GENERATIVE AI

WITH LOCAL LLM

A comprehensive roadmap for building
AI-Driven applications with local LLMs



SHAMIM BHUIYAN
TIMUR ISACHENKO

Generative AI with local LLM

A comprehensive roadmap for building AI-Driven applications with local LLMs

Shamim Bhuiyan and Timur Isachenko

This book is available at https://leanpub.com/quickstartwithai

This version was published on 2025-07-09



This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2025 Shamim Bhuiyan

Tweet This Book!

Please help Shamim Bhuiyan and Timur Isachenko by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Just purchased ”Generative AI with local LLM”
https://leanpub.com/quickstartwithai by @shamim_ru #AI #LLM #GenerativeAI #OLLAMA #LLAMA #AIAGENT #BOOK

To my loving son, Mishel, whose curious questions about why I spend so much time at my computer inspire me every day.

Contents

Preface **1**
What this book covers 4
Code Samples 6
Readership 6
Conventions 6
Reader feedback 8

About the authors **9**

Acknowledgments **10**

Chapter 1: Getting started with Local LLM **11**
Local LLM inference tools 12
Tools and frameworks used in this book 14
Installing and setting up the local LLM inference 16
Useful commands and interfaces 21
Additional setup 23
Uninstall LLM inference 24
Installing a graphical user interface (GUI) client to work with local LLM . . 24
Local LLM performance tuning 25
Configure a Python virtual environment for AI development 26
Install Python 3 26
Install Python package manager pip3 26
Installing and configuring Miniconda 26
Install IDE: Jupyter lab and notebook 27
Install and configure SQLLite database 27
Additional setups 27

Develop your first application with local LLM 27
Troubleshooting 30
Hardware acceleration 31
Using a Workstation with GPU 33
Enabling AVX/AVX2 for CPU acceleration 34
Using 3rd party ASIC platform or VPS with GPU support 34
Using Google Colab or Kaggle service 35
Conclusion 35

Chapter 2: Deep dive into the theories of Generative AI 37
Artificial Intelligence (AI) 37
Machine Learning (ML) 38
Deep Learning (DL) 41
Natural Language Processing (NLP) 44
Transformer 45
Self-Attention mechanism 46
Encoder-Decoder architecture 47
Generative AI 50
What is Generative AI and what is not? 50
Categories of Generative AI 52
Large Language Model 56
How LLM works internally? 57
Tokenization 59
Vector 62
Embedding 65
Transformers 66
Training LLM 67
Pre-training 67
Fine-tuning 67
RAG 67
AI Agents 69
Prompt engineering 69
Resources 70
Conclusion 70

Chapter 3: RAG, enrich LLM models with private datasets 71

RAG vs fine-tuning LLMs . . . 71
Key concepts of RAG . . . 73
Embeddings . . . 73
Vector database . . . 74
Semantic Search . . . 76
How semantic search is different from full text search? . . . 78
Real world use cases of using RAG . . . 79
Implementing RAG in a private company . . . 79
Step-by-Step Example: Loading, retrieving, and processing custom documents with LLM . . . 79
Conclusion . . . 80

Chapter 4: Text-to-SQL, enhance your LLM responses by integrating data from the Database . . . 81
What is Text-to-SQL? . . . 81
Challenges of Text-to-SQL . . . 81
LLM for Text-to-SQL . . . 81
System design patterns of using Text-to-SQL with examples . . . 81
Design pattern 1. Generating and executing SQL queries . . . 82
Design pattern 2. Using Agent's for error handling and ensure correctness . . . 82
Design pattern 3. Text-To-SQL with RAG . . . 82
Conclusion . . . 82

Chapter 5: Fine-tuning LLMs . . . 83
Steps for Fine-tuning a pre-trained model . . . 85
Fine-tuning technics . . . 89
Full Fine-Tuning . . . 89
Parameter-Efficient Fine-Tuning (PEFT) . . . 91
LoRA (Low-Rank Adaptation) . . . 92
Quantized LoRA (QLoRA) . . . 93
Knowledge Distillation (KD) . . . 94
Popular frameworks used for fine-tuning LLMs . . . 96
Step-by-step example of fine-tuning an LLM . . . 98
Prerequisites . . . 98

Part 1. Analyze business requirements, choosing a base model and environment setup . . . 99
Part 2. Exploring the training dataset . . . 99
Part 3. Dataset pre-processing and adapter configuration . . . 99
Part 4. Train the model . . . 99
Part 5. Evaluate the model . . . 99
Part 6. Save & deploy the final model . . . 99
Conclusion . . . 100

Chapter 6: Image processing & generating with LLM . . . 101
Image visioning . . . 101
Possibilities and Functionalities of LLaVA-v1.6 . . . 101
LLaVa architecture . . . 101
Step-by-Step Example: Utilizing LLaVA-v1.6 for Image Visioning . . . 101
Incorporating LLaVA into your application for image processing . . . 101
Image processing . . . 102
Tips for Better Results . . . 102
References . . . 102
Conclusion . . . 102

Chapter 7: Developing and utilizing AI agents . . . 103
The future of AI agents . . . 103
Difference between AI Agents and AI Tools . . . 105
Use cases of AI agents in Generative AI . . . 105
Use cases from a developer's perspective . . . 105
Use cases from a product manager's perspective . . . 106
Classification of AI Agents in Generative AI . . . 106
AI agents architecture . . . 106
Frameworks for developing AI Agents . . . 109
Langchain and LangGraph . . . 109
CrewAI . . . 109
OpenAI Agents SDK . . . 109
Developing AI Agents with different frameworks: a step-by-step guide . . . 113
Conclusion . . . 114

Chapter 8: Advanced Tools and Techniques . . . 115

A Simple Protocol for Communicating with Local and Cloud LLMs : Minion(s) . . . 115
Minion . . . 116
Minions . . . 118
Examples . . . 122
Connecting Local LLMs to Various Data Sources: Model Context Protocol (MCP) . . . 122
Core Architecture . . . 124
Communication details . . . 126
Using MCP: A Practical Walkthrough . . . 127
Extending the Practical Walkthrough: From CLI to MCP Agent based automation . . . 127
Why RESTful applications still matter . . . 127
Application integration overview . . . 128
What Is MCP Agent? . . . 128
How the entire work flow works? . . . 128
Benefits of Integrating MCP Server with REST API . . . 131
Use Case: AI-Driven warehouse automation . . . 131
Why not integrate MCP server directly with the Database? . . . 132
Understanding MCP Agent architecture . . . 143
Potential Improvements to the Pipeline . . . 150
Google's Agent-to-Agent Protocol: enabling cross-agent communication . . . 151
Pros and cons of the A2A Protocol . . . 152
A2A vs. MCP — not a battle, but a partnership . . . 153
Let's Build: a travel planner AI Agent using A2A protocol . . . 154
Conclusion . . . 155

Final words . . . 157

Preface

Artificial Intelligence (AI) has become incredibly popular in recent times, making headlines everywhere—from school discussions to debates in the US Senate. This fast-growing field is grabbing attention from both experts in machine learning and everyday people alike. While some worry about where AI is headed, others even suggest extreme actions, like destroying data centers. Even the *White House* is weighing in on the future of AI.

But rather than fearing AI's rise, we believe it's more productive to focus on how it can help us in our daily lives—whether through Siri or other AI-powered assistants.

There are countless articles about AI, especially about Generative AI and large language models, aimed at readers with different levels of expertise. Unfortunately, most of these require some technical background to fully understand.

And there's no straightforward set of guidelines for learning everything about AI, which is why we wrote this book. It's designed to be a clear and easy-to-follow roadmap for beginners, guiding you through the complicated language and concepts. Don't worry, you won't need to be a machine learning expert to get the hang of it!

After weeks of research and planning, we've come up with a practical learning path for Generative AI, and we're excited to share it with you.

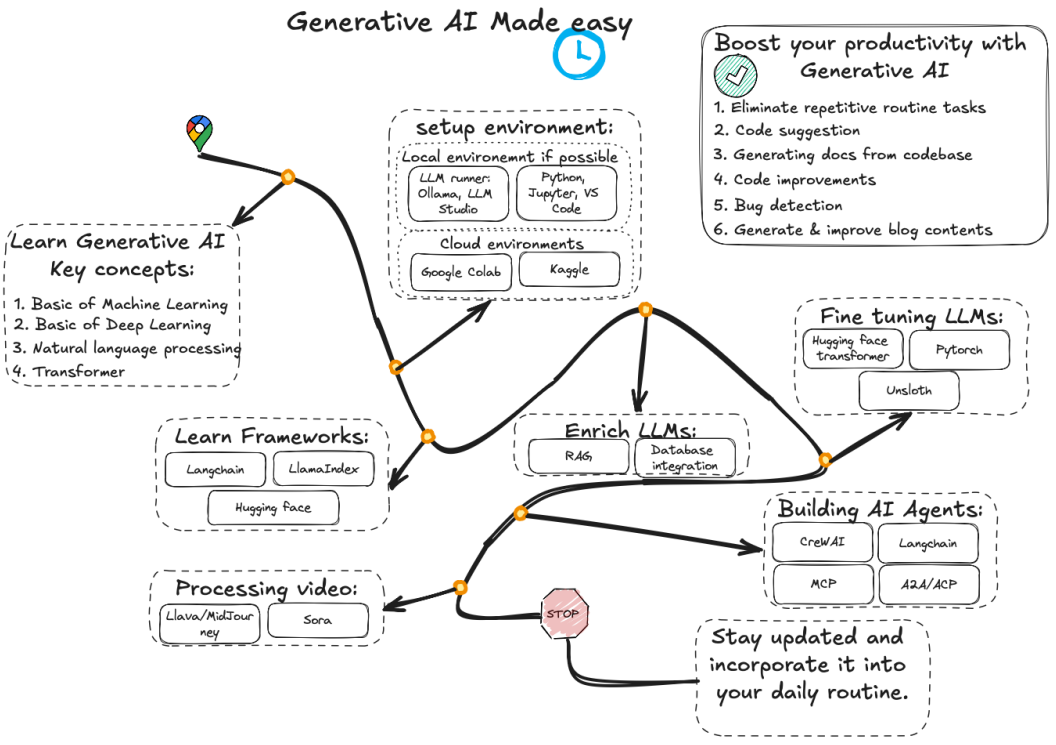


Figure 1.

The Generative AI learning path is structured into 7 interconnected stages:

1. **Understand Generative AI key concepts**:

    - Artificial Intelligence: Learn the basic principles of AI, including what it is and how it's applied.
    - Machine Learning: Study the core concepts of machine learning, such as supervised and unsupervised learning, algorithms, and model training.
    - Deep Learning: Dive into deep learning basics, focusing on neural networks and how they work, including popular architectures like CNNs and RNNs.
    - Natural Language Processing (NLP): Get familiar with NLP, which deals with the interaction between computers and human language.

These fundamentals are crucial for learning more advanced AI topics.

2. **Setup environment with local LLMs**:

    - Run Local Models: Experiment with running local models using tools like *Ollama* or other open-source implementations. Learn how to set up and use models on your local machine.
    - Set up your local Python environment with Miniconda and JupyterLab to begin your journey into Generative AI.
    - Explore Model Sharing Platforms: Use platforms like *Hugging Face* to access pre-trained models and datasets. This will allow you to experiment with a wide range of generative AI models.

3. **Learn frameworks**:

    - LangChain: Start using LangChain to build applications that leverage language models. This will help you integrate generative AI into real-world applications.
    - LlmIndex: Explore LlmIndex for indexing and searching through large language models.
    - Python: Enhance your programming skills in Python, the most widely used language for AI development. Focus on libraries like *TensorFlow*, *PyTorch*, and *Hugging Face's* Transformers.

4. **Fine-tuning/Enrich LLMs**:

    - Fine-Tuning Techniques: Learn how to fine-tune pre-trained models to better suit specific tasks or datasets.
    - Enrichment Techniques: Study methods like Retrieval-Augmented Generation (RAG) to improve the output quality of your models.

5. **Build projects and Agents**:

    - Small Projects: Start with small projects, such as generating text, images, or other media, to reinforce your learning.
    - Agents: Developing agents that can handle some of your daily routine tasks, such as searching the internet and generating quality text for blog writing on specific topics.

6. **Advanced topics**: Once you're familiar with the basics, move on to more advanced concepts such as the *Model Context Protocol (MCP)* for tool integrations and Google's Agent-to-Agent (A2A) protocol for secure, structured agent communication. Learn how Minions enable seamless integration between edge devices and cloud-based models. You'll also explore reinforcement learning, advanced model architectures, real-time AI systems, and performance optimization for edge computing.
7. **Stay updated and keep learning**:

    - Follow AI Trends: Stay updated with the latest advancements in AI by following relevant blogs, YouTube channels, and online courses.
    - Collaborate and Share: Engage with online communities, participate in AI challenges, and share your projects on platforms like GitHub.

This path will give you a strong foundation in Generative AI, allowing you to progressively build your expertise and apply it to real-world scenarios.

What this book covers

This book provides a comprehensive guide to leveraging Generative AI and Large Language Models (LLMs) in a local development environment. It is designed to take you from the basics of Generative AI to advanced techniques like fine-tuning models, enriching them with private datasets, and applying them in practical scenarios such as SQL querying, image processing, and developing Agents. Whether you're a *product owners*, *developers*, *data scientists*, and *AI enthusiasts*, this book will equip you with the knowledge and tools needed to effectively utilize AI in your projects.

Chapter 1. Getting started with Local LLM

Before diving into AI model implementation, it's essential to set up a robust local development environment. This chapter guides you through the process of configuring your machine for AI development, including installing necessary software, setting up Python environments, and choosing the right hardware (like GPUs) for optimal performance. The chapter also introduces tools like Jupyter notebooks which will help streamline your AI development workflow.

Chapter 2. Deep dive into the theories of Generative AI

The journey begins with an introduction to Generative AI, explaining what it is, how it works, and why it is revolutionizing various fields. This chapter covers the fundamental concepts of AI, machine learning, and deep learning, laying the groundwork for understanding how LLMs generate content such as text, code, and images. You'll explore various types of generative models, including the Self-Attention mechanism, Encoder-Decoder architecture, and Transformers, along with their applications across different industries.

Chapter 3: RAG, enrich LLM models with private datasets

This chapter explores the concept of Retrieval-Augmented Generation (RAG), an advanced technique that enhances LLMs by incorporating private datasets. You'll discover how to enrich your AI models with domain-specific knowledge, allowing them to generate more accurate and relevant outputs. The chapter outlines the technical steps for setting up RAG, including Vector DB, indexing private data, and configuring the model to retrieve and generate responses based on this enriched information.

Chapter 4: Text-to-SQL, enhance your LLM responses by integrating data from the Database

AI-driven automation in database interactions is increasingly valuable, and this chapter focuses on using LLMs for Text-to-SQL conversion. You'll learn how to use models that can translate natural language queries into SQL commands, specifically tailored for interactions with BigQuery. The chapter includes practical examples and code snippets, showing how to create systems that allow users to interact with databases without needing to know SQL, making data retrieval more accessible and efficient.

Chapter 5: Fine tuning LLM model

This chapter takes a deep dive into the fine-tuning process, a crucial step in customizing LLMs to better suit your specific needs. You'll explore how to adapt pre-trained models to your data, improving their performance on tasks relevant to your domain. The chapter covers the technical aspects of fine-tuning, including data preparation, adjusting hyperparameters, and evaluating model performance. By the end of this chapter, you'll be able to refine models to achieve higher accuracy and relevance in your applications.

Chapter 6: Image processing & generating with LLM

Generative AI is not limited to text; it also plays a significant role in image processing and generation. This chapter explores how LLMs can be used to create and manipulate images. The chapter provides practical examples and tools, including using models like LLaVa, OpenJourney, to help integrate image generation into your projects.

Chapter 7: Developing and utilizing AI agents

This chapter focuses on one of the most exciting applications of LLMs: developing AI Agents. You'll learn about the purpose and benefits of using AI Agents to automate daily tasks. The chapter explores the architecture of AI agents and their key components. Toward the end, we introduce a multi-agent framework that enables non-technical users to manage AI agents and tasks using intuitive, high-level abstractions.

Chapter 8: Advanced Tools and Techniques

The final chapter provided a comprehensive exploration of advanced tools and techniques for enhancing AI applications using the Model Context Protocol (MCP) framework and Minion(s) protocol. The chapter highlighted how tools like mcp-cli empower developers to experiment with language models and interact with MCP servers programmatically. This opens up possibilities for automating tasks, summarizing content, and accessing structured data through AI-driven prompts.

Code Samples

All code samples, scripts, and more in-depth examples can be found on the GitHub repository.

Readership

The target audiences of this book are *product owners*, *developers*, *data scientists*, and *AI enthusiasts* with minimum programming knowledge. No excessive knowledge is required, though it would be good to be familiar with *Python*, *Java* and tools like *Conda*.

Conventions

The following typographical conventions are used in this book:

1. *Italic* and **Bold** indicates new terms, important words, URL's, filenames, and file extensions.
2. A block code is set as follows:

```
def process_image(image_file_path, prompt):
    print(f"\nProcessing {image_file_path} file \n")
    image = Image.open(image_file_path)
    display(image)

    with image as img:
        with BytesIO() as buffer:
            img.save(buffer, format='PNG')
            image_bytes = buffer.getvalue()

    # Generate a description of the image
    for response in generate(model='llava:34b',
                             prompt=prompt,
                             images=[image_bytes],
                             stream=True):
        # Print the response to the console and add it to the full resp\
onse
        print(response['response'], end='', flush=True)
```

3. Any command-line **input** or **output** is written as follows:

```
!pip install ollama

export OLLAMA_HOST="192.168.1.124"

Processing ./Text2SQL.png file
```



Tip

This icon signifies a tip, suggestion.



Warning

This icon indicates a warning or caution.



Info

This icon signifies general note.

Reader feedback

We would like to hear your comment such as what you think, like or dislike about the content of the book. Your feedback will help us to write a better book and help others to clear all the concepts. To submit your feedback, please use the feedback link.

