease with which you can customize a LCEL chain is one of its greatest strengths. You can tailor every aspect of the chain to fit your specific needs. Let us explore a few ways to customize your LCEL chains:

Adding Custom Logic: You can add custom logic between steps in the LCEL chain by defining your own functions or classes and including them in the chain's workflow. This allows you to perform specific data transformations or apply business rules.

Integrating External Services: You can use tools to interact with APIs, databases, or third-party libraries. You will simply define the necessary parameters and authentication details, and LangChain will do the rest.

Modifying Prompt Templates: LCEL chains allow you to define and modify prompt templates easily using variables, conditional logic, and even generate dynamic prompts based on the input data.

Here is an example of customizing an LCEL chain:

```
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import load_chain
from langchain.tools import APITool

llm = OpenAI(temperature=0.9)
prompt = PromptTemplate(
    input_variables=["product"],
    template="What is a good name for a company that makes
    {product}?",
)

api_tool = APITool(api_url="https://example.com/api")

chain = load_chain("llm_chain", llm=llm, prompt=prompt,
tools=[api_tool])
```

In this example, we are customizing the LCEL chain by adding an APITool to interact with an external API. We define the `api_url` parameter to specify the API endpoint and include the tool in the tools list when constructing the chain.

Executing LCEL Chains

Once you have constructed and customized the LCEL chain, it is time to put it into action. You can use various execution modes to cater to different scenarios and optimize performance. Let us explore the three main execution modes below.

Streaming Execution

When working with large datasets, instead of waiting for the entire dataset to be loaded into memory, you can process data as it arrives. You can handle live data streams as well as process data in chunks.

To enable streaming execution, you can use the `streaming=True` parameter when calling the chain. Here is how you do it:

```
chain.run({"product": "smartphone"}, streaming=True)
```

Async Execution

With the help of asynchronous execution, you can run multiple tasks concurrently and make the most of your system's resources. This way, you can execute multiple independent tasks in parallel.

To use async execution, you can leverage the `arun` method provided by LCEL chains. Here is an example:

```
import asyncio
async def generate_names(product):
    return await chain.arun({"product": product})
product_names = asyncio.run(generate_names("smartphone"))
```

Batch Execution

You can use batch execution to process multiple inputs in a single call to the chain. This can significantly improve performance by reducing the overhead of multiple individual calls.

To perform batch execution, you will have to pass a list of inputs to the chain's `apply` method. Here is an example:

```
products = ["smartphone", "laptop", "smartwatch"]
product_names = chain.apply([{"product": product} for product
in products])
```

Observability in LCEL Chains

Observability is crucial when building and debugging LCEL chains. You can gain insights into the inner workings of your chain, track data flow, and identify potential bottlenecks or issues.

LCEL chains provide built-in observability features, such as logging and tracing. You can access detailed logs and traces for each step of the chain, such as input and output data, execution time, and any errors or exceptions that may occur.

To enable observability in your LCEL chains, you can use the `verbose=True` parameter when constructing the chain. Here is an example:

```
chain = load_chain("llm_chain", llm=llm, prompt=prompt,
verbose=True)
```

With the `RunnableSequence`, your code may look like this:

```
from langchain.globals import set_debug
set_debug(True)
```

Once you have observability enabled, you can easily monitor the execution of your chain and gain valuable insights for debugging and optimization.

Types of LCEL Chains

Let us discuss some of the types of LCEL chains mentioned in LangChain documentation, which I have shown as a table.

Chain Constructor	When to Use
<code>create_stuff_documents_chain</code>	This chain is your go-to when you want to format a list of documents into a prompt and pass it to an LLM. Just make sure the documents fit within the context window of the LLM you are using.
<code>create_openai_fn_runnable</code>	If you want to use OpenAI function calling to optionally structure an output response, this is your chain of choice. You can pass in multiple functions, but it is not required to call them all.
<code>create_structured_output_runnable</code>	When you want to use OpenAI function calling to force the LLM to respond with a specific function, this is your go-to chain. You can only pass in one function, and the chain will always return this response. It is useful when you need a structured output every time.
<code>load_query_constructor_runnable</code>	This chain is useful to generate queries. You specify a list of allowed operations, and it returns a runnable that converts natural language queries into those operations.
<code>create_sql_query_chain</code>	When you want to construct a query for a SQL database from natural language, you can use this chain. No more struggling with complex SQL syntax!
<code>create_history_aware_retriever</code>	This chain takes in conversation history and uses it to generate a search query, which is then passed to the underlying retriever.
<code>create_retrieval_chain</code>	When you want to retrieve relevant documents based on a user inquiry, this chain is your go-to. It takes the user's input, passes it to the retriever to fetch relevant documents, and then combines those documents with the original input to generate a response using an LLM.

Command Generation Using Query Constructor Chain

Now, let us see an example of how you can use the `load_query_constructor_runnable` chain in your code:

```
from langchain.chains import load_query_constructor_runnable

# Define the allowed operations
allowed_operations = ["search", "filter", "sort"]

# Load the query constructor chain
query_constructor = load_query_constructor_runnable(allowed_
operations)

# Use the query constructor to convert natural language to
allowed operations
natural_language_query = "Find all products priced below $100
and sort them by price"
structured_query = query_constructor(natural_language_query)

print(structured_query)
```

In this example, we define a list of allowed operations: “search,” “filter,” and “sort.” We then load the `query_constructor_runnable` chain, passing in the allowed operations. This chain takes a natural language query and converts it into the specified allowed operations.

You provide a natural language query: “Find all products priced below \$100 and sort them by price.” The `query_constructor` chain processes this query and returns a structured query based on the allowed operations.

And here is the output you will get:

```
{
  "operations": [
    {
      "operation": "search",
      "query": "products"
    },
    {
      "operation": "filter",
      "condition": "price < 100"
    },
    {
      "operation": "sort",
      "key": "price"
    }
  ]
}
```

In the example above, the natural language query is transformed into a structured query with specific operations like “search,” “filter,” and “sort.” This structured format is easier for your application to process and execute.

Building with Legacy Chains

Constructing legacy chains is a straightforward process. You simply create an instance of the desired chain class and provide the necessary parameters. You will have to refer to the documentation for the specific chain you are using because each chain has its own set of required and optional parameters.

Constructing Legacy Chains

Here is an example of constructing a legacy chain:

```
from langchain.chains import LLMChain
from langchain.llms import OpenAI

llm = OpenAI(temperature=0.9)
chain = LLMChain(llm=llm, param1=value1, param2=value2)
```

In this example, you import the desired chain class (`MyCustomChain`) and create an instance of it. You then provide the necessary parameters, such as the language model (`llm`) and any custom parameters (`param1` and `param2`).

Executing Legacy Chains

Executing legacy chains is just as simple as constructing them. Most legacy chains provide a `run` method that takes the input data and returns the result. Here is an example:

```
result = chain.run(input_data) print(result)
```

In this example, you simply call the `run` method on the chain instance, passing in the `input_data`. The chain processes the input and returns the result, which you then print.

Types of Legacy Chains

Now that you have a general understanding of legacy chains, let us explore some of the most commonly used ones.

Chain	When to Use
APIChain	When you want to convert a query into an API request, execute it, and then use an LLM to interpret the response.
OpenAPIEndpointChain	Similar to APIChain but optimized for OpenAPI endpoints.
ConversationalRetrievalChain	For having conversations with a document, using previous conversation history to refine queries.
StuffDocumentsChain	When you have a list of documents that fit within the context window of your LLM.
ReduceDocumentsChain	For processing a large number of documents in parallel by iteratively reducing them.
MapReduceDocumentsChain	Similar to ReduceDocumentsChain, but with an initial LLM call before reducing the documents.
RefineDocumentsChain	For sequentially refining an answer based on multiple documents.
MapRerankDocumentsChain	When you want to answer based on a single document with the highest confidence score.
ConstitutionalChain	For enforcing constitutional principles in a chain's answer.
LLMChain	A basic chain for interacting with an LLM.
ElasticsearchDatabaseChain	For asking natural language questions against an Elasticsearch database.

(continued)

Chain	When to Use
FlareChain	An advanced retrieval technique for exploratory purposes.
GraphCypherQChain	For constructing Cypher queries from natural language and executing them against a graph.
LLMMath	For converting user questions to math problems and executing them.
LLMCheckerChain	Uses a second LLM call to verify the initial answer.
LLMSummarizationChecker	Creates a summary using multiple LLM calls for increased accuracy.

These are just a few examples from the extensive list of legacy chains available. Each chain has its unique strengths and use cases, allowing you to tackle a wide range of tasks.

Building Real-World Apps with Legacy Chains

You can build very effective real-world generative AI apps using legacy chains. Below are some popular examples.

Document Chatbot App Using ConversationalRetrievalChain

Let us say you want to have a conversation with a document using the `ConversationalRetrievalChain`. Here is a simplified example:

```
from langchain.chains import ConversationalRetrievalChain
from langchain.chat_models import ChatOpenAI
from langchain.retrievers import ContextualCompressionRetriever

# Initialize the retriever and the LLM
retriever = ContextualCompressionRetriever(...)
llm = ChatOpenAI(temperature=0)

# Create the ConversationalRetrievalChain
qa = ConversationalRetrievalChain.from_llm(llm,
retriever=retriever)

# Start the conversation
chat_history = []
while True:
    question = input("User: ")
    result = qa({"question": question, "chat_history": chat_
history})
    chat_history.append((question, result['answer']))
    print(f"Assistant: {result['answer']}")
```

In this example, you initialize the retriever and the LLM, create the `ConversationalRetrievalChain`, and then start a conversation loop. The chain takes care of using the chat history to refine the queries and provide contextual responses.

Building Text Generation Apps Using LLMChain

You can use the LLMChain to combine a language model (LLM) with a prompt template to generate text. Here's a quick example:

```
from langchain.chains import LLMChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate

template = "What is a good name for a company that makes
{product}?"
prompt = PromptTemplate(template=template, input_
variables=["product"])
llm = OpenAI(temperature=0.9)

chain = LLMChain(llm=llm, prompt=prompt)
company_name = chain.run("colorful socks")
print(company_name)
```

In this example, you first define a prompt template and an OpenAI language model. You then create an LLMChain by combining the llm and prompt and run the chain with a product input and print the generated company name.

Building Conversational Apps with ConversationChain

You will use this chain when you have to have a conversation with a language model and keep track of the conversation history. Here is an example:

```
from langchain.chains import ConversationChain
from langchain.llms import OpenAI

llm = OpenAI(temperature=0.9)
conversation = ConversationChain(llm=llm)
```

```
output = conversation.predict(input="Hi, how are you?")
print(output)
output = conversation.predict(input="What's the weather like
today?")
print(output)
```

In this example, you create a `ConversationChain` with an `OpenAI` language model and then use the `predict` method to have a conversation with the AI assistant. The chain keeps track of the conversation history and maintains a more natural and contextual interaction.

Building Q&A Apps Using RetrievalQA

You will use the `RetrievalQA` chain to perform question answering over a collection of documents. Here is an example:

```
from langchain.chains import RetrievalQA
from langchain.llms import OpenAI
from langchain.document_loaders import TextLoader
from langchain.indexes import VectorstoreIndexCreator
from langchain_openai.embeddings import OpenAIEmbeddings

loader = TextLoader("path/to/documents.txt")
index = VectorstoreIndexCreator().from_loaders([loader])
embedding = OpenAIEmbeddings(openai_api_key=MY_OPENAI_API_KEY)
llm = OpenAI(temperature=0.9)
chain = RetrievalQA.from_chain_type(llm=llm, chain_
type="stuff", retriever=index.vectorstore.as_retriever())

query = "What is the capital of France?"
result = chain.run(query)
print(result)
```

In this example, you load a collection of documents using the `TextLoader` and then create an index using the `VectorstoreIndexCreator`. Next, you instantiate a `RetrievalQA` chain with an OpenAI language model and specify the retrieval strategy as “stuff.” Finally, you run the chain with a query and print the result.

Document Processing App with MapReduceChain

The `MapReduceChain` helps you to parallelize the processing of documents by splitting them into chunks, applying a mapping function to each chunk, and then reducing the results. Here is an example:

```
from langchain.chains import MapReduceChain
from langchain.chains.mapreduce import combine_results
from langchain.prompts import PromptTemplate
from langchain.llms import OpenAI
from langchain.document_loaders import TextLoader

loader = TextLoader("path/to/documents.txt")
documents = loader.load()

llm = OpenAI(temperature=0.9)
prompt = PromptTemplate(template="Summarize this text: {text}",
input_variables=["text"])

chain = MapReduceChain.from_params(
    map_prompt=prompt,
    combine_prompt=prompt,
    llm=llm,
```

```
    chunk_size=1000,  
    reduce_chunk_overlap=0,  
)  
  
result = chain.run(documents)  
print(result)
```

In this example, you load a collection of documents using the `TextLoader` and then create a `MapReduceChain` by specifying the mapping and combining prompts, the language model, and the chunk size. Finally, you run the chain on the documents and print the summarized result.

Education Case Study: Analyzing Educational Data with MapReduce Chains

An educational research institute used MapReduce Chains to handle large volumes of academic data for analysis and reporting.

Implementation: The MapReduce Chains segmented datasets into manageable chunks, applied analytical models to each segment, and then aggregated the results to generate comprehensive reports on student performance and learning trends.

Outcome: The efficient processing led to quicker turnaround times for research papers and provided deeper insights into educational effectiveness, supporting targeted improvements in teaching strategies.

More Complex Workflow Apps Using Chain Composition Strategies

We have already learned about the different types of chains and how to work with them individually. In this section, we will explore how to compose chains and tackle even the most complex workflows.

Chain composition is all about creating a workflow by connecting multiple chains together. You would start by breaking down a complex task into smaller, manageable steps and assigning each step to a specific chain.

Let us explore some strategies you can use to compose chains to build complex real-world gen AI applications.

Data Summarization App with Sequential Chains

You will use a sequential chain when the output of one chain becomes the input for the next chain to create an application with a linear flow of data processing.

Here is an example of a sequential chain workflow:

```
from langchain.chains import LLMChain, SequentialChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate

# Chain 1: Generate keywords from a topic
topic_prompt_template = "Generate relevant keywords for the
topic: {topic}"
topic_prompt = PromptTemplate(template=topic_prompt_template,
input_variables=["topic"])
topic_chain = LLMChain(llm=OpenAI(), prompt=topic_prompt)

# Chain 2: Fetch relevant data based on keywords
data_prompt_template = "Fetch data related to the following
keywords: {keywords}"
data_prompt = PromptTemplate(template=data_prompt_template,
input_variables=["keywords"])
data_chain = LLMChain(llm=OpenAI(), prompt=data_prompt)

# Chain 3: Summarize the fetched data
summary_prompt_template = "Summarize the following
data: {data}"
```

CHAPTER 6 BUILDING INTELLIGENT CHATBOTS AND AUTOMATED ANALYSIS SYSTEMS USING CHAINS

```
summary_prompt = PromptTemplate(template=summary_prompt_
template, input_variables=["data"])
summary_chain = LLMChain(llm=OpenAI(), prompt=summary_prompt)

# Compose the chains sequentially
sequential_chain = SequentialChain(chains=[topic_chain,
data_chain, summary_chain], input_variables=["topic"],
output_variables=["summary"])

# Run the sequential chain
topic = "Artificial Intelligence"
result = sequential_chain({"topic": topic})
print(result["summary"])
```

In this example, you have three chains: `topic_chain`, `data_chain`, and `summary_chain`. We compose them sequentially using the `SequentialChain` class, specifying the order in which they should be executed. The output of `topic_chain` becomes the input for `data_chain`, and the output of `data_chain` becomes the input for `summary_chain`. Finally, you run the sequential chain with a topic and print the summarized result. It is as simple as that!

SequentialChain Use Case Example 1: Customer Support Chatbot App

Imagine building a customer support chatbot that can handle various types of queries. You can compose chains to create a powerful and efficient chatbot workflow.

You can use a `RouterChain` to route the user's query to the appropriate chain based on the intent (e.g., sales query, technical support, general inquiry).

Within each intent-specific chain, use a `SequentialChain` to break down the query into smaller steps, such as understanding the problem, retrieving relevant information, and generating a response.

If needed, you can use a `ConditionalChain` to handle specific cases or exceptions within each intent-specific chain.

SequentialChain Use Case Example 2: Content Generation Pipeline App

Consider a content generation pipeline that takes a topic and generates a well-structured article. You can compose chains to streamline the process.

Use a `SequentialChain` to define the overall pipeline flow:

Chain 1: First, generate an outline for the article based on the topic.

Chain 2: For each outline point, generate a detailed paragraph.

Chain 3: Then, combine the generated paragraphs into a coherent article.

Chain 4: Finally, proofread and refine the article.

Within each step of the sequential chain, you can use additional chains or tools to enhance the generation process, such as retrieving relevant information from a knowledge base or applying language models for text refinement.

SequentialChain Use Case Example 3: Automated Fraud Detection in Finance

A financial institution integrated SequentialChains to enhance its fraud detection systems, automating the analysis of transaction patterns for signs of fraudulent activity.

Implementation: The SequentialChains processed transactions in stages, initially flagging unusual patterns, then cross-referencing with historical data, and finally applying predictive models to assess fraud probability.

Outcome: This approach increased the detection rate of fraudulent transactions by 25%, significantly reducing losses and improving the security of customer accounts.

These are just a few examples to give you an idea of how chain composition can be applied in real-world scenarios.

Task Allocation App Using Router Chains

You will use the router chains when you want to route the input to different chains based on predefined rules or criteria.

Here is an example of a router chain workflow:

```
from langchain.chains import LLMChain, RouterChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate

# Chain 1: Handle sales queries
sales_prompt_template = "Respond to the following sales query:
{query}"
```

CHAPTER 6 BUILDING INTELLIGENT CHATBOTS AND AUTOMATED ANALYSIS SYSTEMS
USING CHAINS

```
sales_prompt = PromptTemplate(template=sales_prompt_template,
input_variables=["query"])
sales_chain = LLMChain(llm=OpenAI(), prompt=sales_prompt)

# Chain 2: Handle support queries
support_prompt_template = "Respond to the following support
query: {query}"
support_prompt = PromptTemplate(template=support_prompt_
template, input_variables=["query"])
support_chain = LLMChain(llm=OpenAI(), prompt=support_prompt)

# Router chain to direct queries based on the topic
router_chain = RouterChain.from_routes(
    routes=[
        ("sales", sales_chain),
        ("support", support_chain),
    ],
    default_chain=sales_chain,
    input_key="query",
    output_key="response",
)

# Run the router chain
query = "I have a question about my order"
result = router_chain.run(query)
print(result)
```

In this example, you have two chains, namely, `sales_chain` and `support_chain`. You created a `RouterChain` that routes the input query to the appropriate chain based on the topic. If the query contains the word “sales,” it is directed to the `sales_chain`. If it contains the word “support,” it is directed to the `support_chain`. If none of the conditions match, it falls back to the default `sales_chain`. Finally, you run the router chain with a query and print the response.

Case Study: Enhancing Customer Service with LangChain Router Chains

A major retail company implemented LangChain Router Chains to automate customer inquiry routing, aiming to improve response times and accuracy in addressing customer concerns.

Implementation and Challenges

The Router Chain categorized inquiries based on keywords like “return,” “warranty,” and “payment,” directing them to the appropriate departments. Challenges included initial misrouting and staff resistance to automated systems. Solutions involved refining the routing logic with feedback from service representatives and providing staff training to increase adoption.

Outcomes

Post implementation, the company saw a 30% reduction in handling time per inquiry and improved customer satisfaction due to quicker, more accurate responses. The Router Chain’s adaptability was enhanced by a learning mechanism that improved its accuracy over time.

Conclusion

This case study showcases the Router Chain’s effectiveness in streamlining customer service operations, demonstrating its potential to significantly enhance operational efficiency and customer satisfaction in retail settings.

Sentiment Analysis App Using Conditional Chains

You will use a conditional chain to execute different chains based on certain conditions.

Here is an example of a conditional chain workflow:

```
from langchain.chains import LLMChain, ConditionalChain
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate

# Chain 1: Positive sentiment
positive_prompt_template = "Respond positively to the following
query: {query}"
positive_prompt = PromptTemplate(template=positive_prompt_
template, input_variables=["query"])
positive_chain = LLMChain(llm=OpenAI(), prompt=positive_prompt)

# Chain 2: Negative sentiment
negative_prompt_template = "Respond cautiously to the following
query: {query}"
negative_prompt = PromptTemplate(template=negative_prompt_
template, input_variables=["query"])
negative_chain = LLMChain(llm=OpenAI(), prompt=negative_prompt)

# Conditional chain based on sentiment analysis
sentiment_conditions = [
    ("positive", positive_chain),
    ("negative", negative_chain),
]

conditional_chain = ConditionalChain.from_conditions(
    conditions=sentiment_conditions,
    default_chain=positive_chain,
```

CHAPTER 6 BUILDING INTELLIGENT CHATBOTS AND AUTOMATED ANALYSIS SYSTEMS USING CHAINS

```
    input_key="query",  
    output_key="response",  
)  
  
# Run the conditional chain  
query = "I love your product!"  
result = conditional_chain.run(query)  
print(result)
```

In this example, you have two chains: `positive_chain` and `negative_chain`. You create a `ConditionalChain` that executes the appropriate chain based on the sentiment of the input query. If the sentiment is positive, it executes the `positive_chain`. If the sentiment is negative, it executes the `negative_chain`. If the sentiment is neutral or cannot be determined, it falls back to the default `positive_chain`. Finally, you run the conditional chain with a query and print the response.

Healthcare Case Study: Handling Patient Inquiries with Conditional Chains

A healthcare provider implemented Conditional Chains to streamline patient communication and effectively triage emergency situations.

Implementation: Conditional Chains were used to analyze patient inquiries based on symptoms described. Non-urgent queries were automatically routed to scheduling or general advice, while potential emergencies triggered immediate alerts to medical staff.

Outcome: The system improved response times for critical cases by 40%, ensuring quick medical attention where needed, and efficiently managed routine inquiries, reducing overall administrative burden.

Advanced Chain Techniques

In this section, we will look at how to handle large datasets, errors, exceptions, optimize chain performance, testing, and debugging.

Handling Large Datasets with Chains

One of the things you will need to manage when working with large datasets is the issue of memory and slow processing times.

MapReduceChain allows you to process the documents in parallel by splitting them into chunks and then apply a mapping function to each chunk and reduce the results. Here is an example:

```
from langchain.chains import MapReduceChain
from langchain.prompts import PromptTemplate
from langchain.llms import OpenAI

llm = OpenAI(temperature=0.9)
prompt = PromptTemplate(template="Summarize this text: {text}",
input_variables=["text"])
map_reduce_chain = MapReduceChain.from_params(
    llm=llm,
    map_prompt=prompt,
    combine_prompt=prompt,
    reduce_llm=OpenAI(temperature=0),
)

result = map_reduce_chain.run(large_dataset)
print(result)
```

CHAPTER 6 BUILDING INTELLIGENT CHATBOTS AND AUTOMATED ANALYSIS SYSTEMS USING CHAINS

In this example, you create a `MapReduceChain` and pass in a large dataset and then apply a summarization prompt to each chunk using the mapping function and then combine the results using the reducing function. You use the `reduce_llm` parameter to specify the language model to use for the final reduction step.

You can also use the `StuffDocumentsChain` in combination with a vector store, which allows you to efficiently retrieve relevant documents based on a query and process them in chunks. Here is an example:

```
from langchain.chains import StuffDocumentsChain
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings

embeddings = OpenAIEmbeddings()
vectorstore = FAISS.from_documents(large_dataset, embeddings)

chain = StuffDocumentsChain.from_llm(OpenAI(temperature=0),
document_variable_name="doc")

query = "What is the main topic of these documents?"
docs = vectorstore.similarity_search(query)
result = chain.run(input_documents=docs, question=query)
print(result)
```

In this example, you create a vector store using the `FAISS` library and the `OpenAI` embeddings. You then use the `StuffDocumentsChain` to retrieve relevant documents based on the query and process them in chunks. The `document_variable_name` parameter contains the variable name you use for the input documents in the prompt template.

Dealing with Errors and Exceptions in Chains

When working with chains, you will have to handle errors gracefully and provide meaningful feedback to users.

One way you can handle errors is by using try-except blocks within your chains. You can catch specific exceptions and provide custom error messages or fallback behavior. Here is an example:

```
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
from langchain.llms import OpenAI
prompt_template = "What is the capital of {country}?"
prompt = PromptTemplate(template=prompt_template, input_variables=["country"])

chain = LLMChain(llm=OpenAI(), prompt=prompt)
try:
    result = chain.run("United States")
    print(result)
except Exception as e:
    print(f"An error occurred: {str(e)}")
    # Fallback behavior or error handling logic
```

In this example, you use a try-except block to wrap the `chain.run()` call. If an exception occurs during the execution of the chain, you catch it and print an error message. You can add your own fallback behavior or error handling logic based on your specific requirements.

You can also use the `verbose` parameter to enable verbose output, which helps you to debug and understand what is happening under the hood. Here is an example:

```
result = chain.run("United States", verbose=True)
```

When you set `verbose=True`, you will get a detailed output of each step in the chain execution, including the input and output of each component. This will make it easy to identify issues and optimize your chains.

Optimizing Chain Performance

When dealing with large datasets or complex workflows, you will need to take care of performance issues. Here are a few tips to optimize your chain performance:

Use Caching: You can use the built-in caching mechanism within LangChain to speed up your chains by storing and reusing the results of expensive operations. You can use the `cache` parameter or enable global caching using the `langchain.cache` module. Two common classes you can use are

InMemoryCache: This class stores cached results in memory, making it fast but ephemeral. Once the program stops, the cache is cleared.

SQLiteCache: This class stores cached results in a local SQLite database, making the cache persistent across sessions.

Parallelize When Possible: You should take advantage of parallelization techniques like the *MapReduceChain* to process data in parallel and reduce overall execution time.

Choose the Right Language Model: You should also experiment with different models and find the one that provides the best combination of speed and accuracy for your specific use case.

Optimize Your Prompts: You can also greatly improve the performance by developing your prompts carefully to obtain the desired response from the language model. Avoiding verbosity and focusing on the essential information will greatly improve performance.

Monitor and Profile Your Chains: Use the *langchain.debug* module to profile and monitor chain execution. Using these profiling tools can help you identify performance bottlenecks so that you can optimize the critical parts of your chains.

Testing and Debugging Chains

As a developer, you will need to have strong testing and debugging skills especially when working with chains. Here are some strategies you should adopt:

Write Unit Tests: Create unit tests for individual components of your chains, such as prompts, language models, and tools. This helps ensure that each component behaves as expected and makes it easier to identify issues when they arise.

Use Test Datasets: You need to have robust test datasets to cover various scenarios and edge cases. Test your chains against these datasets to verify their behavior and accuracy.

Use Logging: Implement logging in your chains to capture important information during execution and track the flow of data and identify any unexpected behavior.

Debug with Breakpoints: Use debugging tools like breakpoints to pause the execution of your chains at specific points and inspect the state of variables and components.

Collaborate and Seek Feedback: Also talk to other developers and seek feedback from the LangChain community. Learn from their perspectives to uncover issues and improve your chains.

Key Takeaways

In this chapter, you have learned everything from the fundamentals of LangChain chains to building sophisticated generative AI applications, such as intelligent chatbots and automated analysis systems. Let us take a moment to reflect on the key concepts we have covered so far.

Recap of Key Concepts

Here is a list of the key concepts we have explored throughout this chapter:

- **Understanding the Basics of Chains:** You learned about the different components of a chain, such as triggers, steps, decision points, and end points. You also explored the two main types of chains in LangChain: LCEL chains and legacy chains.
- **Working with LCEL Chains:** You then explored the world of LCEL chains, discovered their benefits, such as flexibility, performance, and extensibility. You learned how to construct, customize, and execute LCEL chains using various techniques like streaming, async, and batch execution.

- **Composing Chains for Complex Workflows:**
You explored chain composition and learned how to combine multiple chains using strategies like sequential chains, router chains, and conditional chains. You also looked at examples of how chain composition can be used in the real world.
- **Advanced Chain Techniques:** You tackled advanced topics like handling large datasets, dealing with errors and exceptions, optimizing chain performance, and testing and debugging chains. These techniques will help you to build complex generative AI applications.
- **Building Generative AI Applications with Chains:**
You looked at four exciting case studies, such as building a chatbot, a question-answering system, a report generation tool, and a personalized recommendation engine.

Future Possibilities with LangChain Chains

The future of generative AI is incredibly bright, and LangChain chains are in the middle of this exciting field. Here are some future possibilities you need to watch out for:

- **Advancements in Language Models:** As language models like GPT-4, Gemini, and beyond continue to evolve, the capabilities of LangChain chains will also expand. You should watch out for more powerful language models that will enable even more sophisticated human-like interactions.

- **Integration with Other AI Technologies:** You can combine LangChain chains with other AI technologies, such as computer vision, speech recognition, and robotics, to create truly intelligent and multimodal systems. You will be able to build a virtual assistant that can not only understand and respond to text but also analyze images and engage in spoken conversations.
- **Domain-Specific Applications:** You can leverage LangChain chains to build domain-specific gen AI apps in various domains, such as healthcare, finance, education, and entertainment.
- **Collaborative Development and Community Involvement:** You should also get involved in the community, participate in discussions, and collaborate with fellow developers to push the boundaries of what's possible with LangChain chains.

Glossary

Here is a list of terms to help you grasp their meaning better:

Chain: A series of interconnected steps each designed to execute a specific function, ultimately achieving a desired outcome within a LangChain application.

Trigger: The initial action that starts a chain, which could be a user request, a scheduled task, or an event within the application.

Steps (or Nodes): Individual tasks within a chain that process data, make decisions, or interact with external systems. These are executed in a predefined order where each step's output can serve as input for the next.

Decision Points: Points within a chain where choices are made based on the data processed so far. These guide the flow of the chain based on predefined conditions.

End Point: The final stage of a chain where it concludes its execution and delivers the final output.

Tools: Components that allow chains to interact with the outside world, such as fetching data from APIs or querying databases.

Memory Component: Some chains include a memory feature to retain context across multiple interactions, enhancing the continuity and relevance of tasks.

Review Questions

These questions should help clarify the integral roles and functions of various components within a LangChain Chain and their collaborative contribution to executing sophisticated tasks.

CHAPTER 6 BUILDING INTELLIGENT CHATBOTS AND AUTOMATED ANALYSIS SYSTEMS USING CHAINS

1. How do decision points within a Chain influence its execution flow?
 - A. By triggering the start of a Chain
 - B. By altering the sequence of steps based on dynamic conditions evaluated during the Chain's execution
 - C. By providing the necessary tools for interaction with external databases
 - D. By serving as the communication point with the user

2. In the context of LangChain, what is the primary role of the language model (LLM) in a Chain?
 - A. To act solely as a trigger for initiating Chains
 - B. To facilitate data storage and retrieval through external APIs
 - C. To process and generate text, guiding the execution of tasks based on human-like understanding
 - D. To function as the memory component retaining information between interactions

3. Which of the following best describes the function of tools in a LangChain Chain?
 - A. To provide the final output of a Chain
 - B. To exclusively manage user inputs and outputs

CHAPTER 6 BUILDING INTELLIGENT CHATBOTS AND AUTOMATED ANALYSIS SYSTEMS
USING CHAINS

- C. To enable the Chain to perform external interactions, such as API calls or database queries
 - D. To determine the path of execution within a Chain
4. What advantage does incorporating a memory component in a Chain provide in sequential interactions?
- A. It allows the Chain to execute without any user input.
 - B. It enhances the Chain's ability to maintain context over time, thereby improving the relevance and coherence of tasks across sessions.
 - C. It serves as the sole decision-making entity within the Chain.
 - D. It triggers the end point after each interaction.
5. Considering the role of prompts in a Chain, how do they specifically assist the language model (LLM)?
- A. By terminating the Chain upon completion of tasks
 - B. By structuring the input to the LLM to ensure task-specific responses are generated accurately
 - C. By providing a graphical interface for user interaction
 - D. By managing the data flow between different steps in a Chain

Answers

Here are the answers to the questions above:

1. How do decision points within a Chain influence its execution flow?

Answer: B. By altering the sequence of steps based on dynamic conditions evaluated during the Chain's execution

2. In the context of LangChain, what is the primary role of the language model (LLM) in a Chain?

Answer: C. To process and generate text, guiding the execution of tasks based on human-like understanding

3. Which of the following best describes the function of tools in a LangChain Chain?

Answer: C. To enable the chain to perform external interactions, such as API calls or database queries

4. What advantage does incorporating a memory component in a Chain provide in sequential interactions?

Answer: B. It enhances the Chain's ability to maintain context over time, thereby improving the relevance and coherence of tasks across sessions.

5. Considering the role of prompts in a Chain, how do they specifically assist the language model (LLM)?

Answer: B. By structuring the input to the LLM to ensure task-specific responses are generated accurately

Further Reading

Here are some links for further reading specific to LangChain chains:

LangChain Documentation: It covers various aspects of chains, such as

- Getting started with chains
- Types of chains (e.g., LLMChain, SequentialChain, TransformChain)
- Utility chains (e.g., MapReduceChain, AnalyzeDocumentChain)
- Async support for chains
- Tracing and callbacks in chains

<https://python.langchain.com/v0.1/docs/modules/chains/>

LangChain Playground: You can use the LangChain Playground to experiment with different types of chains. You can use the pre-built examples and modify and run the code directly in the browser.

<https://smith.langchain.com/hub/bagatur/chat-langchain-response/playground>

<https://weblangchain.fly.dev/chat/playground/?ref=blog.langchain.dev>

LangChain GitHub Repository: You can access a large collection of example notebooks and scripts to learn about chains in various scenarios. You can find examples related to

- Question answering with chains
- Summarization with chains
- Chatbot implementation using chains
- Text generation with chains

<https://github.com/langchain-ai/langchain/tree/master/libs/langchain/langchain/chains>

LangChain Community Forum: Using the LangChain community forum, you can engage with other developers, ask questions, and learn from their experiences. You can find discussions and insights related to chains, as well as share your own projects and seek feedback from the community.

<https://js.langchain.com/docs/community>

LangChain Blog: The LangChain blog contains articles and tutorials on various topics related to LangChain, including chains.

<https://blog.langchain.dev/>

YouTube Tutorials: You can access several tutorials for step-by-step explanations and practical demonstrations of working with chains.

LangChain Official Channel: <https://www.youtube.com/@LangChain>

CHAPTER 7

Building Advanced Q&A and Search Applications Using Retrieval-Augmented Generation (RAG)

In this chapter, you will learn about Retrieval-Augmented Generation (RAG), an innovative approach that combines retrieval-based methods with large language models (LLMs) to improve the accuracy and relevance of generated responses. By the end of this chapter, you will understand the fundamental concepts of RAG, how it integrates with LLMs, and how to implement RAG using LangChain, Pinecone, and OpenAI. You will explore various strategies to enhance the RAG process for specific use cases. You will gain practical skills building advanced question-answering and information retrieval systems Using Retrieval-Augmented Generation (RAG).

By the end of this chapter, you will learn how to build advanced question-answering and information retrieval systems Using Retrieval-Augmented Generation (RAG).

Importance of RAG

Let us talk about why RAG (Retrieval-Augmented Generation) is so important. You see, many LLM applications require user-specific data that was not part of the model's training set. This presents several challenges:

1. **Recent Information:** When you want to ask about events that occurred after the model's knowledge cutoff date, the model is unable to give you a meaningful response. It may even lead to hallucinations.
2. **Private Documents:** When you need answers based on your own private documents, traditional LLMs fall short.
3. **Enterprise Data Security:** For your business, you need to keep your sensitive information secure.

This is where RAG comes in because it allows you to retrieve external data and pass it to the LLM during the content generation step. It addresses these challenges in several ways:

1. **Up-to-Date Information:** RAG can ensure your responses are current by pulling in the latest data.
2. **Personalized Knowledge:** It can access your private documents to provide tailored answers.

3. **Data Privacy and Security:** Crucially, RAG helps keep highly prized enterprise data mostly out of the hands of LLM vendors such as OpenAI. Here is how:
 - **Selective Data Sharing:** Only the minimal amount of data required to answer a specific question is sent to the LLM. The rest remains securely within your company's control.
 - **Data Residency:** The bulk of your company's data stays on your own servers or chosen cloud infrastructure and never leaves your company's security perimeter.
 - **Contextual Retrieval:** You can design your RAG systems to retrieve and send only nonsensitive portions of documents and thus protect your sensitive information.
 - **Compliance:** You can ensure compliance with data protection regulations by limiting exposure of sensitive data to third-party services.

Thus, you can strike a balance between leveraging advanced LLM capabilities and maintaining strict data security protocols. This is why RAG is an essential technology for enterprises looking to adopt AI solutions responsibly.

How Does RAG Work?

Let us look at the visual representation of the RAG process in Figure 7-1.