About the authors

Shamim Bhuiyan is currently working as an Enterprise architect; where he's responsible for designing and building out highly scalable, and high-load middleware solutions. He received his Ph.D. in Computer Science from the University of Vladimir, Russia in 2007. He has been in the IT field for over 22 years and is specialized in Middleware solutions, Big Data and Data science. Also, he is a former SOA solution designer, speaker, and Big data evangelist. Actively participates in the development and designing of high-performance software for IT, telecommunication and the banking industry. In spare times, he usually writes the blog frommywork-shop and shares ideas with others.

Timur Isachenko is an accomplished Technical Lead and Solution Architect, known for his expertise in backend development and microservices architecture. With over 14 years in the IT industry, he has a rich history of developing and implementing high-load systems across the finance, healthcare, and online banking sectors. Timur holds a Specialist Degree in Computer Science and Applied Mathematics from Kuban State University.

He has led numerous projects, including the development of educational platforms, billing systems for betting and gambling platforms, and integration services for major enterprises. Timur is also a co-author of "High Performance In-Memory Data Grid with Ignite" and has been a speaker at various conferences.

In his personal life, Timur is a dedicated father to three daughters and an avid runner who once ran every day for an entire year.

Acknowledgments

This book would not have been possible without the support and contributions of several people.

First and foremost, I would like to express my deepest gratitude to my co-author **Timur Isachenko**. Your insights, expertise, and dedication were invaluable throughout this journey. Thank you for your collaboration and commitment to making this project a reality.

To my son **Mishel**, thank you for your unwavering encouragement and understanding during the long hours spent working on this book. Your patience and belief in me have been a constant source of motivation.

I would also like to extend a special thanks to the bloggers and vloggers on YouTube who generously shared their knowledge and experiences. Your valuable information and support played a crucial role in shaping this book, and I am grateful for the content you provide to the community.

Finally, I want to extend a heartfelt thank you to our readers who purchased and read our book at the early stage and provides useful commentaries and reviews. Your curiosity and passion for learning are the true driving forces behind this work. I hope that our work will inspire you to explore the exciting possibilities of AI and language modeling, and that you'll find the information and examples in these pages helpful as you embark on your own journey.

Your feedback and engagement are greatly appreciated and will continue to inspire future endeavors.

Chapter 1: Getting started with Local LLM

I still remember my summer vacation in Sharm-el-sheikh, Egypt, many years ago. This beautiful resort town is situated on the banks of the Red Sea. On the second day of my trip, I decided to take a swim in the sea and was amazed by the underwater world teeming with corals, fish, and other marine life. From that moment on, I fell in love with the sea. My friend and I were inspired to try scuba diving for the first time. We were both excited but also slightly apprehensive about the process.

Our trainer noticed our fear, smiled, and said something that stayed with me: *Underwater is another beautiful world. It may not be comfortable at first, but you have to try it to know if it's yours or not.* With his motivation, I took the plunge (literally!) and was hooked on diving from that day forward. Sadly, my friend couldn't handle the ear pressure pain and never dove again. However, for me, that experience taught a valuable lesson: when faced with something new and daunting, just try it out first to see if it's right for you.

Why am I sharing this story? When I started learning Generative AI, I felt similarly overwhelmed by the fragmented information and warnings from others that it was *very hard* to learn and required top-notch hardware. But as soon as I found a clear study roadmap and tried my hand at applying it, I understood why it was mine – because I had taken the first step and made it my own.

So, my advice is: just try out your first application or project in Generative AI, and see if it's something you enjoy. If not, no worries! But if it sparks your interest, you'll be hooked from there on.

Anyway, in my opinion the best way to learn something new is to jump right in and start with a simple example to experiment with. By doing this, you'll gain the necessary knowledge and become familiar with the most common aspects of the particular technology. Once you have a basic understanding of what you can do with it, you can always explore more details later.

As a starting point, this chapter provides basic instructions for installing the **Ollama LLM runner**. Later, we will discuss how to set up a local environment for Python so you can begin developing applications with an open-source local LLM.

In a nutshell, the following topics are covered in this chapter.

1. Tools and frameworks used in this book.
2. Installing and setting up the LLM inference tool: Ollama.
3. Installing a graphical user interface (GUI) client to work with Ollama.
4. Configuring a Python environment for AI development.
5. Hardware acceleration.

Before we start

Let's explore why we chose to use a local Large Language Model (LLM) instead of a provider like OpenAI. There are four key reasons that summarize this decision:

1. Fine-tuning: You may have complex use cases that require customizing the LLM model, which is easier to manage locally.
2. Privacy and security: Your company might enforce strict policies against sharing proprietary code or sensitive information with third-party providers, due to potential risks of data collection and misuse.
3. Global availability: In many countries, private or third-party LLM providers may not be accessible, making a local solution necessary.
4. Cost-effectiveness: For large-scale, user-facing production services, you may need a cost-efficient solution based on your financial model.

While some in the open-source community may view local LLM development as a fun and interesting hobby, our experience has shown that it's also viable for business use cases. By choosing to host our own LLM locally, we can ensure that our needs are met while maintaining control over sensitive information.

Local LLM inference tools

To run an LLM locally, we need a specialized tool or application capable of executing the model on our device. This type of tool is known as an **LLM inference** tool or **LLM runner**. An LLM inference tool is a software framework or library designed to run pre-trained Large Language Models (LLMs) on a local machine, cloud server, or edge device. These tools optimize performance, manage memory efficiently, and enable seamless text generation without requiring deep machine learning expertise.

One of the earliest and most widely used tools for local LLM inference is llama.cpp. It enables efficient model execution with minimal setup while delivering state-of-the-art performance across various hardware platforms—both locally and in the cloud. Written in plain C/C++, llama.cpp serves as the core for many other LLM inference tools, including **Ollama** and **RamaLlama**.

While llama.cpp is a robust tool, it requires manual compilation and command-line usage, making it more suitable for advanced and technical users. However, as of the time of writing this book, several other open-source LLM inference tools are available, offering varying levels of usability and features.

Below is a detailed comparison table of some popular LLM inference tools for running and managing LLMs locally. This table evaluates Ollama, llama.cpp,RamaLlama, LM Studio, Nexa SDK, TransferLab, and vLLM vLLM across key aspects such as interface, ease of use, customization, model management, performance, operating system support, and unique features.

Tool	Interface	Ease of Use	Customization & Flexibility	Performance & Optimization	OS Support	Unique Features
Ollama	CLI with REST API support	Simple & beginner-friendly with straightforward commands	- Flexible via modelfiles; - supports multiple LLM libraries (CPU, GPU, etc.).	- Optimized autodetection of hardware; - leverages GPU acceleration	Cross-platform	- Experimental OpenAI API compatibility; - multimodal input support (e.g., images).
llama.cpp	CLI	- Requires compilation and command-line usage; - more technical.	- Highly flexible; - extensive parameter tuning; - open-source and community maintained.	Lightweight with efficient CPU performance and low memory footprint	Cross-platform	- Supports various quantization techniques; - - highly community-driven.
Ramallama	CLI	Designed to streamline model switching and ease repetitive tasks	Focuses on auto-unload and robust memory management for frequent model swapping	Optimized for constrained systems with rapid model loading and low memory use	Cross-platform	Emphasizes automated unloading and memory optimization.
LM Studio	GUI-based	- Extremely user-friendly; - ideal for newcomers.	Extensive configuration options via a polished GUI interface	- Provides solid local performance plus a local inference server; - High memory foot print.	Windows, macOS, Linux (beta)	- Polished chat interface; - built-in model discovery; - integrated local inference server.
Nexa SDK	SDK/CLI toolkit	- Developer-focused; - requires technical integrationю	Offers advanced quantization options and optimization parameters	Reduces model file size and RAM usage while preserving accuracy (e.g., NexaQuant)	Cross-platform	- Specializes in quantization/optimization; - works seamlessly with tools like Ollama or LM Studio
vLLM	CLI with OpenAI-compatible API server	- Developer-friendly; - integrates seamlessly with Hugging Face models.	- Supports various decoding algorithms; tensor and pipeline parallelism; - quantization	- High throughput with efficient memory management via PagedAttention; - supports GPUs and CPUs	Cross-platform	- PagedAttention mechanism for efficient KV cache management; - supports speculative decoding and structured outputs.

Figure 1.2

Tools and frameworks used in this book

In this book, we will explore a variety of tools and frameworks that are essential for working with LLMs and artificial intelligence (AI) applications. These tools have been carefully selected based on their widespread use, robust features, and ability to handle the demanding computational requirements associated with AI development. By the end of this book, you will have a strong understanding of how to utilize these tools to create, fine-tune, and deploy AI models effectively.

Python is the primary programming language used throughout this book, given its popularity in the AI community and the vast ecosystem of libraries and frameworks available. Python's simplicity and readability make it an ideal choice for both beginners and experienced developers working with AI.

For building, training and enrich LLM models, we will rely on powerful frameworks like Langchain, TensorFlow and PyTorch. These frameworks provide comprehensive

tools for creating neural networks, running computations on GPUs for accelerated performance, and building custom AI solutions. They are highly flexible and have strong community support, making them indispensable for modern AI research and development. In particular, PyTorch's dynamic computation graph and TensorFlow's production-ready capabilities make them suitable for a wide range of applications, from academic research to industrial deployment.

To handle natural language processing (NLP) tasks, we will use the **Hugging Face** Transformers library. This library provides access to pre-trained models and tools for fine-tuning state-of-the-art transformer models, such as BERT, GPT, and T5. Hugging Face's intuitive interface and extensive documentation make it a go-to resource for working with LLMs, enabling developers to build sophisticated NLP applications with minimal effort.

In addition to these core tools, we will explore **Ollama** for setting up and managing local LLMs. Ollama simplifies the deployment of LLMs on local machines, offering a user-friendly interface for running models, managing dependencies, and optimizing performance. By using Ollama, developers can maintain control over their data and models, ensuring privacy and security while benefiting from the power of AI.



Info

Please note that, If you are an advanced user, you can choose any other LLM inference tool instead of Ollama. However, we recommend using Ollama because it is extremely easy to install and use, requiring no additional prerequisites - making it a true plug-and-play solution

Let's summarize everything in a table.

Features	Tools and framework
LLM runner	Ollama, Groq
LLM	Llama, codestral, Llava, Qwen, Openjourney
Programming language	Python
App developing	Langchain, Vanna, CrewAI, OpenAI SDK
ML platform	Hugging face
Fine-tuning	Pytorch
Image processing	Llava, Openjourney

Together, these tools and frameworks provide a comprehensive toolkit for developing, fine-tuning, and deploying AI models. Throughout this book, we will guide you step-by-step on how to integrate these tools into your projects, helping you unlock the full potential of AI and LLMs.

Running an LLM locally requires sophisticated software components and hardware, including a multicore CPU, sufficient RAM, and possibly a GPU. This setup can be achieved on various platforms, such as a workstation, a home lab server, or a rented dedicated VPS from a provider like Amazon. When installing an LLM locally, several considerations must be taken into account.

Components	Description
CPU	Octa-core (minimum) Intel/AMD processor
RAM	Minimum 16 Gb
OS	Linux, MacOs
Network	Connected to internet

Installing and setting up the local LLM inference

As previously mentioned, our AI-related applications are built around Ollama, an open-source tool that efficiently manages LLMs. This versatile tool can be used to run a wide range of LLMs locally. One key advantage of Ollama over other tools

like *LM Studio* is its exceptional performance: it is fast and has a small memory footprint. Additionally, Ollama provides a necessary API for interacting with it, making integration into your system easy.

For our local inference needs, we will be using an open-source LLM in conjunction with Ollama. This combination will enable us to leverage the full potential of Ollama while ensuring seamless and efficient inference operations.

Ollama can be tried in various ways, depending on your desired level of involvement and interest. For a quick start, you can try using the binary distribution, or if you prefer to have more control, you can build Ollama from its source code. In this section, we will guide you through the installation process using the binary distribution.

Step 1: Download and Install Ollama.

- Visit the Ollama website to download the latest version of the Ollama installer for your operating system. Look for a direct download link or an installation guide.
- For **macOS**: Open the downloaded file and follow the on-screen instructions to install Ollama on your system. This typically involves dragging the application to the Applications folder on macOS.
- For **Linux**:
  - execute the command `curl -fsSL https://ollama.com/install.sh | sh`
  - After successful installation, you should see an output similar to the one shown below.

```
>>> Making ollama accessible in the PATH in /usr/local/bin
>>> Adding ollama user to render group...
>>> Adding ollama user to video group...
>>> Adding current user to ollama group...
>>> Creating ollama systemd service...
>>> Enabling and starting ollama service...
>>> The Ollama API is now available at 127.0.0.1:11434.
>>> Install complete. Run "ollama" from the command line.
WARNING: No NVIDIA/AMD GPU detected. Ollama will run in CPU-only mode.
```

- For **Windows**: At the time of writing this book, Ollama is available as a preview version specifically for the Windows operating system.
- For **Docker**: Make sure that, your Docker instance is up and running. Execute the following command to run Ollama inside a Docker container.

```
docker run -d -v ollama:/root/.ollama -p 11434:11434 --name ollama olla\
ma/ollama
```

Once installed, you can verify the installation by opening a terminal and running the following command:

```
ollama --version
```

This command should display the current version of Ollama installed on your machine, confirming that the installation was successful.

```
Warning: could not connect to a running Ollama instance
Warning: client version is 0.3.6
```

The output above indicates that the Ollama instance is not running and the client version is 0.3.6

Step 2: Start the Ollama instance.

There are two easy steps to start an Ollama instance:

- To start an Ollama instance *without* LLM, use the following command:

```
ollama serve
```

This command initializes the Ollama server, allowing it to run and manage LLMs locally. The output should be similar to what is shown below.

```
2024/08/27 15:17:09 routes.go:1125: INFO server config env="map[OLLAMA_\
DEBUG:false OLLAMA_FLASH_ATTENTION:false OLLAMA_HOST:http://127.0.0.1:1\
1434 OLLAMA_KEEP_ALIVE:5m0s OLLAMA_LLM_LIBRARY: OLLAMA_MAX_LOADED_MODEL\
S:0 OLLAMA_MAX_QUEUE:512 OLLAMA_MODELS:/Users/shamim/.ollama/models OLL\
AMA_NOHISTORY:false OLLAMA_NOPRUNE:false OLLAMA_NUM_PARALLEL:0 OLLAMA_O\
RIGINS:[http://localhost https://localhost http://localhost:* https://l\
ocalhost:* http://127.0.0.1 https://127.0.0.1 http://127.0.0.1:* https:\
//127.0.0.1:* http://0.0.0.0 https://0.0.0.0 http://0.0.0.0:* https://0\
.0.0.0:* app://* file://* tauri://*] OLLAMA_RUNNERS_DIR: OLLAMA_SCHED_S\
PREAD:false OLLAMA_TMPDIR:]"
time=2024-08-27T15:17:09.034+03:00 level=INFO source=images.go:782 msg=\
"total blobs: 10"
time=2024-08-27T15:17:09.036+03:00 level=INFO source=images.go:790 msg=\
"total unused blobs removed: 0"
time=2024-08-27T15:17:09.037+03:00 level=INFO source=routes.go:1172 msg\
="Listening on 127.0.0.1:11434 (version 0.3.6)"
time=2024-08-27T15:17:09.038+03:00 level=INFO source=payload.go:30 msg=\
"extracting embedded files" dir=/var/folders/lj/qs6zzt3j7jvb5yrh4x1x39x\
00000gn/T/ollama883440228/runners
time=2024-08-27T15:17:09.087+03:00 level=INFO source=payload.go:44 msg=\
"Dynamic LLM libraries [cpu cpu_avx cpu_avx2]"
time=2024-08-27T15:17:09.087+03:00 level=INFO source=types.go:105 msg="\
inference compute" id="" library=cpu compute="" driver=0.0 name="" tota\
l="16.0 GiB" available="6.3 GiB"
```

- To start an Ollama instance *with* a LLM. Use the following command to start Ollama instance with model Llama3.1:

```
ollama run llama3.1
```

This command will pull the `llama3.1 8b` model from the repository, start an Ollama server, and load the llama3.1 model for processing queries or tasks locally. The following output shows the result of Ollama pulling and loading the LLM `llama3.1`.

```
ollama run llama3.1buj
pulling manifest
pulling 8eeb52dfb3bb... 100% ▕▕▕▕▕▕▕▕▕▕▕▕ 4.7 GB/4.7 GB  2.7 MB/s      \
0s
pulling 11ce4ee3e170... 100% ▕▕▕▕▕▕▕▕▕▕▕▕ 1.7 KB
pulling 0ba8f0e314b4... 100% ▕▕▕▕▕▕▕▕▕▕▕▕  12 KB
pulling 56bb8bd477a5... 100% ▕▕▕▕▕▕▕▕▕▕▕▕   96 B
pulling 1a4c3c319823... 100% ▕▕▕▕▕▕▕▕▕▕▕▕  485 B
verifying sha256 digest
writing manifest
removing any unused layers
success
>>> Send a message (/? for help)
```

The entire list of models supported by Ollama is available here.



Info

Please note that, running a LLM locally requires sufficient RAM. Specifically: for 7B models, at least 8 GB of free RAM is required, for 13B models, 16 GB of free RAM is recommended and for 33B models, 32 GB of free RAM is necessary.

Ollama will only pull a model during its initial setup. Subsequent runs of the command `ollama run llama3.1` will use the locally cached LLM, rather than pulling it from the repository.

Now that the LLM is set up, you can interact with it by sending messages or chatting with it. Try typing something like `tell me a joke` to get started.

At this moment, executing the command `ollama ps` in another terminal window will produce a similar output to what is shown below.

```
NAME            ID              SIZE    PROCESSOR UNTIL
llama3.1:latest 91ab477bec9d    6.2 GB  100% CPU  4 minutes from now
```

If you have a GPU-enabled system, the output should look something like this:

```
NAME            ID           SIZE   PROCESSOR UNTIL
llama3.1:latest 62757c860e01 6.7 GB 100% GPU  4 minutes from now
```

To exit the chat, simply type `/bye` in the command line. This will not only end the conversation but also stop the currently running LLM.

Let's summarize the difference between the two commands: `ollama serve` and `ollama run llama3.1`.

The command `ollama serve` starts the Ollama instance but does not initialize any specific LLM. It sets up the server, making it ready to handle requests, but leaves the choice of which model to use to subsequent commands. On the other hand, the command `ollama run llama3.1` does both: it starts the Ollama server and immediately initializes the `llama3.1` model. This command sets up the server and loads the specified model for processing queries or tasks locally.

Useful commands and interfaces

Here are some useful Ollama commands which provides basic control over the Ollama system, allowing you to check on its status or shut it down when you're finished using it.

- `ollama list-models`: Lists all available models that can be used with Ollama. This command helps you view which models are installed and ready for use.
- `ollama pull <model_name>`: Downloads the specified model (e.g., llama3.1) from the repository to your local system. Use this command to obtain models that are not yet installed.
- `ollama rm llama3.1`: Remove a LLM from the local repository.

- `ollama create <model_name>`: Creates a new model from a specified `GGUF` Modelfile. Use this command to initialize and set up a model based on the configuration and parameters defined in the Modelfile. This is useful for creating custom models tailored to specific tasks or datasets. To that:

  - Download a pre-trained Large Language Model (LLM) file in the GGUF format from Hugging Face or another source - for example, `llama-3-sqlcoder-8b-Q6_K.gguf`. Save this file to a specific directory on your computer.
  - Next, create a new text file called `Modelfile` within the same directory. Inside the `Modelfile`, add the following line of code: `FROM ./llama-3-sqlcoder-8b-Q6_K.gguf`.
  - Finally, navigate back to your terminal and run the command `ollama create llama-sqlcoderq6 -f Modelfile` from within the same directory.
  - After a while, you should see Ollama create and configure a new LLM model for you to work with.



Info

Alternatively, starting from Ollama 3.2, you can directly download and run GGUF models from Hugging Face without any extra manipulation. Use the following command: `ollama run hf.co/bartowski/Llama-3.2-1B-Instruct-GGUF`. In this command, `bartowski` is the username, and `Llama-3.2-1B-Instruct-GGUF` is the repository name.

Ollama provides a useful REST API for interacting with and managing models. To use the REST API, you need to have `CURL` or `WGET` installed in your terminal, or you can use any REST client like `Postman`.

- To load a model, execute the following request:

```
curl http://localhost_OR_IP_ADDRESS:11434/api/generate -d '{
  "model": "gemma2"
}'
```

- To getting a list of models that are currently loaded into memory:

```
curl http://localhost_OR_IP_ADDRESS:11434/api/ps
```

- To generate a Stream request:

```
curl http://localhost_OR_IP_ADDRESS:11434/api/generate -d '{
  "model": "llama3",
  "prompt": "Tell me a joke"
}'
```

The response should be return as a stream. The complete REST API documentation can be found here.

Additional setup

If you're running Ollama on a Linux system, such as Ubuntu, there's an essential command you'll want to know:

- `systemctl stop ollama.service`: This command stops the ollama.service using the systemctl command, which is part of the system and service manager on Linux-based systems. By executing this command, you terminate the Ollama service that is currently running. It is more commonly used in environments where Ollama is set up as a systemd service.
- `systemctl disable ollama.service`: This command disables the ollama.service, preventing it from starting automatically during the system boot. Disabling the service is useful if you want to stop Ollama from running on startup, allowing for manual control over when and how the service is launched.

- `systemctl status ollama.service`: This command checks and displays the current status of the ollama.service. It provides information on whether the service is active, inactive, running, or stopped.
- `sudo journalctl -u ollama.service > ollama_logs.txt`: This command uses `journalctl` to fetch the logs for the `ollama.service` and redirects the output into a file named `ollama_logs.txt`.

These commands are essential for managing the Ollama service effectively, allowing you to start, stop, disable, and check the status of the service as needed. They provide a structured way to control the behavior of Ollama in a production or development environment.

Uninstall LLM inference

If you need to remove Ollama runner from your Linux system or start fresh with a new installation, follow these steps:

- Stop the Ollama server: `systemctl stop ollama.service`.
- Disable the Ollama service: `systemctl disable ollama.service`.
- Remove the service file: `sudo rm /etc/systemd/system/ollama.service`.
- Delete the Ollama binary distribution: `sudo rm $(which ollama)`.
- Remove the downloaded models: `sudo rm -r /usr/share/ollama`.
- Optionally you can also delete the Ollama user and group: `sudo userdel ollama` `sudo groupdel ollama`.

The above commands should successfully uninstall Ollama from your Linux system. If you decide to reinstall the Ollama runner in the future, refer to the section on `Installing and Setting up the Local LLM Inference` for step-by-step instructions.

For macOS users, two manual steps are required to fully remove Ollama:

- Firstly, delete the `Ollama` application from your `Applications` folder.
- Secondly, delete the `.ollama` folder from your user directory to erase all downloaded LLMs.

Installing a graphical user interface (GUI) client to work with local LLM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Local LLM performance tuning

While using Ollama to run local LLMs, optimizing its performance can make a big difference, especially when handling large models or long contexts (> 1024). Here are a few practical tips to speed things up and make your experience smoother.

1. Enable Flash Attention. Modern LLMs often support Flash Attention, a technique that reduces memory usage as the context size increases. This not only speeds up model inference but also prevents slowdowns when processing longer inputs.

   ```
   export OLLAMA_FLASH_ATTENTION=1
   ollama serve
   ```

2. Implement Quantization. Quantization is a technique that compresses model weights, reducing their size and improving inference speed while maintaining reasonable accuracy. Ollama supports GGUF models, which are already quantized and optimized for better performance. When selecting models, prioritize quantized versions (e.g., q4_0, q8_0) for faster execution. For even greater efficiency, use lower-bit quantization (like q4_0), but keep in mind that smaller models may slightly reduce output quality.
3. Leverage GPU Acceleration. If your system has a compatible GPU, make sure Ollama is using it. GPU acceleration significantly speeds up model performance compared to CPU-only execution. Ensure the necessary drivers are installed, and Ollama will automatically detect and use the GPU (if available).

   ```
   export OLLAMA_CUDA=1
   ```

4. Limit Context Length. Longer contexts require more computation and memory. If performance is a concern, try reducing the context length or using models specifically optimized for shorter inputs. Adjusting the prompt size can also help maintain responsiveness without sacrificing accuracy.
5. Optimize System Resources. Ensure your system has enough RAM and CPU cores to handle LLM workloads efficiently. Closing unnecessary applications frees up resources for Ollama, improving its speed. Additionally, running Ollama on a dedicated server or a machine with modern hardware can provide substantial performance gains. Moreover, you can set the number of threads per CPU core as follows:

   ```
   export OLLAMA_NUM_THREADS=8
   ```

   This command allows Ollama to utilize multiple CPU cores efficiently.

By following these tips, you can significantly enhance the speed and efficiency of Ollama, allowing for a smoother and faster experience when working with local LLMs.

Configure a Python virtual environment for AI development

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Install Python 3

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Install Python package manager pip3

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Installing and configuring Miniconda

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Install IDE: Jupyter lab and notebook

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Install and configure SQLLite database

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Additional setups

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Develop your first application with local LLM

Now that we've completed our Large Language Model (LLM) inference and environment setup, we're ready to move on to developing a simple Python program using our local LLM. The primary objective of these exercises is to ensure that our entire setup, including local LLM inference, is working correctly. First and foremost, we want to verify that everything is functioning as expected.

For this exercise, we'll be using Ollama's native Python API to keep things simple and straightforward.

Step 1: Before proceeding, ensure that your Ollama Runner is up and running with an LLM model loaded. If not, execute the following command in your terminal:

```
ollama run llama3.1:latest
```

Step 2: Activate the Miniconda environment named `jupyterlab-env` using the following command:

```
conda activate jupyterlab-env
```

Step 3: Start the Jupyter lab by typing the next command:

```
jupyter lab
```

Step 4: Create a new Notebook on Jupyter.

Step 5: Rename the file `HelloWorld`.

Step 6: In the Jupyter Notebook, add the following Python code:

```
!pip install ollama

import ollama
from ollama import chat

messages = [
  {
    'role': 'user',
    'content': 'Write a python program to calculate factorial',
  },
]

from ollama import Client
client = Client(host='http://IP_ADDRESS:11434')
response = client.chat(model='llama3.1', messages=messages)

print(response['message']['content'])
```

The Python code is using the `ollama` library to interact with a chat-based AI model named `llama3.1`. Below is a step-by-step explanation of what each part of the code does:

- The `!pip install ollama` command is used to install the ollama Python package. The exclamation mark ! at the beginning indicates that this is a shell command, which is commonly used in Jupyter notebooks. The ollama package likely provides functionality to interact with AI models.
- `import ollama`: This line imports the ollama module, making all of its functions and classes available to use in the script.
- `from ollama import chat`: This line imports the chat function specifically from the ollama module, allowing you to use it directly without prefixing it with ollama.
- `messages`: A list named messages is defined with a single dictionary inside it. This dictionary represents a message that will be sent to the AI model. The key `role` indicates the role of the message sender, which is `user` in this case. The key `content` contains the actual text message: `Write a python program to calculate factorial`. This is the prompt that we are asking the AI model to generate Python code for calculating factorial.
- `from ollama import Client`: This line imports the `Client` class from the ollama module.
- `client = Client(host='http://IP_ADDRESS:11434')`: Here, we create an instance of the Client class, specifying the host URL of the LLM runner. If you're using a local Ollama Runner in the same host as your Python environment, you can use `localhost`.



Info

In this example, we're using a custom `Client` instance with explicit host specification to connect to an LLM (remote) runner. This code will work regardless of the Ollama runner topology: local or remote.

- `print(response['message']['content'])`: This line prints the content of the AI model's response to the console. In this case, you should see a working implementation of the Snake game printed out in Python code..
- Run the notebook by clicking the button `Restart the kernel and run all cells`.

After running the `llama3.1` model for a bit, you should see a Python program output that looks something like this:

```
def factorial(n):
    """
    Calculate the factorial of a number.

    Args:
        n (int): The input number.

    Returns:
        int: The factorial of the input number.
    """
    if not isinstance(n, int):
        raise TypeError("Input must be an integer.")

    if n < 0:
        raise ValueError("Input must be non-negative.")

    elif n == 0 or n == 1:
        return 1
    else:
        result = 1
        for i in range(2, n + 1):
            result *= i
        return result

# Example usage
print(factorial(5))  # Output: 120
```

You can copy and paste the above code into another Jupyter Notebook and attempt to run. The program should return Output:120. Don't worry if it doesn't work, getting the calculator programming code from the `llama3.1` model is what matters. If you're able to retrieve this code, it indicates that everything is working fine and your setup has passed a basic (smoke) test. The source code of the notebook is available here.

Troubleshooting

When working with the Python code involving the `ollama` library as shown before, you might encounter several common errors. Here are some possible issues and their explanations:

- `ConnectError: [Errno 61] Connection refused:`
  - Cause: This error happens if the server at the specified host address is not running, is down, or the IP address/port is incorrect.
  - Solution: Verify that the server is running at the given IP address and port. Ensure that you are using the correct host address and port.
- `ResponseError: model "XYZ" not found, try pulling it first`
  - Cause: The specified model (XYZ) may not be available on the server.
  - Solution: Check if the model name is correct. Verify that the model is loaded and available on the server.

These errors and issues can arise due to configuration problems, incorrect usage of the ollama library, network connectivity issues, or unexpected responses from the server. To avoid these issues, make sure you're using the correct configuration.

Hardware acceleration

When working with AI, performance issues are inevitable. These problems can arise from loading large models or fine-tuning models on massive data sets. Furthermore, traditional CPUs struggle to meet the computational demands of modern AI applications. Here are some reasons why:

1. **Complexity**: AI models, especially deep learning models, involve complex mathematical operations that require a large number of calculations. CPUs, even high-performance ones, struggle to keep up with these demands.

2. **Scalability**: As AI models grow in size and complexity, the computational requirements increase exponentially. CPUs become bottlenecks, limiting the model's performance and scalability.
3. **Matrix Operations**: Many AI algorithms involve matrix operations (e.g., matrix multiplications), which are not optimized for CPU execution. Specialized hardware accelerators can efficiently handle these operations, reducing computation time.
4. **Memory Bandwidth**: AI models often require large amounts of memory to store weights, biases, and intermediate results. CPUs typically have limited memory bandwidth, leading to slow data transfer rates and increased processing times.
5. **Parallelization**: Hardware accelerators like GPUs (Graphics Processing Units) or TPUs (Tensor Processing Units) can perform thousands of calculations in parallel, whereas CPUs are limited to executing a few dozen instructions at once.

> Note that, for studying and developing basic AI concepts, you don't need any special GPU or high-performance hardware (except LLM fine-tuning process). An octa-core Intel or AMD processor with 16 GB of RAM is sufficient to test and implement AI features such as embedding, image processing, and simple agent-based application. While the process may be slower without hardware acceleration, it's still tolerable for educational purposes.

To overcome these above limitations, you can use following hardware acceleration technics, which includes:

- GPUs: Designed for matrix operations and parallel processing.
- TPUs: Optimized for deep learning computations, such as convolutional neural networks (CNNs).
- ASICs (Application-Specific Integrated Circuits): Custom-designed chips for specific AI tasks, like image or speech recognition. For example, Groq inference platform.

- AVX/AVX2. Advance vector processing from Intel amd AMD.

By leveraging these specialized hardware accelerators, you can:

- Reduce computation time: Speed up training and inference times for AI models.
- Increase model size: Train larger, more complex models without sacrificing performance.
- Improve scalability: Handle large-scale AI deployments with greater efficiency.

If you urgently need to increase productivity when working with AI, there are several options you can consider. In the final section of this chapter, we will provide a roadmap to help you address low performance.

Using a Workstation with GPU

As we've mentioned earlier, GPUs are designed to excel in parallel processing tasks, making them highly effective for AI workloads such as training deep learning models. With their ability to perform thousands of calculations simultaneously, they significantly accelerate computation compared to traditional CPUs.

If budget permits, investing in a high-performance workstation equipped with a dedicated GPU (e.g., NVIDIA RTX or Tesla series) can be a worthwhile investment. However, it's essential to ensure the GPU has sufficient VRAM (Video RAM) to handle large datasets and models. While this option is cost-effective, we recommend considering other alternatives for general AI use cases.

If you're using an older workstation with an Nvidia video card that supports CUDA, such as the 2014 MacBook Pro model, you can try enabling GPU acceleration in your AI software frameworks like TensorFlow or PyTorch by installing the necessary drivers and libraries (including CUDA and cuDNN).

It's worth noting that Apple silicon-based MacBooks, including M1/M2/M3 models, leverage their onboard GPUs for performing LLM operations. In fact, here's an example output from running Ollama on a MacBook Pro with an M1 processor:

```
NAME            ID           SIZE   PROCESSOR UNTIL
llama3.1:latest 62757c860e01 6.7 GB 100% GPU  4 minutes from now
```

Enabling AVX/AVX2 for CPU acceleration

Advanced Vector Extensions (AVX) and its successor, AVX2, are CPU instruction sets that enhance performance for demanding applications, including AI and machine learning. They allow the CPU to process multiple data points with a single instruction, optimizing computation-heavy tasks.

Most modern Intel and AMD processors come equipped with AVX/AVX2 support out-of-the-box. To check whether your processor supports these advanced instruction sets, you can use the following command in Linux:

```
grep -o 'avx[^ ]*' /proc/cpuinfo
```

Enabling AVX/AVX2 on your processor unlocks significant performance gains, especially when running LLM on Ollama. By default, Ollama leverages this option to utilize CPU resources more efficiently. Below is a fragment of Ollama log that demonstrates its use of AVX/AVX2 extensions while processing LLMs:

```
time=2024-08-27T15:17:09.087+03:00 level=INFO source=payload.go:44 msg=\
"Dynamic LLM libraries [cpu cpu_avx cpu_avx2]"
```

Enabling AVX/AVX2 is the most cost-effective option to improved overall system performance without the need for additional hardware.

Using 3rd party ASIC platform or VPS with GPU support

Application-Specific Integrated Circuits (ASICs) are custom-built hardware optimized for specific tasks, such as AI and deep learning. They offer superior performance and energy efficiency for AI workloads. Groq is a pioneer in leveraging ASIC technology for their inference platform. Groq provides a free API for developers to utilize their language models, offering flexibility without upfront costs. While there

are some usage limits in place – including rates per minute, day, and token count – these restrictions are currently quite generous and should suffice for individual users' daily needs.

Alternatively, Virtual Private Servers (VPS) with GPU support provide access to powerful computational resources without needing to purchase and maintain your own hardware. Examples include Google's TPU (Tensor Processing Unit) service, AWS with NVIDIA GPUs, or other cloud providers like Azure and IBM Cloud. Configure your AI environment on these platforms, ensuring you utilize the available resources effectively.

Using Google Colab or Kaggle service

Google Colab and Kaggle Kaggle are free cloud-based platforms that provide access to powerful computational resources, including GPUs and TPUs. They are ideal for individuals and small teams who need high performance without investing in expensive hardware.

Simply sign up for Google Colab or Kaggle. You can write and execute Jupyter notebooks directly in your browser. Both platforms offer pre-installed libraries and seamless integration with popular deep learning frameworks like TensorFlow, PyTorch, and Keras. They also provide options to use GPU and TPU acceleration with a few clicks.

These third-party services have their own limitations, and using them with personal datasets can potentially lead to data leakage.

By implementing these strategies, you can significantly enhance the performance of AI tasks, making your workflows more efficient and reducing the time needed for training and deploying AI models.

Conclusion

In this chapter, we explored the foundational steps for setting up and optimizing local LLMs for AI programming. From installing essential tools and configuring environments to generating access tokens, these preparatory tasks ensure a smooth transition into AI development. The practical example of developing a Python

application using a local LLM not only reinforces the learning but also demonstrates the potential of these models in real-world scenarios.

We also covered troubleshooting common errors, emphasizing the importance of properly setting up and managing dependencies and configurations. As AI models become more complex, hardware acceleration plays a key role. Using specialized hardware like GPUs and TPUs, or optimizing CPUs with AVX/AVX2 instructions can greatly improve the performance and scalability of AI tasks.

Lastly, we discussed various solutions to boost productivity, from using dedicated workstations to cloud-based services, providing a roadmap for overcoming performance bottlenecks. With these insights, you're now equipped to embark on more complex AI projects, leveraging the power of local LLMs and hardware acceleration to achieve efficient and effective results.