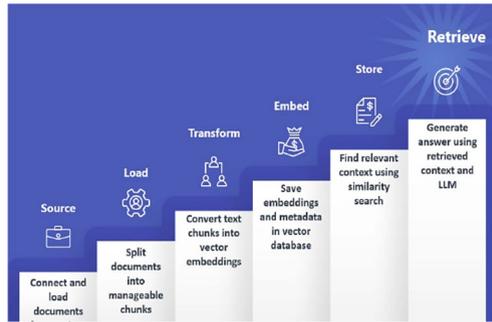


Figure 7-1. RAG Process

As you can see from Figure 7-1, it all starts with the Source, which is where you gather your precious data from. This source could be either a vast collection of documents, web pages, or any other textual data that you want to use. It is a one-time process for each source.

It is important to note that data ingestion and processing is not a one-time process, but rather an ongoing one that requires careful management:

Initial Setup: The initial ingestion of your data source is the first step in setting up your RAG system.

Regular Updates: As your source data changes or grows, note that you will need to update your processed data accordingly. This could involve

Incremental updates for new or modified content

Full reprocessing in cases of significant changes or to maintain consistency

Load Phase: This stage involves extracting data from identified sources and bringing it into your processing environment. Key steps include the following:

- **Extraction:** Retrieving raw data from original sources
- **Initial Ingestion:** Storing data in your system's working environment
- **Validation:** Performing basic checks for completeness and integrity
- **Logging:** Recording metadata about the loaded data

The goal is to make raw data available for further processing.

Transform Phase: Then we move on to the Transform phase, where you preprocess and prepare the data for the next steps. You will be involved in tasks like cleaning the text, removing irrelevant information, or even splitting the data into smaller chunks.

Embed Phase: Next is the real game changer in RAG, which is the Embed phase. This is where you take the transformed data and convert it into a numerical representation called embeddings using text embedding models like OpenAI's text-embedding-ada-002. Embeddings are like a secret code that captures the semantic meaning of the text. By embedding your data, you enable the model to understand the relationships and similarities between different pieces of information.

Store Phase: After the Embed phase, you store the embeddings in a special data structure called a vector database or an index. You will be using a

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

vector database like Pinecone, Chroma, Milvus, or Qdrant. This allows you to quickly retrieve relevant information when you need it.

Retrieve Phase: And finally, during the Retrieve phase, you will provide a query or a prompt to the RAG model to generate the text. Using LangChain, you will generate an embedding for that question using the same embedding model. You then use the question's embedding to find the most similar chunk embeddings from your vector database. This is done by ranking the vectors based on their cosine similarity or Euclidean distance to the question's embedding. The top-ranking vectors represent the chunks most relevant to the question. The model then uses the query to retrieve the most relevant information from the stored embeddings.

Once the relevant information is retrieved, the model uses it to generate new text that incorporates the retrieved knowledge as well as its preexisting knowledge.

By following this pipeline, you can build a powerful question-answering system that leverages the best of both worlds – the language model's inherent knowledge and the ability to retrieve relevant information from external sources.

RAG Use Case Example

Let me share an example to make it more concrete. Imagine you are building a chatbot that answers questions about your company's products, such as detailed specifications, user manuals, or sales data. In such cases, you want the answers to be restricted to your products only. This is where RAG shines because it provides accurate information from your custom data and avoids any generic content.

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

Here is how you would go through the RAG process.

Load your custom data (e.g., user manuals, product specifications) into your system.

Transform the data by cleaning and preprocessing it to ensure consistency and quality.

Embed the transformed data to capture its semantic meaning. Embedding allows for precise and context-aware retrieval.

Store the embeddings in a vector database for efficient and quick retrieval.

Now, when a user asks a question like “What are the installation steps for Product X?”, the RAG model gets into action. Instead of providing a generic answer that a general LLM might generate, the RAG model will

Retrieve the most relevant information from your custom data.

Generate an accurate and context-specific response based on that data.

This ensures that the chatbot provides precise and relevant answers that are strictly related to your products. You get a well-informed and context-aware answer that combines the best of both worlds: the vast knowledge stored in external data sources and the language understanding capabilities of modern language models.

Try It Yourself

There are a number of RAG approaches and architectures like the RAG Token model and the RAG Sequence model. Each has its own strengths and use cases, and I encourage you to explore them all.

Remember, the key to a successful RAG is having high-quality, relevant data and fine-tuning the retrieval and generation components, so they work seamlessly together. It is an iterative process. Try to experiment and iterate until you find the perfect RAG technique for your specific task.

LangChain Components

To deliver the above RAG process, LangChain offers a number of components such as

- Document loaders
- Text splitters
- Text embedding models
- Vector stores
- Retrievers
- Indexes

These tools will come in handy to access the most relevant and up-to-date information when building applications such as a simple FAQ bot or a complex research assistant.

Let us discuss each one of them now.

Document Loaders

At the core of any retrieval system are document loaders. Document loaders allow you to load documents from various sources, such as HTML, PDF, or even code files. You can choose from over 100 different document loaders from LangChain to integrate with sources such as a simple .txt file, HTML from a public website, transcript of a YouTube video, PDF files from private S3 buckets, or code snippets from various repositories. You can even integrate with popular providers like AirByte and Unstructured. These documents could be stored in private S3 buckets or even on public websites.

Document Loaders in Action

Using a document loader is easy. You will start by loading data from various sources and transform them into Document objects. A document is like a neat little container that contains a piece of text along with its associated metadata.

Every loader has two key methods: “Load” and “Load and split.” The “Load” method loads documents from the configured source. The “Load and split” method takes it a step further by loading the documents and then splitting them using a text splitter of your choice.

Some document loaders also provide a “Lazy load” method. This method allows you to load documents into memory lazily, which you will find to be very helpful when dealing with large datasets. It helps you by fetching your documents on demand.

Let us say you have a basic .txt file that you want to load. Here is how you can do it:

```
from langchain_community.document_loaders import TextLoader

# Load a single text file
loader = TextLoader("/media/file1.txt")

# Load multiple text files
#loader = TextLoader(["/media/file1.txt", "/media/file2.txt"])

# Load all text files in a directory
# loader = TextLoader("media")

# Load the documents
documents = loader.load()

# Print the loaded documents
for doc in documents:
```

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

```
print(f"Content: {doc.page_content}\n")
print(f"Metadata: {doc.metadata}\n")
print("----")
```

Here is the output I got.

```
Content: this is just a test file
Metadata: {'source': '/media/file1.txt'}
---
```

Amazing, isn't it? With just three lines of code, you have successfully loaded the content of file1.txt into a list of Document objects. It is that simple!

Now, let us break it down:

1. You import the TextLoader class from the langchain_community.document_loaders module for handling plain text files.
2. You create an instance of TextLoader by passing the file path ("/media/file1.txt) as a parameter. This tells the loader where to find your precious data.
3. Finally, you call the load() method on your loader instance, which reads the file and returns a list of Document objects. Each Document contains the text content and metadata associated with the file.

Try It Yourself

Don't hesitate to experiment and try out different ones. Each loader has its own unique features and capabilities, so find the ones that best suit your needs.

Working with PDFs

Let us now look at one of the most common data formats out there, namely, PDFs.

Most of our data is stored in various formats like PDFs, CSV files, JSON, HTML, or even office documents. Wouldn't it be nice if you could ask questions and get answers based on the content of those documents? That is exactly what you are going to achieve!

First, make sure you have the necessary libraries installed. In this case, you will be using the `pypdf` library to handle PDFs. You can easily install it by running the code below. The `pypdf` library is a pure-Python library for parsing PDF documents and is used by LangChain's `PyPDFLoader` to read and extract text from PDF files:

```
!pip install pypdf langchain langchain_community
```

Next, import the necessary modules such as the `PyPDFLoader` class from the `langchain_community.document_loaders` module. The `langchain_community` package is a community-driven extension of the main `langchain` library and provides additional functionality and utilities:

```
from langchain_community.document_loaders import PyPDFLoader
```

Then create an instance of the `PyPDFLoader` class and pass the path to the PDF file as an argument. In this example, the PDF file is located at `"/media/2022 Annual Report ACME.pdf"`. Make sure you provide the correct path to your PDF file:

```
loader = PyPDFLoader("/media/2022 Annual Report ACME.pdf")
```

Next, you should call the `load_and_split()` method of the `PyPDFLoader` instance to load the PDF document and automatically split it into individual pages or sections. The resulting pages or sections are stored in the `pages` variable as a list of *Document* objects:

```
pages = loader.load_and_split()
```

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

The `pages` variable now contains a list of Document objects, where each object represents a page or section of the PDF document. You can access the text content and metadata of each page using the attributes of the Document object, such as `page_content` and `metadata`:

```
print(pages[0].page_content)
```

This will display the text content of the first page or section of the PDF document.

You can iterate over the `pages` list to access and process each page or section of the PDF document as needed.

But what if you want to access the metadata of a Document? No problem! Each Document object has a `metadata` attribute that stores a dictionary of metadata information. For instance, to view the metadata of the 11th page:

```
print(data[10].metadata)
```

If you are curious about how many pages are in the loaded PDF, you can easily find out using the `len()` function:

```
print(f"{len(pages)} pages in your data")
```

And if you want to know the number of characters on a specific page, you can do something like this:

```
print(f"{len(pages[0].page_content)} characters in the page")
```

Dealing with CSV Files

Let us look at how to load CSV files using the `CSVLoader` class from the `langchain_community.document_loaders` module.

In a CSV file, each line represents a record, and the values within each record are separated by commas. It is a simple way to organize and share data.

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

You will use `CSVLoader` to load CSV files and convert each row into a Document object.

To get started, make sure you have the `langchain_community` package installed. You can install it using `pip`:

```
! pip install langchain_community
```

Once you have the package installed, you can import the `CSVLoader` class and start loading your CSV files. Here is an example:

```
from langchain_community.document_loaders.csv_loader
import CSVLoader loader = CSVLoader(file_path='./ sample_data/
california_housing_test..csv')
data = loader.load() print(data)
```

Here, you create an instance of the `CSVLoader` class and pass the file path of our CSV file to the `file_path` parameter. Then, you call the `load()` method to load the CSV data. The loaded data is stored in the `data` variable, which contains a list of Document objects, where each object represents a row from the CSV file.

When you print the `data` variable, you will see the loaded documents, including their content and metadata. The metadata includes information like the source file path and the row number.

The `CSVLoader` allows you to customize how the CSV file is parsed. You can pass additional arguments to the `csv_args` parameter to control the delimiter, quote character, and field names. For example:

```
loader = CSVLoader(file_path='./example_data/mlb_teams_2012.
csv', csv_args={ 'delimiter': ',', 'quotechar': '"',
'fieldnames': ['MLB Team', 'Payroll in millions', 'Wins'] })
data = loader.load() print(data)
```

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

In this case, we specify the delimiter as a comma, the quote character as a double quote, and provide custom field names for the columns. This way, you can ensure that the CSV file is parsed correctly and the data is loaded as expected.

Below is the results you will get:

```
[Document(page_content='MLB Team: Team\nPayroll in millions:
"Payroll (millions)" \nWins: "Wins"', metadata={'source': './
sample_data/mlb_teams_2012.csv', 'row': 0}),
Document(page_content='MLB Team: Nationals\nPayroll in
millions: 81.34\nWins: 98', metadata={'source': './sample_data/
mlb_teams_2012.csv', 'row': 1}),
.....
```

Another cool feature of the CSVLoader is the ability to specify a source column. By default, the file path is used as the source for all documents. However, if you want to use a specific column from the CSV file as the source, you can use the `source_column` parameter:

```
loader = CSVLoader(file_path='./example_data/mlb_teams_2012.
csv', source_column="Team") data = loader.load() print(data)
```

In this example, you set the `source_column` to “Team,” which means that the value from the “Team” column will be used as the source for each document. This can be particularly useful when working with chains that answer questions using sources.

```
[Document(page_content='Team: Nationals\n"Payroll (millions)":
81.34\n"Wins": 98', metadata={'source': 'Nationals',
'row': 0}),
Document(page_content='Team: Reds\n"Payroll (millions)":
82.20\n"Wins": 97', metadata={'source': 'Reds', 'row': 1}),
.....
```

Working with JSON Files

Now, let us look at how to do this. To handle JSON files effectively, you can use Python's built-in `json` module along with `pathlib` to manage file paths.

First, make sure you have the necessary dependencies installed. Then, you must load the JSON data from a file using the import statement below. In this code snippet, you use `pathlib` to handle the file path and read the contents of the JSON file as a string. Then, you use `json.loads()` to parse the string into a Python dictionary:

```
import json
from pathlib import Path
file_path = './sample_data/products.json'
data = json.loads(Path(file_path).read_text())
```

Check the resources section in the downloads section for the product.json file.

Here, you are using the `json` module to load the JSON data from a file. You do that by specifying the file path and use `Path(file_path).read_text()` to read the contents of the file as a string. Then, you pass that string to `json.loads()` to parse it into a Python dictionary.

Using the `JSONLoader` to extract specific data:

```
from langchain_community.document_loaders import JSONLoader
loader = JSONLoader( file_path='./sample_data/products.json',
jq_schema='.messages[].content', text_content=False)
data = loader.load()
```

You will notice that `JSONLoader` is a handy tool that allows you to extract specific data from a JSON file using a jq schema. In this example, you specify the file path and provide a jq schema (`.messages[].content`) to extract the values under the content field within the messages key of the JSON data. The `text_content=False` parameter indicates that you don't want to load the entire JSON file as text content.

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

Here is the output:

```
Collecting jq
  Downloading jq-1.7.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (657 kB)
-----
657.6/657.6 kB 7.6 MB/s eta 0:00:00

Installing collected packages: jq
Successfully installed jq-1.7.0
{'company': 'Acme Inc.',
 'sales_data': [{'products': [{'name': 'Product A', 'revenue': 200000},
                              {'name': 'Product B', 'revenue': 150000},
                              {'name': 'Product C', 'revenue': 150000}],
                 'region': 'North America',
                 'total_revenue': 500000,
                 'year': 2021},
                {'products': [{'name': 'Product A', 'revenue': 100000},
                              {'name': 'Product B', 'revenue': 120000},
                              {'name': 'Product C', 'revenue': 80000}],
                 'region': 'Europe',
                 'total_revenue': 300000,
                 'year': 2021},
                {'products': [{'name': 'Product A', 'revenue': 150000},
                              {'name': 'Product B', 'revenue': 100000},
                              {'name': 'Product C', 'revenue': 150000}],
                 'region': 'Asia',
                 'total_revenue': 400000,
                 'year': 2021}]}
```

Let us review the code to handle JSON Lines files below:

```
loader = JSONLoader( file_path='./sample_data/products.jsonl',
jq_schema='.content', text_content=False, json_lines=True)
```

If you are working with a JSON Lines file, where each line represents a valid JSON object, you can set `json_lines=True` in the `JSONLoader` constructor. This tells the loader to treat each line as a separate JSON object. You can then specify the `jq_schema` to extract the desired data from each JSON object.

Here is the output:

```
'400000', 'product_name': 'Product C', 'product_revenue': 150000}')
[Document(page_content='', metadata={'source': '/content/sample_data/products.jsonl', 'seq_num': 1}),
 Document(page_content='', metadata={'source': '/content/sample_data/products.jsonl', 'seq_num': 2}),
 Document(page_content='', metadata={'source': '/content/sample_data/products.jsonl', 'seq_num': 3}),
 Document(page_content='', metadata={'source': '/content/sample_data/products.jsonl', 'seq_num': 4}),
 Document(page_content='', metadata={'source': '/content/sample_data/products.jsonl', 'seq_num': 5}),
 Document(page_content='', metadata={'source': '/content/sample_data/products.jsonl', 'seq_num': 6}),
 Document(page_content='', metadata={'source': '/content/sample_data/products.jsonl', 'seq_num': 7}),
 Document(page_content='', metadata={'source': '/content/sample_data/products.jsonl', 'seq_num': 8}),
 Document(page_content='', metadata={'source': '/content/sample_data/products.jsonl', 'seq_num': 9})]
```

Text Splitters

We already discussed that when dealing with large documents, we may need to transform the documents to better suit the application. For example, when you have a lengthy document that exceeds your model's context window, you will need to split it into smaller and semantically meaningful chunks. That is where text splitters come in by keeping related pieces of text together, which ensures that the model can understand the context and provide accurate results.

You can choose from a variety of text splitters based on your specific needs. Each splitter divides text in its own way and some will even add metadata to provide extra information about the chunks. Below are some of the popular options:

1. **RecursiveCharacterTextSplitter:** You can use this splitter to recursively chop up your text based on a list of characters you define. It will help to keep related bits of text close to each other.
2. **HtmlTextSplitter:** This splitter will be your go-to when you are working with HTML documents. It splits text based on HTML-specific characters and also adds metadata about the origin of each chunk.
3. **MarkdownTextSplitter:** This is similar to the HTML splitter but is designed for Markdown documents. It splits text based on Markdown-specific characters and includes metadata about the source of the chunk.
4. **TokenTextSplitter:** You will use this when splitting text based on the number of tokens. You can decide how you want to chunk your text by using the different ways available to measure tokens.

Fully Working Code Example for Text Splitting

To get started, you will need to install the langchain-text-splitters package:

```
pip install langchain-text-splitters
```

Once you have the package installed, you can easily create a text splitter instance and start chunking your text. Here is an example using the `CharacterTextSplitter`:

```
# This is a long document we can split up.
with open("./sample_data/The Art of Money Getting.txt") as f:
    art_of_money_getting = f.read()
```

In the code above, you are opening a text file named “The Art of Money Getting.txt” located in the “./sample_data” directory. You use the `with` statement to ensure that the file is properly closed after you are done reading from it.

You use the `read()` method to read the entire contents of the file and store it in the `art_of_money_getting` variable. This variable now holds the complete text of the document.

```
from langchain_text_splitters import CharacterTextSplitter
```

Here, you will import the `CharacterTextSplitter` class from the `langchain_text_splitters` package. You use this class to split the text into smaller chunks based on a specified character or set of characters.

```
text_splitter = CharacterTextSplitter( separator="\n\n",
chunk_size=1000, chunk_overlap=200, length_function=len,
is_separator_regex=False, )
```

In this block of code, you are creating an instance of the `CharacterTextSplitter` class and configuring it with the following parameters:

- `separator="\n\n"`: You specify the separator as two newline characters (`\n\n`). This means that the text will be split whenever two consecutive newline characters are encountered.
- `chunk_size=1000`: You set the maximum size of each chunk to 1000 characters. If a chunk exceeds this size, it will be further split.
- `chunk_overlap=200`: You specify an overlap of 200 characters between consecutive chunks. This overlap helps you to maintain context between the chunks.
- `length_function=len`: You use the built-in `len` function to calculate the length of each chunk in characters.
- `is_separator_regex=False`: You indicate that the separator is not a regular expression but a simple string.

```
documents = text_splitter.create_documents([art_of_money_getting])
```

Here, you use the `create_documents` method of the `text_splitter` instance to split the `art_of_money_getting` text into smaller chunks. The method takes a list of texts as input (in this case, you provide a single text) and returns a list of `Document` objects. Each `Document` object represents a chunk of text.

```
print(documents[4])
```

Finally, you print the fifth `Document` object from the `documents` list (index 4). This will display the content and metadata of the fifth chunk of text.

The output will look something like this:

```
page_content='...' metadata={}
```

The `page_content` attribute contains the actual text content of the chunk, and the `metadata` attribute is an empty dictionary since we didn't provide any metadata in this example.

Overall, this code demonstrates how to use the `CharacterTextSplitter` from the `langchain_text_splitters` package to split a long document into smaller chunks based on a specified separator (in this case, two newline characters). The resulting chunks are stored as `Document` objects in the `documents` list, which you can access and manipulate as needed.

In order to test how your text splitter is working, you can use the `Chunkviz` tool created by Greg Kamradt that allows you to visualize how your text is being split. Using it, you can fine-tune your splitting parameters.

Note that text splitting is just one example of the transformations you can apply to your documents before feeding the text to an LLM. Feel free to choose from a wide range of document transformer integrations with third-party tools that `LangChain` provides.

Try It Yourself

You must strike the right balance between chunk size and semantic coherence for successful text splitting. I encourage you to experiment with different splitters and parameters to find the one that works best for your specific use case.

Recursive Splitting

Next is recursive splitting. The `RecursiveCharacterTextSplitter` also splits the text based on a specified separator character or a set of characters. However, it uses a recursive approach to split the text into chunks. First, it starts by splitting the entire text based on the separator and creates initial chunks. If any of the resulting chunks exceed the specified `chunk_size`, the splitter recursively applies the splitting process to those chunks. And this recursive splitting continues until all the chunks are within the desired `chunk_size` limit. The `chunk_overlap` parameter is used to maintain context between the recursively split chunks.

This recursive approach ensures that the chunks are split more evenly, especially when you are dealing with long paragraphs or sections that exceed the `chunk_size`.

The difference between the `CharacterTextSplitter` and `RecursiveCharacterTextSplitter` is subtle, and you will find it particularly useful when you want to split the text into smaller, more manageable chunks while also preserving the logical structure of the content.

Here is an example to illustrate the difference:

```
from langchain_text_splitters import CharacterTextSplitter,
RecursiveCharacterTextSplitter
text = "This is a sample text. It consists of multiple
sentences. Some sentences are longer than others. We will split
this text into chunks."
# CharacterTextSplitter
text_splitter = CharacterTextSplitter(separator=". ", chunk_
size=30, chunk_overlap=5)
documents = text_splitter.create_documents([text]) print("Chara
cterTextSplitter:")
for doc in documents:
print(doc.page_content)
```

```
print("---")
# RecursiveCharacterTextSplitter
recursive_text_splitter = RecursiveCharacterTextSplitter(separa
tors=[". ", "! ", "? "], chunk_size=30, chunk_overlap=5)
recursive_documents = recursive_text_splitter.create_
documents([text]) print("RecursiveCharacterTextSplitter:")
for doc in recursive_documents:
print(doc.page_content)
print("---")
```

Below is the output:

```
CharacterTextSplitter:
This is a sample text.
---
It consists of multiple sentences.
---
Some sentences are longer than others.
---
We will split this text into chunks.
---
RecursiveCharacterTextSplitter:
This is a sample text.
---
It consists of multiple sentences.
---
Some sentences are longer than
---
others. We will split this text
---
into chunks.
---
```

As you can see, the `CharacterTextSplitter` splits the text based on the specified separator (“.” in this case) and creates chunks accordingly. On the other hand, the `RecursiveCharacterTextSplitter` recursively splits the chunks for those chunks that exceed the `chunk_size` limit, resulting in more evenly distributed chunks.

Ultimately, you will have to choose the appropriate text splitter based on your specific requirements. If you have long paragraphs or sections that need to be split into smaller chunks while preserving the logical structure, the `RecursiveCharacterTextSplitter` might be a better choice. Otherwise, the `CharacterTextSplitter` is a simpler and straightforward option for most text splitting tasks.

CodeTextSplitter

You can also use the `CodeTextSplitter` when working with source code files or code snippets. It helps you to split code into meaningful chunks while factoring in the programming language’s structure and syntax as well. Here are some examples of where you can use it:

1. **Code Analysis and Understanding:** When you are dealing with a large amount of code, the `CodeTextSplitter` can help you break it down into manageable pieces so that you can analyze and understand the code better.
2. **Code Search and Retrieval:** If you need to quickly find specific code snippets or functions, you can use the `CodeTextSplitter` to create an index of code chunks. You can then find the relevant code in no time based on keywords or criteria.

3. **Code Summarization and Documentation:**

You can generate documentation for your code easily with the `CodeTextSplitter` by splitting your code into logical units and then creating targeted documentation. It will help you and others to understand the purpose and usage of different parts of the codebase.

4. **Code Comparison and Diff:** You can easily compare different versions of your code by using `CodeTextSplitter` to split the code into comparable chunks and identify the changes between versions.

5. **Code Formatting and Style Checking:** You can use the `CodeTextSplitter` as a preprocessing step for formatting and style checking tools. It helps you apply formatting rules and style guidelines to specific parts of your code and ensure readability across your codebase.

Here is an example of how you can use the `CodeTextSplitter` to split a Python code snippet:

```
from langchain_text_splitters import CodeTextSplitter

code_snippet = '''
def greet(name):
    print(f"Hello, {name}!")

def main():
    name = input("Enter your name: ")
    greet(name)

if __name__ == "__main__":
    main()
'''
```

```
code_splitter = CodeTextSplitter(language="python", chunk_
size=50, chunk_overlap=0)
code_chunks = code_splitter.create_documents([code_snippet])

for chunk in code_chunks:
    print(chunk.page_content)
    print("---")
And here is the output:
def greet(name):
    print(f"Hello, {name}!")

---
def main():
    name = input("Enter your name: ")
    greet(name)

---
if __name__ == "__main__":
    main()

---
```

As you can see, the `CodeTextSplitter` splits the Python code snippet into logical units based on functions and code blocks. You can then easily analyze, understand, and work with your code in smaller segments.

Splitting by Token

In the real world, sometimes you may be required to split text into chunks while factoring in the token limits for the LLMs in use. Let us say you are required to process your company's knowledge base articles and feed them into your company chatbot's training pipeline. However, you want to ensure that each chunk of text fits within the token limit of the language model to avoid any issues during training.

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

First, you must install the necessary `text_splitters` and `tiktoken` packages:

```
pip install --upgrade --quiet langchain-text_splitters tiktoken
```

Let us say you have a knowledge base article stored in a file called `returns_policy.txt`. You can split it into chunks while keeping an eye on the token limit as shown below:

```
from langchain_text_splitters import
RecursiveCharacterTextSplitter

# Load the knowledge base article
with open("returns_policy.txt") as f:
    returns_policy = f.read()

# Create a RecursiveCharacterTextSplitter instance
text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_
encoder(
    model_name="gpt-3.5-turbo", # Specify the language model
    you're using
    chunk_size=500, # Set the desired chunk size in tokens
    chunk_overlap=50, # Allow some overlap between chunks
    for context
)

# Split the article into chunks
chunks = text_splitter.split_text(returns_policy)

# Print the number of chunks generated
print(f"Number of chunks: {len(chunks)}")

# Print the first chunk
print("First chunk:")
print(chunks[0])
```

In this example, you will use the `RecursiveCharacterTextSplitter` with the `from_tiktoken_encoder()` method. You specify the language model you are using (“gpt-3.5-turbo” in this case) and set the desired `chunk_size` to 500 tokens. You also add a `chunk_overlap` of 50 tokens to provide some context between chunks.

You can then use the `split_text()` method that takes care of splitting the `returns_policy` article into chunks based on the specified parameters. It ensures that each chunk is within the token limit and recursively splits any oversized chunks until they fit.

After splitting, you print the number of chunks generated and display the first chunk to give you a glimpse of what the output looks like.

By using this approach, you can confidently process your company’s knowledge base articles and prepare them to train your customer support chatbot. The `RecursiveCharacterTextSplitter` takes care of the heavy lifting by ensuring that each chunk is compatible with the language model’s token limit.

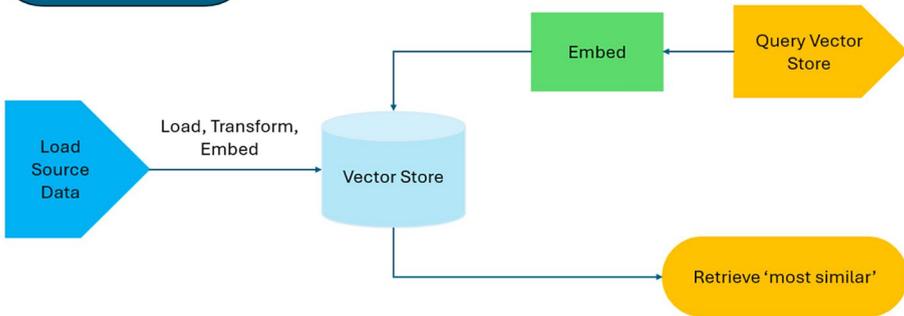
Try It Yourself

The choice of language model and the `chunk_size` and `chunk_overlap` parameters can be adjusted based on your specific requirements. Go ahead and experiment with different values to find the optimal balance between chunk size and context preservation.

Vector Stores

In this section, let us discuss vector stores in greater detail, what they are and why they are important.

Vector Stores



Vector stores come into play when you have a bunch of data that you want to search through quickly and efficiently. They store your data in a special format called vectors, which makes it easy to find similar items.

Now, let us break down the process of how to use vector stores, and then we can make it real with an example:

1. **Load Source Data:** Your first step is to gather all your data and load it into the vector store so that you can search through it later. You can load data from various sources, such as text files, databases, or even web pages.
2. **Query Vector Store:** Once your data is loaded, you can use a query to ask the vector store to find the most similar items to your query. For example, you can ask the vector store to find you the items that are most like your query, such as a search term or a question. The vector store will then, behind the scenes, convert your query into a vector representation. It then compares this query vector with all the vectors in the store and retrieves the most similar ones.

3. **Retrieve “Most Similar”:** The vector store returns the items that are most similar to your query. It is like getting a list of the best matches. You can specify how many items you want to retrieve, and the vector store will give you the top results. These retrieved items are the ones that have the closest vector representations to your query vector. If you really think about it, it is almost like finding the needle in the haystack, but the vector store makes it effortless.

Text Embedding Models

Continuing our discussion on vectors, let us discuss text embeddings a bit deeper. A key part of working with vector stores is creating the vectors using embeddings so that you can search later. Whether you are building a semantic search engine, a recommendation system, or any application that requires understanding the meaning behind words, text embeddings would be your answer. Text embeddings will help you find the documents that are most similar to a specific query from a bunch of documents. You could try comparing the words directly, but that is like searching for a needle in a haystack. Instead, text embeddings come to the rescue!

Keyword Search vs. Text Embeddings

It is worth noting that sometimes plain keyword search can be more effective for straightforward queries. Recent recommendations suggest combining keyword search with embeddings to leverage the strengths of both approaches. This hybrid method can provide the precision of keyword search along with the depth of semantic understanding from embeddings.

Here is how you can implement this combined approach.

Keyword Search

Start by using keyword search to filter the documents that contain the relevant terms. This step can be helpful when handling large datasets and quickly identifying potential matches.

Text Embeddings

Apply text embeddings to the filtered documents to capture their semantic meaning. Use these embeddings to rank the filtered results based on their relevance to the query.

This combination ensures that you benefit from the speed and precision of keyword search while also leveraging the semantic depth provided by embeddings.

You can think of text embeddings as magical representations of text in the form of vectors. They capture the semantic meaning of words and sentences, which helps you to compare them in a meaningful way. It is like assigning coordinates to each piece of text in a high-dimensional space, where similar texts are closer together.

You will be using the Embeddings class in LangChain when working with text embedding models. It provides a standard interface for interacting with various embedding model providers, such as OpenAI, Cohere, and Hugging Face. You can choose from over 25 different embedding providers and methods, ranging from open source to proprietary APIs. No matter which provider you choose, the Embeddings class will help you. The best part is that you can use this interface to easily switch between models, so it is easy to find the one that suits your needs.

Code Walk-Through for Text Embeddings

Let us walk through a real-life example of how to use text embeddings to extract insights from a list of customer reviews for a product and get valuable feedback and sentiment from your customers.

Below is a fully working end-to-end code.

You start by installing the required packages (langchain and langchain-openai) and importing the necessary modules:

```
# Install the required packages
!pip install langchain
!pip install langchain-openai

# Import the necessary modules
from langchain_openai import OpenAIEmbeddings
import os
```

You set the OpenAI API key using the environment variable `OPENAI_API_KEY`. Make sure to replace “your_api_key_here” with your actual OpenAI API key:

```
# Set the OpenAI API key
os.environ["OPENAI_API_KEY"] = "your_api_key_here"
```

You initialize an instance of the `OpenAIEmbeddings` class called `embeddings_model`, which will be our embedding model for the rest of the code:

```
# Initialize the OpenAIEmbeddings class
embeddings_model = OpenAIEmbeddings()
```

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

You define a list of customer reviews called reviews. Each review is a string containing feedback about a product or service:

```
# Define a list of customer reviews to embed
reviews = [
    "I absolutely love this product! It has made my life so
    much easier and I can't imagine going back to the way
    things were before.",
    "I was a bit skeptical at first, but after using this
    service for a few weeks, I'm really impressed. The customer
    support is top-notch and the features are exactly what I
    needed.",
    "I've been using this software for a while now and it has
    definitely improved my productivity. The user interface
    is intuitive and the integration with other tools is
    seamless.",
    "I had a great experience with this company. They
    delivered the project on time and the quality exceeded my
    expectations. I would definitely work with them again.",
    "I'm not entirely satisfied with this product. While it has
    some useful features, I found it lacking in certain areas
    and the price seems a bit steep for what you get."
]
```

You use the `embed_documents` method of the `embeddings_model` to embed the list of reviews. The resulting embeddings are stored in the `embeddings` variable:

```
# Embed the list of reviews using the embed_documents method
embeddings = embeddings_model.embed_documents(reviews)
```

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

You print the number of embeddings (which should be equal to the number of reviews) and the length of each embedding:

```
# Print the length of the embeddings and the length of each
embedding
print(f"Number of embeddings: {len(embeddings)}")
print(f"Length of each embedding: {len(embeddings[0])}")
```

You define a query text called `query_text` that asks about the positive aspects mentioned in the customer reviews:

```
# Define a query text to embed
query_text = "What are the positive aspects mentioned in the
customer reviews?"
```

You use the `embed_query` method of the `embeddings_model` to embed the query text. The resulting embedding is stored in the `embedded_query` variable:

```
# Embed the query text using the embed_query method
embedded_query = embeddings_model.embed_query(query_text)
```

You print the length of the embedded query and the first five elements of the embedded query:

```
# Print the length of the embedded query
print(f"Length of the embedded query: {len(embedded_query)}")

# Print the first 5 elements of the embedded query
print(f"First 5 elements of the embedded query: {embedded_
query[:5]}")
```

You can embed the customer reviews and use the query text to perform various analyses and tasks, such as the following:

- Identifying common themes or topics mentioned in the reviews
- Clustering similar reviews together based on their embeddings
- Searching for reviews that are most relevant to a specific query or topic
- Sentiment analysis to determine the overall sentiment expressed in the reviews
- Comparing the embeddings of different reviews to find similarities or differences

In subsequent sections, we will see how to use these queries to store in a vector store and then get the results for these queries.

Caching the Embeddings

Let us explore the topic of caching the embeddings which is important because it can save you time, money, and computational resources. Caching in the context of embeddings means storing or temporarily saving the computed embeddings to avoid the need to recalculate them every time you need them.

You can use the `CacheBackedEmbeddings` class for caching. Think of it as a wrapper that goes around your embedder. It caches the embeddings in a key-value store using the hashed text as the key, which can be used to store and retrieve embeddings efficiently.

You should start by creating a `CacheBackedEmbeddings` instance and call the `from_bytes_store` method. You will need to provide a few important pieces of information such as the following:

1. `underlying_embedder`: This is the embedder you want to use for embedding your text.
2. `document_embedding_cache`: You will need a `ByteStore` to store your cached document embeddings. Think of it as a secure vault where your embeddings will be kept safe and sound.
3. `batch_size` (optional): If you want to embed multiple documents in batches, you can specify the number of documents to embed between store updates. It is like telling your caching machine how many items to process at once.
4. `namespace` (optional): To avoid collisions with other caches, you can provide a namespace for your document cache. It is a unique name tag to avoid mixing caches with each other.

Now, here is an important tip: make sure to set the `namespace` parameter to avoid any collisions if you are using different embedding models. You don't want your caches to get confused and start mixing up embeddings from different models!

Code Walk-Through for Cache Embeddings

Let us see an example of how you can use `CacheBackedEmbeddings` with a vector store. You will use the local file system for storing embeddings and the FAISS vector store for retrieval.

Below is the full, end-to-end working code.

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

First, you must install or upgrade the `langchain-openai` and `faiss-cpu` packages, which are required for your code to run:

```
# Install the necessary packages
!pip install --upgrade --quiet langchain-openai faiss-cpu
```

You import the `LocalFileStore` class from the `langchain.storage` module for storing embeddings locally:

```
# Import the required modules
from langchain.storage import LocalFileStore
```

Then you import the `TextLoader` class from the `langchain.document_loaders` module for loading text documents:

```
from langchain.document_loaders import TextLoader
```

You import the `FAISS` class from the `langchain.vectorstores` module to implement a vector store using the `FAISS` library:

```
from langchain.vectorstores import FAISS
```

Why FAISS-CPU? `FAISS` (Facebook AI Similarity Search) is a library developed by Facebook AI Research for efficient similarity search and clustering of dense vectors. Let us see why `faiss-cpu` is essential in our context:

Efficient Vector Storage and Retrieval: `FAISS` is designed to handle large-scale vector search efficiently. It provides optimized algorithms for storing and retrieving high-dimensional vectors which makes it ideal for applications involving large datasets.

Scalability: FAISS can scale to millions or even billions of vectors and ensure that your application remains performant even as the dataset grows.

Versatility: FAISS supports various indexing strategies, and hence you can choose the best one for your specific use case. For example, you can choose between fast approximate nearest neighbor search and exact search.

CPU-Based: The faiss-cpu package is optimized for CPU usage and hence is a good alternative for environments where GPU resources are limited or unavailable. This ensures that you can deploy and run your application on a wide range of hardware configurations.

As you can see, these advantages can help you to build a robust vector search system.

Below, you are importing the `OpenAIEmbeddings` class from the `langchain_openai` module for generating embeddings using OpenAI's API:

```
from langchain_openai import OpenAIEmbeddings
```

Then import the `CharacterTextSplitter` class from the `langchain.text_splitter` module, which is used for splitting text into chunks:

```
from langchain.text_splitter import CharacterTextSplitter
```

Lastly, you must import the `CacheBackedEmbeddings` class from the `langchain.embeddings` module which is used for caching embeddings:

```
from langchain.embeddings import CacheBackedEmbeddings
```

```
# Set up the OpenAI API key
```

```
import os
```

```
os.environ["OPENAI_API_KEY"] = "Your OpenAI key"
```

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

In order to generate embeddings, you must create an instance of the `OpenAIEmbeddings` class:

```
# Create an instance of the OpenAI embeddings
underlying_embeddings = OpenAIEmbeddings()
```

Then you must create an instance of the `LocalFileStore` class and specify the directory where the cached embeddings will be stored:

```
# Create a local file store for caching embeddings
store = LocalFileStore("./cache/")
```

This is the most important step, where you create an instance of the `CacheBackedEmbeddings` class and wrap up the underlying embeddings that you previously created with `OpenAI`. You store the embedding in the local file store for caching. You use the `namespace` parameter and point it to the model name of the underlying embeddings:

```
# Create a CacheBackedEmbeddings instance
cached_embedder = CacheBackedEmbeddings.from_bytes_store(
    underlying_embeddings, store, namespace=underlying_
    embeddings.model
)
```

Building the Information Retrieval System

Now you must load the document:

```
# Load the document
raw_documents = TextLoader("./sample_data/The Art of Money
Getting.txt").load()
```

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

Once loaded, you split the document into chunks using the text splitter:

```
# Split the document into chunks
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_
overlap=0)
documents = text_splitter.split_documents(raw_documents)
```

Here, you are creating the vector store using the FAISS vector store and pass in the document chunks and the cached embedder:

```
# Create the vector store
db = FAISS.from_documents(documents, cached_embedder)
```

Below, you are defining the search query:

```
# Perform a similarity search
query = "What advice does the author give about getting rich?"
```

You then perform a similarity search using the query and retrieve the top three most relevant chunks:

```
results = db.similarity_search(query, k=3)
```

Finally, you will iterate over the search results and print each result, including its content and a separator line:

```
# Print the search results
print("Search Results:")
for i, doc in enumerate(results):
    print(f"Result {i+1}:")
    print(doc.page_content)
    print("---")
```

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

Here is a sample of the results I got:

```
WARNING:LangChain_text_splitters.Base.Created a chunk of size 1074, which is longer than
Search Results:
Result 1:
There is no greater mistake than when a young man believes he will
succeed with borrowed money. Why? Because every man's experience
coincides with that of Mr. Astor, who said, "it was more difficult for
him to accumulate his first thousand dollars, than all the succeeding
millions that made up his colossal fortune." Money is good for nothing
unless you know the value of it by experience. Give a boy twenty
thousand dollars and put him in business, and the chances are that he
will lose every dollar of it before he is a year older. Like buying a
ticket in the lottery; and drawing a prize, it is "easy come, easy go."
He does not know the value of it; nothing is worth anything, unless
it costs effort. Without self-denial and economy; patience and
perseverance, and commencing with capital which you have not earned, you
are not sure to succeed in accumulating. Young men, instead of "waiting
for dead men's shoes," should be up and doing, for there is no class of
persons who are so unaccommodating in regard to dying as these rich old
people, and it is fortunate for the expectant heirs that it is so. Nine
out of ten of the rich men of our country to-day, started out in life
as poor boys, with determined wills, industry, perseverance, economy and
good habits. They went on gradually. made their own money and saved it:
```

Congratulations! You have successfully used LangChain and OpenAI embeddings to load a document, split it into chunks, generate embeddings, create a vector store, and perform a similarity search to find relevant information based on a given query.

Try It Yourself!

Remember, caching embeddings is a powerful technique that can save you time and computational resources. So go ahead, experiment with different embedders, storage mechanisms, and namespaces. Have fun exploring the world of cached embeddings and see how they can streamline your development process.

Calling the Vector Store Asynchronously

Let us discuss why calling vector stores asynchronously is helpful. You see, vector stores are usually run as separate services that require some input/output (IO) operations. If you were to call these operations synchronously, you would be wasting precious time waiting for responses from external services.

This is where async operations can be of help because you can let your code continue executing other tasks while waiting for the vector store to respond. The good news is that you can call all the methods asynchronously by simply prefixing a.

First, you will need to install Qdrant, a vector store that fully supports async operations. You will have to install the qdrant-client package. You can do that by running

```
!pip install qdrant-client
```

Once you have the package installed, you can import the Qdrant class from langchain_community.vectorstores.

```
from langchain_community.vectorstores import Qdrant
```

Then, you must create a vector store asynchronously by calling the `afrom_documents` method:

```
db = await Qdrant.afrom_documents(documents, embeddings,  
"http://localhost:6333")
```

Here, `documents` is your collection of documents, `embeddings` is the embedding model you are using, and `"http://localhost:6333"` is the URL of your Qdrant server. Note that you can install the Qdrant server on your local machine using the docker statement:

```
docker run -p 6333:6333 qdrant/qdrant
```

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

You need to have docker installed for this command to run. This command will pull the Qdrant Docker image and start a Qdrant server on your local machine, exposing it on port 6333. You can also use a hosted Qdrant server provided by Qdrant cloud or other cloud providers.

With your vector store created, you can perform similarity searches asynchronously. Let us say you have a query and want to find the most similar documents. Here is how you can do it:

```
query = "What advice did the author give about money" docs =  
await db.asimilarity_search(query) print(docs[0].page_content)
```

The `asimilarity_search` method takes your query and returns a list of the most similar documents. You can access the content of the first document using `docs[0].page_content`.

You can also perform similarity searches using vector embeddings directly. Check this out:

```
embedding_vector = embeddings.embed_query(query) docs = await  
db.asimilarity_search_by_vector(embedding_vector)
```

In this case, you first embed your query using the `embed_query` method of your embedding model. Then, you pass the resulting `embedding_vector` to the `asimilarity_search_by_vector` method to find the most similar documents.

Retrievers

We have already discussed how to retrieve answers to a query from a vector store. Let us discuss retrievers in greater detail.

You can use a retriever to pass in an unstructured query and get back a list of relevant documents as output. It helps you to quickly fetch the information you need based on your question or query.

You can use a number of retrievers from LangChain based on their unique characteristics and use cases, so understanding them will help you choose the best one for your specific needs:

1. **Vectorstore Retriever:** If you are just getting started and looking for something quick and easy, the Vectorstore Retriever is your go-to choice. It is the simplest method using which you will create embeddings for each piece of text, which makes it easy to find similar ones.
2. **ParentDocument Retriever:** You can use ParentDocument Retriever when you have documents with lots of smaller, distinct pieces of information that are indexed separately but can be retrieved together. It indexes multiple chunks for each document and finds the most similar chunks based on their embeddings. What is really helpful is that instead of returning individual chunks, it retrieves the entire parent document for a more comprehensive result.
3. **Multi-vector Retriever:** Sometimes, you might want to extract additional information from documents that you think is more relevant to index than the text itself. The Multi-vector Retriever allows you to create multiple vectors for each document, giving you the flexibility to capture different aspects of the content. For example, you could create vectors just based on summaries or hypothetical questions related to the document.

4. **Self-Query Retriever:** Have you ever encountered situations where users ask questions that are better answered by fetching documents based on metadata rather than text similarity? The Self-Query Retriever uses an LLM (language model) to transform user input into two things: a string to look up semantically and a metadata filter to apply. This is incredibly useful when questions are more about the metadata of documents rather than their content.
5. **Contextual Compression Retriever:** Sometimes, retrieved documents can contain a lot of irrelevant information that distracts the LLM. The Contextual Compression Retriever comes to your rescue by adding a post-processing step to extract only the most relevant information from the retrieved documents. It can be done using embeddings or an LLM to ensure that your LLM stays focused on what matters most.

LangChain offers several other advanced retrieval types, such as Time-Weighted Vectorstore Retriever, Multi-query Retriever, Ensemble Retriever, and Long-Context Reorder Retriever. Each of these has its own unique strengths and use cases, allowing you to tailor your retrieval process to your specific needs.

Code Walk-Through for Information Retrieval

To give you a concrete example, let us say you want to use the Vectorstore Retriever. Here is a simple code snippet:

```
from langchain.retrievers import VectorstoreRetriever
# Assuming you have a vectorstore called 'documents_
vectorstore'
retriever = VectorstoreRetriever(documents_vectorstore)
query = "What are the main benefits of using a RAG?"
relevant_docs = retriever.get_relevant_documents(query)
for doc in relevant_docs:
    print(doc.page_content)
```

In this example, you create an instance of the `VectorstoreRetriever` by passing in your `vectorstore`. Then, you define a query and use the `get_relevant_documents` method to retrieve the most relevant documents based on your query. Finally, you iterate over the retrieved documents and print their content.

Try It Yourself

Remember, choosing the right retriever depends on your specific use case and the nature of your data. Go ahead and experiment and explore different options to find the one that works best for you.

Indexing

In this section, let us discuss the power of indexing to manage our documents. The LangChain indexing API allows you to load and keep your documents in sync with a vector store and avoids duplicating content, rewriting unchanged documents, and recomputing embeddings unnecessarily. It is all about saving you time and money while improving your vector search results.

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

Under the hood, the indexing API uses a RecordManager, which keeps track of document writes into the vector store by computing hashes for each document and storing some key information such as the following:

- The document hash (a unique fingerprint of the page content and metadata)
- The write time (when the document was added or updated)
- The source ID (a way to identify the original source of the document)

You can also use the deletion modes offered by the indexing API to control how existing documents are handled when new ones are indexed. You can choose from

1. **None:** This mode gives you the freedom to manually manage your content as the API doesn't do any automatic cleanup.
2. **Incremental:** This mode continuously cleans up previous versions of content if the source document or derived documents have changed.
3. **Full:** This mode does a thorough cleanup at the end of the indexing process by removing any documents that are no longer present in the current indexing batch.

These deletion modes help you to ensure your vector store stays lean.

Key Takeaways

In this chapter, you explored Retrieval-Augmented Generation (RAG), a powerful technique for enhancing large language models (LLMs) by incorporating additional context through retrieval.

Here is what we learned.

Understanding RAG: We defined RAG and understood its importance for enhancing LLMs, especially in cases where up-to-date or domain-specific knowledge is crucial (OpenAI Cookbook) (Pinecone).

We saw how RAG can reduce hallucinations, fact-check, provide domain-specific knowledge, and enhance LLM responses.

RAG Architecture: We examined the two main components of RAG – indexing and retrieval and generation.

The indexing component involves loading, splitting, and storing data, while the retrieval and generation component involves retrieving relevant data and generating an answer.

Implementing RAG: We explored practical implementations of RAG using LangChain, Pinecone, and OpenAI.

We learned how to set up a knowledge base, implement retrieval, and develop a question-answering application using RAG.

Review Questions

1. What is the primary purpose of Retrieval-Augmented Generation (RAG)?
 - A. To improve the speed of LLM responses
 - B. To combine retrieval-based methods with large language models to enhance the accuracy and relevance of generated responses

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

- C. To reduce the size of LLM models
 - D. To increase the training time of LLM models
2. Which of the following components are part of the RAG architecture?
- A. Indexing and retrieval and generation
 - B. Tokenization and normalization
 - C. Encryption and decryption
 - D. Caching and logging
3. What does the Embed phase in the RAG process involve?
- A. Cleaning the text and removing irrelevant information
 - B. Converting transformed data into a numerical representation called embeddings
 - C. Fetching data from the source and formatting it for processing
 - D. Storing embeddings in a database
4. Which LangChain component is responsible for splitting text into smaller chunks?
- A. Document loader
 - B. Text splitter
 - C. Vector store
 - D. Retriever

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

5. What is the purpose of using a `CacheBackedEmbeddings` class?
 - A. To store embeddings on a remote server
 - B. To cache embeddings and avoid recalculating them
 - C. To convert text into embeddings
 - D. To store document metadata
6. Which vector store is mentioned as supporting asynchronous operations?
 - A. Chroma
 - B. Milvus
 - C. Qdrant
 - D. Pinecone
7. What does the Self-Query Retriever use to transform user input?
 - A. A regular expression
 - B. A machine learning model
 - C. A predefined set of keywords
 - D. A language model
8. In the context of LangChain, what is an embedding?
 - A. A piece of text data
 - B. A numerical representation of text
 - C. A storage format for documents
 - D. A method for splitting text

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

9. Which of the following is NOT a deletion mode in the LangChain indexing API?
 - A. None
 - B. Incremental
 - C. Full
 - D. Partial

10. What is the benefit of using text embeddings in an information retrieval system?
 - A. They increase the storage capacity of databases.
 - B. They capture the semantic meaning of text for comparison and retrieval.
 - C. They reduce the complexity of text processing.
 - D. They simplify the tokenization process.

Answers

1. B. To combine retrieval-based methods with large language models to enhance the accuracy and relevance of generated responses.
2. A. Indexing and retrieval and generation
3. B. Converting transformed data into a numerical representation called embeddings
4. B. Text splitter
5. B. To cache embeddings and avoid recalculating them
6. C. Qdrant

7. D. A language model
8. B. A numerical representation of text
9. D. Partial
10. B. They capture the semantic meaning of text for comparison and retrieval.

Glossary

This glossary provides definitions for key technical terms related to Retrieval-Augmented Generation (RAG) and information retrieval systems as discussed in this chapter.

CacheBackedEmbeddings: A wrapper around an embedder that caches the embeddings in a key-value store, using the hashed text as the key to store and retrieve embeddings efficiently

CharacterTextSplitter: A class used to split text into smaller chunks based on a specified character or set of characters, useful for maintaining context and ensuring manageable chunk sizes

Document Loader: A tool used to load documents from various sources, such as HTML, PDF, or code files, into a format that can be processed by an application

Document: A container that holds a piece of text along with its associated metadata, used for processing and analysis in information retrieval systems

Embedding: A numerical representation of text that captures its semantic meaning, allowing for comparison and retrieval based on similarity

FAISS (Facebook AI Similarity Search): An open source library for efficient similarity search and clustering of dense vectors, often used for implementing vector stores

Indexing: The process of organizing data to improve the speed and efficiency of information retrieval, typically by creating a structure that allows for quick searches and updates

JSONLoader: A tool used to load JSON data and extract specific information based on a jq schema, useful for processing and analyzing JSON files

Multi-vector Retriever: A retriever that creates multiple vectors for each document, capturing different aspects of the content to improve the accuracy of retrieval

ParentDocument Retriever: A retriever that indexes multiple chunks for each document and finds the most similar chunks based on their embeddings, returning the entire parent document for comprehensive results

Qdrant: A vector store that supports async operations, used for efficient similarity search and retrieval in information retrieval systems

RAG (Retrieval-Augmented Generation): A technique that integrates retrieval-based methods with large language models to improve the accuracy and relevance of generated responses by incorporating additional context through retrieval

RecursiveCharacterTextSplitter: A text splitter that uses a recursive approach to divide text into smaller chunks based on specified separators, ensuring even distribution and maintaining logical structure

Retriever: A tool used to fetch relevant documents based on an unstructured query, facilitating quick access to information in an information retrieval system

Self-Query Retriever: A retriever that uses a language model to transform user input into a query string and a metadata filter, useful for retrieving documents based on metadata rather than text similarity

Text Embedding: A numerical representation of text in vector form, used to capture semantic meaning and facilitate comparison and retrieval in information retrieval systems

Text Splitter: A tool used to divide text into smaller chunks, ensuring that each chunk is within a manageable size and maintaining semantic coherence

Tokenization: The process of breaking down text into smaller units (tokens) such as words or phrases for processing by machine learning models

Vector Store: A database that stores data in vector format, allowing for efficient similarity search and retrieval based on vector comparisons

Vectorstore Retriever: A retriever that uses a vector store to find similar documents based on their embeddings, suitable for quick and easy retrieval tasks

References

These references provide foundational knowledge for implementing Retrieval-Augmented Generation (RAG) systems within the context of LangChain, Pinecone, and OpenAI technologies.

1. Lewis, P., et al. (2020). “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.” arXiv:2005.11401.

This research paper introduces the RAG model and explains its architecture and performance in knowledge-intensive tasks.

2. LangChain Documentation. “Question Answering Use Cases.” https://python.langchain.com/v0.1/docs/use_cases/question_answering/

This official documentation is a detailed guide on using LangChain for question-answering systems.

3. Pinecone Documentation. “Retrieval-Augmented Generation with Pinecone.” <https://www.pinecone.io/learn/retrieval-augmented-generation/>

CHAPTER 7 BUILDING ADVANCED Q&A AND SEARCH APPLICATIONS USING RETRIEVAL-AUGMENTED GENERATION (RAG)

This is a documentation from Pinecone that discussed how to use Pinecone's vector database for RAG.

4. Kumar, A. (2023). "Retrieval-Augmented Generation (RAG): From Theory to LangChain Implementation." <https://towardsdatascience.com/retrieval-augmented-generation-rag-from-theory-to-langchain-implementation-4e9bd5f6a4f2>

This article provides a practical overview and implementation guide for RAG using LangChain.

5. OpenAI Help Center. "Retrieval-Augmented Generation (RAG) and Semantic Search for GPTs." <https://help.openai.com/en/articles/8868588-retrieval-augmented-generation-rag-and-semantic-search-for-gpts>

This is an OpenAI resource on applying RAG with OpenAI models.

CHAPTER 8

Your First Agent App

In this chapter, we will explore intelligent autonomous agents and learn how to create them using the LangChain framework. We will start by understanding the fundamental concepts of agents, their key features, and their thought processes. Then, we will build an end-to-end agent application and cover everything from setting up the environment to implementing memory capabilities.

Throughout the chapter, you will gain hands-on experience working with LangChain Agent and learn how to use its powerful tools and libraries to create agents that can perceive, reason, and take actions autonomously. I will provide step-by-step guidance, code snippets, and practical examples to ensure you have a solid grasp of the concepts and can apply them in real-world scenarios.

Introduction

Let us understand what agents are and their thought process.

What Are Agents?

When you have a task at hand, and you are looking for an intelligent, autonomous app or tool to handle it for you, LangChain Agents will come to your rescue! You can employ an Agent to perceive its environment, make decisions, and take actions to achieve a specific goal. In my opinion, it is a leap in the world of software development, with a lot of potential.

Types of LangChain Agents and Their Capabilities

LangChain offers various types of agents that are designed for specific use cases and environments. Here are the main types:

1. **General-Purpose Agents:** These are versatile agents that can handle a wide range of tasks using different tools.
2. **Domain-Specific Agents:** These agents are specialized for particular fields or industries, such as finance, healthcare, or legal domains. They have in-depth knowledge and capabilities tailored to their specific area of expertise.
3. **Simulation Agents:** These agents are designed to operate in simulated environments, often used for training, testing, or scenario planning. They can interact with virtual worlds and respond to simulated stimuli.
4. **Task-Specific Agents:** Designed to excel at particular tasks like text summarization, question answering, or code generation.
5. **Multi-agent Systems:** These involve multiple agents that are working together and often simulating complex systems or handle intricate, multistep processes.

Some of the key features of agents are

1. **Autonomy:** You can use Agents to make decisions and take actions without direct human intervention based on AI-driven logic and the data they process.

2. **Dynamic Tool Selection:** Agents can dynamically select the appropriate tools based on the given query by understanding its requirements and choose the best tools to generate a comprehensive response.
3. **Reactivity:** Agents respond to changes in their environment, such as user input or new data.
4. **Goal-Oriented:** Agents can help you to achieve specific goals or tasks.
5. **Context Awareness:** LangChain Agents can grasp the intent behind the question, identify relevant information, and provide responses that are tailored to the specific needs of the user.
6. **Integration:** Agents can interact with users, other agents, external systems, tools, APIs, and data sources to provide very rich functionality.
7. **Memory:** LangChain Agents can maintain context and retain information throughout a conversation or a series of interactions to provide coherent and contextually relevant answers.
8. **Customization and Extensibility:** You can define your own tools, create custom Agent classes, and extend the existing functionality to suit your specific requirements.

Agents typically work by

- **Perceiving:** They gather information about their environment through sensors, APIs, or other data sources.

- **Creating a Profile:** Based on the perceived information, agents build a profile or model of their environment and the task at hand.
- **Building Memory:** Agents accumulate and organize information from past actions and interactions, creating a knowledge base to inform future decisions.
- **Planning:** Using their profile and memory, agents formulate strategies or plans to achieve their goals.
- **Reasoning:** They analyze the available information, their plans, and potential outcomes to make informed decisions.
- **Acting:** Based on their reasoning, agents perform tasks, interact with users, or manipulate their environment.
- **Learning:** After acting, agents update their memory and refine their decision-making processes based on the outcomes of their actions.

Example of an Agent's Workflow

Consider a customer support agent designed to handle inquiries.

Perceiving

The agent receives a new support ticket via an API.

It extracts relevant details from the ticket, such as the customer's name, issue description, and previous interaction history.

Reasoning

The agent updates its profile with the new ticket information.

It reviews the customer's past interactions to understand context and previous issues.

The agent plans a response by considering potential solutions and the most effective way to address the customer's issue.

It decides to provide troubleshooting steps or escalate the ticket based on the complexity of the issue.

Acting

The agent sends a response to the customer with the planned troubleshooting steps.

It logs the interaction and updates the customer's profile with the new details.

If the issue is resolved, the agent closes the ticket. If not, it provides further assistance or escalates the issue to a human agent.

Agents aren't limited to chatbots though. They can also be used in other scenarios such as

- **Task Automation:** You can use Agents to handle repetitive tasks, like data entry or file organization, which saves you time and effort.
- **Recommendation Systems:** Agents can analyze user preferences and provide personalized recommendations and improve user experience.
- **Intelligent Search:** You can use Agents to retrieve the most relevant information from a vast knowledge base using their ability to understand natural language queries.

Agent's Thought Process

Some of the key parts of an Agent are

- **Tools:** These give it the ability to interact with the world, whether that is searching the Web, executing Python code, or even sending emails.

- **Language Model (LLM):** This is the agent's brain that provides the reasoning power needed to make decisions, understand language, and generate text.
- **Prompt:** These are the instructions you give your agent. A well-crafted prompt can guide the agent's behavior and help it focus on the right tools and actions.

The benefit of Agents lies in the way they abstract the complexities of natural language understanding, decision-making, and action execution which allows you to focus on the higher-level goals of your application. They don't just follow instructions; they make their own decisions. When you give them a task, they find out the best way to accomplish the task using the tools and resources available.

Let us illustrate this with an example. Let us say your task for the agent is to find out the current weather in NY city and text it to your friend.

Here is how it will go about doing the task.

Task: Get the weather in NY city and text it to my friend.

Agent's Thought Process: Agent starts to think the process to solve.

- 1) **Tool Selection:** The Agent starts to think in terms of what tool it can use.

Then it finds out that the SerpAPI tool can help to search for weather in NY City.

- 2) **Action:** The agent uses the SerpAPI tool to search for weather in NY City.

- 3) **Result:** The agent gets the results.

Reflection: The agent evaluates the results, determining if the goal has been met or if further action is required. If needed, it repeats steps 1-3 until the goal is achieved.

More Thinking: The agent starts to think how to send the message to the friend.

Agent: The agent finds out it would need a messaging tool and then goes on to send the message as a text.

Why Agents Matter

Agents open a whole new world of possibilities for building applications. With agents, you can create

- **Research Assistants:** They go beyond simple web searches to synthesize information and generate insightful reports.
- **Autonomous Coders:** They can write, refactor, or debug code based on your natural language instructions.
- **Creative Collaborators:** They help you brainstorm ideas, write stories, or even compose music.

Imagine you're tasked with creating a content generation system that can write engaging articles on various topics. Instead of manually coding each step of the process, you can leverage Agents to handle the heavy lifting.

Here is how Agents can revolutionize your generative AI application:

1. **Content Understanding:** You can use Agents to analyze the input data, such as user prompts or existing articles, and extract key information to guide the content generation process. They can identify relevant topics, sentiments, and styles to ensure the generated content aligns with the desired output.

2. **Contextual Decision-Making:** Agents can make intelligent decisions based on the context of the input data and the goals of the application. They can determine the most appropriate content structure, tone, and length based on the target audience and the purpose of the generated content.
3. **Dynamic Content Generation:** With the help of Agents, you can create dynamic and adaptive content generation systems. Agents can interact with multiple components, such as language models, knowledge bases, and external APIs, to generate coherent and informative content that meets the specified requirements.
4. **Iterative Refinement:** Agents can continuously evaluate and refine the generated content based on predefined criteria or user feedback. They can identify areas for improvement, make necessary adjustments, and regenerate content until it reaches the desired quality level.

Agents for Content Generation

Let us take a look at a code snippet that demonstrates how Agents can be used in a content generation application. Please note that this code is for illustrative purposes only. I will share full working code later in this chapter.

Before diving into the code, let us discuss two crucial aspects:

- **ReAct (Reasoning and Acting) Format:** The ReAct format is a prompt engineering technique that combines reasoning and acting. It allows the agent to

break down complex tasks into a series of thought-action pairs. The agent first reasons about what to do, then takes an action, observes the result, and reasons again. This process continues until the task is completed.

- **SerpAPI Key:** SerpAPI is a search engine results API that requires an API key. You will need to sign up for a SerpAPI account and obtain an API key to use this tool.

```

from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate

# Set up API keys (replace with your actual keys)
os.environ["OPENAI_API_KEY"] = "your_openai_api_key"
os.environ["SERPAPI_API_KEY"] = "your_serpapi_api_key"

# Load the necessary tools
tools = load_tools(["serpapi", "llm-math"],
llm=OpenAI(temperature=0))

# Define the prompt template
prompt_template = PromptTemplate(
    input_variables=["topic"],
    template="Generate an engaging article about {topic}."
)

# Initialize the agent
agent = initialize_agent(tools, OpenAI(temperature=0),
agent="zero-shot-react-description", verbose=True)

# Run the agent with a user prompt

```

```
user_prompt = "What is the revenue increase due to the  
benefits of AI"  
agent.run(prompt_template.format(topic=user_prompt))
```

In this example, you are initializing the Agent with the necessary tools and a prompt template. The user provides a prompt, such as “What is the revenue increase due to the benefits of AI,” and the Agent uses the prompt template to generate an engaging article on the given topic.

The Agent is smart enough to leverage the appropriate loaded tool, such as SerpAPI for retrieving relevant information and LLM-Math for performing any required calculations, to generate comprehensive and accurate content.

Agents As Task Managers

You can view agents as project managers for your AI tasks. You give them a high-level objective, and they break it down into manageable steps, delegate tasks to appropriate tools (like language models or external APIs), and orchestrate the entire process. They analyze the results, make decisions based on feedback, and even learn from their experiences to improve future performance.

Examples in Action

1. **Code Generation:** Instead of asking a language model to simply “write Python code to calculate the Fibonacci sequence,” an agent can take this further. It can
 - **Refine Your Request:** “Write efficient Python code for the Fibonacci sequence up to the 50th number.”
 - **Test the Code:** Execute the generated code and check for correctness.

- **Optimize:** If errors are found, prompt the model to fix them.
 - **Explain:** Provide a clear explanation of how the code works.
2. **Creative Writing:** Let us say you want a story about a time-traveling detective. An agent can
- Break down the plot into scenes.
 - Generate each scene using a language model.
 - Ensure consistency in character development and plot points.
 - Revise and refine the narrative for coherence and style.

In essence, agents enhance the capabilities of generative AI models by

- **Expanding Scope:** Going beyond basic text or image generation to tackle complex, multistep tasks.
- **Adding Practicality:** Focusing on real-world outcomes and solving problems.
- **Providing Autonomy:** Making decisions and adapting to changing circumstances.

How Do Chains Differ from Agents?

Now that you have got a solid grasp on what Agents are, you might be wondering how they differ from Chains.

Think of Chains as a series of connected steps or tasks that are executed in a predefined order. Each step in the Chain takes the output of the previous step as its input and processes it to produce an output for the next step. Chains are ideal for scenarios where you have a fixed sequence of operations to perform.

Here is a simple example of a Chain. Note that this code is for illustration purposes only:

```
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
from langchain.llms import OpenAI

# Define the prompt template
prompt_template = PromptTemplate(
    input_variables=["product"],
    template="What are the benefits of {product}?",
)

# Initialize the LLM
llm = OpenAI(temperature=0.9)

# Create the Chain
chain = LLMChain(llm=llm, prompt=prompt_template)

# Run the Chain
product_name = "Customer Service Chatbot"
response = chain.run(product_name)
print(response)
```

In this example, the Chain consists of a single step: generating the benefits of a given product using a language model (OpenAI) and a prompt template. The Chain takes the product name as input, applies the prompt template, and produces the generated response as output.

Choosing Your Approach

Here is a quick guide to help you decide when to use chains vs. agents:

Factor	Chains	Agents
Task Structure	Well-defined, sequential steps	Open-ended, requires exploration or decision-making
Flexibility	Limited to the predefined sequence	Adapts to new information and can choose from multiple tools
Ease of Use	Generally easier to set up and understand	Requires a bit more configuration, but offers greater power
Example Use Cases	Summarizing text, answering questions based on a document, extracting data	Creative writing, code generation, complex research tasks, autonomous decision-making scenarios

So, what are the key differences between agents and chains?

Autonomy: Agents are autonomous, while chains are not. Agents can make decisions and take actions without direct human intervention, whereas chains rely on human input to function.

Goal-Oriented: Agents are goal-oriented, meaning they are designed to achieve specific goals or tasks. Chains, on the other hand, are more flexible and can be used for a wide range of applications. Note that there are also exploratory agents that are designed to explore and learn from their environment, often without a predefined goal.

Contextual Understanding: Agents can understand the context in which they are operating, allowing them to generate more meaningful and relevant responses. Chains, while capable of processing context, are not designed to understand it in the same way as agents.

In the world of agents, the language model takes center stage as the decision-maker, determining the sequence of actions to take based on the given context.

Think of it this way: in chains, the sequence of actions is predefined and hardcoded. It is like following a recipe step by step, with no room for deviation. But with agents, the language model acts as a reasoning engine, dynamically choosing the actions to take and the order in which to take them. It is able to improvise and adapt the steps based on the tools and the desired outcome.

Your First End-to-End Working Agent App

Now, let us talk about building your first Agent. We already discussed the code behind an agent, but here is an end-to-end fully working Agent:

```
# Install the required packages
!pip install langchain==0.0.153
!pip install openai==0.27.6
!pip install python-dotenv==1.0.0
!pip install google-search-results==2.4.2

import os
from dotenv import load_dotenv
import openai
from langchain.agents import load_tools, initialize_agent
```

```
from langchain.llms import OpenAI
from langchain.chat_models import ChatOpenAI

# Load environment variables from the .env file
load_dotenv()

# Get the OpenAI API key from the environment variable
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")

# Import the new Chat Completion API:
os.environ["OPENAI_API_KEY"] = "your open AI key"

os.environ["SERPAPI_API_KEY"] = "your SerpAPI key"

# Initialize the OpenAI client
openai.api_key = OPENAI_API_KEY

# Confirm that the API key is set correctly
openai.api_key = os.getenv("OPENAI_API_KEY")
SERPAPI_API_KEY = os.getenv("SERPAPI_API_KEY")

# Initialize the ChatOpenAI model
chat_model = ChatOpenAI(model_name="gpt-4", temperature=0)

# Load the necessary tools
tools = load_tools(["serpapi", "llm-math"], llm=chat_model)

# Initialize the Agent
agent = initialize_agent(tools, chat_model, agent="zero-shot-
react-description", verbose=True)

# Run the Agent with a query
query = "A software company is planning to develop a new
mobile app. They estimate that the initial development cost
will be $200,000, and the app will generate a monthly revenue
```

of \$15,000. The company wants to know how many months it will take to break even on their investment, assuming a monthly maintenance cost of \$5,000. Can you help calculate the breakeven point?"

```
response = agent.run(query)
print(response)
```

Getting the SerpAPI Key

To obtain a SerpAPI API key, follow these steps:

Go to the SerpAPI website (<https://serpapi.com/>) and click the “Sign Up” button in the top-right corner of the page.

Fill in the required information, such as your name, email address, and desired password, to create an account.

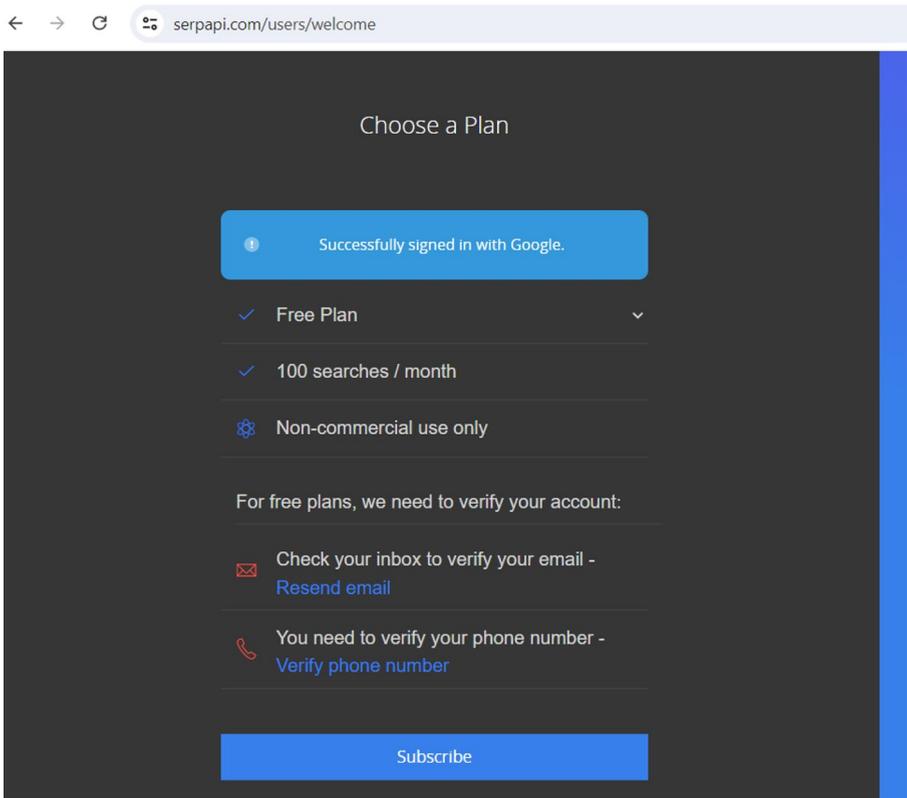
After creating your account, log in to the SerpAPI dashboard using your credentials.

Make sure you confirm your email account as well as the phone number.

Once logged in, you will be directed to the dashboard page. Look for the “API Key” section on this page.

In the “API Key” section, you will find your personal API key. It will be a string of characters that you will use to authenticate your API requests.

Copy the API key and store it securely. Avoid sharing it publicly, as it grants access to your SerpAPI account and resources.



In this example, you have initialized the Agent with a set of tools (SerpAPI and LLM-Math) and a language model (OpenAI). When given a query, the Agent analyzes the query, determines which tools to use, and dynamically generates a response by combining the results from the selected tools.

The Agent can handle multipart questions and adapt its approach based on the information needed to provide a comprehensive answer. It is able to understand your needs and find the most effective way to fulfill them.

Code Explanation

Let us now walk through the code step by step.

- **Installing dependencies**
 - `!pip install serpapi`: First, you must install the `serpapi` library, which is used to interact with the SerpAPI search engine.
 - `!pip install google-search-results`: Then you must install the `google-search-results` library, which provides an alternative way to interact with the Google Search API.
- **Importing necessary modules**
 - `os`: Import the `os` module to interact with the operating system.
 - `dotenv`: This module is used to load environment variables from a `.env` file.
 - `openai`: This module is the OpenAI API client library.
 - `load_tools` and `initialize_agent` from `langchain.agents`: Use the `load_tools` and `initialize_agent` functions from `langchain.agents` to load tools and initialize an agent.
 - `OpenAI` from `langchain.llms`: This class represents the OpenAI language model.
- **Loading environment variables**
 - `load_dotenv()`: Use the `load_dotenv` function to load the environment variables from the `.env` file.

- `OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")`: This line retrieves the OpenAI API key from the environment variable.
- `os.environ["OPENAI_API_KEY"] = "Your OpenAI API Key"`: This line sets the OpenAI API key in the environment variables.
- `os.environ["SERPAPI_API_KEY"] = "Your SERPAPI key"`: This line sets the SerpAPI API key in the environment variables.
- Initializing the OpenAI client
 - `openai.api_key = OPENAI_API_KEY`: This line sets the OpenAI API key for the OpenAI client.
 - `openai.api_key = os.getenv("OPENAI_API_KEY")`: This line confirms that the OpenAI API key is set correctly.
 - `SERPAPI_API_KEY = os.getenv("SERPAPI_API_KEY")`: This line retrieves the SerpAPI API key from the environment variable.
- Initializing the language model (LLM)
 - `llm = OpenAI(openai_api_key=OPENAI_API_KEY, temperature=0)`: Initialize the OpenAI language model with the provided API key and set the temperature to zero for deterministic responses.
- Loading the necessary tools
 - `tools = load_tools(["serpapi", "llm-math"], llm=llm)`: Using this line, you are loading the SerpAPI and LLM-Math tools using the `load_tools` function and passing the initialized language model.