Chapter 2: Deep dive into the theories of Generative AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Artificial Intelligence (AI)

Artificial Intelligence, or AI, is a branch of computer science focused on developing theories and methods to create applications or machines that can mimic human behavior. Therefore, AI is a discipline, much like physics is a discipline within science.

The key features of AI are:

1. **Reasoning**: AI systems should be able to process information and make logical deductions, similar to how humans do.
2. **Learning**: AI aims to enable machines to learn from data, identify patterns, and improve their performance over time without explicit programming for every scenario.
3. **Acting**: AI systems should be able to take actions in the real world based on the information they process and their goals.

There are several distinct types of artificial intelligence:

- **Narrow or Weak AI**: Designed to perform specific tasks, like playing chess or recommending products.
- **General or Strong AI**: Hypothetical AI with human-level intelligence and the ability to perform any intellectual task that a human can.
- **Super AI**: Hypothetical AI that surpasses human intelligence in all aspects.

AI is rapidly evolving and impacting various fields, including healthcare, finance tech, transportation, and entertainment.

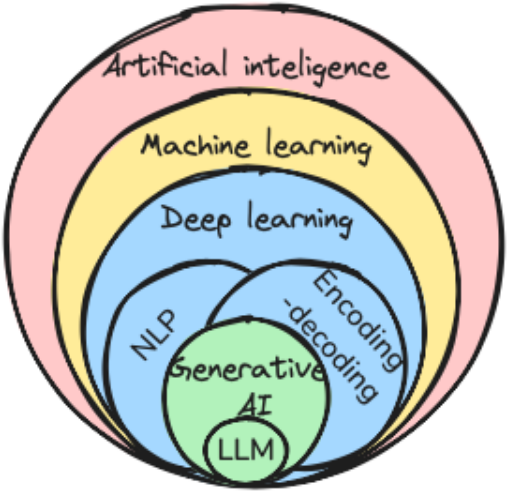


Figure 2.1

As mentioned earlier, AI is a discipline rather than a natural science and consists of several subfields, as illustrated in Figure 2.1. You have likely come across similar images on the internet, as Figure 2.1 is a common illustration used to explain AI and its subfields. The first subset of the AI discipline is machine learning, AI fundamentally begins here.

Machine Learning (ML)

Machine learning is a subfield of Artificial Intelligence (AI) that enables computers to learn from data without being explicitly programmed.

If you have ever shopped on Amazon or eBay, you have likely interacted with a machine learning model. Recommendation systems, a type of machine learning model, are commonly used on e-commerce platforms like Amazon to suggest books or products based on users past behavior or preferences. Let's explore an example to clarify the concept of machine learning.

Imagine we have the following dataset on our e-commerce site.

User ID	Book Title	Genre	Author	Rating (1-5)
1	The Apache Ignite book	Technology	Shamim Ahmed	5
1	Getting started with generative AI	Technology	Shamim Ahmed	5
2	High performance in-memory computing	Technology	Shamim Ahmed	4
3	The Martian	Sci-Fi	Andy Weir	4
3	Dune	Sci-Fi	Frank Herbert	5

We want to predict and recommend a book to a user based on factors such as **genre preferences** (e.g., Technology, Non-fiction, Sci-Fi), **previous ratings** (books the user has rated highly), **book popularity** (average rating from other users), and **reading history** (types of books the user frequently reads).

Machine learning models can be used to recommend books by identifying patterns in the behavior of multiple users. For example, if both User A and User B have highly rated certain books, the system might suggest books that User A liked but User B hasn't read yet, and vice versa.



Info

The columns **Genre** and **Rating (1-5)** from the dataset are called **features** in machine learning terms. These features are used by algorithms to make **predictions** or **classifications**.

Here's the basic idea of ML:

1. **Data**: Machine learning algorithms need lots of data to learn from. This data can be anything: text, numbers, sensor readings, etc.
2. **Algorithm**: This is the *learning recipe* that the computer uses to analyze the data and find patterns. There are many different types of algorithms, each suited for different tasks.
3. **Training**: The algorithm is fed the data and adjusts its internal parameters to improve its performance on a specific task. Think of it like the child learning from the cat pictures – they gradually get better at recognizing cats.
4. **Prediction**: Once trained, the algorithm can make predictions on new, unseen data. For example, it could recommend a book that a user hasn't read yet.

This idea can be illustrated as a machine learning process, as shown in Figure 2.2.

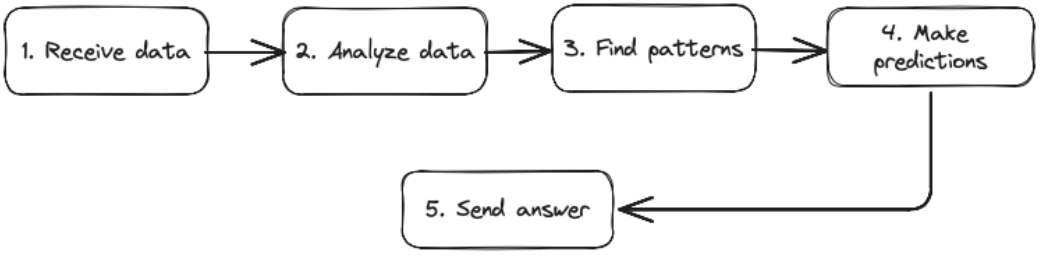


Figure 2.2

Once the ML engineer finished training and optimized the *Algorithm*, it will follow the standard process:

1. Recieve a new input as a User query.
2. Analyze the data.
3. Find a pattern.
4. Make a prediction, recomendation or classification.
5. Send a result back to the User.

There are a few types of ML:

- **Supervised Learning**: The algorithm is trained on labeled data (e.g., data from database tables). It learns to map input data to the correct output label.

- **Unsupervised Learning**: The algorithm is trained on unlabeled data (e.g., images of dog, cat etc.) and must discover patterns and relationships on its own.
- **Reinforcement Learning**: The algorithm learns through trial and error, receiving rewards for correct actions and penalties for incorrect ones.

Examples of Machine Learning in action is as follows:

- **Recommendation systems**: Netflix suggesting movies you might like or Amazon product recommendations.
- **Spam filters**: Identifying and blocking unwanted emails.
- **Fraud detection**: Flagging suspicious transactions on banking system.

Machine learning is revolutionizing many industries and aspects of our lives, and its potential continues to grow. If you're interested in machine learning and want to learn more, we highly recommend picking up a book on the topic and experimenting with some frameworks to get hands-on experience.

Deep Learning (DL)

Deep learning is a subfield of Machine Learning or can be said as a special kind of ML. It works technically in the same way as machine learning does, but with different capabilities and approaches. DL uses Artificial Neural Networks (ANNs) to learn patterns and relationships from data.

The key difference is, machine learning algorithms learn from data by identifying *simple patterns* and *relationships* when Deep learning uses artificial neural networks with multiple layers (hence **deep**) of interconnected nodes to learn *complex patterns* and *representations*. These layers allow the algorithm to gradually build a deeper understanding of the data, similar to how our brain processes information.



Info

Artificial Neural Network (ANN) in plain English: Imagine your brain has billions of tiny, specialized computing units called *neurons* (or *nodes*) that communicate with each other through connections (like tiny synapses). An ANN is a simplified model of this process, using artificial nodes and layers to make predictions or classify data. Each layer analyzes and transforms the input data in some way, allowing the network to *learn* more complex patterns and relationships.

The diagram shown as Figure 2.3 displays a basic structure of an Artificial Neural Network, where it consists of multiple layers called the **input layer**, **output layer**, and **hidden layers**. In each layer every **node (neuron)** is connected to all nodes (neurons) in the next layer with parameters called **weights**.

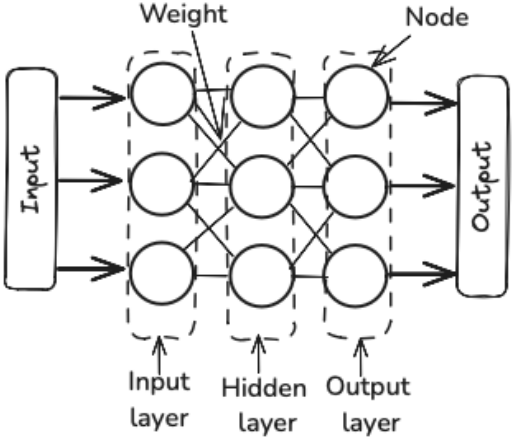


Figure 2.3

Let's consider the example, you're trying to recognize a **Bicycle** picture. A simple computer program might look for basic features like **Handlebar** or **Wheels**. But with Deep Learning, we create an ANN that looks at multiple layers of detail:

- Layer 1: Basic shapes (e.g., circles, lines)
- Layer 2: Object parts (e.g., Handlebar, Wheels)
- Layer 3: Overall object recognition (e.g., Bicycle)

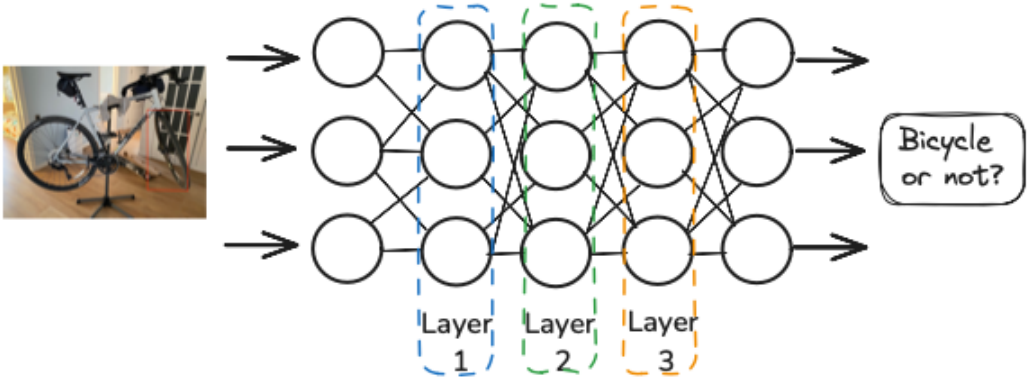


Figure 2.4

Each layer builds upon the previous one, allowing the network to learn more complex and nuanced features of the image. This process is called training, where we feed the network with many examples of **Bicycle** pictures and adjust the connections between nodes based on the errors or **misclassifications**.



Info

Please note that in the context of Deep Learning, the following terms will be used extensively throughout this book: **Nodes**, **Layers**, **Weights**. These terms are fundamental to understanding various concepts, including fine-tuning large language models and image processing.

Deep learning uses the following features:

- **Feature extraction**: Deep learning algorithms can automatically learn relevant features from raw data, eliminating the need for manual feature engineering.

- **Handling complexity**: Deep networks can handle vast amounts of complex data, making them suitable for tasks like image recognition, natural language processing, and speech recognition.
- **Generalization**: Deep learning models tend to generalize well to new data, meaning they can perform well on unseen examples.

Examples of Deep Learning in Real-World Scenarios:

- **Image recognition**: Identifying objects, faces, and scenes in images.
- **Speech recognition**: Converting spoken words into text.
- **Self-driving cars**: Perceiving the environment and making driving decisions.

Deep Learning models can be initially trained on vast amounts of data, and then tailored to perform specific tasks or operate within particular domains. This capability to process massive datasets and learn from raw text has made Deep Learning an essential component of cutting-edge applications such as Natural Language Processing (NLP).

Natural Language Processing (NLP)

NLP is a subfield of AI (this is where the confusion begins!!) that focuses on enabling computers to understand, interpret, and generate human language in a way that is both meaningful and useful. It combines computational linguistics, computer science, and data science to bridge the gap between human communication and machine understanding.

NLP and deep learning are closely related, with deep learning providing powerful tools and techniques that have significantly advanced the capabilities of NLP. Here's how they are related:

- **Neural Networks**: Deep learning employs neural networks with multiple layers (deep neural networks) to model complex patterns in data. In NLP, these models are used to process and understand text by learning from large corpora of language data.

- **Recurrent Neural Networks (RNNs)**: Early deep learning approaches in NLP used RNNs to handle sequential data and capture dependencies in text. Variants like Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRUs) were introduced to address limitations in handling long-range dependencies.
- **Convolutional Neural Networks (CNNs)**: Though traditionally used for image processing, CNNs are also applied to NLP for tasks like sentence classification and named entity recognition by capturing local patterns in text.



Info

NLP is not a subset of deep learning. Instead, deep neural networks, particularly deep learning models, are used to enhance NLP capabilities by learning complex patterns and representations from large datasets.

NLP can be broadly divided into two areas: **Natural Language Understanding (NLU)** and **Natural Language Generation (NLG)**. NLU focuses on helping machines understand and analyze human language by identifying concepts, entities, emotions, and keywords. NLG, on the other hand, involves creating coherent and meaningful text, such as phrases, sentences, and paragraphs, from internal data representations.

Models designed to understand and generate human language are commonly referred to as **Language Models (LMs)**. Typically, these models are based on recurrent neural networks (RNNs), which have proven effective at capturing long-range dependencies within sequential data, such as text. However, RNN-LMs can be computationally intensive to train and susceptible to overfitting, especially when dealing with longer sequences of words

The rise of deep learning models in 2016, especially **Transformers**, has revolutionized both NLU and NLG, leading to significant advancements in natural language understanding and generation, which have greatly improved the accuracy and effectiveness of NLP applications.

We will now smoothly transition to explore the Transformer model in detail, examining its structure and how it works. For further reading on NLP concepts, you can find more comprehensive information here.

Transformer

The Transformer model was first introduced in the ground-breaking paper Attention Is All You Need, which presented a revolutionary new approach to deep learning neural networks that has had a profound impact on Natural Language Processing (NLP). Unlike traditional models such as Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs), Transformer models utilize **self-attention** mechanisms to more effectively capture the complex relationships between words within a sentence. This ability to handle long-range dependencies, regardless of their distance in the text, has made Transformers highly effective for a wide range of NLP tasks.

Key components (there are much more in the original paper) of Transformer models include:

- Self-Attention Mechanism: This enables the model to assess the importance of different words in a sentence relative to each other, capturing complex relationships.
- Encoder-Decoder Architecture: Essential for tasks like machine translation, where the encoder processes the input text, and the decoder generates the output text.

In the following sections, we will explore each of these components in detail.

Self-Attention mechanism

The self-attention mechanism is a key part of how Transformer models understand language. It helps the model figure out which **words** in a sentence are **most important** and how they **relate** to each other.

Imagine you are reading a sentence like `The dog barked loudly because it saw a stranger.` When you read the word `it`, you automatically know that `it` refers to `the dog` because of the context. The self-attention mechanism does something similar—it looks at all the words in the sentence and focuses on the ones that are important for understanding each other.

Let's try to understand how it works with this above example sentence: `The dog barked loudly because it saw a stranger.`

- **Looking at Each Word**: The self-attention mechanism looks at every word in the sentence. When it gets to the word `it`, it has to figure out what `it` refers to. Is `it` the dog? The barking? The stranger?
- **Focusing on the Important Words**: The mechanism looks at the surrounding words like `dog` and `saw` to figure out that `it` refers to the dog. It also focuses less on less important words like `loudly` or `because` that don't help in understanding the main point.
- **Assigning Importance**: It assigns a weight or score to each word, giving more attention to the words that matter. In this case, `dog` would get a higher score because it's more relevant to the meaning of `it`.
- **Using the Connections**: The model uses these connections to understand the sentence as a whole. Now it knows that the dog is the one barking because it saw a stranger, making the sentence clear.

Before the self-attention mechanism, older models would look at words one by one, often losing track of important relationships between them. Self-attention allows the model to understand the meaning of a word based on all the other words in the sentence, making it much smarter at tasks like translation, summarizing, or even completing your sentences.

The self-attention mechanism is widely used in everyday applications such as Google's Gmail or search functions. For instance, consider this scenario: when you type into Google Search `I need a hotel that is close to the airport`, the self-attention mechanism recognizes that `close` refers to the location of the hotel and `airport` indicates its proximity to the desired destination. By understanding the entire context, rather than just individual words, it provides more accurate search results.

Encoder-Decoder architecture

The encoder-decoder architecture is a mechanism used by many language models, especially in tasks like translation. It works like a two-step process where one part

the encoder takes the input information, and the other part **the decoder** uses that information to produce an output.

Imagine it like a translator who first reads and understands a sentence in one language (the encoding part) and then translates it into another language (the decoding part).

The encoder-decoder mechanism is crucial part of any models because it allows models to handle complex tasks like language translation, summarization, and even chatbots. The encoder helps the model understand the input (whether it's a sentence, paragraph, or image), and the decoder generates the corresponding output, ensuring that the meaning stays accurate.

To fully understand how Transformer models process information, let's break down their architecture into its two primary components: the encoder and the decoder. We'll explore each of these crucial elements in greater detail to gain insight into their roles and functionality.

- **Encoder**: Understanding the Input.
  - The encoder's job is to take the input, like a sentence, and understand its meaning. For example, if the input sentence is `The cat is sleeping on the sofa`, the encoder processes each word, understands the relationships between the words, and converts the whole sentence into a kind of code or summary (called a *hidden representation*).
  - This summary captures the essence of the sentence.
- **Decoder**: Producing the Output.
  - The decoder's job is to take the summary from the encoder and use it to generate the desired output. For instance, if we want to translate `The cat is sleeping on the sofa` into French, the decoder will use the encoded summary and generate the French sentence: `Le chat dort sur le canapé`.
  - The decoder starts with the encoded summary and predicts one word at a time, checking back with the encoder to make sure it's on track.

We can illustrate the process as shown in figure 2.6.

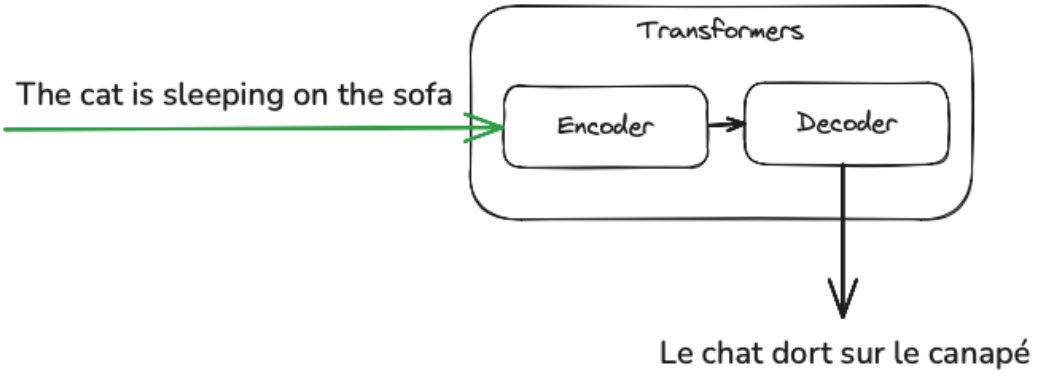


Figure 2.6

Let's try another example, imagine the encoder-decoder mechanism as a travel guide who helps you translate signs at an airport.