- Initializing the Agent
 - `agent = initialize_agent(tools, llm, agent="zero-shot-react-description", verbose=True)`: You must then initialize the agent using the loaded tools, language model, and the "zero-shot-react-description" agent type. The `verbose=True` argument enables verbose output.
- Running the Agent with a query
 - `query = "A software company is planning to develop a new mobile app. They estimate that the initial development cost will be $200,000, and the app will generate a monthly revenue of $15,000. The company wants to know how many months it will take to break even on their investment, assuming a monthly maintenance cost of $5,000. Can you help calculate the breakeven point?":` This line defines the query to be asked to the agent.
 - `response = agent.run(query)`: You then run the agent with the provided query and store the response.
 - `print(response)`: Finally, you print the agent's response.

Results After Running the Code

Let us now analyze the results to understand how the agent works. Below is the result of the code when run:

Entering new AgentExecutor chain...

***To calculate the breakeven point, I need to find the formula for breakeven analysis. I will search for this information.

Action: Search

Action Input: “breakeven analysis formula”***

Observation: ***The breakeven point is the point at which total revenue equals total costs. The basic formula for breakeven analysis is:

Breakeven Point = Fixed Costs ÷ (Revenue per Unit - Variable Costs per Unit)

In this case, we have:

- Fixed Costs: The initial development cost of \$200,000
- Revenue per Unit: The monthly revenue of \$15,000
- Variable Costs per Unit: The monthly maintenance cost of \$5,000

So, the formula would be:

Breakeven Point (in months) = \$200,000 ÷ (\$15,000 - \$5,000) =
\$200,000 ÷ \$10,000

To calculate the breakeven point in months, divide the initial development cost by the difference between the monthly revenue and monthly maintenance cost.***

Thought: ***The search results provide the formula I need to calculate the breakeven point. I have the initial development cost, monthly revenue, and monthly maintenance cost. I can now calculate the breakeven point in months.

Action: Calculator

Action Input: 200000 / (15000 - 5000)***

Observation: ***Answer: 20.0***

Thought: ***Using the breakeven analysis formula and the given information, I have calculated that it will take the company 20 months to break even on their investment in the mobile app.

Final Answer: Given an initial development cost of \$200,000, a monthly revenue of \$15,000, and a monthly maintenance cost of \$5,000, it will take the software company 20 months to break even on their investment in the new mobile app.***

> **Finished chain.**

Given an initial development cost of \$200,000, a monthly revenue of \$15,000, and a monthly maintenance cost of \$5,000, it will take the software company 20 months to break even on their investment in the new mobile app.

Interpreting the Results

Let us analyze the steps taken by the agent to solve the business query – “A software company is planning to develop a new mobile app. They estimate that the initial development cost will be \$200,000, and the app will generate a monthly revenue of \$15,000. The company wants to know how many months it will take to break even on their investment, assuming a monthly maintenance cost of \$5,000. Can you help calculate the breakeven point?”

1. The agent starts by recognizing that it needs to find the formula for breakeven analysis to calculate the breakeven point. It decides to use the “search” tool for this information.
2. The search results provide the basic formula for breakeven analysis, which involves dividing the fixed costs by the difference between revenue per unit and variable costs per unit.
3. The agent identifies the information from the query to proceed with the calculation of the breakeven period:

- The initial development cost (\$200,000) as the fixed cost
 - The monthly revenue (\$15,000) as the revenue per unit
 - The monthly maintenance cost (\$5,000) as the variable cost per unit
4. Once it got the formula, the agent is smart enough to use a calculator to apply the breakeven analysis formula, dividing the initial development cost by the difference between the monthly revenue and monthly maintenance cost: $\$200,000 / (\$15,000 - \$5,000)$.
 5. The calculator returns the result of 20, indicating that it will take 20 months to break even.
 6. The agent provides the final answer, stating that given the initial development cost, monthly revenue, and monthly maintenance cost, it will take the software company 20 months to break even on their investment in the new mobile app.

This example shows how the agent can handle a business-related query that requires both information retrieval (searching for the breakeven analysis formula) and mathematical calculations (applying the formula to the given values).

The agent's thought process and actions are transparently visible in the verbose output. It shows the step-by-step approach taken by the agent to solve the problem.

So, to recap:

- Chains are best suited for fixed sequences of operations where the steps are predefined.
- Agents are ideal for more complex and open-ended tasks that require dynamic decision-making and adaptability.

Key Takeaways

In this chapter, we have explored the incredible potential of agents and how they can transform the way we build intelligent applications. We started by understanding what agents are, their key features, and how they differ from traditional chains. You learned that agents are autonomous entities that can perceive, reason, and act to achieve specific goals, making them perfect for tackling complex and dynamic tasks.

We even walked through the thought process of agents, discovering how they select tools, make decisions, and take actions based on the queries they receive. We also discussed some examples of how agents can be used in various domains, such as content generation, task automation, and intelligent search.

More importantly, we got our hands dirty and built our first end-to-end working agent application. Together, we set up the environment, loaded language models, defined tools, and created prompts to guide the agent's behavior.

Review Questions

Let us test your understanding of this chapter's content.

1. What is a key feature of LangChain agents?
 - A. They can only execute predefined sequences of tasks.
 - B. They require constant human intervention to function.
 - C. They can make decisions and take actions autonomously.
 - D. They are limited to processing text data only.
2. Which of the following best describes the thought process of a LangChain agent?
 - A. Receiving inputs, generating random outputs
 - B. Perceiving information, reasoning, and taking actions
 - C. Following hardcoded rules without adaptation
 - D. Interacting with users only through predefined prompts
3. What is one advantage of using dynamic tool selection in agents?
 - A. It reduces the need for context awareness.
 - B. It ensures that the agent always uses the same tool for all tasks.

- C. It allows the agent to choose the best tools based on the query.
 - D. It limits the agent's ability to handle multiple tasks.
4. How does memory integration enhance an agent's conversational abilities?
- A. By storing the conversation history in a database
 - B. By enabling the agent to recall and build upon previous interactions
 - C. By limiting the agent's responses to predefined answers
 - D. By allowing the agent to forget old conversations automatically
5. Which of the following is NOT a practical application of LangChain agents?
- A. Automating customer support
 - B. Generating content for marketing
 - C. Conducting repetitive data entry tasks
 - D. Creating static HTML web pages
6. What is the purpose of setting up environment variables when developing a LangChain agent?
- A. To hardcode sensitive information into the script
 - B. To configure the agent's appearance and user interface
 - C. To securely store API keys and other credentials
 - D. To define the agent's conversational topics

7. How does the LangChain framework simplify the development of intelligent agents?
 - A. By providing a single model for all tasks
 - B. By abstracting the complexities of direct API calls
 - C. By requiring no external libraries or dependencies
 - D. By limiting the customization options available
8. What is a significant difference between agents and chains in LangChain?
 - A. Chains can adapt to new information, while agents cannot.
 - B. Agents follow a fixed sequence of steps, while chains are goal-oriented.
 - C. Chains require manual intervention for each step, while agents operate autonomously.
 - D. Agents can make dynamic decisions, while chains follow predefined sequences.
9. In the context of LangChain, what does “context awareness” refer to?
 - A. The agent’s ability to operate without any human input
 - B. The agent’s capability to understand and incorporate the context of a query
 - C. The necessity for agents to have Internet connectivity
 - D. The predefined set of rules an agent follows

10. Which tool in LangChain helps manage conversation history for agents?
 - A. ChatOpenAI
 - B. ConversationBufferMemory
 - C. LLMChain
 - D. PromptTemplate

Answers

Below are the answers to the questions above:

1. C
2. B
3. C
4. B
5. D
6. C
7. B
8. D
9. B
10. B

Further Reading

These references will help deepen your understanding of LangChain agents and provide practical insights into building and optimizing your AI applications:

1. Understanding Agents in LangChain

LangChain Documentation on Agents: This section provides an in-depth explanation of what agents are, how they function, and the various use cases where they can be applied.

<https://python.langchain.com/v0.1/docs/modules/agents/>

2. Dynamic Tool Selection and Integration

LangChain Tools Integration Guide: Learn how to dynamically select and integrate various tools that agents can use to perform tasks.

<https://python.langchain.com/v0.2/docs/integrations/tools/>

3. Implementing Memory in Agents

LangChain Memory Modules: Explore the different memory options available in LangChain, such as ConversationBufferMemory, and how to integrate them into your agents.

<https://python.langchain.com/v0.1/docs/modules/memory/>

4. Building End-to-End Agent Applications

LangChain Tutorial on Building Agents: Follow a step-by-step guide to building a complete agent application from scratch, including setting up the environment, loading language models, and defining tools.

<https://python.langchain.com/v0.2/docs/tutorials/agents/>

5. Advanced Use Cases and Applications

LangChain Use Cases: Discover practical use cases of agents, such as task automation, content generation, and intelligent search, with detailed examples and code snippets.

https://js.langchain.com/v0.1/docs/use_cases/autonomous_agents/

CHAPTER 9

Building Different Types of Agents

In this chapter, we will be designing and implementing agents using LangChain. We will explore how to define an agent's objectives, manage its inputs and outputs, and use various tools and toolkits to supercharge its capabilities.

We will start by discussing how to define an agent's objectives. This is crucial because it sets the foundation for everything that follows. I will walk you through the process of articulating the problem your agent is meant to solve and outlining the specific tasks it needs to perform to achieve its goals. We will also identify the tools and resources your agent will require to get the job done.

Next, we will explore some core agent concepts like `AgentAction`, `AgentFinish`, and intermediate steps. These components are the building blocks that allow agents to reason and act dynamically.

One of the exciting aspects of working with LangChain is the variety of agent types available. We will explore Zero-Shot-React agents, structured chat agents, ReAct agents, and more. Each agent type has its own unique characteristics and use cases, and I will provide you with practical, working code examples to illustrate how they work.

As we progress through the chapter, you will get hands-on experience implementing both basic and advanced agents. We will start with simple examples and gradually build up to more complex scenarios.

We will also explore how to integrate built-in tools and create custom tools to extend their capabilities. You will learn about toolkits to group related tools together for specific tasks. This will help you organize your agent's functionality and make it more efficient.

Finally, we will discuss adding memory to your agents, which enables your agents to remember previous interactions and maintain context over multiple conversations. I will show you how to implement memory capabilities and use chat history to enhance your agents' context awareness and coherence.

Learning Objectives

By the end of this chapter, you will be able to

1. Define your agent's objectives clearly.
2. Understand core agent concepts like `AgentAction`, `AgentFinish`, and intermediate steps.
3. Differentiate between various agent types and their use cases.
4. Implement basic and advanced agents using `LangChain`.
5. Use built-in tools and create custom tools to extend your agent's capabilities.
6. Add memory to your agents to enable context-aware conversations.

Designing and Implementing an Agent

In this section, let us review the aspects of agent design and implementation, including defining objectives, understanding core concepts, managing inputs and outputs, executing tasks, and utilizing tools and toolkits. Knowing this will help you create efficient and effective agents.

Defining the Agent's Objective

Let us take a moment to discuss the most crucial aspect of building an Agent: defining its objective.

To define your Agent's objective, ask yourself the following questions:

1. What problem do I want my Agent to solve?
 - Are you building a chatbot to assist customers with their inquiries?
 - Do you need an Agent to analyze data and provide insights?
 - Is your Agent responsible for generating content based on specific prompts?
2. Will my Agent need to collaborate with other agents?
 - Does the task require multiple specialized agents working together?
 - Would different agents handle different aspects of a complex problem?
 - Could collaboration lead to more comprehensive or efficient solutions?

3. What tasks should my Agent perform to achieve its goal?
 - You should start by breaking down the problem into smaller, manageable tasks.
 - Identify the specific actions your Agent needs to take to solve the problem.
 - You should carefully consider the input and output requirements for each task.
4. What tools and resources will my Agent need?
 - You must determine the libraries, APIs, and external services your Agent will interact with.
 - Assess whether you need to integrate any domain-specific knowledge or databases.
 - Identify the language models or pretrained models that align with your Agent's objectives.

Let us say you want to create an Agent business app that assists users in finding the best restaurants in a given city. Your Agent's objective could be defined as follows:

Objective: The Restaurant Recommendation Agent will help users discover top-rated restaurants in a specified city based on their cuisine preferences and budget constraints. The Agent will interact with users through a conversational interface, understand their requirements, and provide personalized restaurant recommendations along with relevant information such as ratings, reviews, and contact details.

With this objective in mind, you can now outline the tasks your Agent needs to perform:

1. Understand user input and extract relevant information (city, cuisine, budget).
2. Interact with a restaurant database or API to fetch restaurant data.
3. Apply filtering and ranking algorithms to select the best restaurants based on user preferences.
4. Generate a natural language response presenting the recommended restaurants and their details.
5. Handle follow-up questions and provide additional information as requested by the user.

Now that you have a clear objective and a set of tasks, you can start gathering the necessary tools and resources. In this case, you might need the following:

- A language model or conversational AI framework for natural language understanding and generation
- A restaurant database or API to access restaurant information
- Libraries for data manipulation, filtering, and ranking
- Integration with external services for fetching reviews, ratings, and contact details

Concepts

Alright, let us discuss the core concepts behind agents and explore the key components that make agents tick:

1. AgentAction

- `AgentAction` is a dataclass that represents the action an agent should take. It is like a blueprint for the agent's next move.
- It has two important properties:
 - `tool`: This is the name of the tool that the agent should invoke. Think of it as the specific skill or capability the agent will use.
 - `tool_input`: This is the input that should be provided to the tool. It is like giving the agent the necessary information to perform the action effectively.

2. AgentFinish

- `AgentFinish` represents the final result from an agent when it has completed its task and is ready to return the output to the user.
- It contains a `return_values` key-value mapping, which holds the final agent output. This is where the agent's response or solution is stored.
- Typically, the `return_values` mapping includes an `output` key, which contains a string representing the agent's final response.

3. Intermediate Steps

- Intermediate steps are like the agent's memory of previous actions and their corresponding outputs within the current agent run.
- They are crucial for passing information to future iterations, allowing the agent to know what work it has already done and build upon it.
- Intermediate steps are represented as a list of tuples, where each tuple contains an `AgentAction` and its corresponding output.
- The type of the intermediate steps is `List[Tuple[AgentAction, Any]]`. The `Any` type is used for the output to provide maximum flexibility, as the output can be of various types depending on the tool used. In most cases, the output is a string, but it can be other types as well.

Here is a code snippet to illustrate the usage of these components. Note that this is for illustrative purposes only.

```
from langchain_core.agents import AgentAction, AgentFinish

# Create an AgentAction
action = AgentAction(tool="search", tool_input="What is the
annual revenue of Amazon in 2023?")

# Perform the action and get the output
output = perform_action(action)

# Store the intermediate step
intermediate_steps = [(action, output)]
```

```
# Create an AgentFinish with the final output
final_output = AgentFinish(return_values={"output": "The annual
revenue of Amazon is 574.8 billion."})
```

In this example, you create an `AgentAction` specifying the tool to use (e.g., “search”) and the input to provide to the tool. You then perform the action and store the output. The intermediate step, consisting of the action and its output, is added to the `intermediate_steps` list. Finally, you create an `AgentFinish` with the final output, which in this case is a string stating the annual revenue of Amazon.

You can see that agents can leverage the power of these LLMs and tools to dynamically reason and determine the best course of action to solve a given problem. They can adapt and make decisions based on the available tools, the input provided, and the intermediate results obtained along the way.

Agent

Alright, let us talk about the heart of the agent, namely, the chain responsible for deciding the next step to take. It is usually powered by a language model, a prompt, and an output parser.

You should remember that different agents have their own unique styles when it comes to reasoning, encoding inputs, and parsing outputs. `LangChain` provides a variety of built-in agents that you can choose from. Each has their own strengths and characteristics. You can find a full list of these agents in the `agent types` documentation that I have included in the “Further Reading” section of this chapter.

If you need more control or have specific requirements, you can easily build your own custom agents. Building custom agents allows you to define your own prompting style, input encoding, and output parsing logic. We will discuss this as well later in this chapter.

Let us dive into the inputs and outputs of an agent.

Agent Inputs

When it comes to the inputs of an agent, it is all about key-value pairs. The only required key is `intermediate_steps`, which corresponds to the `Intermediate Steps` we discussed earlier. These steps are crucial because they provide the agent with the context of what has been done so far.

But here is where the `PromptTemplate` comes in. It takes care of transforming these key-value pairs into a format that can be easily understood by the language model.

Agent Outputs

Next is Agent output. The output of an agent can be either the next action(s) to take or the final response to send back to the user. In technical terms, these outputs are represented by `AgentActions` or `AgentFinish`. You can think of them as the agent's decisions or the final verdict.

The output can be one of three types:

- `AgentAction`: A single action the agent wants to take next
- `List[AgentAction]`: A list of actions the agent wants to take next
- `AgentFinish`: The final response the agent wants to send back to the user

It is like the agent is saying what the next step or answer is. The output parser is responsible for taking the raw output from the language model and transforming it into one of those three types. In other words, it interprets the agent's thoughts and turns them into concrete actions or responses.

Here is a code snippet to illustrate the usage of agent inputs and outputs:

```

from langchain_core.agents import AgentAction, AgentFinish

# Create the agent using the new method agent = create_react_
agent(llm=llm, tools=tools, prompt=prompt)
# Define the agent's input
agent_input = {
    "intermediate_steps": [
        (AgentAction(tool="Search", tool_input="What is the
        capital of France?", log="Searching for the capital of
        France"), "Paris is the capital of France.")    ]
}

# Create an AgentExecutor agent_executor =
AgentExecutor(agent=agent, tools=tools, verbose=True)

# Call the agent with the input
agent_output = agent.run(agent_input)

# Check the type of the agent's output
if isinstance(agent_output, AgentAction):
    print("Agent wants to take the following action:",
    agent_output)
elif isinstance(agent_output, list):
    print("Agent wants to take the following actions:",
    agent_output)
elif isinstance(agent_output, AgentFinish):
    print("Agent has finished with the following response:",
    agent_output)

```

In this example, you define the agent's input by providing the `intermediate_steps`, which is a list of tuples containing previous actions and their outputs. You then call the agent with this input and store the output in `agent_output`.

Finally, you check the type of the agent's output using `isinstance()` to determine whether it is an `AgentAction`, a list of `AgentActions`, or an `AgentFinish`. Based on the type, you can take the appropriate action or send the final response back to the user.

You now have a solid understanding of how agents make decisions, take actions, and provide responses.

AgentExecutor

`AgentExecutor` is the core engine behind the scenes which provides the runtime for the agent to run smoothly and efficiently. It is responsible for calling the agent, executing the actions it chooses, passing the action outputs back to the agent, and repeating this process until the agent reaches a conclusion. It is like a loop of communication between the agent and the executor, with the executor facilitating the flow of information and actions.

Here is a simplified pseudocode representation of how the `AgentExecutor` works, again for illustrative purposes only:

```
next_action = agent.get_action(...)
while next_action != AgentFinish:
    observation = run(next_action)
    next_action = agent.get_action(..., next_action,
    observation)
return next_action
```

It may seem straightforward, but the `AgentExecutor` handles several complexities behind the scenes to make your life easier. Let us review some of the scenarios:

1. When the agent selects a nonexistent tool, the executor gracefully handles the situation and keeps the agent on track.
2. If a tool encounters an error during execution, the executor catches the exception and manages it appropriately to ensure the agent can continue its work.
3. In cases where the agent produces output that cannot be parsed into a valid tool invocation, the executor handles the situation and guides the agent back to a valid path.
4. The executor provides comprehensive logging and observability at all levels, such as agent decisions and tool calls. It can output this information to stdout and/or send it to LangSmith for further analysis and visualization.

Tools and Toolkits

Let us take a quick look at the tools and toolkits.

Tools

Tools are interfaces that an agent, chain, or LLM (large language model) can use to interact with the world. They combine a few essential elements:

1. **Name of the Tool:** A concise, descriptive label that tells you what the tool does.
2. **A Description:** A brief explanation of the tool's purpose and functionality.

3. **JSON Schema:** A structured definition of the inputs required by the tool. Think of it as a blueprint for how to use the tool correctly.
4. **The Function to Call:** The actual code that executes the tool's action.
5. **Flag:** A flag that determines if the tool's output should be immediately visible or processed further.

The name, description, and JSON schema help the LLM understand how to specify the desired action, while the function to call is the equivalent of actually taking that action.

A Tool abstraction in LangChain consists of two key components:

1. **The Input Schema for the Tool:** This is like a blueprint that tells the language model (LLM) what parameters are needed to call the tool. It is crucial to provide sensibly named and well-described parameters, so the LLM knows exactly what inputs to provide when invoking the tool.
2. **The Function to Run:** This is the actual Python function that gets executed when the tool is invoked. It is the code that performs the desired action based on the provided inputs.

One important thing you should keep in mind is that the simpler the input to a tool, the easier it is for an LLM to use it. I recommend you to use tools that have a single string input because agents work well with them. LangChain has documentation on which agent types can handle more complex inputs. Please see the “Further Reading” section for the link to the documentation.

CHAPTER 9 BUILDING DIFFERENT TYPES OF AGENTS

Now, let us use the `WikipediaQueryRun` tool, which is a handy wrapper around Wikipedia:

```
!pip install langchain==0.2.5 langchain_openai==0.2.5 wikipedia
from langchain_community.tools import WikipediaQueryRun
from langchain_community.utilities import WikipediaAPIWrapper

# Initialize the tool with custom configurations
api_wrapper = WikipediaAPIWrapper(top_k_results=1, doc_content_
chars_max=100)
tool = WikipediaQueryRun(api_wrapper=api_wrapper)
```

Now, let us explore some of the tool's properties:

```
# Check the default name
print(tool.name) # Output: 'Wikipedia'

# Check the default description
print(tool.description) # Output: 'A wrapper around Wikipedia.
Useful for when you need to answer general questions about
people, places, companies, facts, historical events, or other
subjects. Input should be a search query.'

# Check the default JSON schema of the inputs
print(tool.args) # Output: {'query': {'title': 'Query',
'type': 'string'}}

# Check if the tool should return directly to the user
print(tool.return_direct) # Output: False
```

Let us try searching for information about LangChain itself:

```
# Call the tool with a dictionary input
print(tool.run({"query": "langchain"}))

# Output: 'Page: LangChain\nSummary: LangChain is a framework
designed to simplify the creation of applications '
```

```
# Call the tool with a single string input (since it expects
only one input)
print(tool.run("langchain"))
# Output: 'Page: LangChain\nSummary: LangChain is a framework
designed to simplify the creation of applications '
```

But what if we want to customize the tool's name, description, or JSON schema? Let us create a custom schema for the Wikipedia tool:

```
from langchain_core.pydantic_v1 import BaseModel, Field

class WikiInputs(BaseModel):
    """Inputs to the wikipedia tool."""
    query: str = Field(description="query to look up in
Wikipedia, should be 3 or less words")

# Now, let's create a new instance of the tool with our custom
settings
tool = WikipediaQueryRun(
    name="wiki-tool",
    description="look up things in wikipedia",
    args_schema=WikiInputs,
    api_wrapper=api_wrapper,
    return_direct=True,
)

# Check the updated properties
print(tool.name) # Output: 'wiki-tool'
print(tool.description) # Output: 'look up things in
wikipedia'
print(tool.args) # Output: {'query': {'title': 'Query',
'description': 'query to look up in Wikipedia, should be 3 or
less words', 'type': 'string'}}
print(tool.return_direct) # Output: True
```

```
# Use the tool with the updated settings
print(tool.run("langchain"))
# Output: 'Page: LangChain\nSummary: LangChain is a framework
designed to simplify the creation of applications '
```

You have just learned how to work with built-in tools and customize them to your liking. LangChain offers a wealth of resources to help you on your journey:

- **Built-In Tools:** Check out the official documentation for a comprehensive list of all built-in tools.
- **Custom Tools:** While built-in tools are handy, you will likely need to define your own tools for your specific use cases. LangChain provides a guide on how to create custom tools.
- **Toolkits:** Toolkits are collections of tools that work well together. The documentation offers an in-depth description and a list of all built-in toolkits.
- **Tools As OpenAI Functions:** Tools in LangChain are similar to OpenAI Functions, and you can easily convert them to that format. Check out the official notebook for instructions on how to do that.

Toolkits

Toolkits are carefully curated collections of tools that are designed to work seamlessly together for specific tasks. Sometimes, accomplishing a task requires a set of related tools working together. That is where toolkits come in handy. They come with handy loading methods, which makes it easier to get started with the tools you need. For example, the GitHub

toolkit includes tools for searching through GitHub issues, reading files, commenting on issues, and more. LangChain provides a comprehensive list of ready-made toolkits, and you can locate them in the Integrations section of the documentation.

First, you need to initialize the toolkit you want to use. Let us say you are working with the `ExampleToolkit` (it is just a placeholder for now):

```
toolkit = ExampleToolkit(...)
```

Every toolkit exposes a `get_tools` method, which returns a list of the tools contained within that toolkit. Here is how you can access them:

```
tools = toolkit.get_tools()
```

Now, with these tools at your disposal, you can create an agent that can harness their collective power. LangChain provides a `create_agent_method` function that allows you to do just that. Simply pass in your LLM (large language model), the list of tools, and a prompt (if needed), and you have got yourself an agent ready to tackle any task as shown below:

```
agent = create_react_agent(llm=llm, tools=tools, prompt=prompt)
```

With this agent at your command, you can effortlessly orchestrate the tools to perform complex tasks, streamline workflows, and achieve results that would have been otherwise challenging or time-consuming.

The Integration section of LangChain's documentation provides a comprehensive list of ready-made toolkits. You have toolkits ranging from web scraping, working with databases, social media platforms, and data processing to natural language processing and more. By combining the right tools, you can create powerful workflows and automate complex tasks with ease.

Considerations

When working with tools, there are two important design considerations to keep in mind:

1. **Giving the Agent Access to the Right Tools:** It is essential to equip your agent with the necessary tools to accomplish its objectives. Without the right set of tools, your agent will be limited in its capabilities and may struggle to complete the tasks at hand.
2. **Describing the Tools in a Way That Is Most Helpful to the Agent:** The way you describe the tools plays a crucial role in how effectively the agent can use them. You should remember to provide clear descriptions that explain the purpose of each tool, so the agent can make informed decisions on when and how to use them.

Building an Agent Using LangGraph for Enhanced Capabilities

Let us look at how you can use LangGraph to create more complex, capable, and responsive agents by leveraging graph structures to manage and process information.

What Is LangGraph?

LangGraph allows you to structure information in a graph format, where nodes represent pieces of data or tasks, and edges represent the relationships between them. This structure makes it easier for agents to handle complex workflows, understand context, and perform multistep tasks efficiently.

Imagine you have an agent that needs to perform a series of tasks based on user input. Without a structured way to manage these tasks, your code can get messy and hard to maintain. LangGraph helps you by organizing tasks and data in a way that is both logical and scalable.

Setting Up LangGraph

First, let us set up your environment. Make sure you have LangChain installed:

1. **Install LangChain:** If you haven't already, install LangChain using pip.

```
!pip install langchain==0.2.5 langchain_
openai==0.1.8 langgraph==0.1.8
```

2. **Import Necessary Modules:** You will import the classes you need from LangChain.

```
from langchain_openai import OpenAI
from langchain.agents import Agent
from langchain.graph import LangGraph, Node
```

3. **Initialize the Language Model:** Set up your language model using the OpenAI API key.

```
llm = OpenAI(api_key="your_openai_api_key")
```

Creating a Simple LangGraph

Let us create a simple LangGraph that represents a sequence of tasks for your agent. Let us assume you are building a travel agent that helps users plan a trip:

1. **Define Nodes:** Nodes are the individual tasks or pieces of information in the graph.

```
# Define the nodes
node1 = Node(name="GreetUser", action=lambda:
"Hello! How can I assist you with your travel
plans today?")
node2 = Node(name="GetDestination",
action=lambda user_input: f"Great choice!
{user_input} sounds like a fantastic
destination.")
node3 = Node(name="SuggestActivities",
action=lambda: "Here are some activities you
might enjoy: visiting museums, trying local
cuisine, and exploring nature trails.")
```

2. **Create the Graph:** Link the nodes together to form the workflow.

```
# Create the LangGraph
travel_graph = LangGraph()
travel_graph.add_node(node1)
travel_graph.add_node(node2)
travel_graph.add_node(node3)

# Define the edges (relationships between nodes)
travel_graph.add_edge("GreetUser", "GetDestination")
travel_graph.add_edge("GetDestination",
"SuggestActivities")
```

3. **Define the Agent:** Create an agent that uses this graph to interact with users.

```

class TravelAgent(Agent):
    def __init__(self, llm, graph):
        super().__init__(llm=llm)
        self.graph = graph

    def run(self, user_input):
        # Start from the first node and
        # process through the graph
        current_node = self.graph.get_
        node("GreetUser")
        response = current_node.action()
        print(response)

        current_node = self.graph.get_
        node("GetDestination")
        response = current_node.
        action(user_input)
        print(response)

        current_node = self.graph.get_
        node("SuggestActivities")
        response = current_node.action()
        print(response)

# Instantiate the agent
agent = TravelAgent(llm=llm,
graph=travel_graph)

# Test the agent
agent.run("Hawaii")

```

In this illustrative example, the agent starts by greeting the user, then asks for a destination, and finally suggests some activities based on the destination. As you can see, the graph structure makes it easy to manage these steps and ensures that the agent follows a logical sequence.

Agent Types

LangChain offers a variety of agent types, each with its own unique characteristics and capabilities that you can choose from, depending on your specific needs and the models you are working with.

Criteria for Choosing Agent Types

First, let us talk about the intended model type. Some agents are designed specifically for chat models, which take in messages and output messages. Others are tailored for language models (LLMs), which take in strings and output strings. The main difference lies in the prompting strategy used. While you can mix and match agent types with different model types, sticking to the intended model type will likely yield the best results:

Chat History Support: Some agent types are built to handle chat history, making them ideal for chatbot applications. These agents can maintain context and provide a seamless conversational experience. On the other hand, agent types that don't support chat history are better suited for single tasks or one-off interactions.

Multi-input Tools: Some agent types are equipped to handle tools that require multiple inputs, while others are designed for simpler tools with a single input. If you're working with complex tools that need multiple parameters, make sure to choose an agent type that supports multi-input tools.

Parallel Function Calling: Parallel function calling is a powerful feature that allows agents to call multiple tools simultaneously to significantly

speed up the execution of certain tasks. However, this capability is more challenging for LLMs to handle, so not all agent types support it. Consider your specific requirements and the complexity of your tasks when deciding whether parallel function calling is essential.

Model Parameters: Some agent types also have specific requirements for model parameters. For example, certain agents take advantage of OpenAI's function calling capabilities, which require additional model parameters. If an agent type doesn't have any required model parameters, it means that everything is handled through prompting.

Types of LangChain Agents

Now that you have a solid understanding of what Agents are and how they differ from Chains, let us take a closer look at the wonderful world of LangChain Agents as of this writing.

1. API Tool Calling

- Intended Model Type: Chat
- Supports Chat History:
- Supports Multi-input Tools:
- Supports Parallel Function Calling:
- Required Model Params: tools
- When to Use: If you are using a tool-calling model

2. OpenAI Tools

- Intended Model Type: Chat
- Supports Chat History: ✓
- Supports Multi-input Tools: ✓
- Supports Parallel Function Calling: ✓
- Required Model Params: tools
- When to Use: [Legacy] If you are using a recent OpenAI model (1106 onward). Generic Tool Calling agent recommended instead

3. OpenAI Functions

- Intended Model Type: Chat
- Supports Chat History: ✓
- Supports Multi-input Tools: ✓
- Supports Parallel Function Calling: ✓
- Required Model Params: functions
- When to Use: [Legacy] If you are using an OpenAI model or an open source model that has been fine-tuned for function calling and exposes the same functions parameters as OpenAI. Generic Tool Calling agent recommended instead

4. XML

- Intended Model Type: LLM
- Supports Chat History: ✓
- Supports Multi-input Tools: ✓
- Supports Parallel Function Calling: ✓

- Required Model Params: None
 - When to Use: If you are using Anthropic models, or other models good at XML
5. Structured Chat
- Intended Model Type: Chat
 - Supports Chat History:
 - Supports Multi-input Tools:
 - Supports Parallel Function Calling:
 - Required Model Params: None
 - When to Use: If you need to support tools with multiple inputs
6. JSON Chat
- Intended Model Type: Chat
 - Supports Chat History:
 - Supports Multi-input Tools:
 - Supports Parallel Function Calling:
 - Required Model Params: None
 - When to Use: If you are using a model good at JSON
7. ReAct
- Intended Model Type: LLM
 - Supports Chat History:
 - Supports Multi-input Tools:
 - Supports Parallel Function Calling:

- Required Model Params: None
 - When to Use: If you are using a simple model
8. Self-Ask with Search
- Intended Model Type: LLM
 - Supports Chat History: ✗
 - Supports Multi-input Tools: ✗
 - Supports Parallel Function Calling: ✗
 - Required Model Params: None
 - When to Use: If you are using a simple model and only have one search tool

Here is a quick overview of the different types of Agents available in LangChain:

1. **Zero-Shot-React Agent:** You will use this Agent if you want to handle a wide range of tasks without any prior training or fine-tuning. You simply provide it with a set of tools and a language model, and it dynamically generates a response based on the given query. It is capable of adapting to any situation.
2. **Conversation Agent:** As the name suggests, you will use this Agent for conversational AI applications such as a chatbot or a virtual assistant. It can engage in back-and-forth dialogue, maintain context, and provide relevant responses based on the conversation history.

3. **Structured Tool Agent:** Sometimes, you have a specific set of tools or actions that you want your Agent to use in a structured manner. In such situations, you can employ the Structured Tool Agent to define a predetermined sequence of actions and tools which will guide the Agent's behavior.
4. **MRKL Agent:** The MRKL (Mean Reciprocal Rank with Logistic Regression) Agent is a powerful addition to the LangChain family. It uses a combination of search algorithms and logistic regression to rank and select the most relevant tools for a given query. You can use it to prioritize and choose the best course of action.

Zero-Shot-React Agent Example

The Zero-Shot-React Agent is a flexible and adaptable agent that uses a large language model (LLM) to determine which tool to use from a set of available tools, based solely on the tool's description and the user's input:

- **Zero-Shot:** The agent has no prior training on how to specifically use each tool. It figures out how to use them in the moment.
- **ReAct:** This is the underlying framework that guides the agent's decision-making. It is a loop of reasoning and action, allowing the agent to dynamically choose the best tool for the task.

The advantage when using this agent is that you don't need to create complex rules or chains for each tool, and it doesn't need prior training. However, its performance depends heavily on the description of the tool and the LLM's ability to understand the description. It implies therefore that you use a good reasoning LLM, such as gpt-4 or gpt-3.5-turbo.

The second disadvantage is that it does not have a memory. You can address this limitation by passing the `ConversationBufferMemory` object when initializing the agent. There is also a `conversational-react-description` agent type which uses memory.