- Encoder: The travel guide reads a sign in French that says, "Sortie de secours" (which means "emergency exit"). They understand what the sign means in French.
- Decoder: After understanding it, the guide then tells you, "This sign means 'emergency exit' in English."

The encoder understood the French sentence, and the decoder turned that understanding into an English sentence.

This brief overview of the encoder-decoder architecture lays a solid foundation for understanding its core principles. In future sections of this chapter, we will revisit the transformer architecture in more detail once we have a clearer understanding of tokens and vectors.

The Transformer architecture serves as the foundation for the development of a new generation of language models (LMs), which have been trained on a certain corpus of data. Notably, models like BERT and CamemBERT are exemplary of this new breed of LMs. These LMs are often referred to as Gen1 (Generation 1) LMs or Language Models after Transformers. A key characteristic of these models is their relatively small parameter count, making them well-suited for basic language tasks. For example, BERT, developed by Google in 2018, is an encoder-only LLM that boasts

a base variant with 12 layers and 12 attention heads, providing approximately 110 million parameters.

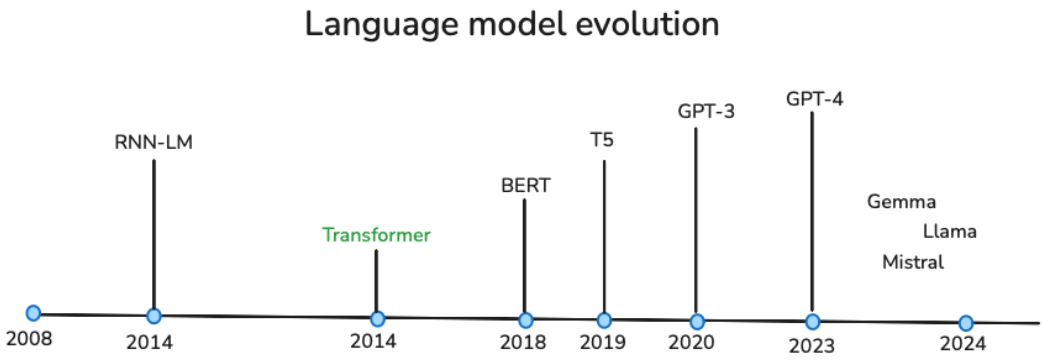


Figure 2.7

However, the capabilities of these transformer-based models have led to the creation of more sophisticated pre-trained models like GPT (Generative Pre-trained Transformer) or T5. These cutting-edge models are trained on enormous amounts of text data and can be tailored for specific NLP tasks, resulting in state-of-the-art performance. The emergence of these advanced models has given rise to the exciting field of generative AI.

Generative AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

What is Generative AI and what is not?

Everything in this field is evolving rapidly, and it's easy for people to become confused by the terms, jargon, and distinctions—especially when it comes to understanding what generative AI is and what it is not. In this section, we aim to clarify the concept: What is Generative AI, and what is not?

The figure 9 shows a simple way to distinguish between what is Gen AI and what's not.

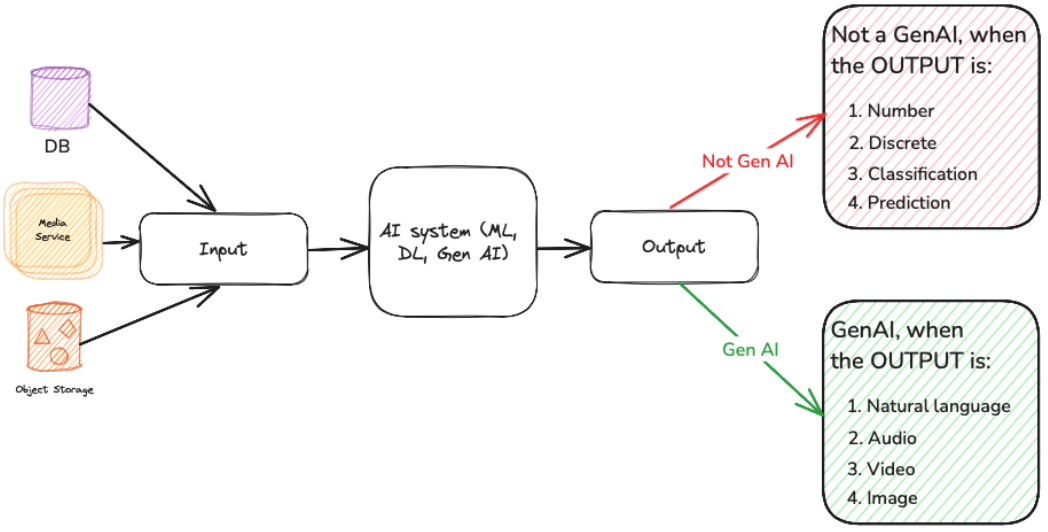


Figure 2.9

If a system—whether it's machine learning, deep learning, or any other AI—produces outputs such as numbers, classifications (e.g., "spam" or "not spam"), or probabilities (e.g., "90% chance of buying a product"), it is not considered generative AI.

However, if the output is something like natural language (e.g., an essay or poem) or images (e.g., pictures of dogs or cats), then it falls under the category of generative AI.

All of the explanations above can also be expressed in mathematical terms as follows:

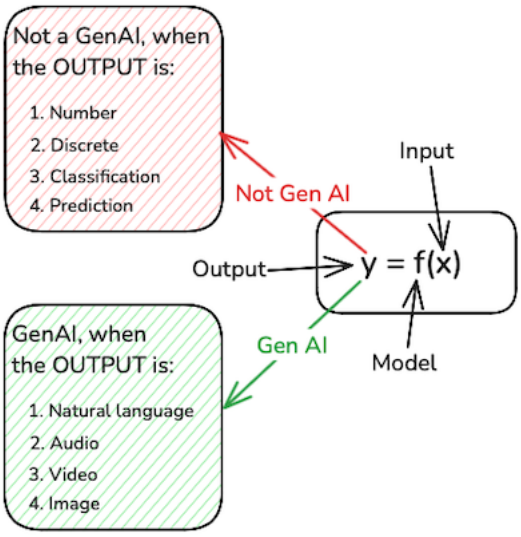


Figure 2.10

The equation `y = f(x)` represents how the output of a process (`Y`) is calculated based on different inputs (`X`). In this case, `Y` is the model's output, `F` represents the function or model used for the calculation, and X stands for the input data, which could be anything from raw text (e.g., Wikipedia articles), CSV files (e.g., from annual reports), or even images. The model's output is determined by all the inputs. If `Y` is a number, such as a predicted stock price, it is not considered generative AI. However, if `Y` is a **sentence**, such as `the share price`, the model is performing generative AI by generating a response.

Categories of Generative AI

In the last few years, the landscape of generative AI (Gen AI) has evolved significantly. In 2024, we see a wide range of generative AI (Gen AI) models emerging, including:

1. Text-to-text models: These models take text as input and generate text as output. The text can range from natural language and programming code to poems and even HTML. Examples include Google's Gemini, GPT-4, Claude Opus, and LLaMA 3.1.

2. Text-to-image models: These models generate images based on a text description. For example, you could ask for an image of a dog or cat with green ears, and the model would generate that image for you. Midjourney is an example of such a model.
3. Image-to-image models: These models transform or combine images that you provide as input.
4. Image-to-text models: These models extract text from images, which is particularly useful for tasks like pulling text from presentations. LLaVA is an example of this type of model.
5. Speech-to-text models: These models transcribe speech into text, which is useful for things like meeting notes, such as those from a Zoom call. Examples include Whisper and DeepSpeech.
6. Text-to-audio models: These models generate music or sound from a text prompt. They are especially useful for creating audio from text, such as audio transcriptions. Deepgram's Aura is an example of this type of model.
7. Text-to-video models: These models generate video from a text prompt. In the near future, we might see entire films being created from scripts. Examples include Sora, Dream Machine, and Kling.

Generative AI applications or models can be broadly classified into two primary categories, based on their implementation approaches:

1. **Foundation models** are large, pre-trained *machine learning* models that serve as the base for various AI applications. These models are trained on vast, diverse datasets (such as text, images, or audio) and can be fine-tuned for specific tasks with minimal additional training. Due to their scale and versatility, foundation models can be adapted to solve a wide range of tasks, from language translation to image recognition.

As an example, `DALL-E`, `DeepMind's Gato` is a foundation model trained on images and text, and it generates original images based on textual descriptions.

Key characteristics:

- foundation models are trained on extensive datasets from diverse sources, allowing them to learn a broad understanding of language, images, or other types of data.

- can be adapted or fine-tuned for many downstream tasks such as sentiment analysis, text generation or object recognaization.
- their knowledge can be transferred to new tasks with minimal extra training.

2. **Large Language Models (LLMs)** are a type of foundation model that leverages the power of *deep learning* to analyze and learn from vast amounts of data. By consuming and training on these massive datasets, LLMs become incredibly proficient in processing and generating human-like language. In fact, they can create entirely new text combinations that mirror the nuances and complexities of natural language, all based on the patterns and structures they've learned from their training data.

GPT, LLaMA is the best examples of LLMs.

Key characteristics:

- designed specifically to handle natural language tasks.
- generate text, process language, and understand human language patterns.
- also trained on massive datasets from various sources.

To put it simply, LLMs are a specific type of foundation model. Both foundation models and LLMs drive Generative AI, allowing machines κ systems to create diverse and innovative content.

If you're feeling confused, don't worry - it's not your fault! The term *foundation model* was introduced by researchers at Stanford's university and the Center for Research on Foundation Models (CRFM), but the original paper isn't entirely clear. What we will try to do here is break down the concept in simpler terms:

- A foundation model refers to a type of AI system that has broad capabilities, making it adaptable to a range of different tasks and applications. Think of it as a base layer - the original model provides a solid foundation on which other models can be built upon. This stands in contrast to many other AI systems, which are specifically trained for a single purpose and aren't easily transferable to other areas.

- To illustrate this point, imagine a building. A foundation model would be like the concrete slab that provides a stable base for the entire structure. You can then build different floors, walls, and features on top of it, each serving a specific purpose. In contrast, many other AI systems are like individual rooms within the building - they're designed to serve a single function and aren't easily modified or repurposed

Examples of foundation models often include many of the systems mentioned earlier as LLMs. To understand how something more specific can be built on a broader base, consider `ChatGPT`. Initially, ChatGPT was based on the `LLM GPT-3.5`, which served as the foundation model. To tailor it for chat, OpenAI used additional data specific to conversational settings to create a modified version of GPT-3.5. This adjusted model was then used to develop `ChatGPT`.



Info

One notable example of a foundation model is DeepMind's Gato model, which has been designed to tackle a diverse range of tasks that extend beyond language processing. In addition to generating text, Gato can also control robot arms and play complex video games. This versatility makes Gato a prime candidate for the label `foundation model`, but it would not be classified as a Large Language Model (LLM).

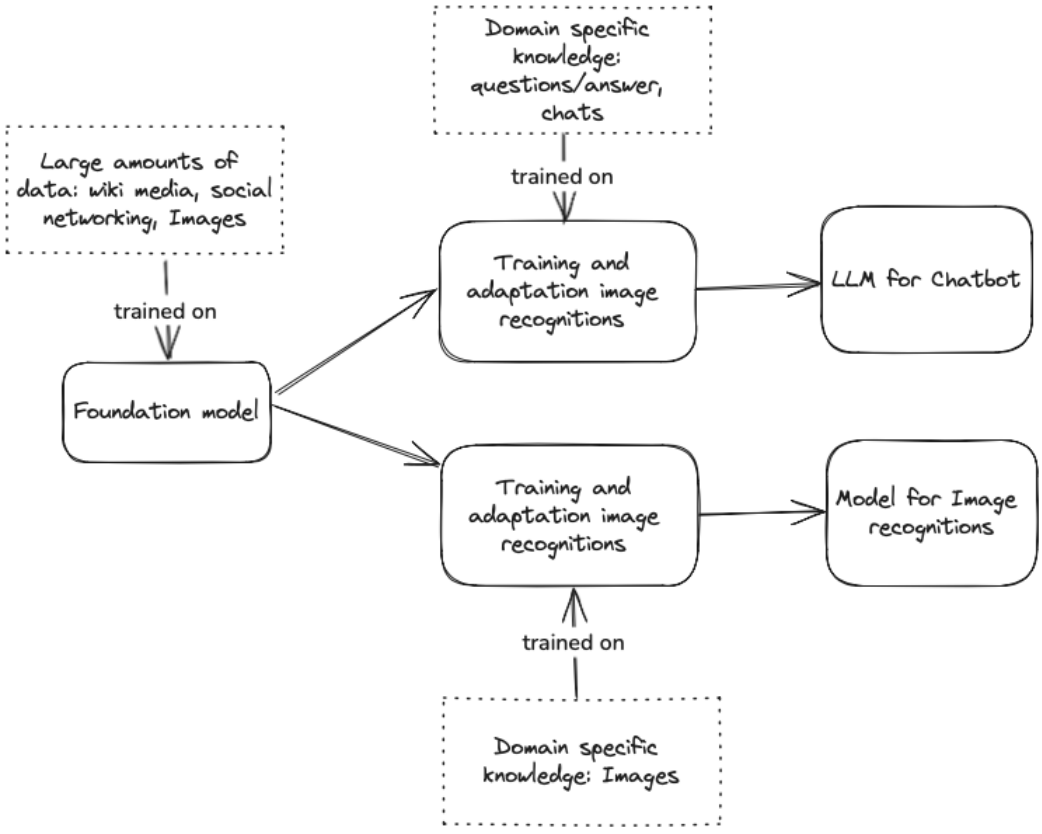


Figure 2.11

Currently, the term *foundation model* is often used interchangeably with *large language model* because language models are the most prominent example of systems with broad capabilities that can be adapted for various purposes. The key difference between the two terms is that *large language models* specifically refers to systems focused on language tasks, while *foundation model* represents a broader concept that could include new types of systems in the future.

In the following sections, we will explore the different concepts behind LLMs and explain how they operate.

Large Language Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

How LLM works internally?

We've often discussed that LLMs are designed specifically for natural language processing (NLP) tasks. But how do they understand the grammatical structure (syntax) and meaning (semantics) of the input text, and generate an output that has the correct syntax and relevant meaning?

In this section, we'll explore the various techniques and mechanisms involved in NLP processing, so get ready!

Large Language Models (LLMs) work by processing vast amounts of text data to learn patterns in language. Internally, they rely on neural networks, which are designed to handle sequential data efficiently.

Here's a simplified workflow:

1. A neural network is made up of numerous numbers, or parameters, that are interconnected. Each of these connections is called a *weight*.
2. The neural network only processes numbers, so when you send an input (a prompt or question), it converts it into numbers. Based on how the parameters are configured, the network outputs a result, also in numerical form.
3. Any type of content—text, images, or otherwise—can be represented as numbers and fed into the model.

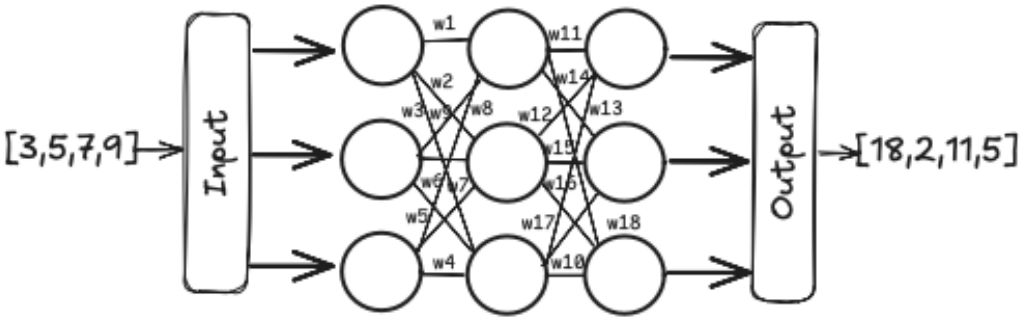


Figure 2.10

For example, if we input the sentence *The sky is* into an LLM, it converts the words into numbers, processes them through the neural network, and then outputs a result like *The sky is blue.* This is a basic demonstration of how an LLM predicts the next word, a common task in machine learning.



Tips

When someone says that an LLM is a collection of pretrained and fine-tuned text models with sizes ranging from 8 billion to 405 billion parameters, they're referring to the number of weights (connections between nodes) in the model's neural network. In Figure 2.10, our network has 18 parameters.

Now, here's where things get really interesting: when you combine the input and output, and feed it back to the LLM, it can continue generating new content by predicting the next word. This process can be repeated indefinitely, allowing us to create a continuous stream of generated text - much like what happens when you engage with ChatGPT or LLaMA.

```
>>> finish the sentence "The sky is"
...painted with colors of sapphire, amethyst, and gold!

Or was that what you were thinking?

Here are a few other options:

* ...always changing, yet always the same.
* ...the perfect backdrop for a beautiful sunset.
* ...a reminder of how small we truly are.

But I have to say, my favorite is:
* ...no limit to its beauty and wonder!
```

At first glance, it seems simple and straightforward, doesn't it? But beneath the surface, there's a lot of fascinating activity going on. Let's break it down step by step.

1. **Tokenization**: The model breaks down text into smaller units, called tokens (words or subwords), which serve as input.
2. **Embedding**: Each token is converted into a numerical representation (vector) that captures its meaning in the context of the text.
3. **Transformer Architecture**: The transformer consists of layers of *attention mechanisms* that help the model understand the relationships between tokens, regardless of their position in a sentence. It helps the model focus on relevant words when generating or analyzing text.

Let's start by the tokenization.

Tokenization

This process involves converting natural language (such as sentences) into bits of data that a program can work with. In simpler terms, it breaks sentences down into

individual words or parts, known as tokens, which serve as the building blocks for semantic analysis.

Thus, A token is a unit of text segmented for efficient processing by a large language model. LLM tries to understand the statistical relationships between these tokens for producing the next token in a sequence of tokens.

Tokens can be entire words, letters, combinations of words, or even punctuation marks. A tokenizer is an algorithm or function that performs this segmentation.

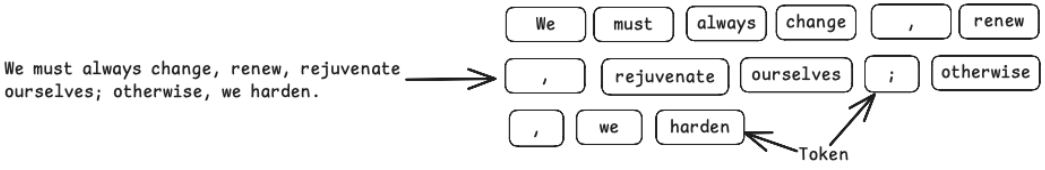


Figure 2.11

There are various tokenizers, each with its own trade-offs. Some well-known ones include NLTK (Natural Language Toolkit), SpaCy, the BERT tokenizer, and Keras.

A simple word-based tokenizer would treat each word as a token, meaning the number of tokens would match the number of words in the sentence. For example, the above sentence would have 15 tokens. Below is an example of how the SpaCy tokenizer splits a sentence into tokens.

```
import spacy
nlp = spacy.blank("en")
tokens = nlp ("We must always change, renew, rejuvenate ourselves; othe\
rwise, we harden.")
len (tokens)

for token in tokens:
  print (token)
```

If you execute the above code, the output should same as shown below:

```
We
must
always
change
,
renew
,
rejuvenate
ourselves
;
otherwise
,
we
harden
.
```

In tokenization, characters are often counted individually, including spaces between words. However, not all tokenizers operate this way, some consider the context in which the text is being used, resulting in varying outcomes.

Google's LLM, BERT, employs a context-dependent tokenizer to achieve its language understanding capabilities. For comparison, using the same tokenizer as GPT-4 (and available online through OpenAI's free online token counter), we can see how the previous sentence is broken down into distinct 17 tokens.

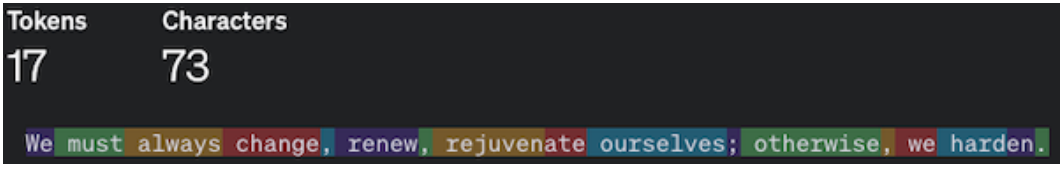


Figure 2.15

Tokens are also a useful unit of measurement. The amount of text an LLM can process or generate is measured in tokens. Additionally, the cost of running LLMs is directly tied to the number of tokens processed, the fewer the tokens, the lower the cost, and vice versa.

The size of an LLM is defined by the number of tokens it can accept as input, and each model has different limitations. Since tokens are stored and processed in memory, these limits help keep the model efficient and optimize resource use. Below are some of the token limitations for different LLMs:

Model	Tokens Limit
Llama 2	4096
GPT 4	8192

After the input text is split into tokens, the tokenizer encodes the input using a specific scheme and generates specialized vectors that the LLM can understand. In the next section, we'll explore vectors and how tokens are represented in vector format.

Vector

As we have split the input, LLMs still struggle to understand the meaning of the text. Therefore, our first step should be to develop a strategy for converting strings into numerical representations (or *vectorizing* the text) before feeding it to the model.

Vectors play a critical role in the functionality of LLMs and overall Generative AI. This is because LLMs can only process numbers in a specific format. To fully appreciate the role and value of vectors in LLMs, it's essential to understand what vectors are, how they're generated, and how they're utilized in these models.

In mathematics, a vector is an object that represents both the value and direction of a quantity in any dimension. Consider the following example, where a vector represents five elements stored in an array.

```
# Creating a vector as an array
!pip install numpy

import numpy as np

vector = np.array([1, 2, 3, 4, 5])
print("Vector of 5 elements:", vector)
```

Output:

```
Vector of 5 elements: [1 2 3 4 5]
```

In the context of LLMs, vectors are used to represent text or data in numerical format, allowing the model to understand and process it. This is necessary because machines only understand numbers, so *text* and *images* must be converted into vectors for the LLM to comprehend.

Since neural networks and transformer architectures can only understand vectors, this representation is essential for successful processing. Moreover, operations on vectors, enable us to determine whether two vectors are identical or distinct. At its core, this capability forms the foundation for similarity search in vector databases or memory storage.



Tips

This is where the new trend of Vector Databases comes in. Vector databases store data as high-dimensional vectors, called embeddings, which capture semantic meaning and relationships. They use specialized indexing techniques, such as hashing, quantization, and graph-based methods, to enable fast querying and similarity or semantic searches. LLMs do not directly use vector databases; however, they are essential when you plan to enhance an LLM with Retrieval-Augmented Generation (RAG), as discussed in Chapter 3.

When text passes through a **tokenizer**, it split and *encodes* the input according to a specific scheme and generates specialized vectors that can be understood by the

LLM. Notably, the encoding scheme is highly dependent on the specific LLM being used. The tokenizer may choose to convert each word and part of a word into a vector based on this encoding scheme. Conversely, when a token passes through a *decoder*, it can be easily translated back into text.

The following is pseudocode that converts our motivational text into tokens using the **Phi-2** model. We use the `AutoTokenizer` class from Hugging Face to encode the text into vectors and decode it back into text.

```
!pip install transformers

from transformers import AutoTokenizer
from huggingface_hub import interpreter_login

# Use your API KEY of Hugging Face here and click enter ;-)
interpreter_login()

model_name='microsoft/phi-2'

tokenizer = AutoTokenizer.from_pretrained(model_name,token="HF_TOKEN")

txt = "We must always change, renew, rejuvenate ourselves; otherwise, w\
e harden."

token = tokenizer.encode(txt)
print(token)

decoded_text = tokenizer.decode(token)

print(decoded_text)
```

Output:

```
[1135, 1276, 1464, 1487, 11, 6931, 11, 46834, 378, 6731, 26, 4306, 11, \
356, 1327, 268, 13]
We must always change, renew, rejuvenate ourselves; otherwise, we harde\
n.
```

Tokenization involves various techniques in practice, such as padding, end-of-line markers, and more. The details of tokenization are covered in Chapter 5 (Fine-tuning LLMs).

Vectors alone are not sufficient for LLMs because they only capture basic numerical features of a token, without encoding its rich semantic meaning. Vectors are simply a mathematical representation that can be fed into the model. To capture the semantic relationships between tokens, we need something more—embeddings.

Embedding

An *embedding* is a more sophisticated version of a *vector*, usually generated through training on large datasets. Unlike raw vectors, embeddings capture semantic relationships between tokens. This means that tokens with similar meanings will have similar embeddings, even if they appear in different contexts.

Embeddings are what enable Large LLMs to grasp the subtleties of language, including context, nuance, and the meanings of words and phrases. They arise from the model's learning process, as it absorbs vast amounts of text data and encodes not just the identity of individual tokens but also their relationships with other tokens.

Through embeddings, LLMs gain a profound understanding of language, empowering them to perform tasks such as sentiment analysis, text summarization, and question answering with precision and nuance. Embeddings serve as the entry point for the model, allowing it to access and process vast amounts of text data. Moreover, they are also used independently to transform text into vectors while preserving its semantic context.

When text is fed through an embedding model, a vector is generated that encapsulates the embeddings. This process retains the essential meaning and relationships between words and phrases, making it possible to analyze, summarize, or answer questions based on the original text.

Typically, embeddings are generated through techniques such as *Word2Vec*, *GloVe*, or using modern neural networks like *transformers*. Here's an example of how OpenAI Embeddings can be used to generate embeddings from input texts: `Lion`, `Tiger` and `IPhone`.

```
from langchain_openai import OpenAIEmbeddings
embeddings_model = OpenAIEmbeddings(api_key="YOUR OPEN API KEY")

txt ="Lion"
embedded_query = embeddings_model.embed_query(txt)

print (len(embedded_query))

embedded_query[:5]
```

Output:

Embedding: Lion	Embedding: Tiger	Embedding: Iphone
[-0.0015009930357336998, -0.010024921968579292, -0.015631644055247307, -0.023150335997343063, -0.01021870318800211]	[-0.013500549830496311, -0.009651594795286655, -0.008970077149569988, -0.0019277227111160755, -0.018589219078421593]	[-0.0076760281808674335, -0.021282033994793892, 0.0060024564154446125, -0.02365402691066265, -0.016182253137230873]

Notice how the embeddings for `Lion` and `Tiger` are more similar to each other compared to `IPhone`. This is because embeddings capture relationships like gender, royalty, or object categories, and represent these concepts in a continuous vector space.

Before we dive into the more technical details of LLMs, such as the *Transformer architecture*, let's quickly review what we've covered so far.

1. **Tokenization**: The process of converting raw text into tokens (words or subwords). Example: `"Machine learning"` → `["Machine", "learning"]`.
2. **Vector & Vectorization**: Each token is mapped to a numerical vector. Example: `"Machine"` → `[0.5, 0.1, 0.7,...]`.
3. **Embedding**: A learned numerical representation that captures semantic meaning. Example: `Machine` and `Learning` would have similar embeddings if the model understands that they are related terms.

Transformers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Training LLM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Pre-training

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Fine-tuning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

RAG

RAG (Retrieval-Augmented Generation) and LLM Fine-Tuning are both techniques used to enhance the capabilities of language models, but they serve different purposes and operate in fundamentally different ways.

RAG is a hybrid approach that combines the strengths of retrieval-based methods with generative language models. It involves two main components:

- **Retrieval Module**: This component searches a large external knowledge base (such as a private Confluence, private cloud document storage, or any custom database) to find the most relevant pieces of information based on the input query.
- **Generative Module**: This is typically a pre-trained language model, such as LLaMA or GPT, which takes the retrieved information along with the original query to generate a coherent and informative response.

Instead of fine-tuning the model, you'll often use RAG to enhance the LLM with your private dataset. The RAG workflow is quite straightforward:

1. When you provide an input to an RAG model, it first retrieves relevant information from a large database or knowledge graph using the retrieval model. The retrieved information is then used to augment the input provided to the model.
2. The generator model then uses this augmented input to generate text based on your specific requirements. This process allows RAG models to leverage existing knowledge and provide more accurate and informative responses.

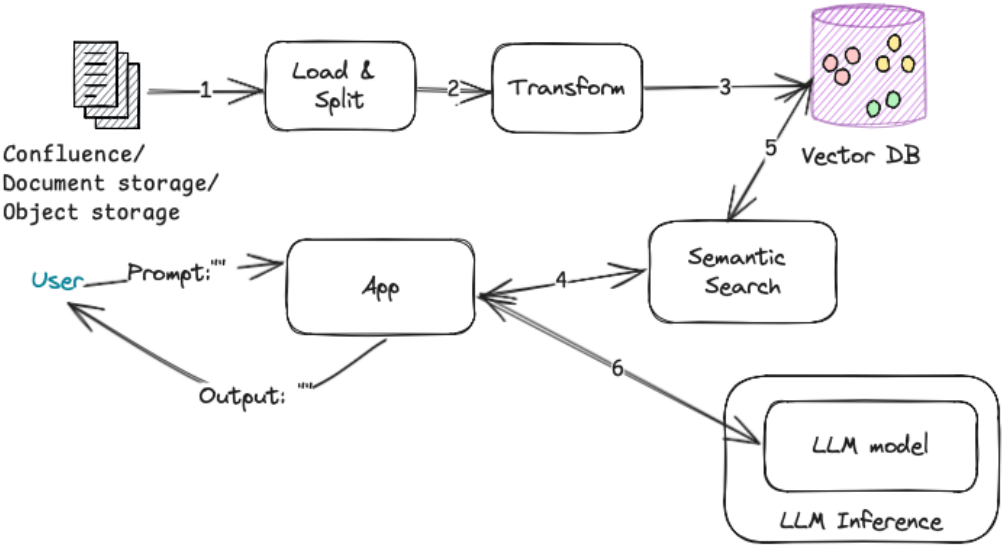


Figure 2.18

Key aspects of RAG:

- **Domain-Specific Knowledge**: RAG models are designed to cover a wide range of topics and deliver more accurate, up-to-date responses.
- **Knowledge Graph Integration**: They often use extensive knowledge graphs from various data sources, providing a more structured and comprehensive understanding of information.

- **Cost-Effectiveness**: RAG is a highly efficient way to enhance an LLM in terms of both development and hardware costs.

Understanding the differences between RAG and LLM fine-tuning is crucial for selecting the best approach based on your application's needs. Let's go over the key distinctions between the two methods:

Feature	RAG	LLM Fine-Tuning
Mechanism	Combines retrieval from an external source with generation.	Modifies model weights (parameters) using task-specific data.
Data Source	Uses an external knowledge base during inference.	Relies on the training data used for fine-tuning.
Adaptability	Dynamically integrates new information without retraining.	Requires retraining to adapt to new information.
Use Cases	Fact-based QA, dynamic knowledge integration.	Task-specific applications like sentiment analysis, translation.
Cost	Low cost of developments	High cost of training LLM

In summary, RAG is ideal for applications where the model needs to access and integrate external, potentially dynamic information in real time. We assign an entire chapter to discussing RAG and provide step-by-step instructions for developing a RAG system tailored for a private company.

AI Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Prompt engineering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Resources

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Chapter 3: RAG, enrich LLM models with private datasets

RAG (Retrieval-Augmented Generation) is an approach in AI within natural language processing that enhances the capabilities of LLM models by integrating them with a retrieval system. This technique enables an LLM model to dynamically access and utilize external information from a knowledge base, such as Wikipedia or a corporate Confluence, during the generation process.

In other words, RAG models allow pre-trained LLMs to retrieve information from large warehouse, custom datasets, or databases, and incorporate it into the generation process.

In this chapter, we'll explore how to use Retrieval-Augmented Generation (RAG) for building more effective and engaging conversational AI applications. We'll cover the basics of RAG, its benefits, and provide step-by-step instructions on how to develop your own RAG mechanism for local use.

RAG vs fine-tuning LLMs

Fine-tuning an LLM (Large Language Model) involves adjusting its pre-training for a specific task. Initially, an LLM is trained on a massive dataset to learn general language patterns. This process is followed by further training on a narrower dataset tailored to a particular application, such as customer service or code generation.

In contrast, RAG is useful when you need your LLM to generate responses based on large amounts of updated and context-specific data. For instance, enriching the response of an LLM model with datasets from your data-lake or archive documents.

RAG offers a cost-efficient alternative to fine-tuning due to its minimal resource and technical knowledge requirements. With just 10 lines of Python code, you can ensure that your model has access to the most relevant and up-to-date data.

Most of the LLM runner client like *Open WebUI* or *AnythingLLM* already incorporate RAG features.

Let's try a quick example. To get started, open the *Open WebUI* and give a prompt: `Who is the author of the book "The apache Ignite book"?`

The response should be very similar to this one.

```
A very specific question!

After conducting a quick search, I found that the book "Apache Ignite C\
ookbook" (not "The Apache Ignite Book") has several authors. However, I\
 couldn't find any information on a single author for a book with the t\
itle "The Apache Ignite Book".

That being said, it's possible that the book you're thinking of is actu\
ally the online documentation or tutorial provided by the Apache Ignite\
 project itself, rather than a traditional book.

If you could provide more context or details about the book you're look\
ing for, I'd be happy to try and help you further.
```

I'm not happy with the answer, so let's provide more context to our LLM model by uploading a relevant file from the book's web page. To do this, click on the + icon in the prompt message box, select `More`, and then choose the file you'd like to upload. Prompt will be the same as before `Who is the author of the book "The apache Ignite book"?`

After processing the file for a few seconds, you can expect to see a specific result, which is outlined below.

```
The authors of the book "The Apache Ignite book" are Shamim Bhuiyan and\
 Michael Zheludkov.
```

For personal use, yes, that seems sufficient. However, for medium-sized private companies, having a customized Retrieval, Analysis, and Generation (RAG) system might be necessary to efficiently upload, retrieve, and process information from documents using LLMs.

The entire RAG mechanism can be summarised as follows:

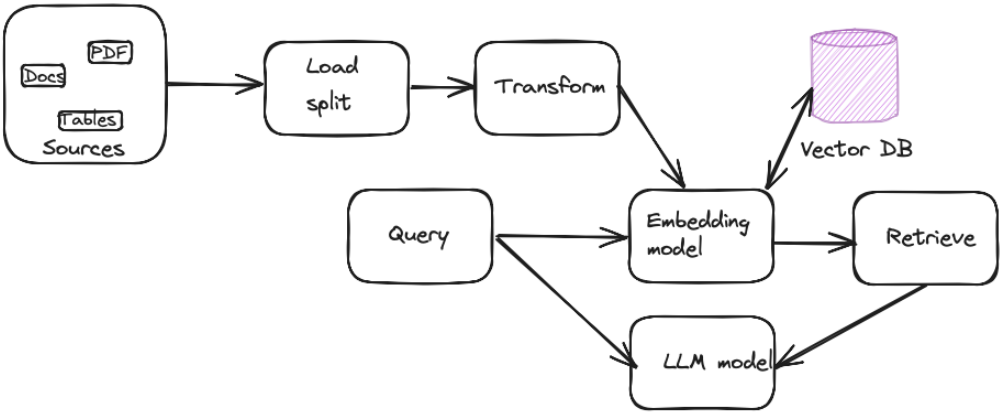


Figure 3.1

Key concepts of RAG

To build a local RAG system, you'll need the following components.