Here is a code snippet that demonstrates how to initialize and use a `Zero-Shot-React Agent`:

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.llms import OpenAI

# Load the necessary tools
tools = load_tools(["serpapi", "llm-math"],
llm=OpenAI(temperature=0))

# Initialize the Zero-Shot-React Agent
agent = initialize_agent(tools, OpenAI(temperature=0),
agent="zero-shot-react-description", verbose=True)

# Run the Agent with a query
query = "What is the capital of France? What is the population
of that city?"
response = agent.run(query)
print(response)
```

In this example, you load the required tools (`serpapi` and `llm-math`) and initialize the `Zero-Shot-React Agent` with those tools and a language model (`OpenAI`). You then provide a query to the Agent, and it generates a response by leveraging the appropriate tools based on the query's requirements.

Tool Calling Agent

Let me introduce you to a powerful concept called tool calling, which helps your agent to detect when it should call one or more tools and respond with the appropriate inputs for those tools.

With tool calling, it can intelligently choose the right tools and know exactly what ingredients to use with each tool. In the API world, you can describe these tools to your agent, and it will output a structured object (like JSON) containing the arguments needed to call the tools.

The goal of tool calling is to ensure that your agent reliably returns valid and useful tool calls, going beyond what a generic text completion or chat API can do. By leveraging this structured output and allowing your agent to choose from multiple tools, you can create an agent that repeatedly calls tools and receives results until it resolves a query.

Setup

To get started with tool calling, you will need a model that supports it. LangChain offers a wide range of options, including Anthropic, Google Gemini, Mistral, and OpenAI. You can check out the supported models in the LangChain documentation.

For this demo, you will be using Tavily, but feel free to swap in any other built-in tool or even add your own custom tools. To use Tavily, you will need to sign up for an API key and set it as `process.env.TAVILY_API_KEY`.

First, let us install the necessary dependencies:

```
pip install -qU langchain-openai langchain tavily-python
```

Next, set your OpenAI API key as an environment variable:

```
import getpass
import os

os.environ["OPENAI_API_KEY"] = getpass.getpass()
os.environ["TAVILY_API_KEY"] = getpass.getpass("Enter your
Tavily API key: ")
```

Now, you will import the ChatOpenAI class from langchain_openai and create an instance of the language model:

```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4")
```

Initializing Tools

Let us create a tool that can search the Web using Tavily:

```
from langchain.agents import AgentExecutor, create_tool_
calling_agent
from langchain_community.tools.tavily_search import
TavilySearchResults
from langchain_core.prompts import ChatPromptTemplate

tools = [TavilySearchResults(max_results=1)]
```

Creating the Agent

Now, you must initialize your tool calling agent by defining a prompt and creating the agent:

```
prompt = ChatPromptTemplate.from_messages(
    [
        (
```

```

        "system",
        "You are a helpful assistant. Make sure to use the
        tavily_search_results_json tool for information.",
    ),
    ("placeholder", "{chat_history}"),
    ("human", "{input}"),
    ("placeholder", "{agent_scratchpad}"),
]
)

agent = create_tool_calling_agent(llm, tools, prompt)

```

Running the Agent

With your agent initialized, you will create an executor to run it and invoke it with a query:

```

# Create an agent executor by passing in the agent and tools
agent_executor = AgentExecutor(agent=agent, tools=tools,
verbose=True, handle_parsing_errors=True)

# Invoke the agent executor with the business-related question
response = agent_executor.invoke({"input": "How can our company
reduce operational costs by leveraging AI technologies?"})

# Print the response
print(response)

```

Watch as your agent springs into action, using the Tavily search tool to find information about your question and provide a concise summary:

› **Entering new AgentExecutor chain...**

Could not parse LLM output: {

"response": "Reducing operational costs by leveraging AI technologies can be achieved through various strategies:",

```

"strategies": [
  {
    "title": "Automating Routine Tasks",
    "description": "Implement AI-powered automation to handle repetitive and time-consuming tasks such as data entry, customer support inquiries, and inventory management. This can reduce the need for manual labor and free up employees to focus on higher-value activities."
  },
  {
    "title": "Predictive Maintenance",
    "description": "Utilize AI for predictive maintenance of equipment and machinery. By analyzing data from sensors and historical maintenance records, AI can predict potential failures, allowing for proactive maintenance and minimizing downtime."
  },
  {
    "title": "Optimizing Supply Chain Management",
    "description": "AI can optimize supply chain operations by analyzing demand patterns, predicting inventory needs, and identifying the most cost-effective shipping routes. This can lead to reduced inventory holding costs and improved efficiency."
  },
  {
    "title": "Enhancing Energy Efficiency",
    "description": "Implement AI-powered systems to optimize energy usage within facilities. AI can analyze patterns of energy consumption and automatically adjust settings for lighting, heating, and cooling, leading to cost savings."
  }
]

```

```

    },
    {
      "title": "Improving Decision Making",
      "description": "AI can provide valuable insights by
analyzing large datasets and identifying cost-saving
opportunities. By leveraging AI for data-driven decision
making, companies can optimize processes and resource
allocation."
    }
  ]
}

```

Using Chat History

One of the coolest features of tool calling agents is their ability to use chat history for a more conversational experience. By passing in previous conversation turns, your agent can respond in a more contextual and natural way.

Here is an example of how to use chat history with your agent:

```

from langchain_core.messages import AIMessage, HumanMessage

agent_executor.invoke(
    {
      "input": "what's my name? Don't use tools to look this
up unless you NEED to",
      "chat_history": [
        HumanMessage(content="hi! my name is Rabi Jay"),
        AIMessage(content="Hello Rabi Jay! How can I assist
you today?"),
      ],
    }
)

```

In this case, the agent remembers the previous conversation and responds accordingly:

Based on what you told me, your name is Rabi Jay. I don't need to use any tools to look that up since you directly provided your name.

You have just witnessed the power of tool calling agents in action. They can intelligently choose and use tools, provide structured outputs, and even engage in conversational interactions using chat history.

OpenAI Tools

Let us talk about an exciting feature from OpenAI called “tools,” which help your agent to detect when it should call one or more functions and respond with the appropriate inputs. Newer OpenAI models have been fine-tuned for this capability, making your agent smarter and more efficient.

In an API call, you can describe functions to your agent, and it will intelligently choose to output a JSON object containing the arguments needed to call those functions.

The goal of OpenAI tools is to ensure that your agent reliably returns valid and useful function calls, going beyond what a generic text completion or chat API can do. It makes your agent more precise and effective.

OpenAI has two related concepts: “functions” and “tools.” Functions allow your agent to invoke a single function, while tools enable it to invoke one or more functions when appropriate. In the OpenAI Chat API, functions are now considered a legacy option and are deprecated in favor of tools.

So, if you are creating agents using OpenAI models, you should be using the OpenAI Tools agent instead of the OpenAI Functions agent.

Using tools has a significant advantage because it allows the model to request that more than one function be called when appropriate. This can help reduce the time it takes for your agent to achieve its goal, making it more efficient and effective.

Now, let us look into the code and see how you can create an OpenAI Tools agent in action.

First, make sure you have the necessary libraries installed:

```
%pip install --upgrade --quiet langchain-openai tavily-python
langchainhub
```

Next, you must import the required modules:

```
from langchain import hub
from langchain.agents import AgentExecutor, create_openai_
tools_agent
from langchain_community.tools.tavily_search import
TavilySearchResults
from langchain_openai import ChatOpenAI
```

Initializing Tools

For this example, you will give our agent the ability to search the Web using Tavily:

```
tools = [TavilySearchResults(max_results=1)]
```

Creating the Agent

Now, you will create your OpenAI Tools agent by pulling the prompt from the LangChain hub and choosing the LLM that will drive the agent:

```
prompt = hub.pull("hwchase17/openai-tools-agent")
llm = ChatOpenAI(model="gpt-3.5-turbo-1106", temperature=0)
agent = create_openai_tools_agent(llm, tools, prompt)
```

Running the Agent

With your agent ready, create an executor to run it and invoke it with a query:

```
agent_executor = AgentExecutor(agent=agent, tools=tools,
    verbose=True)
agent_executor.invoke({"input": "what is LangChain?"})
```

Watch as your agent springs into action, using the Tavily search tool to find information about LangChain and provide a concise summary:

```
LangChain is an open source orchestration framework for the
development of applications using large language models.
It is essentially a library of abstractions for Python and
Javascript, representing common steps and concepts. LangChain
simplifies the process of programming and integration with
external data sources and software workflows. It supports
various large language model providers, including OpenAI,
Google, and IBM. You can find more information about LangChain
on the IBM website: [LangChain - IBM](https://www.ibm.com/
topics/langchain)
```

Using Chat History

One of the coolest features of OpenAI Tools agents is their ability to use chat history for a more conversational experience. By passing in previous conversation turns, your agent can respond in a more contextual and natural way.

Here is an example of how to use chat history with your agent:

```
from langchain_core.messages import AIMessage, HumanMessage

agent_executor.invoke(
    {
        "input": "what's my name? Don't use tools to look this
up unless you NEED to",
        "chat_history": [
            HumanMessage(content="hi! my name is Rabi Jay"),
            AIMessage(content="Hello Rabi Jay! How can I assist you
today?"),
        ],
    }
)
```

In this case, the agent remembers the previous conversation and responds accordingly:

Your name is Rabi Jay.

You have just witnessed the power of OpenAI Tools agents in action. They can intelligently choose and use functions, provide structured outputs, and even engage in conversational interactions using chat history.

Structured Chat Agent

Let me introduce you to the structured chat agent, a powerful tool to use multi-input tools. To get started, make sure you have the necessary libraries installed. In this case, you will be using LangChain, Tavily Search, and OpenAI's ChatGPT. Here is how you can import them from the LangChain library and the LangChain community tools:

```
from langchain import hub
from langchain.agents import AgentExecutor, create_structured_
chat_agent
from langchain_community.tools.tavily_search import
TavilySearchResults
from langchain_openai import ChatOpenAI
```

Initializing Tools

For this example, you will be testing the agent using the Tavily Search tool to search for information online. This line creates a list of tools that the agent will have access to. You set the `max_results` parameter to 1 to indicate that the tool will return a maximum of one search result:

```
tools = [TavilySearchResults(max_results=1)]
```

Creating the Agent

Now, create your structured chat agent by pulling the prompt from the LangChain hub using `hub.pull()` as shown below in the first line. You then customize this prompt to suit your needs by providing specific instructions and examples for the business analyst assistant:

```
prompt = hub.pull("hwchase17/structured-chat-agent")
prompt.messages[0].prompt.template = ""
```

You are a business analyst assistant tasked with helping entrepreneurs and business owners make informed decisions. Use the provided search tool to find relevant information and answer the user's questions as best as you can.

If the question cannot be answered using the search results, provide guidance on where the user can find more information.

Please provide your response in a clear, concise, and well-structured format.

Here are some examples of the types of business-related questions you may be asked:

- What are some effective marketing strategies for a small business?
- How can I improve my company's cash flow?
- What are the key steps in creating a business plan?
- How do I conduct market research for a new product idea?
- What are some common challenges faced by startups, and how can they be overcome?

Remember to provide actionable advice tailored to the user's specific needs and situation.

Let's work together to help businesses thrive!

"""

Next, you will choose the language model (LLM) that will drive our agent. In this case, you will use OpenAI's ChatGPT with a temperature of 0 and the "gpt-3.5-turbo-1106" model:

```
llm = ChatOpenAI(temperature=0, model="gpt-3.5-turbo-1106")
```

Finally, you will construct the agent by calling the `create_structured_chat_agent` function, passing in the LLM, tools, and prompt:

```
agent = create_structured_chat_agent(llm, tools, prompt)
```

Defining a Helper Function

You then create a helper function that processes the output generated by the language model. It checks if the output contains the string “Invalid or incomplete response” and raises a `ValueError` exception if it does.

Otherwise, it returns the output as is:

```
def process_llm_output(output):
    if "Invalid or incomplete response" in output:
        raise ValueError("The language model generated an
            invalid or incomplete response.")
    return output
```

Running the Agent

With your agent ready to go, create an instance of the `AgentExecutor` class to run it. You provide the agent, tools, and various configuration options such as `verbose` (for detailed output), `handle_parsing_errors` (to handle parsing errors gracefully), `max_iterations` (to limit the maximum number of iterations), and `early_stopping_method` (to specify the method for early stopping):

```
agent_executor = AgentExecutor(
    agent=agent,
    tools=tools,
    verbose=True,
    handle_parsing_errors=True,
    max_iterations=3,
    early_stopping_method="force",
)
```

Now, ask your agent a question and see it in action:

```
question = "What are some effective ways to reduce operational
costs in a manufacturing business?"
```

Finally, you use a try-except block to execute the agent and handle any potential errors. You invoke the agent using `agent_executor.invoke({"input":question})` and pass the question as input. The agent generates a result, which you process using the `process_llm_output()` function. If the output is valid, you print the processed result. If a `ValueError` is raised due to an invalid or incomplete response, you catch the exception and print an error message.

Watch as the agent searches for information about LangChain using Tavily Search and provides a concise summary. You can ignore any errors for now as it is out of scope for the exercise. You have now learned how to set up and use a structured chat agent with LangChain to answer business-related questions.

› Entering new AgentExecutor chain...

```
{
  "response": "Reducing operational costs in a manufacturing
business can be achieved through various strategies:",
  "strategies": [
    {
      "title": "Lean Manufacturing",
      "description": "Implement lean manufacturing principles
to eliminate waste, improve efficiency, and reduce
costs. This involves streamlining processes, optimizing
inventory levels, and minimizing downtime."
    },
    {
      "title": "Energy Efficiency",
```

```

    "description": "Invest in energy-efficient equipment and
    processes to lower utility expenses. Conduct an energy
    audit to identify areas for improvement and consider
    renewable energy sources."
  },
  {
    "title": "Supplier Negotiation",
    "description": "Negotiate with suppliers for better
    pricing, discounts, or favorable payment terms.
    Consolidate purchases with key suppliers to leverage
    volume discounts."
  },
  {
    "title": "Inventory Management",
    "description": "Optimize inventory levels to reduce
    carrying costs and minimize the risk of excess or
    obsolete inventory. Implement just-in-time inventory
    practices where feasible."
  },
  {
    "title": "Outsourcing Non-Core Activities",
    "description": "Consider outsourcing non-core functions
    such as janitorial services, maintenance, or certain
    manufacturing processes to specialized third-party
    providers to reduce overhead costs."
  },
  {
    "title": "Process Automation",
    "description": "Invest in automation technologies to
    streamline production processes, improve accuracy, and
    reduce labor costs. This may involve robotics, automated
    assembly lines, or software systems."
  }

```

```

    }
  ]
}

```

Using Chat History

You can use the same approach you used for the previous two agent types to use chat history for a more conversational experience. By passing in previous conversation turns, your agent can respond in a more contextual and natural way.

ReAct Agent

Next is the ReAct agent, a powerful tool that allows you to implement the ReAct logic in your AI applications. It enables the agent to reason and act based on the information it gathers.

To get started, make sure you have the necessary libraries installed. In this case, you will be using LangChain, Tavily Search, and OpenAI. Here is how you can import them:

```

from langchain import hub
from langchain.agents import AgentExecutor, create_react_agent
from langchain_community.tools.tavily_search import
TavilySearchResults
from langchain_openai import OpenAI

```

Initializing Tools

First, let us load some tools for our ReAct agent to use. In this example, you will be using Tavily Search to allow your agent to search for information online:

```

tools = [TavilySearchResults(max_results=1)]

```

Creating the Agent

Now, create your ReAct agent by pulling the prompt from the LangChain hub. You can customize this prompt to suit your needs, but for now, you will use the default one:

```
prompt = hub.pull("hwchase17/react")
```

Next, you will choose the language model (LLM) to use. In this case, let us go with OpenAI:

```
llm = OpenAI()
```

Finally, construct the ReAct agent by calling the `create_react_agent` function, passing in the LLM, tools, and prompt:

```
agent = create_react_agent(llm, tools, prompt)
```

Running the Agent

With your ReAct agent ready to go, create an executor and run it by passing in the agent, tools, and set `verbose=True` to see the agent's thought process:

```
agent_executor = AgentExecutor(agent=agent, tools=tools,
    verbose=True)
```

Now, ask your agent a question and see it in action:

```
agent_executor.invoke({"input": "what is LangChain?"})
```

Watch as the agent goes through a series of thoughts and actions to gather information about LangChain:

```
I should research LangChain to learn more about it.
Action: tavily_search_results_json
Action Input: "LangChain"
```

I should read the summary and look at the different features and integrations of LangChain.

Action: `tavily_search_results_json`

Action Input: "LangChain features and integrations"

I should take note of the launch date and popularity of LangChain.

Action: `tavily_search_results_json`

Action Input: "LangChain launch date and popularity"

I now know the final answer.

Final Answer: LangChain is an open source orchestration framework for building applications using large language models (LLMs) like chatbots and virtual agents. It was launched by Harrison Chase in October 2022 and has gained popularity as the fastest-growing open source project on Github in June 2023.

Using Chat History

When using the ReAct agent with chat history, you will need a prompt that takes that into account. Let us pull the chat-specific prompt from the LangChain hub:

```
prompt = hub.pull("hwchase17/react-chat")
```

Now, let us construct the ReAct agent with this prompt:

```
agent = create_react_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools,
verbose=True)
```

To use chat history, you can pass in a string representing previous conversation turns. Here is an example:

```
from langchain_core.messages import AIMessage, HumanMessage

agent_executor.invoke(
    {
        "input": "what's my name? Only use a tool if needed,
        otherwise respond with Final Answer",
        # Notice that chat_history is a string, since this
        prompt is aimed at LLMs, not chat models
        "chat_history": "Human: Hi! My name is Rabi\nAI: Hello
        Rabi! Nice to meet you",
    }
)
```

In this case, the agent will reason about whether it needs to use a tool or not based on the given chat history:

```
Thought: Do I need to use a tool? No
Final Answer: Your name is Rabi.
```

```
Entering new AgentExecutor chain...
```

Note You may also get different messages depending on what model you use. Here is one such example:

```
As an AI, I don't have access to personal data about individuals
unless it has been shared with me in the course of our conversation.
I am designed to respect user privacy and confidentiality. Therefore, I
don't know the user's name.
```

```
Final Answer: I'm sorry, but I don't have access to that information.
```

```
> Finished chain.
```

```
{'input': "what's my name? Only use a tool if needed, otherwise
respond with Final Answer",
'chat_history': 'Human: Hi! My name is Rabi\nAI: Hello Rabi! Nice to
meet you',
'output': "I'm sorry, but I don't have access to that information."}
```

Self-Ask Agents

Let us look at self-ask agents with search capabilities to find answers to your burning questions.

To get started, make sure you have the necessary tools in your toolkit. In this case, you will be using LangChain, Fireworks LLM, and Tavily Answer. Go ahead and import them:

```
from langchain import hub
from langchain.agents import AgentExecutor, create_self_ask_
with_search_agent
from langchain_community.llms import Fireworks
from langchain_community.tools.tavily_search import
TavilyAnswer
```

Initializing Tools

Now, initialize the tools your self-ask agent will use. For this agent, you will be using Tavily Answer, which provides you with direct answers to your questions.

One important thing to note is that this agent can only use one tool, and it must be named “Intermediate Answer.” So, let us set it up:

```
tools = [TavilyAnswer(max_results=1, name="Intermediate
Answer")]
```

Creating the Agent

With your tools ready, it is time to create your self-ask agent. You will start by pulling the prompt from the LangChain hub. You can customize this prompt to suit your needs, but for now, you will use the default one:

```
prompt = hub.pull("hwchase17/self-ask-with-search")
```

Next, choose the LLM that will power your agent's thinking process. In this example, you will go with Fireworks LLM:

```
Set up the Fireworks API key os.environ["FIREWORKS_API_KEY"] =
"your-fireworks-api-key-here"
# Initialize the Fireworks LLM llm = Fireworks(
model="accounts/fireworks/models/llama-v2-13b-chat", # Choose
an appropriate model max_tokens=1024, temperature=0.7 )
```

Finally, construct the self-ask with search agent by calling the `create_self_ask_with_search_agent` function, passing in the LLM, tools, and prompt:

```
agent = create_self_ask_with_search_agent(llm, tools, prompt)
```

Running the Agent

With your self-ask agent assembled, create an executor and bring it to life by passing in the agent, tools, and set `verbose=True` to see the agent's thought process:

```
agent_executor = AgentExecutor(agent=agent, tools=tools,
verbose=True)
```

Now, ask your agent a question and watch it work its magic:

```
agent_executor.invoke(
    {"input": "What is the headquarters location of the
    company with the largest market capitalization in the tech
    industry?"}
)
```

The agent will start by asking itself a follow-up question to gather more information:

Yes.

Follow up: Which company has the largest market capitalization in the tech industry?

Using the Tavily Answer tool, the agent will find the answer to its own question:

As of June 18, 2024, the company with the largest market capitalization is NVIDIA.

Armed with this information, the agent will then provide the final answer:

So the final answer is: Santa Clara, California.

The self-ask with search agent has successfully found the answer to your question by breaking it down into smaller steps and using the available tools.

Autonomous Decision-Making Capability

Let us take up a more advanced example that showcases the autonomous decision-making capabilities of an agent. In this example, you will create a “Task Manager” agent that can understand user requests, break them down into subtasks, and autonomously decide which tools to use to complete each subtask.

Here is the step-by-step code breakdown:

1. Install the necessary libraries:

```
pip install langchain==0.2.5 openai==1.35.13
google-search-results serpapi
```

2. Import the required modules:

```
from langchain.agents import initialize_
agent, Tool
from langchain.llms import OpenAI
from langchain.utilities import
GoogleSearchAPIWrapper
```

3. Set up the language model and search tool:

```
llm = OpenAI(temperature=0)
search = GoogleSearchAPIWrapper()
```

You are using the OpenAI language model and the Google Search API to enable the agent to search for information online.

4. Define the tools available to the agent:

```
tools = [
    Tool(
        name="Search",
        func=search.run,
        description="Useful for searching
the internet for information."
    )
]
```

In this example, you define a single tool called “Search” that allows the agent to perform Internet searches using the Google Search API.

5. Initialize the agent:

```
agent = initialize_agent(
    tools,
    llm,
    agent="zero-shot-react-description",
    verbose=True
)
```

You initialize the agent with the defined tools, language model, and the “zero-shot-react-description” agent type, which enables the agent to autonomously decide which tools to use based on the user’s request.

6. Provide a task to the agent:

```
task = "I need to plan a trip to Paris. What are
the top tourist attractions I should visit, and
what is the best time of year to go?"
result = agent.run(task)
print(result)
```

You give the agent a task related to planning a trip to Paris. The agent will autonomously break down the task, search for relevant information, and provide a comprehensive response.

When you run this code, the agent will autonomously process the task and provide a response similar to the following:

CHAPTER 9 BUILDING DIFFERENT TYPES OF AGENTS

To plan your trip to Paris, here are the top tourist attractions you should visit and the best time of year to go:

Top tourist attractions in Paris:

1. Eiffel Tower - Iconic landmark with stunning city views
2. Louvre Museum - World-renowned art museum housing the Mona Lisa and other masterpieces
3. Notre-Dame Cathedral - Historic cathedral known for its Gothic architecture (currently under renovation due to a fire in 2019)
4. Arc de Triomphe - Famous monument honoring those who fought and died for France
5. Palace of Versailles - Opulent former royal residence with stunning gardens
6. Montmartre - Charming hilltop neighborhood known for its artistic history and Sacré-Cœur Basilica
7. Seine River Cruise - Scenic boat ride along the Seine River, offering views of the city's landmarks

Best time to visit Paris:

The best time to visit Paris depends on your preferences and priorities. Here's a breakdown by season:

- Spring (March to May): Mild weather, beautiful blooms, and fewer crowds compared to summer. Ideal for outdoor activities and sightseeing.
- Summer (June to August): Warm to hot weather, long days, and peak tourist season. Perfect for outdoor events and festivals, but expect larger crowds and higher prices.
- Fall (September to November): Pleasant weather, fewer tourists, and beautiful autumn foliage. Great for sightseeing, cultural events, and food festivals.

- Winter (December to February): Cold weather, shorter days, and festive holiday decorations. Suitable for indoor activities, museums, and Christmas markets. Expect lower prices and fewer crowds.

Ultimately, the shoulder seasons of spring and fall offer a good balance of pleasant weather, manageable crowds, and reasonable prices. However, Paris is a year-round destination with unique charms in every season.

Intelligent Agent Performing Tasks with Multiple Tools

In this section, we are going to create an intelligent agent that can perform tasks using multiple tools. This agent will search for information online and look up specific data from a preloaded index.

Setup: Introducing LangSmith

Building agents can be tricky, especially when it comes to debugging and observability. That is where LangSmith comes in to make the process of building and debugging agents easy by tracing all the steps in your agent's workflow. To set up LangSmith, you just need to set a couple of environment variables:

```
export LANGCHAIN_TRACING_V2="true"
export LANGCHAIN_API_KEY="<your-api-key>"
```

Make sure to replace `<your-api-key>` with your actual LangSmith API key.

Defining Tools

We will equip your agent with two powerful tools – Tavily for online search and a retriever for querying a local index.

Tool 1: Tavily

Tavily is a built-in tool in LangChain that allows your agent to search the Web effortlessly and gives access to a vast knowledge base. To use Tavily, you will need an API key. They offer a free tier, but if you don't have one or don't want to create one, feel free to skip this step.

Once you have your Tavily API key, export it as an environment variable:

```
export TAVILY_API_KEY="..."
```

Now, let us create an instance of the TavilySearchResults tool:

```
from langchain_community.tools.tavily_search import
TavilySearchResults

search = TavilySearchResults()
```

You can easily invoke the search tool with a query:

```
search.invoke("what is the weather in San Francisco")
```

This will return a list of search results related to the weather in San Francisco.

Tool 2: Retriever

In addition to online search, create a retriever that allows your agent to look up information from a local index. This is particularly useful when you have specific data that you want your agent to access quickly.

To create the retriever, follow these steps:

1. Load the data using WebBaseLoader:

```
from langchain_community.document_loaders import
WebBaseLoader

loader = WebBaseLoader("https://docs.smith.
langchain.com/overview")
docs = loader.load()
```

2. Split the loaded documents into chunks using `RecursiveCharacterTextSplitter`:

```
from langchain_text_splitters import
RecursiveCharacterTextSplitter

documents = RecursiveCharacterTextSplitter(
    chunk_size=1000, chunk_overlap=200
).split_documents(docs)
```

3. Create a vector store using FAISS and `OpenAIEmbeddings`:

```
from langchain_community.vectorstores
import FAISS
from langchain_openai import OpenAIEmbeddings

vector = FAISS.from_documents(documents,
OpenAIEmbeddings())
retriever = vector.as_retriever()
```

Now you have a retriever that can search for information within the indexed documents. You can invoke the retriever with a query:

```
retriever.invoke("how to upload a dataset")[0]
```

This will return the most relevant document chunk based on the query.

Creating a Retriever Tool

To make it easier for your agent to use the retriever, we can convert it into a tool using the `create_retriever_tool` function from `LangChain`:

```
from langchain.tools.retriever import create_retriever_tool

retriever_tool = create_retriever_tool(
```

```

    retriever,
    "langsmith_search",
    "Search for information about LangSmith. For any questions
    about LangSmith, you must use this tool!",
)

```

This creates a retriever tool with a specific name and description, making it more intuitive for your agent to understand when and how to use it.

Putting It All Together

Now that you have your tools ready, create a list that includes both the Tavily search tool and the retriever tool:

```
tools = [search, retriever_tool]
```

This allows your agent to perform online searches and look up information from a local index.

Choosing the LLM

The next step is to choose the language model (LLM) that will serve as the brain of your agent. In this example, you will be using an OpenAI Functions agent, which is a powerful and versatile option.

To get started, import the `ChatOpenAI` class from the `langchain_openai` module and create an instance of the LLM:

```

from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-3.5-turbo-0125", temperature=0)

```

Here, you are using the `gpt-3.5-turbo-0125` model with a temperature of 0. Feel free to adjust these parameters based on your specific requirements.

Selecting the Prompt

Next, you need to choose the prompt that will guide your agent's behavior. The prompt plays a crucial role in shaping how the agent interacts with the tools and processes input.

If you have access to LangSmith, you can explore the contents of a predefined prompt by visiting <https://smith.langchain.com/hub/hwchase17/openai-functions-agent>.

Alternatively, you can use the hub module from LangChain to pull the prompt:

```
from langchain import hub

prompt = hub.pull("hwchase17/openai-functions-agent")
prompt.messages
```

This will retrieve the prompt template and display its messages, which include system messages, placeholders for chat history and agent scratchpad, and human input.

Initializing the Agent

Now, it is time to initialize your agent by combining the LLM, the prompt, and the tools we defined earlier. The agent is responsible for taking input and deciding what actions to take based on that input. It is important to note that the agent itself does not execute the actions – that is the job of the AgentExecutor, which we will discuss in the next step.

To create the agent, you will use the `create_tool_calling_agent` function from the `langchain.agents` module:

```
from langchain.agents import create_tool_calling_agent

agent = create_tool_calling_agent(llm, tools, prompt)
```

This function takes in the LLM, the list of tools, and the prompt and returns an initialized agent ready to tackle our tasks.

Creating the AgentExecutor

The final piece of the puzzle is the AgentExecutor, which brings together the agent (the brains) and the tools (the abilities). The AgentExecutor repeatedly calls the agent to determine the next action and then executes the corresponding tool.

To create an AgentExecutor, you will use the following code:

```
from langchain.agents import AgentExecutor

agent_executor = AgentExecutor(agent=agent, tools=tools,
                               verbose=True)
```

We pass in the initialized agent, the list of tools, and set verbose=True to enable detailed output during execution.

Let us put the agent to the test and see what it can do! You will run a few queries and observe how the agent handles them. Keep in mind that for now, these queries are stateless, meaning the agent won't remember previous interactions.

First, start with a simple greeting:

```
agent_executor.invoke({"input": "hi!"})
```

When you run this code, the agent will process the input and generate a response. Here is what I got:

```
[1m> Entering new AgentExecutor chain...[0m
[32;1m[1;3mHello! How can I assist you today?[0m

[1m> Finished chain.[0m

{'input': 'hi!', 'output': 'Hello! How can I assist you today?'}
As you can see, the agent responds with a friendly greeting,
```

asking how it can assist us. The `verbose=True` setting allows us to see the agent's thought process, indicated by the green text.

Now, let us ask the agent a more specific question related to LangSmith and testing:

```
agent_executor.invoke({"input": "how can langsmith help with testing?"})
```

Here is the response:

```
[1m> Entering new AgentExecutor chain...[0m
[32;1m[1;3m
Invoking: `langsmith_search` with `{'query': 'how can LangSmith help with testing'}`
```

```
[0m[33;1m[1;3mLangSmith is a platform for building production-grade LLM applications that can help with testing in the following ways:
```

1. **Tracing**: LangSmith provides tracing capabilities that allow you to closely monitor and evaluate your application during testing. You can log traces to track the behavior of your application and identify any issues.
2. **Evaluation**: LangSmith offers evaluation capabilities that enable you to assess the performance of your application during testing. This helps you ensure that your application functions as expected and meets the required standards.
3. **Production Monitoring & Automations**: LangSmith allows you to monitor your application in production and automate certain processes, which can be beneficial for testing different scenarios and ensuring the stability of your application.

4. **Prompt Hub**: LangSmith includes a Prompt Hub, a prompt management tool that can streamline the testing process by providing a centralized location for managing prompts and inputs for your application.

Overall, LangSmith can assist with testing by providing tools for monitoring, evaluating, and automating processes to ensure the reliability and performance of your application during testing phases.

```
[1m> Finished chain.[0m
```

```
{'input': 'how can langsmith help with testing?',  
 'output': 'LangSmith is a platform for building production-grade LLM applications that can help with testing in the following ways:\n\n1. Tracing: LangSmith provides tracing capabilities that allow you to closely monitor and evaluate your application during testing. You can log traces to track the behavior of your application and identify any issues.\n\n2. Evaluation: LangSmith offers evaluation capabilities that enable you to assess the performance of your application during testing. This helps you ensure that your application functions as expected and meets the required standards.\n\n3. Production Monitoring & Automations: LangSmith allows you to monitor your application in production and automate certain processes, which can be beneficial for testing different scenarios and ensuring the stability of your application.\n\n4. Prompt Hub: LangSmith includes a Prompt Hub, a prompt management tool that can streamline the testing process by providing a centralized location for managing prompts and inputs for your application.\n\nOverall, LangSmith can assist with testing by providing tools for monitoring, evaluating, and automating processes to ensure the reliability and performance of your application during testing phases.'}
```

In this example, the agent invokes the `langsmith_search` tool with the query “how can LangSmith help with testing.” It retrieves relevant information from the indexed documents and generates a detailed response explaining how LangSmith’s features, such as tracing, evaluation, production monitoring, and the Prompt Hub, can aid in testing LLM applications.

Finally, let us ask the agent about the weather in San Francisco:

```
agent_executor.invoke({"input": "whats the weather in sf?"})
```

Here is the response:

```
[1m> Entering new AgentExecutor chain...[0m
[32;1m[1;3m
Invoking: `tavily_search_results_json` with `{'query': 'weather
in San Francisco'}`

[0m[36;1m[1;3m[{'url': 'https://www.weatherapi.com/',
'content': '{"location': {'name': 'San Francisco', 'region':
'California', 'country': 'United States of America', 'lat':
37.78, 'lon': -122.42, 'tz_id': 'America/Los_Angeles',
'localtime_epoch': 1712847697, 'localtime': '2024-04-11
8:01'}, 'current': {'last_updated_epoch': 1712847600, 'last_
updated': '2024-04-11 08:00', 'temp_c': 11.1, 'temp_f': 52.0,
'is_day': 1, 'condition': {'text': 'Partly cloudy', 'icon':
'//cdn.weatherapi.com/weather/64x64/day/116.png', 'code':
1003}, 'wind_mph': 2.2, 'wind_kph': 3.6, 'wind_degree': 10,
'wind_dir': 'N', 'pressure_mb': 1015.0, 'pressure_in': 29.98,
'precip_mm': 0.0, 'precip_in': 0.0, 'humidity': 97, 'cloud':
25, 'feelslike_c': 11.5, 'feelslike_f': 52.6, 'vis_km': 14.0,
'vis_miles': 8.0, 'uv': 4.0, 'gust_mph': 2.8, 'gust_kph':
4.4}}"}][0m[32;1m[1;3mThe current weather in San Francisco is
partly cloudy with a temperature of 52.0°F (11.1°C). The wind
```

```
speed is 3.6 kph coming from the north, and the humidity is  
at 97%.[0m
```

```
[1m> Finished chain.[0m
```

```
{'input': 'whats the weather in sf?',  
 'output': 'The current weather in San Francisco is partly  
cloudy with a temperature of 52.0°F (11.1°C). The wind speed is  
3.6 kph coming from the north, and the humidity is at 97%.'}
```

For this query, the agent invokes the `tavily_search_results_json` tool to search for weather information in San Francisco. It retrieves the relevant data from the search results and presents a concise summary of the current weather conditions, including temperature, wind speed, and humidity.

These examples demonstrate how our agent can handle different types of queries and use the appropriate tools to generate informative responses.

In this quick start, we covered the basics of creating a simple agent and progressively enhanced it with memory capabilities. We learned how to pass in chat history and structure messages using `AIMessage` and `HumanMessage`.

Try It Yourself

You can dive deeper into different types of agents, experiment with various prompts, and integrate additional tools to expand your agent's capabilities.

Differences Between LangChain v0.1 and v0.2 Agents

As you explore LangChain Agents further, it is important to understand the differences between the two major versions: v0.1 and v0.2.

In LangChain v0.1, Agents were introduced as a powerful tool for building AI applications. They provided a foundation for creating dynamic and adaptable Agents that could interact with various tools and generate responses based on user queries. However, as the library evolved, the LangChain team made significant enhancements to the Agent framework in v0.2.

Let us take a closer look at the major differences between LangChain v0.1 and v0.2 Agents:

1. Simplified Agent Initialization

- In v0.1, initializing an Agent required specifying the Agent type, tools, language model, and other parameters separately.
- With v0.2, the process has been streamlined. You can now use the `initialize_agent` function, which automatically selects the appropriate Agent type based on the provided tools and language model.

2. Enhanced Agent Types

- LangChain v0.2 introduces new and improved Agent types, such as the `zero-shot-react-description` Agent, which provides better support for natural language descriptions of actions.
- The `conversational-react-description` Agent has been optimized for conversational AI applications for more seamless and coherent dialogues.

3. Improved Tool Integration

- v0.2 simplifies the process of integrating tools with Agents. The `load_tools` function now supports a wider range of tools out of the box, making it easier to extend the capabilities of your Agents.

- Custom tools can be created by subclassing the `BaseTool` class and defining the necessary methods, providing greater flexibility for incorporating domain-specific functionality.
4. Enhanced Error Handling and Debugging
- LangChain v0.2 introduces better error handling mechanisms, making it easier to diagnose and fix issues in your Agent implementations.
 - The `verbose` parameter in the `initialize_agent` function allows you to enable detailed logging to understand the Agent's decision-making process and identify potential problems.

Simplified Agent Initialization in LangChain v0.2

Here is an illustrative example that showcases the simplified Agent initialization in v0.2:

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.llms import OpenAI

# Load the necessary tools
tools = load_tools(["serpapi", "llm-math"],
llm=OpenAI(temperature=0))

# Initialize the Agent
agent = initialize_agent(tools, OpenAI(temperature=0),
agent="zero-shot-react-description", verbose=True)
```

```
# Run the Agent with a query
query = "What is the capital of France? What is the population
of that city?"
response = agent.run(query)
print(response)
```

As you can see, the `initialize_agent` function takes care of selecting the appropriate Agent type based on the provided tools and language model, making the initialization process more intuitive and straightforward.

Key Takeaways

In this chapter, we explored the process of designing and implementing various types of agents using LangChain. You learned how to define clear objectives for your agents, understand their core concepts, and leverage a range of tools and toolkits to enhance their capabilities. We delved into different agent types, including Zero-Shot-React agents, structured chat agents, and ReAct agents, providing practical code examples for each. Additionally, we covered the importance of adding memory to your agents, allowing them to engage in more natural, context-aware conversations.

By now, you should have a solid foundation for building, customizing, and deploying intelligent agents tailored to your specific needs. These agents can autonomously handle complex tasks, making your AI applications more efficient and effective.

Review Questions

Let us test your understanding of this chapter's content.

1. What is the first step in designing an agent?
 - A. Implementing the agent's memory
 - B. Defining the agent's objectives
 - C. Loading language models
 - D. Creating custom tools
2. Which of the following is a core concept of an agent in LangChain?
 - A. AgentHandler
 - B. AgentAction
 - C. AgentLoader
 - D. AgentManager
3. What type of agent is designed for conversational AI applications?
 - A. Zero-Shot-React Agent
 - B. Structured Tool Agent
 - C. ReAct Agent
 - D. Conversation Agent
4. Which function is used to initialize an agent in LangChain v0.2?
 - A. load_agent
 - B. start_agent
 - C. initialize_agent
 - D. create_agent

5. What is the benefit of adding memory to an agent?
 - A. It reduces the agent's response time.
 - B. It allows the agent to remember previous interactions and provide context-aware responses.
 - C. It enables the agent to perform parallel processing.
 - D. It simplifies the agent's codebase.
6. Which tool allows an agent to search the Web for information?
 - A. LLMTool
 - B. SerpAPI
 - C. OpenAITool
 - D. WikiQuery
7. What is the role of the AgentExecutor in LangChain?
 - A. It initializes the agent.
 - B. It handles tool loading.
 - C. It runs the agent and executes the actions it chooses.
 - D. It manages the agent's memory.
8. Which agent type is optimized for handling tools with multiple inputs?
 - A. Zero-Shot-React Agent
 - B. ReAct Agent
 - C. Structured Chat Agent
 - D. Self-Ask Agent

9. What is an important consideration when describing tools for an agent?
 - A. Making the tool names as short as possible
 - B. Providing clear and concise descriptions
 - C. Using complex JSON schemas
 - D. Limiting the number of tools available
10. What is the main advantage of using LangGraph for agents?
 - A. Simplifies the process of integrating APIs
 - B. Allows agents to handle complex workflows using graph structures
 - C. Reduces the need for memory integration
 - D. Enhances the speed of language model processing

Answers

1. B
2. B
3. D
4. C
5. B
6. B
7. C

8. C
9. B
10. B

Further Reading

These references will provide you with in-depth knowledge and practical examples to help you understand and implement various types of LangChain agents effectively:

1. Tool Calling Agents

Tool Calling Agents: Information on how to set up and use tool calling agents to handle various tasks dynamically.

https://python.langchain.com/v0.1/docs/modules/agents/agent_types/tool_calling/

2. OpenAI Tools Agents

OpenAI Tools: Detailed documentation on utilizing OpenAI Tools agents for various functions and integrating them with your projects.

https://python.langchain.com/v0.1/docs/modules/agents/agent_types/openai_tools/

3. Structured Chat Agents

Structured Chat Agent Guide: Step-by-step guide on implementing structured chat agents for handling multi-input tools and managing complex interactions.

https://python.langchain.com/v0.1/docs/modules/agents/agent_types/structured_chat/

4. ReAct Agents

ReAct Agent Guide: Detailed instructions and examples for setting up and using ReAct agents to implement reasoning and action capabilities.

5. LangGraph for Enhanced Capabilities

LangGraph Documentation: Learn how to use LangGraph to create more complex and capable agents by leveraging graph structures.

<https://blog.langchain.dev/langgraph/>

6. Agent Types Overview

Types of LangChain Agents: Overview of different types of agents available in LangChain and their specific use cases.

https://python.langchain.com/v0.1/docs/modules/agents/agent_types/

CHAPTER 10

Projects: Building Agent Apps for Common Use Cases

In this chapter, we will explore how to create custom agents using LangChain. By the end of this chapter, you will have a solid understanding of how to load language models, define tools, create prompts, and bind everything together to build a functional agent. We will also cover practical use cases like customer support automation, personalized recommendations, and real-time data analysis and decision-making.

Creating a Custom Agent

Now, let us work on creating a custom agent.

Loading the Language Model

The first step is to load the language model. In this case, you will be using OpenAI's ChatOpenAI model, but feel free to experiment with other models as you progress:

```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)
```

Defining Tools

Next up, you need to equip your agent with tools. Let us start with a simple Python function that calculates the length of a given word:

```
from langchain.agents import tool

@tool
def get_word_length(word: str) -> int:
    """Returns the length of a word."""
    return len(word)

get_word_length.invoke("abc") # Output: 3
```

Pay close attention to the docstring here. It serves as a crucial guide for your agent to understand how to use the tool effectively.

Now, create a list to hold all the tools your agent will have at its disposal:

```
tools = [get_word_length]
```

Creating the Prompt

With the language model and tools in place, it is time to craft the prompt that will guide your agent's behavior. Thanks to OpenAI function calling, you can keep things simple and focus on the essential information your agent needs:

```
from langchain_core.prompts import ChatPromptTemplate,
MessagesPlaceholder

prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "You are a very powerful assistant, but don't know
            current events",
        ),
        ("user", "{input}"),
        MessagesPlaceholder(variable_name="agent_scratchpad"),
    ]
)
```

You have provided your agent with a brief system message, a placeholder for the user's input, and a placeholder for the agent's scratchpad (a space to store intermediate steps and tool outputs).

Binding Tools to the Language Model

Then you must bind your tools to the language model which helps your agent know what capabilities it has at its disposal:

```
llm_with_tools = llm.bind_tools(tools)
```

Creating the Agent

Import a couple of utility functions to help format your agent's intermediate steps and parse its output:

```
from langchain.agents.format_scratchpad.openai_tools import
format_to_openai_tool_messages
from langchain.agents.output_parsers.openai_tools import
OpenAIToolsAgentOutputParser

agent = (
    {
        "input": lambda x: x["input"],
        "agent_scratchpad": lambda x: format_to_openai_tool_
messages(
            x["intermediate_steps"]
        ),
    }
    | prompt
    | llm_with_tools
    | OpenAIToolsAgentOutputParser()
)
```

And just like that, your agent is ready!

```
from langchain.agents import AgentExecutor

agent_executor = AgentExecutor(agent=agent, tools=tools,
verbose=True)
```

You have created an `AgentExecutor` to interact with your agent and see its thought process in action.

Testing Your Agent

Now, let us put your agent to the test! You will ask it a simple question and observe how it performs:

```
list(agent_executor.stream({"input": "How many letters in the
word first"}))
```

You should see output similar to this:

```
> Entering new AgentExecutor chain...
Invoking: `get_word_length` with `{'word': 'eudca'}`
5
There are 5 letters in the word "first".
> Finished chain.
```

Awesome! Your agent successfully used the `get_word_length` tool to answer your question.

Adding Memory

But what if you want your agent to remember previous interactions and engage in a more natural conversation?

First, you need to add a placeholder for chat history in your prompt:

```
MEMORY_KEY = "chat_history"
prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "You are a very powerful assistant, but bad at
            calculating lengths of words.",
        ),
    ],
```

```

        MessagesPlaceholder(variable_name=MEMORY_KEY),
        ("user", "{input}"),
        MessagesPlaceholder(variable_name="agent_scratchpad"),
    ]
)

```

Next, set up a list to track the chat history:

```

from langchain_core.messages import AIMessage, HumanMessage

chat_history = []

```

Finally, update your agent and AgentExecutor to include the chat history:

```

agent = (
    {
        "input": lambda x: x["input"],
        "agent_scratchpad": lambda x: format_to_openai_tool_
messages(
            x["intermediate_steps"]
        ),
        "chat_history": lambda x: x["chat_history"],
    }
    | prompt
    | llm_with_tools
    | OpenAIToolsAgentOutputParser()
)

agent_executor = AgentExecutor(agent=agent, tools=tools,
verbose=True)

```

Now, when interacting with your agent, track the inputs and outputs as chat history:

```
input1 = "how many letters in the word first?"
result = agent_executor.invoke({"input": input1, "chat_
history": chat_history})
chat_history.extend(
    [
        HumanMessage(content=input1),
        AIMessage(content=result["output"]),
    ]
)
agent_executor.invoke({"input": "is that a real word?", "chat_
history": chat_history})
```

Your agent can now engage in a back-and-forth conversation, remembering previous interactions and providing contextual responses.

You have just created your very own custom agent using LangChain.

Practical Use Cases for Agents

Let us review some of the practical use cases of agents.

Customer Support Automation

In this section, let us explore how you can use agents to automate and streamline your customer support processes. If you have a business with a growing customer base, and you want to provide top-notch support without overwhelming your team, this use case can help. You can create a self-service support system that handles common customer inquiries efficiently and effectively.

Let us review the steps in the process:

1. Identify common customer inquiries:
 - Analyze your customer support data to identify the most frequently asked questions and common issues.
 - Categorize these inquiries into different topics or themes, such as product information, troubleshooting, billing, etc.
2. Create a knowledge base:
 - Compile a comprehensive knowledge base that covers the identified topics and provides clear, concise answers to common questions.
 - Organize the knowledge base into a structured format, such as FAQ pages, product documentation, or troubleshooting guides.
3. Set up the agent:
 - Use the LangChain agent framework to build your customer support automation system.
 - Install and import the necessary libraries:

```
%pip install --upgrade --quiet langchain-openai
tavily-python langchain_community langchain_openai
from langchain.agents import initialize_agent, Tool
from langchain.llms import OpenAI
from langchain.chains import ConversationChain
from langchain.memory import
ConversationBufferMemory
```

4. Define the tools and memory:

- Create tools that allow the agent to access and retrieve information from your knowledge base:

```
def search_knowledge_base(query):
    # Implement logic to search the knowledge base
    # based on the query
    # Return the most relevant answer or
    # information
    pass

tools = [
    Tool(
        name="Knowledge Base Search",
        func=search_knowledge_base,
        description="Useful for searching the
        knowledge base for answers to customer
        inquiries."
    )
]
```

- Set up a memory object to store the conversation history:

```
memory = ConversationBufferMemory(memory_key=
"chat_history")
```

5. Initialize the agent:

```
agent = initialize_agent(
    tools,
    OpenAI(temperature=0),
    agent="conversational-react-description",
```

```

        verbose=True,
        memory=memory
    )

```

6. Integrate the agent into your customer support channels:

- Implement a user interface, such as a chatbot or a webform, where customers can interact with the agent.
- Use the agent to handle incoming customer inquiries:

```

def handle_inquiry(inquiry):
    response = agent.run(inquiry)
    return response

# Example usage
customer_inquiry = "How can I reset my account password?"
response = handle_inquiry(customer_inquiry)
print(response)

```

7. Monitor and improve:

- Continuously monitor the performance of your customer support automation system.
- Collect feedback from customers and analyze the effectiveness of the agent's responses.
- Iterate and improve the knowledge base and agent's capabilities based on the feedback and identified areas for enhancement.

By implementing a customer support automation system using agents, you can

- Provide instant, 24/7 support to your customers
- Reduce the workload on your support team by handling common inquiries automatically
- Ensure consistent and accurate responses to customer questions
- Scale your support capabilities as your customer base grows

Here is a complete code example that demonstrates a basic customer support automation:

```
%pip install --upgrade --quiet langchain-openai tavily-python
langchain_community langchain_openai
from langchain.agents import initialize_agent, Tool
from langchain.llms import OpenAI
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory

import os
from dotenv import load_dotenv

# Load environment variables from the .env file
load_dotenv()

# Get the OpenAI API key from the environment variable
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")

# Import the new Chat Completion API:
os.environ["OPENAI_API_KEY"] = "Your openAI key"

# Define the knowledge base search function
def search_knowledge_base(query):
```

```

# Implement logic to search the knowledge base based on
the query
# Return the most relevant answer or information
# Here's a simple example that returns a predefined
response
if "password reset" in query.lower():
    return "To reset your account password, please follow
these steps:\n1. Go to the login page.\n2. Click on the
'Forgot Password' link.\n3. Enter your registered email
address.\n4. Check your email inbox for a password
reset link.\n5. Follow the instructions in the email to
create a new password."
else:
    return "I apologize, but I couldn't find a specific
answer to your question in our knowledge base. Please
provide more details or contact our support team
directly for further assistance."

# Set up the tools and memory
tools = [
    Tool(
        name="Knowledge Base Search",
        func=search_knowledge_base,
        description="Useful for searching the knowledge base
for answers to customer inquiries."
    )
]
memory = ConversationBufferMemory(memory_key="chat_history")

# Initialize the agent
agent = initialize_agent(
    tools,

```

```

OpenAI(temperature=0),
agent="conversational-react-description",
verbose=True,
memory=memory
)

# Handle customer inquiries
def handle_inquiry(inquiry):
    response = agent.run(inquiry)
    return response

# Example usage
customer_inquiry = "How can I reset my account password?"
response = handle_inquiry(customer_inquiry)
print(response)

```

In this example, you define a simple `search_knowledge_base` function that returns predefined responses based on the customer's inquiry. The function can be enhanced to search an actual knowledge base or database for more dynamic responses.

The agent is initialized with the knowledge base search tool and a memory object to store the conversation history. The `handle_inquiry` function takes a customer inquiry as input, passes it to the agent, and returns the generated response.

When you run this code with the example customer inquiry, it will output the predefined response for resetting the account password such as shown below:

Entering new AgentExecutor chain...

Thought: Do I need to use a tool? Yes

Action: Knowledge Base Search

Action Input: "reset account password"

Observation: ***I apologize, but I couldn't find a specific answer to your question in our knowledge base. Please provide more details or contact our support team directly for further assistance.***

Thought: ***Do I need to use a tool? No***

AI: No problem, I can help you reset your account password. Please provide me with your email address and I will send you instructions on how to reset your password.

> Finished chain.

No problem, I can help you reset your account password. Please provide me with your email address and I will send you instructions on how to reset your password.

Congratulations, you just created a customer support automation agent!

Personalized Recommendations

In this section, we will explore how you can leverage agents to build a personalized recommendation system that thrills your users. Whether you are working on an ecommerce platform, a content streaming service, or any application where personalization is key, this use case can help.

Let us dive into the step-by-step process of creating a personalized recommendation system using agents.

1. Collect user data:

- Gather relevant information about your users, such as their preferences, behavior, and interaction history within your application.
- This data can include user profiles, browsing history, purchase records, ratings, and reviews.
- Ensure that you comply with data privacy regulations and obtain necessary user consent.

Here is a sample data for illustrative purposes:

```
user_preferences_data = {
    "1234": {
        "favorite_genres": ["Action", "Sci-Fi"],
        "favorite_actors": ["Tom Cruise", "Brad Pitt"],
        "favorite_directors": ["Christopher Nolan"]
    }
    # Add more user data as needed
}
```

2. Preprocess and analyze the data:

- Clean and preprocess the collected user data to ensure its quality and consistency.
- Perform exploratory data analysis to gain insights into user patterns, preferences, and trends.
- Identify key features or attributes that can be used to generate personalized recommendations.

3. Set up the agent:

- Build your personalized recommendation system using LangChain.
- Install and import the necessary libraries:

```
from langchain.agents import initialize_agent, Tool
from langchain.llms import OpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
```

4. Define the recommendation tools:

- Create tools that allow the agent to access and analyze user data, as well as generate personalized recommendations:

```
# Recommendation tool functions
def user_preference_tool(user_id):
    return user_preferences_data.get(user_id, {})

def recommendation_generator_tool(user_
preferences):
    # Generate personalized recommendations based
    on user preferences
    recommendations = []
    if user_preferences:
        favorite_genres = user_preferences.
        get("favorite_genres", [])
        favorite_actors = user_preferences.
        get("favorite_actors", [])
        favorite_directors = user_preferences.
        get("favorite_directors", [])
        if favorite_genres:
            recommendations.append(f"Based on your
            favorite genres ({', '.join(favorite_
            genres)}), we recommend:")
            recommendations.append("1. Inception
            (Action, Sci-Fi)")
            recommendations.append("2. The Matrix
            (Action, Sci-Fi)")
            recommendations.append("3. Guardians of
            the Galaxy (Action, Comedy, Sci-Fi)")
```

```

if favorite_actors:
    recommendations.append(f"Considering your
    favorite actors ({', '.join(favorite_
    actors)}), you might enjoy:")
    recommendations.append("1. Mission:
    Impossible - Fallout (starring Tom
    Cruise)")
    recommendations.append("2. Once
    Upon a Time in Hollywood (starring
    Brad Pitt)")
    recommendations.append("3. The Hunger
    Games (starring Jennifer Lawrence)")
if favorite_directors:
    recommendations.append(f"Given your
    appreciation for {' and '.join(favorite_
    directors)}}, we suggest:")
    recommendations.append("1. The Dark
    Knight Trilogy (directed by Christopher
    Nolan)")
    recommendations.append("2. Pulp Fiction
    (directed by Quentin Tarantino)")
    recommendations.append("3. Interstellar
    (directed by Christopher Nolan)")
if not recommendations:
    recommendations.append("Oops! We couldn't
    find personalized recommendations based on
    your preferences.")
    recommendations.append("Please provide more
    information about your favorite genres,
    actors, or directors.")
return "\n".join(recommendations)

```

5. Set up the tools:

```
tools = [
    Tool(
        name="User Preference Tool",
        func=user_preference_tool,
        description="Retrieves user preferences based
        on the user ID."
    ),
    Tool(
        name="Recommendation Generator Tool",
        func=recommendation_generator_tool,
        description="Generates personalized
        recommendations based on user preferences."
    )
]
```

6. Set up the recommendation prompt:

- Define a prompt template that guides the agent in generating personalized recommendations:

```
recommendation_prompt = PromptTemplate(
    input_variables=["user_id"],
    template="""
    Given the user ID {user_id}, retrieve their
    preferences and generate personalized
    recommendations.
    Provide a list of top recommendations
    along with a brief explanation for each
    recommendation.
    """)
```

7. Initialize the agent:

```

recommendation_agent = initialize_agent(
    tools,
    OpenAI(temperature=0.7),
    agent="zero-shot-react-description",
    verbose=True
)

```

8. Generate personalized recommendations:

- Use the agent to generate personalized recommendations for a specific user:

```

# Generate personalized recommendations
def generate_recommendations(user_id):
    user_preferences = user_preference_
    tool(user_id)
    recommendations = recommendation_generator_
    tool(user_preferences)
    return recommendations

# Example usage
user_id = "1234"
recommendations = generate_recommendations(user_id)
print(recommendations)

```

9. Here is an example usage:

```

user_id = "1234"
recommendations = generate_recommendations(user_id)
print(recommendations)

```

10. Integrate the recommendations into your application:

- Display the personalized recommendations to the user in a visually appealing and intuitive manner.
- Consider factors such as the placement, timing, and format of the recommendations to maximize their impact.
- Monitor user engagement and gather feedback to continuously improve the recommendation system.

Here is a complete code example that demonstrates a basic personalized recommendation system:

```
from langchain.agents import initialize_agent, Tool
from langchain.llms import OpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate

# Dummy user preferences data
user_preferences_data = {
    "1234": {
        "favorite_genres": ["Action", "Sci-Fi", "Comedy"],
        "favorite_actors": ["Tom Cruise", "Brad Pitt",
            "Jennifer Lawrence"],
        "favorite_directors": ["Christopher Nolan", "Quentin
            Tarantino"]
    }
}

# Recommendation tool functions
def user_preference_tool(user_id):
    return user_preferences_data.get(user_id, {})
```

```

def recommendation_generator_tool(user_preferences):
    # Generate personalized recommendations based on user
    preferences
    recommendations = []
    if user_preferences:
        favorite_genres = user_preferences.get("favorite_
        genres", [])
        favorite_actors = user_preferences.get("favorite_
        actors", [])
        favorite_directors = user_preferences.get("favorite_
        directors", [])
        if favorite_genres:
            recommendations.append(f"Based on your favorite
            genres ({', '.join(favorite_genres)}), we
            recommend:")
            recommendations.append("1. Inception (Action,
            Sci-Fi)")
            recommendations.append("2. The Matrix (Action,
            Sci-Fi)")
            recommendations.append("3. Guardians of the Galaxy
            (Action, Comedy, Sci-Fi)")
        if favorite_actors:
            recommendations.append(f"Considering your favorite
            actors ({', '.join(favorite_actors)}), you might
            enjoy:")
            recommendations.append("1. Mission: Impossible -
            Fallout (starring Tom Cruise)")
            recommendations.append("2. Once Upon a Time in
            Hollywood (starring Brad Pitt)")
            recommendations.append("3. The Hunger Games
            (starring Jennifer Lawrence)")

```

```

    if favorite_directors:
        recommendations.append(f"Given your appreciation
        for {' and '}.join(favorite_directors)}, we
        suggest:")
        recommendations.append("1. The Dark Knight Trilogy
        (directed by Christopher Nolan)")
        recommendations.append("2. Pulp Fiction (directed
        by Quentin Tarantino)")
        recommendations.append("3. Interstellar (directed
        by Christopher Nolan)")
    if not recommendations:
        recommendations.append("Oops! We couldn't find
        personalized recommendations based on your
        preferences.")
        recommendations.append("Please provide more information
        about your favorite genres, actors, or directors.")
    return "\n".join(recommendations)

# Set up the tools
tools = [
    Tool(
        name="User Preference Tool",
        func=user_preference_tool,
        description="Retrieves user preferences based on the
        user ID."
    ),
    Tool(
        name="Recommendation Generator Tool",
        func=recommendation_generator_tool,
        description="Generates personalized recommendations
        based on user preferences."
    )
]

```

```

# Set up the recommendation prompt
recommendation_prompt = PromptTemplate(
    input_variables=["user_id"],
    template="""
    Given the user ID {user_id}, retrieve their preferences and
    generate personalized recommendations.
    Provide a list of top recommendations along with a brief
    explanation for each recommendation.
    """
)

# Initialize the agent
recommendation_agent = initialize_agent(
    tools,
    OpenAI(temperature=0.7),
    agent="zero-shot-react-description",
    verbose=True
)

# Generate personalized recommendations
def generate_recommendations(user_id):
    user_preferences = user_preference_tool(user_id)
    recommendations = recommendation_generator_tool(user_
    preferences)
    return recommendations

# Example usage
user_id = "1234"
recommendations = generate_recommendations(user_id)
print(recommendations)

```

In this example, we have a dummy user preference dataset (`user_preferences_data`) that maps user IDs to their favorite genres, actors, and directors. The `user_preference_tool` retrieves the preferences for a given user ID, while the `recommendation_generator_tool` generates personalized recommendations based on those preferences.

The agent is initialized with the recommendation tools and the recommendation prompt. The `generate_recommendations` function takes a user ID, retrieves their preferences, and generates personalized recommendations using the agent.

When you run this code with the example user ID, it will output a set of personalized movie recommendations based on the user's favorite genres, actors, and directors as shown below:

```
Based on your favorite genres (Action, Sci-Fi, Comedy), we recommend:
```

1. Inception (Action, Sci-Fi)
2. The Matrix (Action, Sci-Fi)
3. Guardians of the Galaxy (Action, Comedy, Sci-Fi)

```
Considering your favorite actors (Tom Cruise, Brad Pitt, Jennifer Lawrence), you might enjoy:
```

1. Mission: Impossible - Fallout (starring Tom Cruise)
2. Once Upon a Time in Hollywood (starring Brad Pitt)
3. The Hunger Games (starring Jennifer Lawrence)

```
Given your appreciation for Christopher Nolan and Quentin Tarantino, we suggest:
```

1. The Dark Knight Trilogy (directed by Christopher Nolan)
2. Pulp Fiction (directed by Quentin Tarantino)
3. Interstellar (directed by Christopher Nolan)

Remember, this is a simplified example to illustrate the concept. In a real-world scenario, you would integrate the agent with your actual user data, recommendation algorithms, and domain-specific knowledge to generate more accurate and diverse recommendations.

Real-Time Data Analysis and Decision-Making

In this use case example, you are leveraging the stream of data flowing into your system to make swift, informed decisions. Whether you are monitoring sensor readings, analyzing user behavior, or optimizing resource allocation, real-time data analysis and decision-making are crucial.

Let us dive into the step-by-step process of building an agent-powered real-time data analysis and decision-making system:

1. Identify the data sources:
 - Determine the sources of your real-time data, such as sensors, log files, APIs, or databases.
 - Ensure that you have the necessary permissions and access to retrieve the data in real time.
2. Set up data ingestion:
 - Implement a mechanism to continuously ingest the real-time data into your system.
 - This can involve setting up data pipelines, streaming frameworks, or event-driven architectures.
 - Consider tools like Apache Kafka, Apache Flink, or AWS Kinesis for efficient data ingestion.
3. Preprocess and transform the data:
 - Perform necessary data preprocessing and transformation steps to ensure data quality and consistency.

- Handle missing values, outliers, and data formatting issues.
 - Apply relevant feature engineering techniques to extract meaningful information from the raw data.
4. Set up the agent:
- Build your real-time data analysis and decision-making system using LangChain.
 - Install and import the necessary libraries:

```
from langchain.agents import initialize_agent, Tool
from langchain.llms import OpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
```

5. Define the data analysis tools:

- Create tools that allow the agent to access and analyze the real-time data:

```
# Dummy data retrieval and analysis functions
def data_retrieval_tool():
    # Retrieve the latest batch of real-time data
    # Return the data in a suitable format for
    # analysis
    # Here's a dummy example that returns
    # random data
    import random
    temperature = random.randint(20, 30)
    humidity = random.randint(40, 60)
    return f"Temperature: {temperature}°C,
    Humidity: {humidity}%"
```

```

def data_analysis_tool(data):
    # Perform data analysis tasks on the
    # provided data
    # Return insights, patterns, or anomalies
    # Here's a dummy example that checks for high
    # temperature and humidity
    try:
        temperature = int(data.split("Temperature: ")
                           [1].split("°C")[0])
        humidity = int(data.split("Humidity: ")[1].
                       split("%")[0])
        if temperature > 25 and humidity > 50:
            return "High temperature and humidity
                   detected. Adjustments may be
                   necessary."
        else:
            return "Temperature and humidity within
                   normal range. No action required."
    except (IndexError, ValueError):
        return "Error: Invalid data format. Unable
               to analyze."

# Set up the tools
tools = [
    Tool(
        name="Data Analysis Tool",
        func=data_analysis_tool,
        description="Performs data analysis tasks
                   on the provided data."
    )
]

```

6. Set up the decision-making prompt:

- Define a prompt template that guides the agent in making decisions based on the analyzed data:

```
decision_prompt = PromptTemplate(
    input_variables=["data_insights"],
    template="""
    Based on the data insights: {data_insights},
    make a decision on the appropriate action
    to take.
    Provide a clear and concise decision along with
    a brief justification.
    """)
```

7. Initialize the agent:

```
decision_agent = initialize_agent(
    tools,
    OpenAI(temperature=0.7),
    agent="zero-shot-react-description",
    verbose=True
)
```

8. Process real-time data and make decisions:

- Continuously retrieve the latest batch of real-time data.
- Use the agent to analyze the data and make decisions based on the insights:

```
# Process real-time data and make decisions
def process_data_and_make_decision():
```

```

    data = data_retrieval_tool()
    decision = decision_agent.run(decision_prompt.
format(data_insights=data))
    return decision

# Example usage
while True:
    decision = process_data_and_make_decision()
    print("Decision:", decision)
    # Perform actions based on the decision
    input("Press Enter to retrieve the next batch
of data...")

```

9. Integrate the decisions into your application:

- Use the decisions made by the agent to trigger appropriate actions or updates in your application.
- This can involve sending alerts, adjusting system parameters, or triggering automated processes.
- Monitor the impact of the decisions and continuously refine the agent's decision-making capabilities.

Here is a complete code example that demonstrates a basic real-time data analysis and decision-making system:

```

%pip install --upgrade --quiet langchain langchain-openai
langchain_community langchain_openai python-dotenv
import os
from dotenv import load_dotenv

# Load environment variables from the .env file
load_dotenv()

```

```

# Get the OpenAI API key from the environment variable
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")

# Import the new Chat Completion API:
os.environ["OPENAI_API_KEY"] = "your Open AI key"
from langchain.agents import initialize_agent, Tool
from langchain.llms import OpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate

# Dummy data retrieval and analysis functions
def data_retrieval_tool():
    # Retrieve the latest batch of real-time data
    # Return the data in a suitable format for analysis
    # Here's a dummy example that returns random data
    import random
    temperature = random.randint(20, 30)
    humidity = random.randint(40, 60)
    return f"Temperature: {temperature}°C, Humidity:
    {humidity}%"

def data_analysis_tool(data):
    # Perform data analysis tasks on the provided data
    # Return insights, patterns, or anomalies
    # Here's a dummy example that checks for high temperature
    and humidity
    try:
        temperature = int(data.split("Temperature: ")[1].
        split("°C")[0])
        humidity = int(data.split("Humidity: ")[1].
        split("%")[0])
        if temperature > 25 and humidity > 50:
            return "High temperature and humidity detected.

```

```

Adjustments may be necessary."
    else:
        return "Temperature and humidity within normal
            range. No action required."
    except (IndexError, ValueError):
        return "Error: Invalid data format. Unable to analyze."

# Set up the tools
tools = [
    Tool(
        name="Data Analysis Tool",
        func=data_analysis_tool,
        description="Performs data analysis tasks on the
            provided data."
    )
]

# Set up the decision-making prompt
decision_prompt = PromptTemplate(
    input_variables=["data_insights"],
    template="""
Based on the data insights: {data_insights},
make a decision on the appropriate action to take.
Provide a clear and concise decision along with a brief
justification.
"""
)

# Initialize the agent
decision_agent = initialize_agent(
    tools,
    OpenAI(temperature=0.7),

```

```

    agent="zero-shot-react-description",
    verbose=True
)

# Process real-time data and make decisions
def process_data_and_make_decision():
    data = data_retrieval_tool()
    decision = decision_agent.run(decision_prompt.format(data_
        insights=data))
    return decision

# Example usage
while True:
    decision = process_data_and_make_decision()
    print("Decision:", decision)
    # Perform actions based on the decision
    input("Press Enter to retrieve the next batch of data...")

```

In this example, you have dummy functions for data retrieval (`data_retrieval_tool`) and data analysis (`data_analysis_tool`). The `data_retrieval_tool` simulates retrieving real-time data by generating random temperature and humidity values. The `data_analysis_tool` performs a simple analysis by checking if the temperature and humidity exceed certain thresholds.

The agent is initialized with the data retrieval and analysis tools, along with the decision-making prompt. The `process_data_and_make_decision` function retrieves the latest data, analyzes it, and uses the agent to make a decision based on the insights.

The example usage demonstrates a continuous loop where the system retrieves data, makes decisions, and prompts the user to retrieve the next batch of data.

This is a simplified example to illustrate the concept. In a real-world scenario, you would integrate the agent with your actual real-time data sources, use more sophisticated data analysis techniques, and implement the necessary actions based on the decisions.

Key Takeaways

In this chapter, we successfully created a custom agent using LangChain. We explored loading language models, defining tools, and creating prompts. Additionally, we covered practical applications for agents, such as customer support automation, personalized recommendations, and real-time data analysis. By adding memory, we enhanced our agents to engage in more natural and coherent conversations.

Review Questions

Let us test your understanding of this chapter's content.

1. What is the first step in creating a custom agent?
 - A. Defining tools
 - B. Loading the language model
 - C. Creating prompts
 - D. Binding tools to the language model
2. Which function allows an agent to remember previous interactions?
 - A. AgentAction
 - B. ConversationBufferMemory
 - C. ChatPromptTemplate
 - D. AgentExecutor

3. What is a practical application of agents in customer support?
 - A. Generating random data
 - B. Handling common customer inquiries automatically
 - C. Managing system updates
 - D. Analyzing server logs
4. Which tool retrieves the latest batch of real-time data?
 - A. Data analysis tool
 - B. Data retrieval tool
 - C. Real-time data tool
 - D. Decision-making tool
5. What is the benefit of adding memory to your agent?
 - A. Faster response times
 - B. Enhanced context-aware interactions
 - C. Increased accuracy in calculations
 - D. Simplified codebase

Answers

1. B
2. B
3. B
4. B
5. B

Further Reading

These references will provide you with in-depth knowledge and practical examples to help you understand and implement various LangChain agent use cases effectively, from customer support automation to real-time data analysis and personalized recommendations:

1. Creating Custom Agents

This is a cookbook that shows you how to build a custom agent using LlamaIndex.

https://docs.llamaindex.ai/en/latest/examples/agent/custom_agent/

2. Defining and Using Tools in Agents

LangChain Tools Integration: Explore how to define and integrate various tools into your agents, enhancing their functionality and effectiveness.

<https://python.langchain.com/v0.2/docs/integrations/platforms/>

3. Crafting Prompts for Agents

Prompt Engineering with LangChain: Best practices and techniques for creating effective prompts that guide agent behavior and optimize responses.

https://python.langchain.com/v0.1/docs/modules/model_io/prompts/

4. Practical Use Cases and Examples

LangChain Use Cases: A collection of practical use cases and examples of LangChain agents in various applications, providing insights and inspiration for your projects.

https://js.langchain.com/v0.1/docs/use_cases/

CHAPTER 11

Building and Deploying a ChatGPT-like App Using Streamlit

In this chapter, we are going to use two powerful tools, namely, LangChain and Streamlit to develop a ChatGPT-like LangChain-based UI application. In particular, we will be transitioning from Jupyter Notebooks to a web application to create something that is not just functional but also user-friendly with sleek web interfaces and production-ready. Here is a sneak peek of what you will be doing:

1. Setting up our development environment
2. Building the core Q&A functionality with LangChain
3. Creating a web interface using Streamlit
4. Testing and refining your application

Setting Up Your Development Environment

Let us set up the development environment to run Streamlit in your desktop environment.

Installing Streamlit Library

Make sure that you have Streamlit installed in your Python environment. You can do so by running the following command in your terminal or command prompt:

```
pip install streamlit
```

If you haven't installed pip, you may see an error message similar to this:

```
C:\Users\rabij>!pip install streamlit
'!pip' is not recognized as an internal or external command,
operable program or batch file.
```

If you already have Streamlit installed, you can skip this step.

Installing Python

If you haven't installed Python in your desktop environment, you may get an error as shown below:

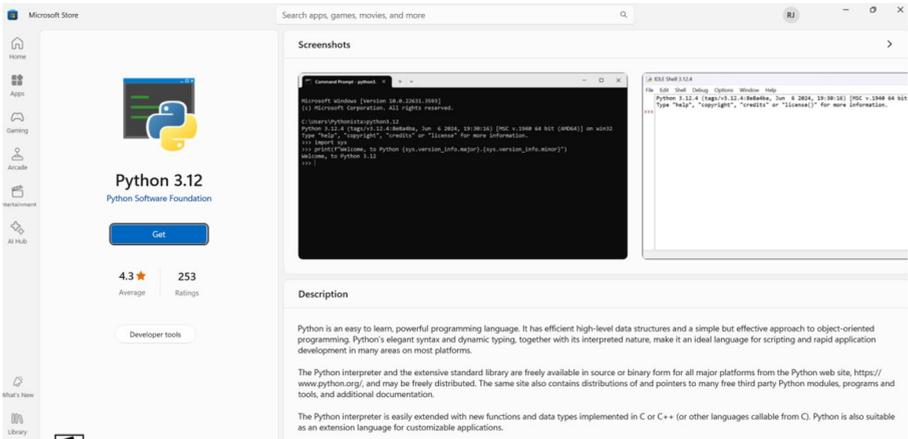
```
C:\Users\rabij>python --version
Python was not found; run without arguments to install from the Microsoft Store, or disable this shortcut from Settings
> Manage App Execution Aliases.
```

The error message suggests that the Python command is not recognized in your Command Prompt. This usually means that either Python is not installed or the Python installation directory is not added to the system's PATH environment variable.

To resolve this issue, you can try the following steps:

1. Check if Python is installed.

- Open a new Command Prompt window.
- Type `python --version` and press Enter.
- If Python is installed correctly, it will display the version number. If not, you need to install Python first.
- Install Python from the Microsoft Store as shown below:



2. Install pip.

- If you have Python installed but pip is missing, you can download the `get-pip.py` script from the official pip website: <https://bootstrap.pypa.io/get-pip.py>.
- Save the `get-pip.py` file to a location on your computer, such as `C:\Users\abc\get-pip.py`.