1. Sources. Source documents, it might be .doc, txt or pdf files located your network.
2. Loader. A loader which will load and split the documents into chunks..
3. Transform. Transform the chunk for embedding.
4. Embedding model. Embedding model takes the input as a chunk and outputs an embedding as a vector representation.
5. Vector DB. Vector database for storing embedding.
6. LLM model. Pre-trained model which will use the *embedding* to answer the user query.

Embeddings

As we covered earlier in chapter 2 that, embedding is a mathematical representation of data, like words or images, in a high-dimensional space. These representations capture the semantic meaning or features of the data, making it easier for machine learning models to process and understand complex inputs.

An embedding is essentially a list of numbers (a vector) that represents an item, like a word, in a multi-dimensional space. Embeddings represent not just the surface form of the data (e.g., a word) but also its meaning and context.

Embeddings in RAG are most commonly used for providing text similarity. For example, word embeddings can help find similar words, such as lion and tiger, which have embeddings that are close to each other in the vector space.



Tip

For more details and examples on embeddings, please refer to Chapter 2.

Vector database

A vector database, also known as a *similarity search engine*, is a specialized database designed to store and efficiently retrieve vectors. These databases are optimized for performing nearest neighbour searches (i.e., finding the most similar item based on their embeddings) in high-dimensional vector spaces. Unlike traditional relational databases, vector databases can compare vectors directly without needing explicit queries about attributes.

Key Characteristics of a Vector Database:

- **Stores embeddings**: Instead of storing raw data (like text or images), it stores the vector representations (embeddings) of this data.
- **Specialized indexing**: Uses techniques like HNSW (Hierarchical Navigable Small World graphs) or FAISS (Facebook AI Similarity Search) to index and search for similar vectors efficiently.

- **Scalability**: Can handle millions or billions of vectors and perform fast similarity searches even in high-dimensional spaces.

Let's say, you have a database of movie descriptions stored as embeddings. When you input the embedding of a new movie description, the vector database can quickly find similar movies based on their descriptions.

Let's go by an example:

- Movie A: "A notorious pirate characterized by his slightly drunken swagger"
  - Embedding: [0.9, 0.1, 0.8, ...]
- Movie B: "A pirate who wants to confront Jack for stealing her ship."
  - Embedding: [0.85, 0.15, 0.75, ...]

If you query the database with the embedding of Movie A, it will also return Movie B because their embeddings are close in the vector space, indicating they have similar content.

Vector databases can be used in various scenarios:

- **Semantic Search**: For example, when you search for *artificial intelligence* in a document database, the vector database can find documents that contain related topics like *machine learning* or *neural networks*.
- **Image Retrieval**: Find similar images based on their visual features, such as finding all images of dogs in a large photo collection.
- **Recommendation Systems**: Quickly find and recommend products, articles, or media similar to what a user is interested in.

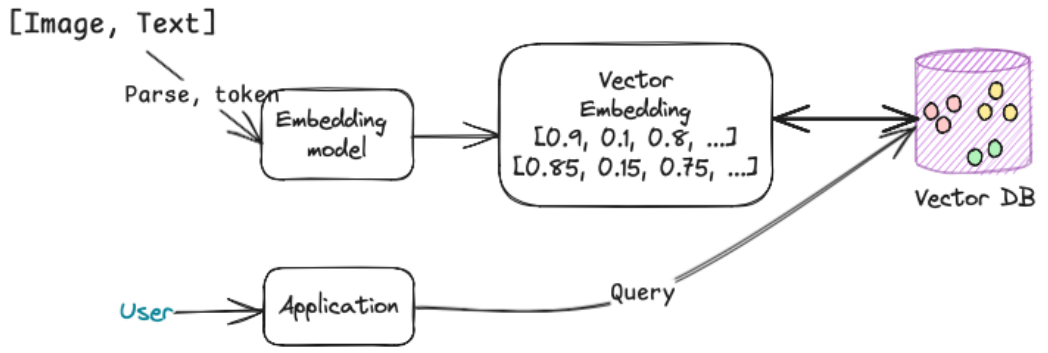


Figure 3.2

In RAG, vector database is commonly used for semantic search. Popular vector databases are:

1. Milvus
2. Chroma
3. Pinot
4. Weaviate

Since vector databases enable semantic and similarity searches over stored vectors, let's explore semantic search in more detail.

Semantic Search

Semantic search is an advanced search technique that leverages the meaning and context of words to deliver more relevant results. This type of search interprets the intent behind a user's query by understanding the meaning of the words and their context, rather than just looking for exact matches. It aims to find information that is conceptually relevant to the query, even if it doesn't contain the exact keywords.

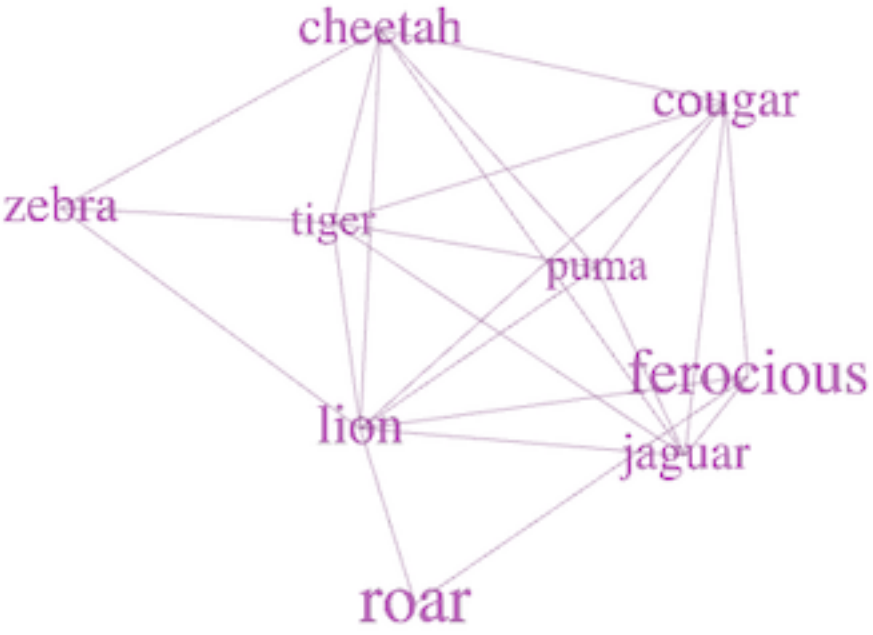


Figure 3.3

Figure 3.3 (generated by the Embedding Projector by Tensor flow) illustrates the relationship between animals like *lions* and *tigers*, which can be queried using semantic search.

Semantic search often uses AI models, like transformers or other *NLP* models, to understand and process natural language queries. These models are capable of capturing the nuances of human language, such as synonyms, related concepts, and the context of words.

Imagine you're searching for a *Big red car* in a rental system. The expected results would include cars that fit that description, such as a *Ferrari* or a *Lamborghini*. In this example, the semantic search engine understands the meaning behind your query and delivers relevant results based on the context and intent of your search, rather than just matching the exact words.

Think of it like this:

- Exact search: `Find a car that matches 'big red car'` (only finds cars with those exact words).
- Semantic search: `Find a car that is big and red"` (finds cars that match the description, even if they don't have the exact words).

Key benefits of Semantic Search:

- Improved Relevance: By focusing on meaning and context, semantic search delivers results that are more aligned with the user's intent, improving the relevance of search results.
- Handling Synonyms and Polysemy: Semantic search can understand that different words might have the same meaning (synonyms) or that the same word can have different meanings in different contexts (polysemy), providing more accurate results.
- Contextual Understanding: It can consider the context in which words are used, making it effective for complex queries where traditional search might fail to capture the nuances.

How semantic search is different from full text search?

Semantic Search and Full Text Search are two different approaches for searching text data. While they share some similarities, there are key differences between them.

Full Text Search:

- Searches for exact matches of words or phrases within a document.
- Typically uses keyword-based matching algorithms (e.g., exact match, prefix matching).
- Focuses on literal meaning of the search query.
- Often used in scenarios where accuracy is crucial (e.g., search engines, database queries).

Semantic Search:

- Searches for matches based on meaning and context rather than literal words.

- Uses natural language processing (NLP) and machine learning-based algorithms to understand the query's intent.
- Can handle complex queries, synonyms, antonyms, and other relationships between concepts.
- Typically used in scenarios where accuracy is important but not as critical as speed (e.g., e-commerce search, customer support)

Key differences between Full Text Search and Semantic Search:

- Matching criteria: Full Text Search relies on exact matching of words, while semantic search uses similarity metrics to match the meaning.
- Query complexity: Semantic search can handle more complex queries with multiple keywords or entities, whereas full text search is often limited to simple queries.
- Contextual understanding: Semantic search takes into account the context and relationships between concepts, whereas full text search focuses solely on literal matching.

Now that we have a solid understanding of the integral parts of the mechanism, let's return to the use case of RAG and its implementation.

Real world use cases of using RAG

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Implementing RAG in a private company

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Step-by-Step Example: Loading, retrieving, and processing custom documents with LLM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Chapter 4: Text-to-SQL, enhance your LLM responses by integrating data from the Database

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

What is Text-to-SQL?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Challenges of Text-to-SQL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

LLM for Text-to-SQL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

System design patterns of using Text-to-SQL with examples

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Design pattern 1. Generating and executing SQL queries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Design pattern 2. Using Agent's for error handling and ensure correctness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Design pattern 3. Text-To-SQL with RAG

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Chapter 5: Fine-tuning LLMs

Fine-tuning a Large Language Model refers to the process of adapting a pre-trained LLM, like Llama or Mistral, for a specific task or domain by modifying its weights and architecture. This is done to make the model more effective at a particular problem, such as text *classification*, *sentiment analysis*, or *document summarization*.

In the fine-tuning process, you start with a pre-trained model, like **LLaMA 3**, which has been trained on a vast amount of general knowledge from various sources. You then fine-tune this model for a specific task, such as code generation or converting text into SQL code.

> To put it in plain English, imagine you have a super smart friend who knows a lot about many topics but hasn't specifically studied your area of interest, like hiking or diving. This friend represents a pre-trained LLM model. Now, you want to teach your friend more about your specific area of interest. You provide him with examples and explanations related to hiking, for instance. As he learns from these new examples, his understanding becomes more specialized and accurate in the context of hiking.

We can visualize the entire process as shown below:

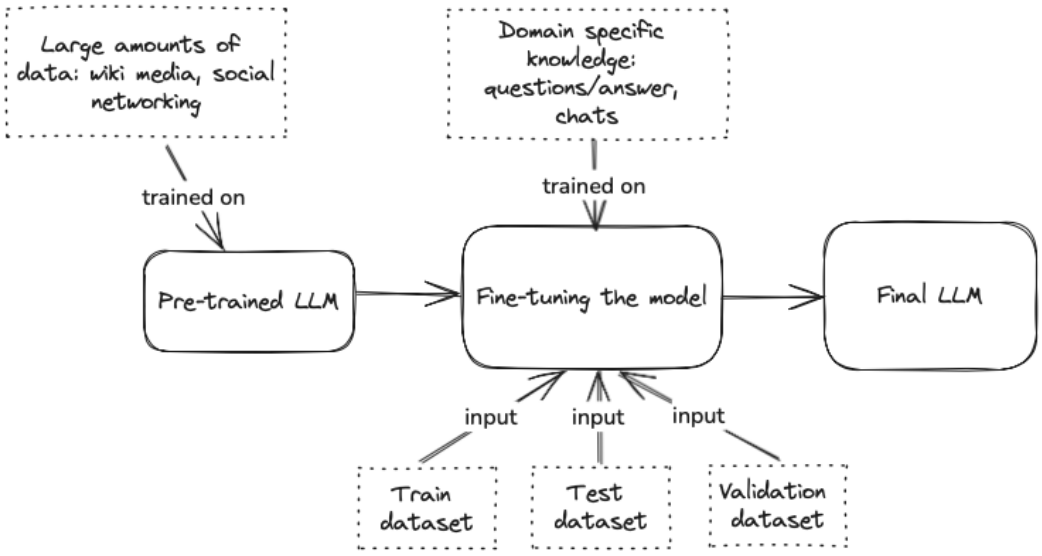


Figure 5.1

LLM fine-tuning offers several benefits:

1. **Cost-effective**: Fine-tuning is less computationally expensive and time-consuming compared to training a model from scratch.
2. **Domain expertise**: By adapting the model for your specific domain, you can leverage its language understanding capabilities in that context.
3. **Flexibility**: Fine-tuned models can be adapted for various tasks and domains, making them versatile assets.

However, fine-tuning a LLM is a resource-intensive process that requires specific knowledge of the overall tuning process. Before investing time and resources into fine-tuning an LLM, it's essential to ensure that you really need to do so. Consider fine-tuning your model when,

- **You have a small dataset**: Fine-tuning is suitable when you don't have a large amount of labeled data available for training from scratch.
- **Domain-specific knowledge is crucial**: When the task or domain requires specialized knowledge, fine-tuning can help adapt the model to better understand the context.

- **Resources are limited**: In cases where resources (e.g., GPU, compute power) are constrained, fine-tuning offers a more efficient alternative.

In this chapter, we'll provide a thorough overview of the fine-tuning process for LLMs, giving you a solid understanding of the theoretical foundations. Then, we'll guide you through practical steps to tune an open-source model like Phi-2 for a specific task. Throughout this chapter, our main goal is to maintain a balance between theoretical concepts and practical applications. To achieve this, we'll first dive into the underlying theories and then move on to concrete examples and hands-on exercises.



Info

Note that, fine-tuning a Large Language Model is a complex topic that requires a comprehensive understanding of various concepts and techniques. One chapter or brief overview is not sufficient to cover all aspects of this process. If you're interested in learning more about fine-tuning an LLM, I recommend exploring additional resources dedicated to this specific process.

Steps for Fine-tuning a pre-trained model

The fine-tuning process of large language models is more complex than what's depicted in Figure 5.1. As we've previously mentioned, fine-tuning involves training a pre-trained model on a specific dataset related to your task, business requirements, or domain. The general steps for fine-tuning an LLM are illustrated in Figure 5.2.

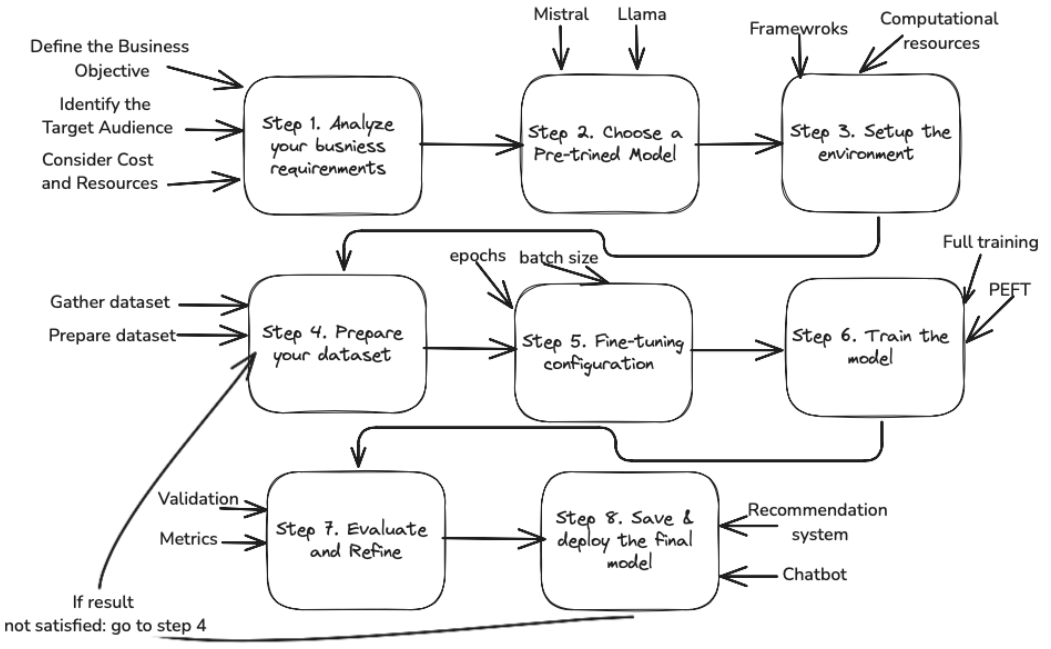


Figure 5.2

While the details may vary depending on the library or framework you are using, here's an outline of general steps to guide you through the process:

1. **Analyze your busniess requirements**: Before fine-tuning LLM, it's crucial to analyze your business requirements to ensure the fine-tuning effort aligns with your goals, maximizes resources, and delivers value. Here's a structured approach to analyzing your business requirements before fine-tuning:

    - *Define the Business Objective.* Clearly define why you want to fine-tune an LLM. Are you aiming to automate customer service, improve content generation, or streamline internal processes? Identify the key business problem or opportunity the model will address.
    - *Identify the Target Audience.* Determine whether the fine-tuned LLM will be customer-facing (e.g., chatbots, content generation) or used internally (e.g., document processing, knowledge retrieval).
    - *Evaluate Data Availability.* Determine whether you have enough high-quality data for fine-tuning. For example, if you're building a customer

service chatbot, do you have enough chat transcripts or support tickets to train the model effectively?
- *Consider Cost and Resources.* Fine-tuning large models requires significant computational resources (e.g., GPUs or TPUs). Assess the budget for cloud services or hardware to handle the fine-tuning process. Ensure your team has the skills in machine learning, data processing, and model evaluation.

2. **Choose a Pre-trined Model**: Start by selecting a pre-trained model that fits your needs. Common models include GPT, LLaMA, Mistral and others. The choice depends on the task (e.g., text generation, classification, or translation) and the availability of models. Ensure that the pre-trained model has been trained on a large corpus of general data, which provides a strong foundation for fine-tuning. Moreover, to compare different pre-trained models, refer to benchmarks (or LLM Leaderboard) such as:

   - *MMLU*: Evaluates the performance of language understanding capabilities.
   - *HellaSwag*: Assesses the ability to generate coherent text.
   - *DROP*: Tests the model's capacity for reasoning and common sense. By considering these factors and comparing pre-trained models through established benchmarks, you can make an informed decision and select the ideal model for your specific needs.

3. **Setup the environment**:

   - Set up the necessary software, frameworks, and tools for fine-tuning. Common libraries include Hugging Face's Transformers, PyTorch, MLX and TensorFlow.
   - Fine-tuning requires significant computational resources, so it's important to use GPUs or TPUs for faster training. Cloud platforms like Google Colab, Kaggle or AWS can provide access to such hardware.