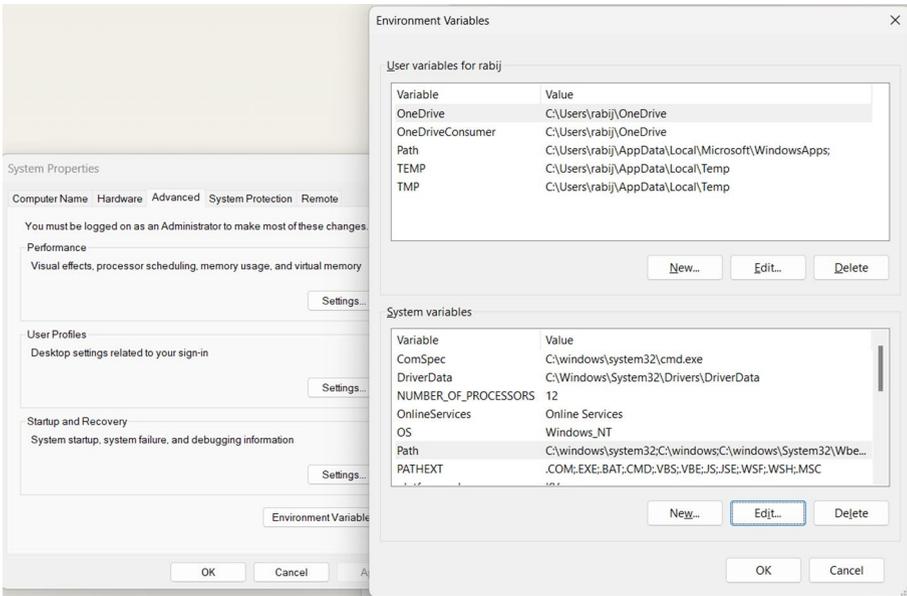
- Open a Command Prompt window and navigate to the directory where you saved the `get-pip.py` file using the `cd` command.
- Run the following command to install pip:

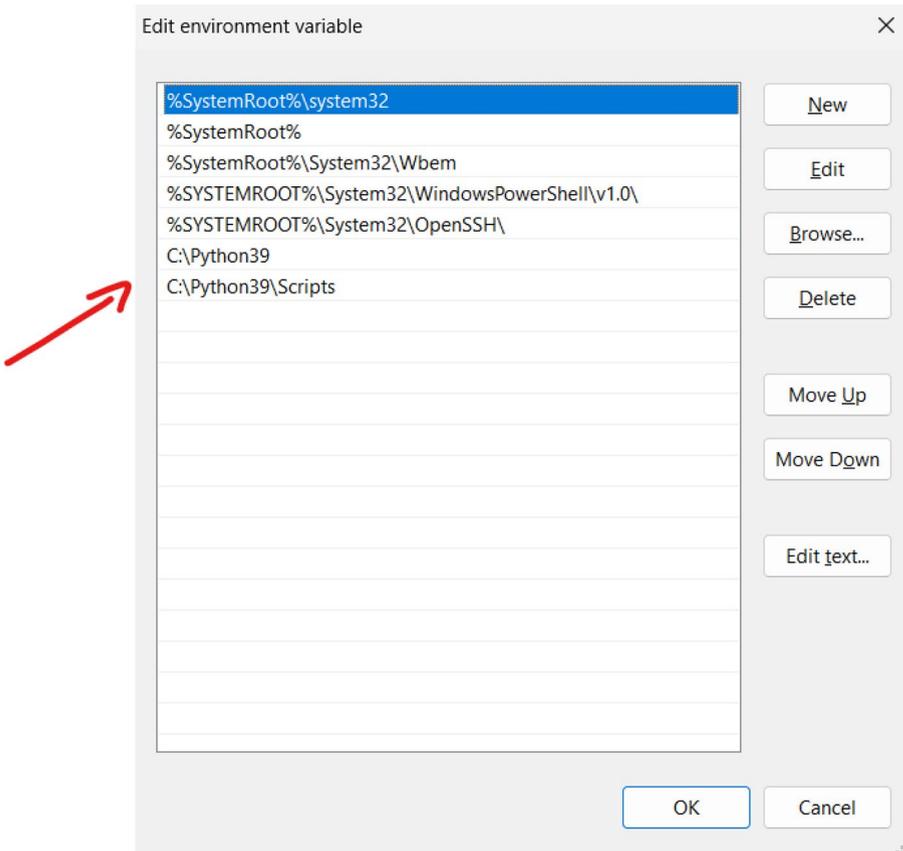
```
C:\Users\rabij\LangChain>python get-pip.py
Defaulting to user installation because normal site-packages is not writeable
Collecting pip
  Using cached pip-24.0-py3-none-any.whl.metadata (3.6 kB)
Using cached pip-24.0-py3-none-any.whl (2.1 MB)
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 24.0
    Uninstalling pip-24.0:
      Successfully uninstalled pip-24.0
Successfully installed pip-24.0
```

3. Add Python and pip to the system's PATH as shown in the figure below.

- Open the Start menu and search for “Environment Variables.”
- Click “Edit the system environment variables.”
- In the System Properties window, click the “Environment Variables” button.
- Under “System variables,” scroll down and find the “Path” variable, then click “Edit.”
- Click “New” and add the path to your Python installation directory, typically `C:\Python39` or similar.

- Click “New” again and add the path to the Scripts directory within your Python installation, typically C:\Python39\Scripts.
- Click “OK” to save the changes.





4. Verify the installation.

- Open a new Command Prompt window.
- Type `pip --version` and press Enter.
- If pip is installed and accessible, it will display the version number.

After completing these steps, you should be able to run the `pip install streamlit` command successfully in your Command Prompt.

If you still encounter issues, make sure you have the necessary permissions to install packages and that your Internet connection is stable. Additionally, you can try running the Command Prompt as an administrator by right-clicking the Command Prompt icon and selecting “Run as administrator.”

Installing Required Dependencies

Next, you must make sure you have the required dependencies installed (streamlit, openai, langchain, pinecone) before running the code. You can install them using pip:

```
pip install streamlit openai langchain pinecone-client
langchain_community
```

After making these changes and ensuring the dependencies are installed, you should be able to run the code on your desktop.

If you still encounter issues, make sure you have the latest versions of the required dependencies installed. You can update them using the following commands:

```
pip install --upgrade langchain langchain_openai langchain_
community openai streamlit pinecone-client
```

Additionally, ensure that you have a compatible version of Python installed (Python 3.7 or above).

Building the Streamlit LangChain UI App

Now that you have set up the development environment in your desktop, it is time for the real work to begin, which is building the actual Streamlit LangChain UI app.

Components of the Streamlit App

A typical Streamlit app consists of the `app.py` file which is a Python script that contains the code for building a Streamlit application. It defines the structure, layout, and functionality of your Streamlit app. In our case, the `app.py` file is named as `LangChainUI.py` file. The `LangChainUI.py` file is the main entry point where you import the necessary libraries, define the user interface components, and specify the logic and interactions of your application.

To run this Streamlit application, use the `streamlit run LangChainUI.py` command in your terminal or command prompt.

Streamlit provides a wide range of components and features that you can use to build interactive web applications, including widgets, charts, tables, maps, and more. The `LangChainUI.py` file serves as the central place where you define and organize these components to create your desired application.

Steps Involved in Building the App

Below are some of the steps involved when building the app:

1. First, import the necessary libraries:

```
import os
import streamlit as st
from langchain.chains import LLMChain
from langchain.prompts import ChatPromptTemplate,
HumanMessagePromptTemplate
from langchain.chat_models import ChatOpenAI
```

These lines import the necessary libraries: `os` for environment variables, `streamlit` for the web interface, and various components from `langchain` for interacting with the language model.

- Next, you must set the API key:

```
os.environ["OPENAI_API_KEY"] = "your_openai_api_key_here"
```

This sets the OpenAI API key as an environment variable, which will be used by the ChatOpenAI model.

- Streamlit UI setup

```
st.title("ChatGPT-like Q&A App")
user_query = st.text_input("Enter your question:")
```

Here, you are creating the title of the app and an input box for the user's question.

- Chat history initialization

```
if 'chat_history' not in st.session_state:
    st.session_state['chat_history'] = []
```

You then initialize an empty chat history if it doesn't exist in the session state.

- Displaying chat history

```
for chat in st.session_state['chat_history']:
    st.write(f"Q: {chat['question']}")
    st.write(f"A: {chat['answer']}")
    st.write("---")
```

You then loop display all previous questions and answers stored in the chat history.

- Handling user input

```
if user_query and st.button("Submit"):
```

You check if there is a user query, and the submit button is clicked.

7. Creating and using the language model

```
llm = ChatOpenAI(temperature=0.7, model_
name='gpt-3.5-turbo')
prompt = ChatPromptTemplate.from_messages([
    HumanMessagePromptTemplate.from_template("{query}")
])
chain = LLMChain(llm=llm, prompt=prompt)
response = chain.run(query=user_query)
```

Next, you create an instance of the ChatOpenAI model, set up a prompt template, create an LLMChain, and generate a response to the user's query.

8. Updating and displaying the response

```
st.session_state['chat_history'].append({"question":
user_query, "answer": response})
st.write("Answer:")
st.write(response)
```

You add the new Q&A pair to the chat history and display the response.

9. Fallback message

```
else:
    st.write("Please enter a question and click
Submit.")
```

Following this, you display a message if no question is entered or the submit button isn't clicked.

Finally, you create a simple web-based chat interface that uses the OpenAI GPT model to generate responses to user queries, maintains a chat history, and displays all interactions.

Indentation Error in the Code

Sometimes, you may get an error because the code is not indented properly. To resolve this issue, make sure that the indentation of the code block is consistent throughout the code. In Python, indentation is used to define code blocks, and each level of indentation should be consistent (usually 4 spaces per level).

Here is an example of how the code should be indented correctly:

```
# Previous code...

if 'my-index' not in pc.list_indexes().indexes:
    pc.create_index(
        name='my-index',
        dimension=1536,
        metric='cosine'
    )

# Rest of the code...
```

Make sure that the `pc.create_index()` function call and its arguments are indented correctly under the `if` statement. The indentation should match the level of the `if` statement.

Review the indentation of the entire code block and ensure that it is consistent throughout. Each line within the same code block should have the same level of indentation.

After fixing the indentation, save the changes and run the Python script again using Streamlit.

Run Your Streamlit Application

To run your Streamlit application, follow these steps:

- Open a terminal or command prompt.
- Navigate to the directory where your `LangChainUI.py` file is located using the `cd` command. For example:

```
cd /path/to/your/app/directory
```
- Once you are in the correct directory, run the following command:

```
streamlit run LangChainUI.py
```
- This command will start the Streamlit server and run your `LangChainUI.py` file.
- Streamlit will provide you with a URL (usually `http://localhost:8501`) that you can open in your web browser to view and interact with your Streamlit application.

View Your Streamlit Application in a Web Browser

- After running the `streamlit run LangChainUI.py` command, Streamlit will display a message in the Command Prompt indicating that your application is running.
- It will provide a URL (usually `http://localhost:8501`) that you can copy and paste into your web browser to view and interact with your Streamlit application.
- Click the URL or copy and paste it into your preferred web browser.

Interact with Your Streamlit Application

- Once your Streamlit application is loaded in the web browser, you can interact with it based on the components and functionality defined in your `LangChainUI.py` file.
- Streamlit automatically updates the application in real time as you make changes to your `LangChainUI.py` file, allowing for quick iteration and development.

Stop the Streamlit Server (When Needed)

- To stop the Streamlit server and exit the application, go back to the Command Prompt where you ran the `streamlit run LangChainUI.py` command.
- Press `Ctrl+C` to interrupt the server and terminate the Streamlit application.

Testing the App

Ask a question and you might see an answer similar to this:

localhost:8501/#ask-your-questions

ChatGPT-like Q&A App

Enter your question:

What major product did this person's predecessor introduce that revolutionized the smartphone indust

Q: Who is the CEO of Apple Inc.?

A: The current CEO of Apple Inc. is Tim Cook.

Submit

Answer:

The predecessor of this person introduced the iPhone, which revolutionized the smartphone industry.

I asked the first question – “Who is the CEO of Apple Inc.?” – and the answer is straightforward (as of 2024, Tim Cook is the CEO of Apple Inc.).

The second question directly refers to the answer of the first question without naming the person explicitly. It also requires the app to know that (a) the “person” referred to is Tim Cook, (b) Tim Cook’s predecessor was Steve Jobs, and (c) the product in question is the iPhone.

If the app’s memory is working correctly, it should be able to

1. Provide the correct answer to the first question (Tim Cook).
2. Understand that “this person” in the second question refers to Tim Cook.

3. Recall that Tim Cook's predecessor was Steve Jobs.
4. Identify that the iPhone was the revolutionary product introduced by Steve Jobs.

A correct response to the second question would demonstrate that the app has maintained context across the two questions and can link information from both to provide a coherent answer.

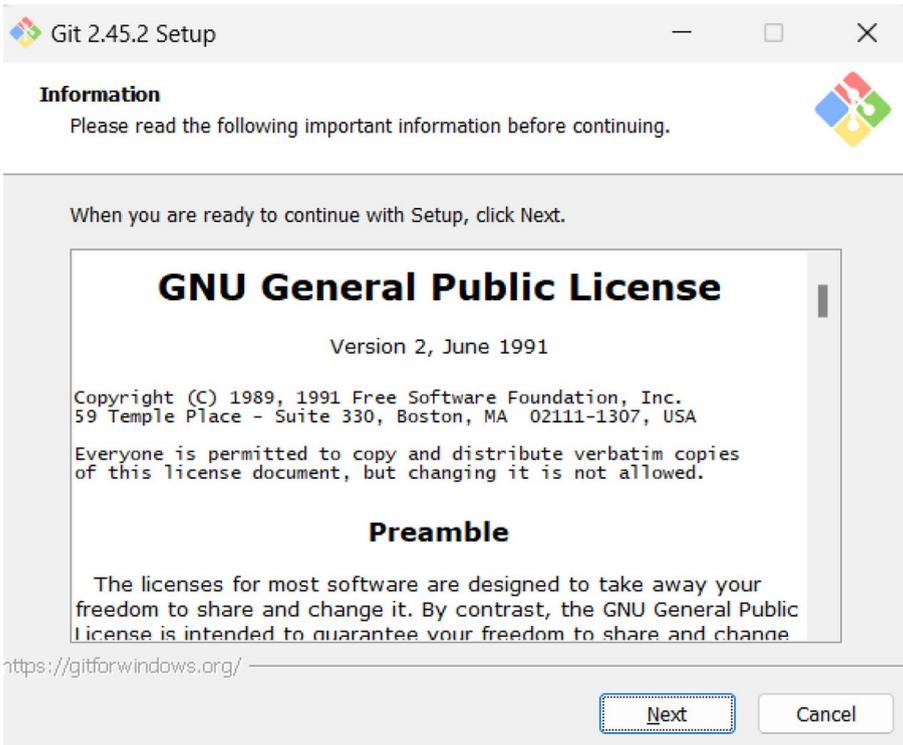
Deploying the LangChain Application

Let us discuss the steps involved in deploying the LangChain application.

Installing Git on Your System

First, you need to ensure that Git is installed on your system:

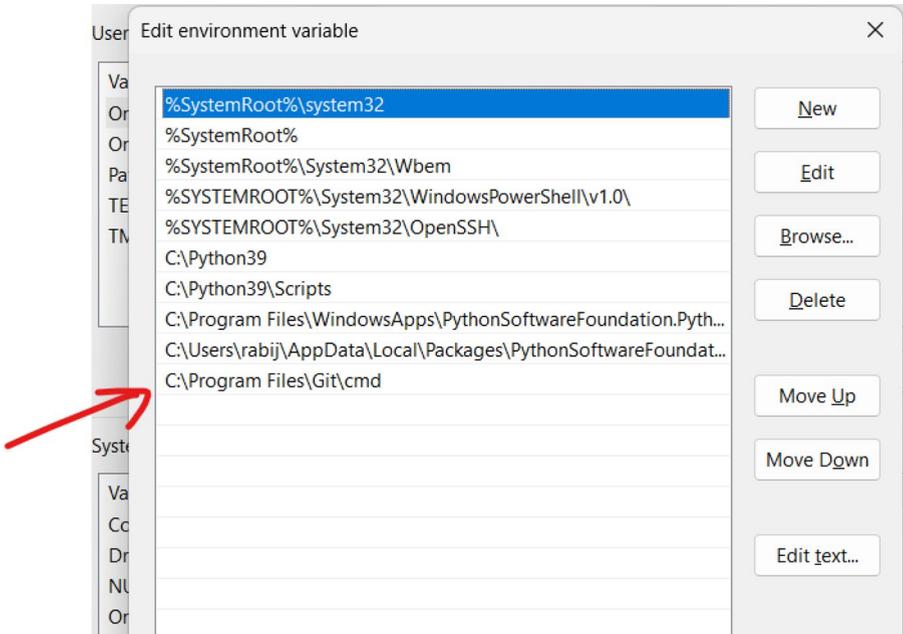
1. **Install Git:** Here are the steps to install it first.
 - a. Go to the official Git website: <https://git-scm.com/download/win>.
 - b. Download the installer for Windows.
 - c. Run the installer and follow the installation wizard. Use the default settings unless you have specific preferences as shown below:



2. **Add Git to PATH:** After installation, Git should automatically be added to your system's PATH. However, if it is not, you can add it manually.
 - a. Right-click "This PC" or "My Computer" and select "Properties."
 - b. Click "Advanced system settings."
 - c. Click "Environment Variables."
 - d. Under "System variables," find and select the "Path" variable, then click "Edit."

- e. Click “New” and add the path to your Git installation. It is typically

C:\Program Files\Git\cmd



- f. Click “OK” to close all dialog boxes.
3. **Restart Command Prompt:** After installing Git or modifying the PATH, you need to close and reopen your Command Prompt for the changes to take effect.
 4. **Verify Installation:** Open a new Command Prompt and type


```
git -version
```

This should display the installed version of Git as shown in the figure below if everything is set up correctly.

```
C:\Users\rabij\LangChain>git --version
git version 2.45.2.windows.1
```

5. **Try Git Init:** Now you should be able to run `git init` in your project directory without any errors.

```
C:\Users\rabij\LangChain>git init
Initialized empty Git repository in C:/Users/rabij/LangChain/.git/
```

Setting Up Your Identity

Git uses your identity information to associate commits with an author. You need to set up your Git configuration with your name and email address. Here is how to do it:

1. **Set Your Email Address:** Run the following command, replacing “your.email@example.com” with your actual email address.

```
git config --global user.email your.email@example.com
```

2. **Set Your Name:** Run this command, replacing “Your Name” with your actual name.

```
git config --global user.name "Your Name"
```

3. **Verify Your Settings:** You can check your settings by running

```
git config -list
```

This will display all your Git configurations, including the email and name you just set.

4. **Try Committing Again:** After setting your email and name, try your commit command again.

```
git commit -m "Initial commit for Streamlit Q&A app"
```

Important Notes

- The `--global` flag sets this configuration for all Git repositories on your computer. If you want to use different settings for different projects, you can omit `--global` and run these commands from within a specific repository.
- Make sure to use an email address that is associated with your GitHub account if you plan to push your commits to GitHub.
- If you are concerned about privacy, GitHub provides options for keeping your email address private. You can use the GitHub-provided no-reply email address in your Git configuration.

After completing these steps, you should be able to commit your changes without any identity-related errors. Remember, this is a one-time setup unless you want to change your Git identity later.

Set Up the OpenAI Key As an Environmental Variable

You should also set up an environment variable for local development for keeping sensitive information like API keys secure. Here is a more detailed explanation of how to do this:

1. For Windows

a. Temporary (for current session only)

- Open Command Prompt.
- Type: `set OPENAI_API_KEY=your_api_key_here`

b. Permanent

- Search for “Environment Variables” in the Start menu.
- Click “Edit the system environment variables.”
- Click “Environment Variables.”
- Under “User variables,” click “New.”
- Variable name: `OPENAI_API_KEY`
- Variable value: `your_api_key_here`
- Click “OK” to save.

2. For macOS/Linux

a. Temporary (for current session only)

- Open Terminal.
- Type: `export OPENAI_API_KEY=your_api_key_here`

b. Permanent

- Open your shell configuration file (e.g., `~/.bash_profile`, `~/.zshrc`).
- Add the line: `export OPENAI_API_KEY=your_api_key_here`
- Save the file and restart your terminal or run `source ~/.bash_profile` (or relevant file).

3. Using a .env file

a. Create a file named `.env` in your project directory.

b. Add the following line to the file:

```
OPENAI_API_KEY=your_api_key_here
```

c. Install the `python-dotenv` package: `pip install python-dotenv`

d. In your Python code, load the environment variables:

```
from dotenv import load_dotenv
import os

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

4. Accessing the environment variable in your code

Once you have set up the environment variable, you can access it in your Python code like this:

```
import os

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

5. Benefits of using environment variables

- **Security:** Your API key isn't hardcoded in your source code.
- **Flexibility:** Easy to change without modifying code.
- **Portability:** Works across different environments (development, staging, production).

6. Best practices

- Never commit your `.env` file to version control.
- Add `.env` to your `.gitignore` file.
- Provide a `.env.example` file with placeholder values for other developers.

By setting up your API key as an environment variable, you can develop locally without risking exposure of your sensitive information. Note that when you deploy your application, you will use the deployment platform's secret management system (like Streamlit Cloud's Secrets management) to securely store and access the API key in the production environment.

Resolving Sensitive Information Issues in Your Repository

If you notice that there are still some issues with sensitive information in your repository, you may follow the steps below:

1. The `.env` file

- Create a file named `.env.txt`, but it should be just `.env` (without the `.txt` extension).

2. Remove the `.env` file from Git tracking:

```
git rm --cached .env.txt
git rm --cached .env
```

3. Rename `.env.txt` to `.env` if needed:

```
ren .env.txt .env
```

4. Update `.gitignore`. Ensure your `.gitignore` file contains

```
.env
*.pyc
__pycache__/
```

5. Remove API keys from other files:
 - Open “`LangChainUI.py`” and replace the hardcoded API key with

```
import os
from dotenv import load_dotenv
load_dotenv()
openai_api_key = os.getenv("OPENAI_API_KEY")
```

6. Commit these changes:

```
git add .gitignore
git add LangChainUI.py "LangChainUI - Copy.py"
git commit -m "Remove API keys and update .gitignore"
```

7. Force push the changes:

```
git push -u origin main -force
```

8. Clean Git history. If the API key is still in your Git history, you may need to clean it:

```
git filter-branch --force --index-filter "git
rm --cached --ignore-unmatch .env.txt .env LangChainUI.
py 'LangChainUI - Copy.py'" --prune-empty --tag-name-
filter cat -- --all
```

Then force push again:

```
git push origin --force -all
```

Remember, after cleaning the Git history, anyone who has cloned your repository will need to re-clone it or perform a forced pull.

These steps should remove the API key from your repository and prevent it from being pushed to GitHub. You should always be cautious with sensitive information and double-check your commits before pushing.

Preventing Email Privacy–Related Issues

Sometimes, you may get privacy-related error messages which indicate that GitHub is preventing the push due to email privacy restrictions. GitHub may be trying to protect your privacy by not allowing commits with a private email address to be pushed to a public repository. Here is how to resolve this issue:

1. Configure Git to use your GitHub-provided no-reply email address:
 - a. Go to GitHub and sign in to your account.
 - b. Go to Settings ► Emails.
 - c. Look for a paragraph that says something like “Keep my email address private. We will use yourusername@users.noreply.github.com when performing web-based Git operations and sending email on your behalf.”
 - d. Copy this email address (it should look like yourusername@users.noreply.github.com).

2. Set this email in your Git configuration. Open your command prompt and run

```
git config --global user.email
yourusername@users.noreply.github.com
```

Replace “yourusername@users.noreply.github.com” with the actual no-reply email address you found in your GitHub settings.

3. Amend the last commit to use the new email:

```
git commit --amend --reset-author
```

This will open your default text editor. Just save and close the editor without making any changes.

4. Force push again:

```
git push origin --force -all
```

Alternatively, if you are comfortable making your commit email public

1. Go to GitHub Settings ► Emails.
2. Uncheck the box that says “Keep my email address private” as shown below.
3. Make sure your public commit email is set to an email you are comfortable being public.

Keep my email addresses private

We'll remove your public profile email and use 15165543+rabijay1@users.noreply.github.com when performing web-based Git operations (e.g. edits and merges) and sending email on your behalf. If you want command line Git operations to use your private email you must [set your email in Git](#).

Previously authored commits associated with a public email will remain public.

Block command line pushes that expose my email

When you push to GitHub, we'll check the most recent commit. If the author email on that commit is a private email on your GitHub account, we will block the push and warn you: [show if warning your private email](#)

After making either of these changes, try pushing your changes again. This should resolve the email privacy restriction issue.

Remember, if you choose to use the no-reply email, you will need to use this email for all your Git commits to avoid this issue in the future. If you are working on a shared machine or multiple projects, you might want to set this on a per-repository basis instead of globally.

```
git config user.email yourusername@users.noreply.github.com
```

Run this command in each repository where you want to use the no-reply email.

Deploying the App in GitHub

Below are the steps to deploy your app in GitHub:

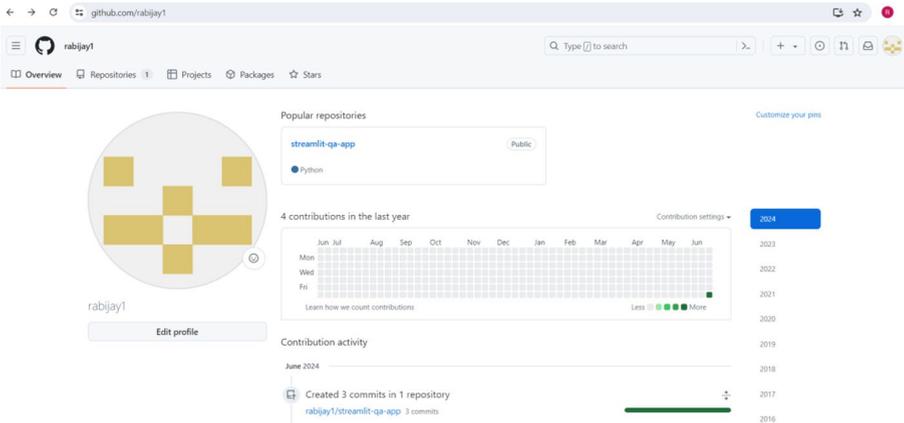
1. First, create a new repository on GitHub if you haven't already. Let us say you named it "streamlit-qa-app."
2. Now, modify the commands with your information:

```
git remote add origin https://github.com/your-actual-username/streamlit-qa-app.git
git branch -M main
git push -u origin main
```

Here is what each command does:

- The first command adds a remote named "origin" pointing to your GitHub repository.
- The second command renames your current branch to "main" (if it is not already named that).
- The third command pushes your local "main" branch to the "origin" remote and sets it up to track the remote branch.

After running these commands, your local repository will be connected to your GitHub repository, and your code will be pushed to GitHub as shown below.

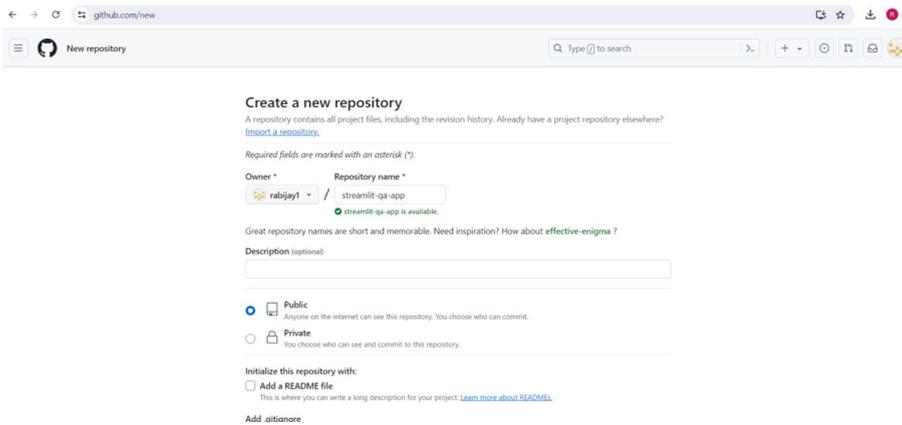
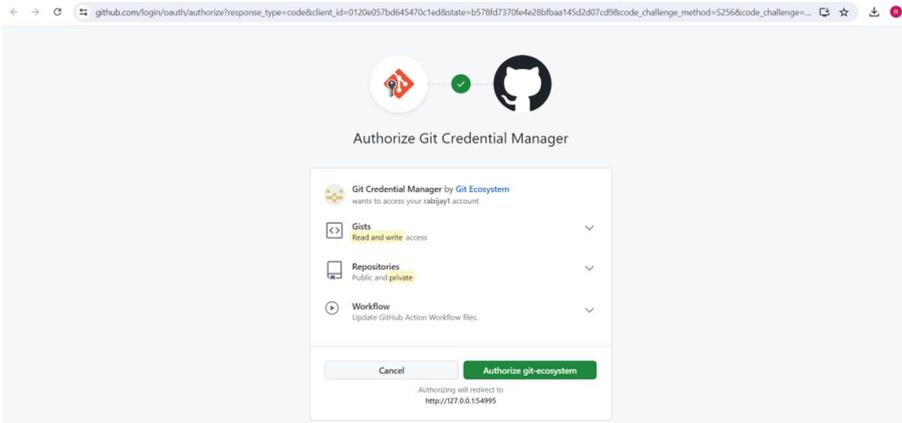


Remember, you only need to run the `git remote add` command once for each repository. For subsequent pushes, you can simply use

```
git remote add origin https://github.com/<your_user_name>/
streamlit-qa-app.git
git branch -M main
git push -u origin main
```

Providing Access to GitHub

Remember that as part of deploying the app to GitHub, you may need to give the appropriate permissions to GitHub.



In situations where you don't see the repository or you don't have access to it, you should follow the steps below to resolve the issue:

1. Verify the repository exists:
 - Go to https://github.com/<your_user_name>/streamlit-qa-app.
 - If you see a 404 error, the repository doesn't exist.

2. If the repository doesn't exist
 - Go to GitHub and create a new repository named "streamlit-qa-app."
 - Make sure it is under your account (your username).
 - Do not initialize the repository with a README, .gitignore, or license if you are pushing an existing repository.
3. Check your remote URL. Run this command to verify the remote URL:

```
git remote -v
```

It should show

```
origin https://github.com/your_user_name/  
streamlit-qa-app.git (fetch)  
origin https://github.com/your_user_name/  
streamlit-qa-app.git (push)
```

4. If the URL is incorrect, update it with

```
git remote set-url origin https://github.com/  
your_user_name /streamlit-qa-app.git
```
5. Ensure you are logged in if you see this error message: "please complete authentication in your browser."

6. Check your GitHub account:

- Ensure you are using the correct GitHub account (`your_user_name`).
- Verify that you have the necessary permissions to push to this repository.

7. Try pushing again:

```
git push -u origin main
```

8. Create a new personal access token:

- Go to GitHub Settings ► Developer settings ► Personal access tokens.
- Generate a new token with repo permissions.
- Use this token as your password when pushing.

9. Use HTTPS instead of SSH. If you are using an SSH URL, try the HTTPS URL instead:

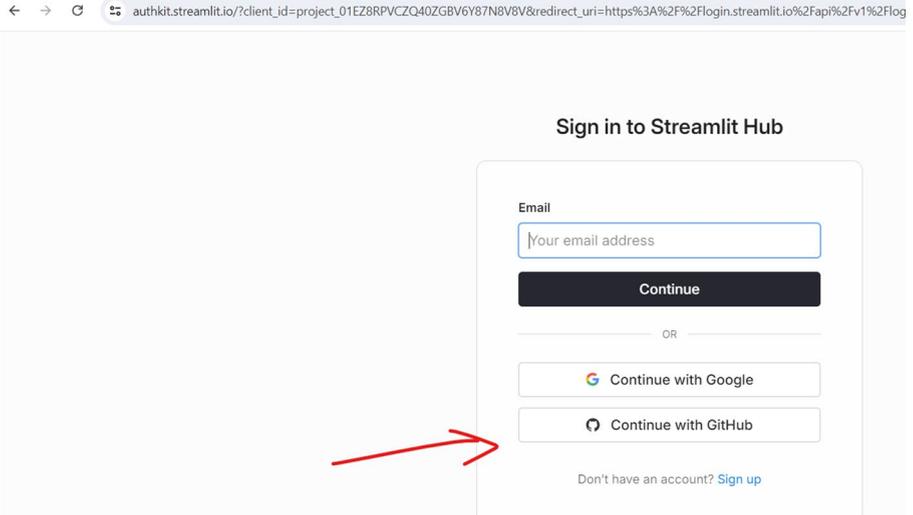
```
git remote set-url origin https://github.com/  
your_user_name/streamlit-qa-app.git
```

10. Check your Internet connection and firewall settings.

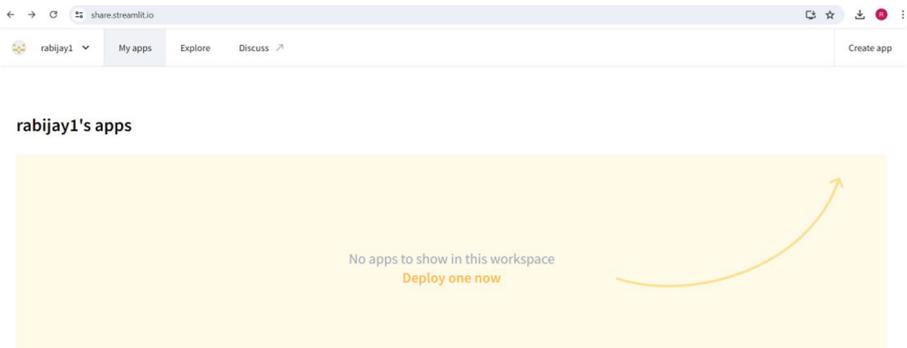
Deploying in the Streamlit Cloud

Let us discuss the steps to deploy using Streamlit Cloud:

1. Go to <https://streamlit.io/cloud> and sign in with your GitHub account as shown below.



2. Click “New app.”



3. Select your GitHub repository, branch (usually “main”), and the main Python file (e.g., “LangChainUI.py”).
4. Click “Deploy.”

Congratulations, if everything goes well, you must have successfully deployed the app to Streamlit Cloud for others to access.

Other Cloud Deployment Options

Let us talk about other ways to deploy your app to the cloud:

1. **LangServe:** This is a tool specifically designed for deploying LangChain applications. It offers easy API creation for your chains and agents and a user interface for testing your endpoints.

To use LangServe, you will typically

- Install it with `pip install langserve`
- Define your chain or agent in a Python file
- Create a FastAPI app and add your chain as a route
- Run your server with `uvicorn your_app:app --reload`

LangServe is particularly useful if you are building an API around your LangChain components. It makes it easy to expose your chains and agents as API endpoints, which can be incredibly handy when integrating your LangChain apps as part of larger systems or applications.

2. **Heroku:** This is often my go-to for quick deployments. It is user-friendly and works well with Python apps. Here is a quick rundown:

- First, sign up for a Heroku account.
- Install the Heroku CLI.
- Create a Procfile in your project directory with this line: `web: streamlit run your_app.py`.
- Run `heroku create your-app-name`.
- Push your code with `git push heroku main`.

3. **AWS Elastic Beanstalk:** If you are looking for something more robust, this is a solid choice. It is a bit more complex, but it scales well. You will need to
 - Set up an AWS account
 - Install the AWS CLI and EB CLI
 - Initialize your EB environment with `eb init`
 - Create an environment with `eb create`
 - Deploy with `eb deploy`
4. **Google Cloud Run:** This is great if you are comfortable with containers. Here is the gist:
 - Create a Dockerfile for your app.
 - Build your container image.
 - Push it to Google Container Registry.
 - Deploy to Cloud Run using the Google Cloud Console or `gcloud` CLI.
5. **DigitalOcean App Platform:** I like this for its simplicity. It is somewhere between Heroku and AWS in terms of complexity:
 - Connect your GitHub repo to DigitalOcean.
 - Choose your project and branch.
 - Select your resource plan.
 - Click deploy!
6. **Streamlit Cloud:** You have just now deployed using Streamlit Cloud.

Remember, each of these options has its pros and cons. Heroku and Streamlit Cloud are great for getting started quickly. AWS and Google Cloud offer more control and scalability but have a steeper learning curve. DigitalOcean sits somewhere in the middle.

My advice? Start with something simple like Heroku or Streamlit Cloud. As you get more comfortable and your needs grow, you can explore the more advanced options.

And don't stress if it doesn't work perfectly the first time. Deployment can be tricky, and even experienced developers sometimes need a few tries to get it right. Before you know it, you will be deploying apps like a pro!

Key Takeaways

Great job! You have just wrapped up an incredible journey from prototype to production.

You have transformed that initial Jupyter code into a fully functional web app using Streamlit. In particular, you have harnessed the power of OpenAI's GPT-3.5-turbo model to create a web app that people can interact with, ask questions, and get meaningful responses. That is huge!

Let us further break down what you have achieved:

1. You have built an interactive Q&A app where users can have a conversation with AI!
2. You have mastered chat history management which allows your app to maintain context over multiple exchanges.
3. You have learned to deploy your application so that it is ready for the world to use.

But here is the real beauty, you haven't just learned to use tools like LangChain and Streamlit, but you have gained skills that are directly applicable to real-world AI development. The next time someone says, "We need an AI-powered web app," you can confidently say, "I can do it!"

So, what is next on your AI development journey? Whatever it is, I know you are ready for it. Keep coding, learning, and, most importantly, keep pushing the boundaries of what is possible with gen AI.

Review Questions

Test your understanding of this chapter.

1. What is Streamlit primarily used for?
 - A. Data storage
 - B. Building interactive web applications
 - C. Machine learning model training
 - D. Network security
2. Which Python library is essential for connecting to OpenAI's GPT-3.5-turbo model in this chapter?
 - A. Pandas
 - B. NumPy
 - C. LangChain
 - D. Matplotlib

3. What is the main purpose of the Streamlit `st.text_input` function in the context of this chapter?
 - A. To display images
 - B. To create a text input field for user queries
 - C. To manage file uploads
 - D. To generate graphs and charts
4. Which feature is implemented to keep track of the conversation history in the Streamlit application?
 - A. Session state
 - B. Global variables
 - C. Database connections
 - D. File logging
5. What is the first step in deploying a Streamlit application?
 - A. Setting up a SQL database
 - B. Writing unit tests
 - C. Setting up the deployment environment on a web server or cloud service
 - D. Creating a Docker container
6. How does LangChain enhance the integration of AI models in the application?
 - A. By providing pretrained machine learning models
 - B. By offering tools to streamline prompt engineering and model management

- C. By replacing the need for any other Python libraries
 - D. By improving data visualization
7. Which Streamlit command is used to run the application locally?
- A. `streamlit deploy`
 - B. `streamlit serve`
 - C. `streamlit start`
 - D. `streamlit run`
8. **What advantage does using Streamlit provide for developing AI applications?**
- A. It offers high performance for computational tasks.
 - B. It simplifies the creation of interactive web interfaces.
 - C. It provides built-in natural language processing tools.
 - D. It ensures data security and encryption by default.
9. What should you do if you need to maintain chat history across different user sessions in Streamlit?
- A. Use session cookies
 - B. Implement a database backend
 - C. Store the data in global variables
 - D. Use a local text file

10. Which of the following is a key step in integrating OpenAI's GPT-3.5-turbo model using LangChain?
 - A. Training the model from scratch
 - B. Setting the model temperature parameter
 - C. Uploading data to the model
 - D. Visualizing the model's architecture

Answers

1. B. Building interactive web applications
2. C. LangChain
3. B. To create a text input field for user queries
4. A. Session state
5. C. Setting up the deployment environment on a web server or cloud service
6. B. By offering tools to streamline prompt engineering and model management
7. D. streamlit run
8. B. It simplifies the creation of interactive web interfaces.
9. B. Implement a database backend
10. B. Setting the model temperature parameter

Further Reading

By exploring these resources, you can deepen your understanding and enhance your ability to build, deploy, and manage AI-powered applications using Streamlit and LangChain.

Setting Up Your Development Environment

- **Streamlit Documentation: Getting Started:** Provides comprehensive guidance on installing and setting up Streamlit, including basic concepts and tutorials.

<https://docs.streamlit.io/>

- **Python Virtual Environments:** Learn about creating and managing virtual environments to keep dependencies isolated.

<https://docs.streamlit.io/>

Running Streamlit in Your Desktop

- **Streamlit Commands:** Explore various commands and options available for running and deploying Streamlit applications.

<https://docs.streamlit.io/develop/api-reference/cli>

Installing Streamlit Library

- **Streamlit Installation Guide:** Step-by-step instructions to install Streamlit using different package managers.

<https://docs.streamlit.io/get-started/installation>

Creating and Using the Language Model

- **OpenAI GPT-3 Documentation:** Detailed documentation on using OpenAI's GPT-3, including API references and usage examples.
<https://platform.openai.com/docs/introduction>
- **LangChain Documentation:** In-depth guide on using LangChain to build and manage language model applications.
<https://langchain.readthedocs.io/en/latest/>

Handling User Input

- **Streamlit Widgets:** Comprehensive list of widgets provided by Streamlit to handle user inputs and interactions.
<https://docs.streamlit.io/develop/api-reference/widgets>

Chat History Initialization

- **Managing State in Streamlit:** Learn about managing state in Streamlit applications to maintain information across user interactions.
https://docs.streamlit.io/develop/api-reference/caching-and-state/st.session_state

Deploying the Application

- **Deploying Streamlit Apps:** Guidelines for deploying Streamlit applications to the cloud or other hosting services.

<https://docs.streamlit.io/deploy/streamlit-community-cloud/deploy-your-app>

GitHub Integration

- **GitHub for Beginners:** Learn how to use GitHub for version control and collaboration.

<https://docs.github.com/en/get-started/start-your-journey/hello-world>

- **Using Environment Variables:** Best practices for managing API keys and other sensitive information using environment variables.

<https://12factor.net/config>

Index

A

- Access token, 144–147
- add_example method, 175–177
- Advanced chain techniques, 251
 - errors and exceptions, 247, 248
 - large datasets with chains, 245, 246
 - optimize your chain
 - performance, 248, 249
 - test and debugging chains, 249, 250
- AgentAction, 345, 346, 350–353, 355
- Agent design and implementation
 - AgentExecutor, 355, 356
 - concepts, 350–352
 - defining Agent's objective, 347, 348
 - defining Agent's tasks, 349
 - design considerations, 362
 - gathering tools and resources, 349
 - inputs, 353
 - LangGraph (*see* LangGraph)
 - outputs, 353
 - toolkits, 360, 361
 - tools, 356–360
- AgentExecutor, 355, 356, 384, 401, 418
- AgentFinish, 345, 346, 350, 352, 353, 355
- Agentic applications, 17–19
- Agents, 352
 - inputs, 353
 - outputs, 353
- Agents as task managers
 - code generation, 324
 - creative writing, 325
 - generative AI enhancing capabilities, 325
- Agent's importance
 - building applications
 - autonomous coders, 321
 - creative collaborators, 321
 - research assistants, 321
 - generative AI application
 - content understanding, 321
 - contextual decision-making, 322
 - dynamic content generation, 322
 - iterative refinement, 322

INDEX

Agent's key parts

- LLM, 320
- prompt, 320
- tools, 319

Agent's Thought Process, 320, 337, 338

Agent types selection criteria

- chat history support, 366
- multi-input tools, 366
- parallel function calling, 366

AIMessage, 381, 406

AI tutoring system, 14, 15

ALBERT, 157

Anthropic's Claude AI models

- Claude 3 model family, 134, 135
- Create Key, 136
- Get API Keys, 136
- SDK, 136
- sign up, 135

API, *see* Application programming interface (API)

API tool calling agents

- agent creation, 374
- agent running, 375, 377
- chat history, 377, 378
- description, 373
- goal, 373
- initializing tools, 374
- setup, 373, 374
- tools selection, 373

Application programming interface (API)

- LLMs (*see* LLM APIs)

app.py file, 458

AR, *see* Augmented reality (AR)

Argument variable, 69

asimilarity_search_by_vector method, 300

Augmented reality (AR), 131

Automated content

- scheduler, 17–19

AutoModelForCausalLM.from_ pretrained(), 150

Autonomous decision-making capabilities, 393–397

AWS Elastic Beanstalk, 483

B

BART, 157

BaseExampleSelector class, 175, 176

BaseLLM, 85

BaseTool class, 408

BLEU score, 47

Building Q&A

vs. chatbot

API key handling, 98

chain *vs.* direct

invocation, 98

interaction, 98

model used, 98

output parsing, 99

prompt handling, 98

single-turn interaction, 99

use case, 99

user input, 99

conversational app, 94–97

- create a prompt template, 92
 - create the chain, 92
 - full end-to-end working
 - code, 93, 94
 - import libraries, 91
 - invoke the chain, 92
 - LLM initialize, 92
 - output, 93
 - Output Parser, 91
- C**
- CacheBackedEmbeddings class,
 - 292, 293, 295, 296
 - Caching embeddings, 292–296, 298
 - Chains, 325, 326
 - vs.* agents
 - autonomy, 327
 - contextual understanding, 328
 - goal-oriented, 327
 - overview, 327
 - components, 214
 - higher-level components,
 - 214, 215
 - internal components, 216
 - LCELs, 217
 - LCEL *vs.* legacy chains
 - advanced features, 219
 - load_chain function, 219
 - legacy chain example, 218
 - legacy chains, 217
 - CharacterTextSplitter class,
 - 276–279, 295
 - Chatbot development, 12, 140
 - Chatbots, 3, 46, 96, 97, 114
 - Chat completion, 64, 199, 203
 - ChatGPT, 94
 - ChatGPT-like LangChain-based UI
 - application, 451
 - Chat models, 76, 90, 91, 94
 - ChatOpenAI, 85, 94, 196, 374, 400
 - Chat prompt templates, 170
 - create chat model instance, 200
 - create prompt template, 201
 - define, 200
 - format prompt with user
 - input, 202
 - generate chat completion, 202
 - import required libraries, 199
 - print assistant’s response, 202, 203
 - print formatted prompt, 202
 - prompt templates, 199
 - set up the OpenAI API key, 200
 - user_input variable, 201
 - Chirp Speech models, 133
 - chunk_overlap parameter, 279
 - Claude 3 Haiku, 134
 - Claude 3 model family
 - Claude 3 Haiku, 134
 - Claude 3 Opus, 134, 137–139
 - Claude 3 Sonnet, 134
 - easy to use, 135
 - following directions, 135
 - legacy models, 135
 - multilingual capabilities, 134
 - transitioning, 135
 - vision and image processing, 135
 - Claude 3 Opus, 134, 137–139

INDEX

- Claude 3 Sonnet, 134
- Code Snippet, 114, 115
- CodeTextSplitter, 281–283
- Codex, 113, 115, 116
- Codey Suite, 131, 132
- Cohere’s AI model, 139–143
- Cohere’s command model, 140–143
- Colab notebooks, 104
- Command-light model, 140
- Command-R model, 140
- Complex workflow apps, chain
 - composition strategies
 - data summarization app,
 - sequential chains, 237, 238
 - sentiment analysis app,
 - conditional chains, 243, 244
 - SequentialChain use
 - case example
 - automated fraud detection,
 - finance, 240
 - content generation pipeline app, 239
 - customer support chatbot app, 238, 239
 - task allocation app, router
 - chains, 240–242
- Conditional chain, 243, 244
- Content generation, agents
 - loaded tool, 324
 - prompt, 324
 - ReAct format, 322
 - SerpAPI, 323
- Content generation platform, 22,
 - 23, 50, 51
- Context-aware applications, 14,
 - 15, 34, 211
- Context-rich applications, 51
- Contextual Compression
 - Retriever, 302
- ConvBERT, 158
- Conversation Agent, 370
- Conversational app
 - ChatOpenAI, 94
 - ChatOpenAI Object creation, 95
 - generate_response, 95, 96
 - importing necessary
 - modules, 94
 - interaction loop, 96, 97
 - setting the API key, 94
- conversational-react-description
 - agent, 407
- ConversationalRetrievalChain,
 - 230, 232
- ConversationBufferMemory
 - object, 372
- create_agent_method function, 361
- create_documents method, 277
- create_react_agent function, 388
- create_tool_calling_agent
 - function, 401
- CSV files, 270–272
- CSVLoader class, 270–272
- CSV Parser, 192
- CTRL, 157
- Custom agent creation
 - adding memory, 419–421
 - binding tools, 417
 - creating agent, 418

- creating prompt, 417
 - defining tools, 416
 - loading language model, 416
 - testing agent, 419
 - Customer support automation
 - agent initialization, 423
 - agent integration, 424
 - agent setup, 422
 - code example, 425
 - defining tools and memory, 423
 - handle_inquiry function, 427
 - identifying common customer inquiries, 422
 - implementation using agents, 425
 - knowledge base creation, 422
 - monitoring and improving, 424
 - search_knowledge_base function, 427
 - self-service support system, 421
 - Customer support systems, 17
 - Custom Example Selector, 176, 177
- D**
- DALL-E 2, 113, 116–120
 - Data aware, 13, 37
 - Data connections, 51, 53
 - Data storage and retrieval
 - graph databases, 78
 - traditional databases, 79
 - vectorstores, 78, 79
 - Data Summarization App,
 - sequential chains, 237
 - Decision Points, 215, 216
 - Dense Passage Retrieval (DPR), 143
 - Development environment
 - LangChain, 65
 - OpenAI’s LLMs, 55–57
 - Python, 57–61
 - Development environment setup
 - installing Python, 452–456
 - installing required dependencies, 457
 - installing streamlit library, 452
 - Development, LangChain
 - agents, 80
 - callbacks, 81
 - chains, 80
 - chat models, 76
 - composition, 79
 - data storage and retrieval, 78, 79
 - document loaders, 77
 - embedding models, 78
 - LLMs, 76, 77
 - memory, 81
 - model I/O, 76
 - prompts, 76
 - retrieval component, 77
 - retrievers, 79
 - text splitters, 78
 - tools, 80
 - Development playground
 - Colab notebooks, 104
 - Hugging Face Spaces, 104
 - Kaggle notebooks, 105
 - LangChain, 103
 - OpenAI API, 104

INDEX

Development productivity, 18
DigitalOcean App Platform, 483
Direct LLM API *vs.* LangChain
 benefits, 51, 52
 content generation
 platform, 50, 51
 development complexity, 48
 flexibility, 48
 generic responses, 49
 integration and scalability, 49
 scalability, 48
 simplicity, 48
 streamlining data
 integration, 51
 text generation task, 62–65
 trade-offs, 52, 53
Document loaders, 77
 in action, 267, 268
 CSV files, 270–272
 JSON files, 273, 274
 load documents, sources, 266
 PDFs, 269, 270
Document objects, 268, 270
Domain-specific applications, 252
DPR, *see* Dense Passage
 Retrieval (DPR)

E

Ecosystem, LangChain
 high-level components, 87–89
 and integration
 LangServe, 89
 LangSmith, 89
 LangTemplates, 89
LangChain-community, 86
LangChain-Core, 85
ELECTRA, 158
Embedding models, 78
embed_documents method, 290
embed_query method, 291
End Point, 215, 216
End-to-end fully working Agent
 code explanation
 importing necessary
 modules, 332
 initializing agent, 334
 initializing LLM, 333
 initializing openAI
 client, 333
 installing dependencies, 332
 loading environment
 variables, 332
 loading tools, 333
 running agent with
 query, 334
 code generation, 328
 interpreting outputs, 336, 337
 outputs, 334–336
 SerpApi API key, 330
.env file, 471
Error handling and
 troubleshooting
 API and reliability, 99
 API connectivity issues, 100
 authentication errors, 100
 diagnosing and resolving
 API connectivity issues, 101

- authentication errors, 102
 - invalid request errors, 102
 - logging and monitoring, 102
 - model-specific
 - limitations, 102
 - rate limiting, 102
 - implementing, 100
 - invalid request errors, 100
 - model-specific limitations, 100
 - rate limiting, 100
 - example_prompt template, 188
 - Example selectors
 - create list of example, 175
 - Custom Example Selector, 177
 - custom implementation, 176
 - define BaseExampleSelector
 - Class, 175
 - factors, 180
 - few-shot learning, 174
 - Length example selector, 180
 - LLM, 174
 - MMR example selector, 179
 - Ngram example selector, 180
 - prompt, 178, 179
 - similarity example selector, 179
 - ExampleToolkit, 361
- F**
- FAISS (Facebook AI Similarity Search), 294, 295
 - FastText, 144
 - Few-shot learning, 168, 174, 181
 - Few-shot prompt template
 - question and answer
 - create FewShotPromptTemplate, 183
 - integrate example selector into prompt template, 186
 - prepare example set, 182
 - review output, 188–190
 - select examples with example selector, 184, 185
 - test prompt templates, 186
 - works, 181
 - FewShotPromptTemplate, 178, 182, 183, 185, 186
 - Flagship Command model, 140
 - Float16 data type, 150
 - Float32 data type, 150
 - format_prompt() method, 202
 - Formatted Prompt, 197, 202
 - from_bytes_store method, 292
 - from_messages() method, 201
 - from_tiktoken_encoder()
 - method, 285
- G**
- Gemini, 1, 3, 6, 8, 21, 31, 94
 - Gemini 1.0 Pro Vision, 120, 130, 131
 - Generative AI Apps, 213, 251
 - LangChain (*see* LangChain)
 - Generative models, 139, 140
 - get-pip.py file, 454
 - get-pip.py script, 453

INDEX

- get_relevant_documents
 - method, 303
- get_tools method, 361
- GitHub, 4, 36
- git remote add command, 477
- global flag, 469
- Google Cloud natural language API
 - API Key creation, 121, 122
 - billing account, 124, 125
 - create and download the JSON key file, 126, 127
 - enabling, 123, 124
 - environment variables, 127–130
 - Google Cloud Console, 121–123
 - Granting Access, 126
 - Python script, 128
 - sentiment analysis, 129
 - sentiment magnitude, 129, 130
 - Service Accounts, 125
- Google Cloud Run, 483
- Google Colab, 4, 105
- Google's AI model
 - Chirp Speech, 133
 - Codey Suite, 131, 132
 - language and chat models
 - Gemini 1.0 Pro, 120
 - Gemini 1.0 Pro Vision, 130, 131
 - Google Cloud natural language API, 121–130
 - PaLM 2, 131
 - multimodal and security, 132, 133
 - text and image data, 132

- GPT, 114, 115
- GPT-3.5-turbo, 97
- GPT-4, 1, 3, 6, 7, 21–24, 28, 31, 45, 46, 62, 65, 113
- GPT-Neo, 157
- GPU capabilities, 150

H

- Heroku, 482–484
- HtmlTextSplitter, 275
- Hugging Face
 - code to use LLaMA model, 148–150
 - explanation for the code, 147, 148
 - pass your access token, 144–147
 - running time, 150, 151
- Hugging Face Spaces, 104, 111
- HumanMessage, 194, 197, 406
- HumanMessagePromptTemplate.
 - from_template(), 201

I

- Image generation wizard, 116–120
- Imagen, 132, 159
- Indexing API
 - deletion modes
 - full mode, 304
 - incremental, 304
 - none, 304
 - key information, 304
 - RecordManager, 304
- initialize_agent function, 407–409

__init__ method, 176
 InMemoryCache, 248
 Intelligent agent tasks performance
 setup LangSmith, 397
 tools
 agent initialization, 401
 creating
 AgentExecutor, 401–423
 creating Retriever tool,
 399, 400
 LLM selection, 400
 prompt selection, 401
 Retriever, 398, 399
 Tavily, 398
 Intermediate steps, 351–353
 Internet connection, 457

J

jq schema, 273
 JSON files, 273, 274
 JSONLoader, 273, 274
 json.loads(), 273
 JSON Parser, 192

K

Kaggle notebooks, 105
 Keyword search, 287, 288

L

LangChain, 1, 192, 198, 199, 221
 accuracy and reliability, 19

active community and
 ecosystem, 54
 adapt to user needs and
 preferences, 19
 advanced functionalities, 53
 advantage, 9, 10, 12, 13, 19
 agentic applications, 17–19
 agents, 22, 26
 applications, 6
 automate complex workflows, 19
 benefits, 3
 building blocks, 32
 chains, 26
 chatbots, 3
 community, 36
 components, 21, 23, 24, 75
 deployment, 84
 development, 76–82
 production, 82, 83
 RAG process, 266
 content generation
 platform, 22, 23
 context-aware applications, 14, 15
 cost optimization, 54
 customer support chatbot,
 215, 216
 data aware, 13
 data connections, 21, 25, 51
 definition, 2
 developers, 21
 development efficiency, 9
vs. Direct LLM API (*see* Direct
 LLM API *vs.* LangChain)
 ecosystem, 10, 84–89

INDEX

LangChain (*cont.*)

- enhanced flexibility, 65–70
 - flexibility and scalability, 53
 - framework, 2, 20
 - generative app, 3–6
 - indexes, 25
 - integration with multiple
 - LLMs, 22, 23
 - libraries, 5
 - and LLMs (*see* Large language models (LLMs))
 - memory concepts, 25
 - models, 21, 24
 - modify your script, 66–69
 - modular and scalable
 - architecture, 9
 - multi-LLM integration, 54
 - no cost barrier, 11
 - and OpenAI, 5
 - open source and community
 - collaboration, 10
 - prompt engineering, 11,
12, 69, 70
 - prompt templates, 25
 - RAG, 15, 16
 - rapid development and
 - prototyping, 53
 - real-world example, 8
 - retrieval types, 302
 - trade-offs, 54
- ### LangChain Agents
- API tool calling, 367, 373–378
 - Conversation Agent, 370
 - description, 315

- JSON chat, 369
 - key features, 316, 317
 - MRKL Agent, 371
 - OpenAI functions, 368
 - OpenAI tools, 368, 378–380
 - ReAct agent, 369, 387–390
 - scenarios
 - intelligent search, 319
 - recommendation
 - systems, 319
 - task automation, 319
 - self-ask agents, 370, 391–393
 - structured chat agent,
369, 382–387
 - Structured Tool Agent, 371
 - types
 - domain-specific agents, 316
 - general-purpose agents, 316
 - multi-agent systems, 316
 - simulation agents, 316
 - task-specific agents, 316
 - workflow, 317–319
 - XML, 368
 - Zero-Shot-React Agent, 370
- ### LangChain Application
- deployment
 - in GitHub, 476, 477
 - installing Git, 465–468
 - OpenAI Key, environment
 - variable, 469–472
 - other cloud deployment
 - options, 482–484
 - preventing email privacy-related error, 474–476

- providing access,
 - GitHub, 477–480
- resolving sensitive information
 - issues, 472–474
- setting up your identity,
 - 468, 469
- in Streamlit Cloud, 480, 481
- LangChain chains
 - example, 212, 213
 - future, generative AI, 251, 252
 - generative AI
 - applications, 212–214
- langchain.chat_models, 194
- LangChain-community, 86
- langchain_community
 - package, 269
- LangChain-Core, 85
- langchain.debug module, 249
- LangChain Documentation on
 - Agents, 343
- LangChain Execution Language
 - (LCEL) chains, 216, 218, 250
 - construct, 221, 222
 - customize, 222, 223
 - execution modes
 - async execution, 224
 - batch execution, 225
 - observability, LCEL
 - chains, 225
 - streaming execution, 224
 - flexibility, 217
 - vs.* legacy chains, 218, 219
 - load_query_constructor_
 - runnable chain, 227
 - natural language query, 228
 - query_constructor chain, 227
 - scalability, 217
 - types, 226
 - use cases, 217, 219
- LangChain indexing API, 303
- LangChain library, 171, 203
- LangChain Memory
 - Modules, 343
- langchain.output_parsers, 194
- langchain.prompts, 194
- langchain.schema, 194
- LangChain’s GitHub
 - repository, 103
- langchain-text-splitters
 - package, 276
- LangChain tools
 - built-in tools, 360
 - custom tools, 360
 - OpenAI functions, 360
 - toolkits, 360
- LangChain Tools Integration
 - Guide, 343
- LangChain Tutorial on Building
 - Agents, 343
- LangChainUI.py file, 458
- LangChain Use Cases, 344
- LangChain v0.1 *vs.* v0.2 Agents
 - enhanced agent types, 407
 - enhanced error handling and
 - debugging, 408
 - improved tool integration, 407
 - simplified agent
 - initialization, 407

INDEX

- LangGraph
 - creation, 363–365
 - description, 362
 - installation, 363
 - organizing tasks and data, 363
 - LangServe, 89, 482
 - LangSmith, 82, 83, 89, 397, 401, 403, 404
 - langsmith_search tool, 405
 - LangTemplates, 89
 - Language models, 165, 251
 - Large Language Model Meta AI (LLaMA), 3, 143
 - using Hugging Face, 144–151
 - Large language models (LLMs), 1, 90, 164, 198, 320
 - Anthropic’s Claude models, 133–139
 - APIs (*see* LLM APIs)
 - app development
 - workflow, 26, 27
 - choose LLM and LangChain integration, 28
 - client-server interaction, 29
 - conceptualization, 27
 - database design, 29
 - define requirements, 28
 - deploy application, 30
 - design application
 - architecture, 28
 - implement LangChain components, 29
 - incorporate data sources, 29
 - iterate and optimize, 30
 - monitor and maintain, 31
 - preparing deployment scripts, 30
 - service-oriented architecture, 29
 - set up development environment, 29
 - train/test with LLMs, 30
 - applications, 24
 - capabilities, 6
 - chain, 216
 - chat models, 90, 91
 - Cohere’s AI model, 139–143
 - and developers, 20
 - diversity applications, 155
 - error handling and troubleshooting, 99–102
 - Gemini, 8
 - general LLM model, 90
 - generating code, 7
 - Google’s AI, 120–133
 - GPT-4, 7
 - on Internet, 6
 - Meta AI, 143–152
 - OpenAI, 114–120
 - PaLM, 8
 - prompts, 216
 - summarizing large texts, 7
 - super-reading robots, 6
 - translating languages, 7
 - writing stories, 7
- “Lazy load” method, 267
 - Learning through experimenting
 - document your findings, 105

- experiment freely, 105
- share and collaborate, 106
- Legacy chains, 218, 219
 - construct, 229
 - conversational apps with
 - ConversationChain, 233, 234
 - direct application, 217
 - Document Chatbot App
 - Conversational RetrievalChain, 232
 - Document Processing App,
 - MapReduceChain, 235, 236
 - execution, 229
 - vs.* LCEL chains, 218, 219
 - Q&A Apps, RetrievalQA, 234, 235
 - simplicity, 217
 - Text Generation Apps,
 - LLMChain, 233
 - types, 230
 - use case, 218, 220
- Legal analysis tool, 16
- LengthBasedExampleSelector, 184, 185, 189, 190
- Length example selector, 180
- LLM APIs
 - business benefits
 - continuous improvement in language capabilities, 41
 - cost-effective access to massive language models, 41
 - domain-specific knowledge integration, 41
 - multilingual and cross-cultural capabilities, 42
 - natural language processing, 40
 - rapid prototyping of
 - AI-powered features, 41
 - scalable applications, 41
 - calling an OpenAI API
 - directly, 61–65
 - challenges
 - debugging and troubleshooting, 44
 - latency, 44
 - managing deprecation, 43
 - rate limits and cost management, 44
 - security concerns, 43
 - choose the right use case, 46
 - development
 - environment, 55–61
 - direct *vs.* LangChain (*see* Direct LLM API *vs.* LangChain)
 - model performance, 47
 - pre-built models and functionalities, 40
 - prepare data, 47
 - select the right model, 46
 - technical benefits
 - advanced prompt engineering capabilities, 43
 - efficient handling of context and memory, 42

INDEX

LLM APIs (*cont.*)

- flexible integration of language models, 42
- seamless integration of multimodal inputs, 43
- simplified complex NLP tasks, 42

LLMChain, 172, 233

- from langchain.chains, 68
- run method, 69

“llm_chain” template, 221, 222

load_and_split() method, 269

load_chain function, 219, 221

load() method, 267, 268, 271

load_query_constructor_runnable chain, 227

LocalFileStore class, 294

Longformer, 158

M

MapReduceChain, 235, 236, 245, 246, 248

MarkdownTextSplitter, 275

Market analysis tool, 16

Maximal Marginal Relevance (MMR) Example Selector, 179

MegatronLM, 157

Memory component, 216, 253

Memory concepts, 25, 53

Meta AI models

- DPR, 143
- FastText, 144

LLaMA, 143–151

M2M-100, 143

NLLB, 143

OPT, 143

PyTorch, 152

RoBERTa, 143

WaVE, 144

Meta-Llama-3-8B model, 147, 150

M2M-100, 143

Model fine-tuning, 168

Model input/output (I/O), 76, 89, 90

MRKL Agent, 371

Multimodal and security models, 132, 133

Multi-vector Retriever, 301, 310

N

Natural language processing, 40, 120, 159, 361

Natural language query, 227, 228

Next-generation language models, 114

Ngram example selector, 180, 181

No Language Left Behind (NLLB), 143

O

Observability, LCEL chains, 225

OpenAI

- API key, 68
- chat model, 203

- ChatOpenAI model, 416
- GPT-3.5-turbo model, 484
- from langchain.llms, 67
- OpenAIEmbeddings class, 289, 295, 296
- OpenAI function calling, 192, 226, 417
- OpenAIFunctions Parser, 192
- OpenAI models
 - Codex, 115, 116
 - DALL-E 2, 116–120
 - GPT, 114, 115
- OpenAI's LLMs
 - API key, 56
 - billing information, 55
 - create/open an account, 55
- OpenAI tools agent
 - advantage, 379
 - agent creation, 379, 380
 - agent running, 380
 - chat history, 381
 - description, 378
 - goal, 378
 - initializing tools, 379
 - necessary libraries, 379
- OpenAITools Parser, 192
- Open Pretrained Transformer (OPT), 143
- os.environ.get(), 196
- OutputFixingParser, 192, 198
- Output parsers, 353
 - functions, 191
 - LLM, 191
 - OutputFixingParser, 198

- parse with Prompt, 191
- PydanticOutputParser,
 - movie data
 - choose the LLM and its settings, 196
 - create prompt template, 195
 - define movie data model, 194
 - generate movie information, 197
 - import libraries, 194
 - parse the LLM's response, 197, 198
 - set up the OpenAI API key, 196
 - types, 192, 193

P, Q

- page_content attribute, 278
- PaLM, 1, 3, 6, 8, 21–23, 28, 31
- PaLM 2, 131
- ParentDocument Retriever, 301, 310
- pc.create_index() function, 461
- PDFs, 25, 269, 270
- Personalized
 - recommendation system
 - agent setup, 429
 - code example, 434
 - data preprocess and analysis, 429
 - defining recommendation tools, 430

INDEX

- Personalized recommendation system (*cont.*)
 - ecommerce platform, 428
 - generate_recommendations function, 438
 - generating personalized recommendations, 433
 - initializing the agent, 433
 - integrating recommendations into application, 434
 - recommendation prompt setup, 432
 - tools setup, 432
 - user data collection, 428
 - user preference dataset, 438
 - user_preference_tool, 438
- pip install streamlit command, 456
- Practical use cases
 - customer support
 - automation, 421–428
 - personalized recommendation system, 428–438
 - real-time data analysis and decision-making, 439–447
- Pretrained model, 147
- Prompt engineering, 11, 12, 43, 69, 70
 - access to knowledge, 162
 - career opportunities, 162
 - cost efficiency, 162
 - formats, 162
 - LLM models, 161
 - multiple models, 162
 - scalability, 163
 - steps, 164–169
 - vs.* model fine-tuning, 168
- Prompts, 76
 - chaining, 53
 - components
 - output parsers (*see* Output parsers)
 - prompt templates, 170–174
 - streamlining customer service, case study, 203, 204
 - advanced engineering, 206
 - customization, 205
 - impact, 206, 207
 - initial design, 205
- Prompt templates, 68, 165, 195, 353
 - advantages, 173, 174
 - chat, 170
 - context and questions, 171
 - create multiline string, 171
 - example selectors
 - create list of example, 175
 - Custom Example Selector, 177
 - custom implementation, 176
 - define BaseExampleSelector Class, 175
 - factors, 180, 181
 - few-shot learning, 174
 - Length example selector, 180
 - LLM, 174
 - MMR example selector, 179
 - Ngram example selector, 180
 - prompt, 178, 179

- similarity example
 - selector, 179
 - types, 179
 - few-shot examples, 170
 - instructions, 170
 - from langchain.prompts, 67
 - LLM, 172, 173
 - regular, 170
 - pydantic, 192–194
 - PydanticOutputParser, 195, 198
 - Pydantic Parser, 193
 - PyPDFLoader class, 269
 - Python command, 452
 - Python development environment
 - configure API key, 58
 - Google Colaboratory, 57
 - install OpenAI library, 57
 - securing the API keys
 - configuration files, 60, 61
 - environment
 - variables, 59, 60
 - test the setup, 58, 59
 - PyTorch, 147, 152
- R**
- RAG, *see* Retrieval-augmented generation (RAG)
 - ReAct agent, 345, 371
 - chat history, 389, 390
 - creating agent, 388
 - definition, 387
 - initializing tools, 387
 - necessary libraries, 387
 - running agent, 388
 - ReAct (Reasoning and Acting)
 - format, 322
 - read() method, 276
 - Real-time customer service
 - chatbot, 45
 - Real-time data analysis and decision-making
 - data analysis tools, 440
 - data preprocess and transform, 439, 440
 - data retrieval, 446
 - identifying data sources, 439
 - initializing agent, 442
 - integrating decisions, 443
 - process_data_and_make_decision function, 446
 - process real-time data and make decisions, 442, 443
 - real-world scenario, 447
 - setup agent, 440
 - setup data ingestion, 439
 - setup the decision-making prompt, 442
 - Real-world AI development, 485
 - recommendation_generator_tool, 438
 - RecursiveCharacterTextSplitter, 275, 279, 281, 285
 - Recursive splitting, 279, 281
 - Regular prompt templates, 170
 - Representative models, 139, 140
 - Research assistant, 16, 321
 - response = agent.run(query), 334

INDEX

- Retrieval-augmented generation
 - (RAG), 11, 15, 16, 77
 - architecture, 305
 - embed phase, 263
 - high-quality, 265
 - implementations, 305
 - importance, 260, 261
 - LangChain components, 266
 - Document loaders (*see* Document loaders)
 - indexing API (*see* Indexing API)
 - retrievers (*see* Retrievers)
 - text embedding models (*see* Text embedding models)
 - text splitters (*see* Text splitters)
 - vector stores, 286, 287
 - load phase, 263
 - retrieve phase, 264
 - Source, 262
 - store phase, 263
 - transform phase, 263
 - use case example, 264, 265
 - visual representation, 262
 - Retrieval-based question-answering system, 88
 - Retrieval component, 77
 - RetrievalQA chain, 234
 - Retrievers, 398, 399
 - code, 302, 303
 - Contextual Compression Retriever, 302
 - LangChain, 302
 - Multi-vector Retriever, 301
 - ParentDocument Retriever, 301
 - Self-Query Retriever, 302
 - unstructured query, 300
 - Vectorstore Retriever, 301
 - RetryWithError Parser, 192
 - returns_policy.txt, 284
 - RoBERTa, 143, 144
- ## S
- Sec-PaLM2, 133, 159
 - select_examples method, 175–177, 185
 - Self-ask agents
 - creating agent, 392
 - initializing tools, 391
 - necessary tools, 391
 - running agent, 392, 393
 - search capabilities, 391
 - Self-Query Retri, 302
 - Sentiment analysis, 129, 206
 - Sentiment analysis app,
 - conditional chains, 243, 244
 - Sentiment magnitude, 129, 130
 - SequentialChain class, 237–240
 - SerpAPI, 323
 - Similarity example selector, 179
 - Simplified Agent initialization in v0.2, 408, 409
 - Simulation agents, 316
 - Software Development Kit (SDK), 62

Speech recognition (Whisper), 120
 split_text() method, 285
 SQLiteCache, 248
 Steps (or Nodes), 214, 215
 Streamlit Cloud, 483
 Streamlit LangChain UI app
 building steps, 458–461
 components, 458
 indentation error in code, 461
 running
 interaction, 463
 steps, 462
 stopping Streamlit
 server, 463
 viewing, web browser, 462
 testing, 464, 465
 streamlit run LangChainUI.py
 command, 458
 Structured chat agent, 345
 chat history, 387
 creating agent, 382, 383
 defining helper
 function, 384
 definition, 382
 initializing tools, 382
 LangChain community
 tools, 382
 running agent, 384–386
 Structured Tool Agent, 371
 StuffDocumentsChain, 246
 Super-reading robots, 6
 SystemMessagePromptTemplate,
 199, 201

System prompt, 166

T

Task Allocation App, router chains
 challenges, 242
 example, 240, 241
 implementation, 242
 outcomes, 242
 streamlining customer service
 operations, 242
 Task-specific agents, 316
 tavily_search_results_json tool, 406
 Tavily search tool, 373–375, 379,
 380, 385, 398
 Text and image processing
 models, 132
 Text completion model, 90
 text_content=False parameter, 273
 Text embedding models
 cache embeddings, 292, 293,
 295, 296
 customer reviews, 292
 embed_documents
 method, 290
 embed_query method, 291
 find the documents, 287
 information retrieval
 system, 297–299
 install required packages, 289
 vs. keyword search, 287, 288
 LangChain, 288
 OpenAI API key, 289

INDEX

- OpenAIEmbeddings class, 289
 - query text, 291
 - reviews, 290
 - vectors, 288
 - vector stores asynchronously,
 - 299, 300
 - TextLoader class, 236, 294
 - Text splitters, 78
 - code example, 276–278
 - CodeTextSplitter, 281–283
 - HtmlTextSplitter, 275
 - langchain-text-splitters
 - package, 276
 - MarkdownTextSplitter, 275
 - RecursiveCharacter
 - TextSplitter, 275
 - Recursive
 - splitting, 279, 281
 - token, 283–285
 - TokenTextSplitter, 275
 - text_splitters and tiktoken
 - packages, 284
 - Text-to-speech (TTS), 120
 - The Restaurant Recommendation
 - Agent, 348
 - Tokenizer, 147
 - TokenTextSplitter, 275
 - to_messages() method, 202
 - Toolkits, 360, 361
 - Tools
 - abstraction, key
 - components, 357
 - custom schema, 359
 - definition, 356
 - essential elements, 356
 - properties, 358
 - WikipediaQueryRun, 358
 - Transformer-XL, 157
 - Trigger, 214, 215
 - try-except block, 385
 - TTS, *see* Text-to-speech (TTS)
 - TwitterTweetLoader, 86
- ## U
- user_input variable, 68, 201
 - User prompt, 64, 69, 166
- ## V
- ValueError exception, 384
 - Vectorstore Retriever, 301–303, 312
 - Vector stores
 - load source data, 286
 - query vector store, 286
 - retrieve “most similar”, 287
 - verbose parameter, 247, 408
 - Visual Question Answering (VQA), 132
- ## W
- WaVE, 144
 - Web application, 18, 19, 451
 - WebBaseLoader, 398
 - WikipediaQueryRun tool, 358

X

XLNet, 156

XML Parser, 192

Y

YAML Parser, 193

Z

Zero-Shot-React agents, 345, 370, 371

 advantage, 372

 definition, 371

 disadvantage, 372

 required tools, 372

zero-shot-react-description

 agent, 408