4. **Prepare your dataset**:

   - Gather or create a synthetic dataset that is relevant to the task you want to fine-tune the model for. This could include labeled examples (e.g., text with categories for classification, dialogue for chatbots).

- You'll need to prepare your dataset by splitting it into two categories: Training and Testing. *Training set* portion of the dataset will be used to train the LLM. On the other hand, *Testing set* part of the dataset will be used to evaluate the performance of the fine-tuned model. - Ensure the data is clean and pre-processed. For text-based tasks, this might involve removing irrelevant information, tokenizing the text, or converting data into the format required by the model (e.g., converting text to tokens).
- Organize the dataset in a format the model can understand, such as input-output pairs for supervised learning tasks (e.g., input: text, output: summary).

5. **Fine-tuning configuration**: Define hyperparameters such as *learning rate*, *batch size*, and the number of training *epochs*. These may vary based on your specific task.
6. **Train the model**: Train the model on the task-specific dataset. During this step, the model's parameters are adjusted based on the new data to make it more effective for the target task. Monitor performance metrics like loss, accuracy, or other relevant evaluation metrics. This helps in tracking progress and determining when to stop training to avoid overfitting. You can consider W&B platform for monitoring metrics during fine-tuning process.
7. **Evaluate and Refine**: After fine-tuning, evaluate the model on a separate validation or test dataset to check its performance on unseen data. Use task-specific metrics such as accuracy, precision, ROUGE, F1 score, or BLEU score to assess how well the fine-tuned model performs. Human evaluation is also commonly used to gauge the model's effectiveness.
8. **Save & deploy the final model**: Once fine-tuning is complete, save the fine-tuned model, including all relevant weights and configurations. Deploy the fine-tuned model into production or integrate it into your application, such as a chatbot, recommendation system, or document processing pipeline. After deployment, continue to monitor the model's performance in real-world scenarios and retrain or fine-tune further if needed.

If the initial fine-tuning process doesn't deliver the desired results on production, there are several steps you can take to refine your model:

- **Augmenting the Dataset**. Consider augmenting your dataset by:

  - Adding More Training Data: Supplement your existing data with additional examples related to your specific task or domain.
  - Refining the Dataset: Review and refine the existing dataset for improved quality.

- **Adjusting Hyperparameters**. After evaluating the model's performance, adjust hyperparameters as needed:

  - Learning Rate: Adjust the learning rate to optimize the model's convergence speed.
  - Number of Training Epochs: Increase or decrease the number of training epochs to achieve better results.

- **Fine-Tuning Again**. Once you've made the necessary adjustments, fine-tune the model again using the updated dataset and hyperparameters. This process might require several iterations, but it's a normal part of the fine-tuning workflow.

Fine-tuning technics

So far, we have learned about the steps required for fine-tuning an LLM. However, a lot of technical jargon is involved in the process of fine-tuning. During this process, the model's weights (refer to Chapter 2 for more details) are adjusted to better align with the specific task or data. Fine-tuning techniques focus on finding the optimal way to modify these weights so the model can perform well on a target task without losing the general knowledge gained during pre-training.

There are various fine-tuning techniques that can be applied to LLMs, depending on the task, available data, and computational resources. In this section, we will discuss the most prominent fine-tuning methods used for LLMs:

Full Fine-Tuning

In this approach the entire model's weights are updated based on a task-specific dataset. The model learns to adapt its internal representations to the new task. It

demands significant memory and computational resources, similar to pre-training, to handle storage and processing during training.

The weights of the model are updated using an optimization algorithm, typically **Adam or Stochastic Gradient Descent (SGD)**, based on the computed gradients. Moreover, all layers of the model are updated. This includes early layers that capture general language features and deeper layers that focus on more abstract representations. The magnitude of these updates depends on the learning rate, a **hyperparameter** that controls how large the adjustments are. Lower learning rates prevent drastic changes to the weights, ensuring that the model retains valuable pre-trained knowledge while adjusting to the new task.

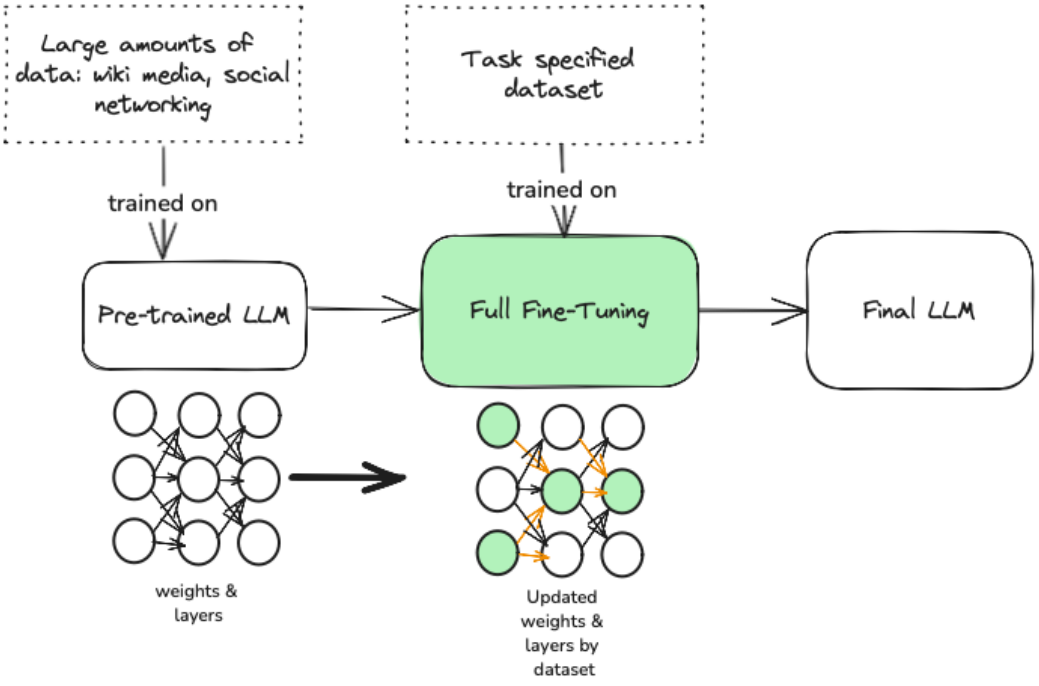


Figure 5.3

Key Characteristics of Full Fine-Tuning:

- All layers are updated: In full fine-tuning, every weight in the model is adjusted,

allowing the model to deeply specialize in the new task.
- Computationally expensive: Since all weights are updated, full fine-tuning requires significant computational resources, especially for large models.
- Highly effective for domain-specific tasks: Full fine-tuning allows the model to adapt comprehensively to the nuances of the task-specific data, leading to high performance on specialized tasks.
- Risk of overfitting: Because the model is fully updated, there is a risk of overfitting to the fine-tuning dataset, especially if it is small.

However, it produces the best task-specific performance because all layers are adjusted.

Parameter-Efficient Fine-Tuning (PEFT)

This is one of the most common methods of fine-tuning LLMs today. With PEFT, only a subset of the model's parameters is updated. Instead of modifying all the weights (as in full fine-tuning), PEFT focuses on adjusting a smaller subset of parameters, often by introducing additional components or adapting specific layers. During the process, the rest of the model is typically `frozen`, meaning it remains unchanged. This allows the model to specialize in a new task without retraining the entire network.

Key Characteristics of PEFT:

- Efficiency in Memory and Computation: PEFT significantly reduces the number of trainable parameters, which means less memory and computational power is required. This is especially important for very large models, which can be resource-intensive to fine-tune fully.
- Preserves Pre-trained Knowledge: Since most of the model's parameters remain unchanged, PEFT retains much of the general knowledge acquired during the pre-training phase. This minimizes the risk of catastrophic forgetting, where the model loses its ability to perform well on tasks it was initially trained on.
- Adaptability to Multiple Tasks: With PEFT, it's possible to fine-tune a model for multiple tasks by introducing small, task-specific components (e.g., adapters) without altering the core model. This enables multi-task learning or domain adaptation without needing to retrain the entire model from scratch.

- Scalability: PEFT methods allow for efficient scaling across different tasks or domains. This is particularly useful for organizations that need to deploy LLMs across various applications (e.g., customer support, content creation, translation) while keeping costs low.

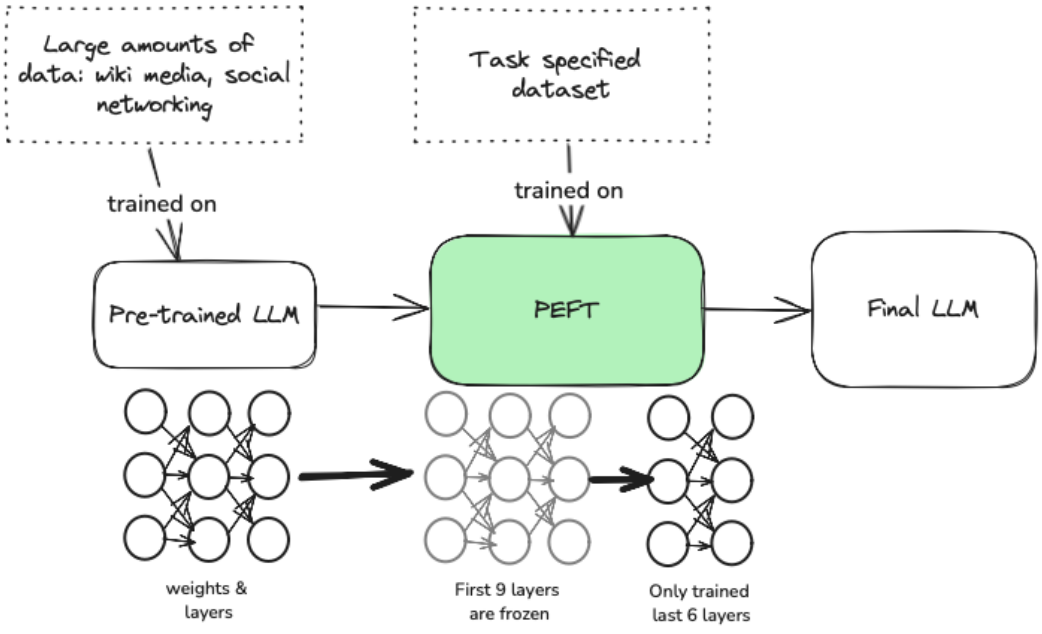


Figure 5.4

While PEFT is highly efficient, it may not always achieve the same level of performance as full fine-tuning. Since only part of the model is being adapted, it may not capture the full complexity of certain tasks.

There are a few ways of achieving Parameter efficient fine-tuning. **Low-Rank Adaptation LoRA** & **QLoRA** are the most widely used and effective.

LoRA (Low-Rank Adaptation)

LoRA is a technique that introduces low-rank update matrices (matrix) into the attention layers of the transformer model. Instead of updating all weights in the

attention layers, LoRA applies low-rank transformations to capture task-specific variations.

LoRA injects small, trainable matrices that modify only a small subset of the model's internal components. The model learns task-specific representations by adjusting only these low-rank matrices, while the bulk of the model's parameters remain frozen. Such a way, it reduces the number of trainable parameters significantly, making fine-tuning more efficient without sacrificing much in terms of performance.

After LoRA fine-tuning for a specific task or use case, the original LLM remains unchanged, while a much smaller `LoRA adapter` is created—often only a single-digit percentage of the original LLM's size (measured in MBs rather than GBs).



Info

A LoRA adapter is a smaller, task-specific version of the original LLM that is created after LoRA fine-tuning. You can think of the adapter as a plugin for the pre-trained model, typically accounting for only a small fraction of the original LLM's size. Multiple adapters can be generated for different purposes. During inference, the adapter is combined with the original LLM to produce accurate results, offering flexibility to deploy the same LLM for various tasks.

Quantized LoRA (QLoRA)

QLoRA is a variant of the Low-Rank Adaptation (LoRA) technique that incorporates model quantization to make fine-tuning even more memory and computationally efficient. QLoRA combines the principles of LoRA, which fine-tunes small low-rank matrices while freezing the rest of the model, with quantization.



Info

Quantization reduces the precision of a model's parameters, such as converting from 16-bit floating point to 4-bit integers. This process allows the same LLM to be more flexible and efficient for different tasks.

By using 4-bit quantization, QLoRA reduces the memory footprint of large language models (LLMs) by representing their weights with lower precision. This significantly decreases memory requirements and accelerates both training and inference.

Like LoRA, it adds small, trainable low-rank matrices to the model. These matrices are fine-tuned for the specific task while the rest of the quantized model remains frozen. Despite quantization, QLoRA can achieve performance close to full-precision models, balancing efficiency and task accuracy.

> The 4-bit quantization significantly reduces memory usage, making it possible to fine-tune very large models even on smaller GPUs, such as the Google Colab T4 with 16 GB of GPU RAM.

Knowledge Distillation (KD)

In these methods a smaller `student` model is trained to mimic the behavior of a larger `teacher` model. The teacher is often a fine-tuned or pre-trained LLM, and the student learns to replicate its behavior on a task while being smaller and more efficient. It reduces the size of the model while preserving much of its performance. Most commonly, these types of LLMs are used in resource-constrained environments, such as mobile devices.

As shown in Figure 5.5 (original source Arxiv), in knowledge distillation, a small `student` model learns to mimic a large `teacher` model and leverage the knowledge of the teacher to obtain similar or higher accuracy.

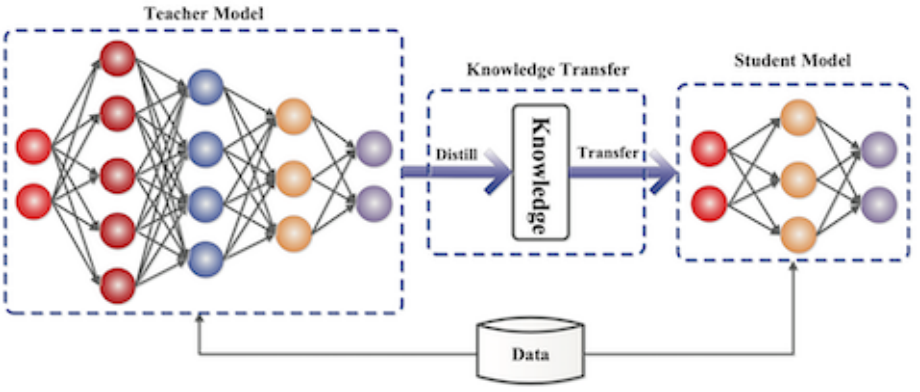


Figure 5.5

Key Characteristics of knowledge distillation:

- Teacher-Student Paradigm: The process involves two models: the teacher and the student. The teacher is typically a large, well-trained model that has been fine-tuned on a specific task or dataset.
- Knowledge Transfer: The goal of KD fine-tuning is to transfer knowledge from the teacher to the student. This is done by training the student on the same data as the teacher, but with an additional "temperature" term in the loss function that encourages the student to mimic the output probabilities (or logits) of the teacher.
- Soft Targets: The student learns to predict soft targets, which are probability distributions over the output classes, rather than hard labels. This allows the student to learn more general and abstract representations.
- Improved Transfer Learning: KD fine-tuning improves transfer learning by allowing the student to inherit the knowledge and representations learned by the teacher. This enables the student to learn more efficiently from a smaller dataset or with fewer resources.

Several techniques can be used to implement knowledge distillation fine-tuning:

- Softmax Temperature Scaling: This involves scaling the output of the teacher model using a temperature parameter to produce softer targets.

- Output Regularization: Use output regularization techniques such as dropout or weight decay to encourage the student model to learn diverse and robust representations.
- Teacher-Student Weight Sharing: Allow certain weights in the pre-trained LLM (the teacher) to be shared with the new task or dataset, effectively transferring knowledge from the teacher to the student.

Note that, KD fine-tuning is not a replacement for traditional fine-tuning methods, but rather an additional tool to improve the performance and efficiency of LLMs.

Popular frameworks used for fine-tuning LLMs

Several frameworks are commonly used for fine-tuning LLMs. These frameworks provide the tools, libraries, and infrastructure needed to adapt pre-trained models to specific tasks. As this book is written, the field of AI is rapidly evolving, with new frameworks and libraries emerging to address changing needs. Despite these changes, here's a list of popular frameworks used for fine-tuning LLMs:

1. **Hugging Face Transformers**.

   Hugging Face is one of the most popular frameworks for fine-tuning LLMs. It provides pre-trained models, tokenizers, and easy-to-use APIs for adapting models to various tasks (e.g., text classification, summarization, question-answering).

   Key features:

   - allows downloading/uploading pre-trained models and datasets.
   - provides easy-to-use APIs for loading, fine-tuning, and deploying models.
   - seamless integration with PyTorch and TensorFlow.
   - `Trainer` class simplifies fine-tuning by handling training loops and evaluation.

2. **PyTorch**.

PyTorch is another popular deep learning framework that allows fine-tuning of LLMs using its high-level API. It's often used in conjunction with the `transformers` library from Hugging Face. It allows for custom fine-tuning of LLMs by providing low-level control over the training process.

Key features:

- provides dynamic computation graphs.
- integration with Hugging Face Transformers and other high-level libraries.
- very strong community support with many tutorials and libraries.

3. **TensorFlow**.

   TensorFlow is a widely used open-source machine learning framework that supports fine-tuning LLMs. TensorFlow, with its high-level API `Keras` offers scalability for large models and allows for distributed training.

   Key features:

   - supports for distributed training across GPUs.
   - integration with `TensorFlow Hub`, which provides pre-trained models for easy fine-tuning.
   - extensive ecosystem for machine learning and deep learning.

4. **Unsloth**.

   Unsloth is a Python library designed to make fine-tuning pre-trained language models, particularly those from the Hugging Face Transformers library. It provides significant improvements in memory efficiency and speed, making it possible to fine-tune large models on smaller hardware. Unsloth leverages techniques like 4-bit quantization, LoRA (Low-Rank Adaptation), and gradient checkpointing to reduce memory usage while preserving performance. This allows models to handle longer context lengths (>2048), which is particularly useful for tasks requiring extended sequences.

   Key features:

   - provides an interface for efficiently fine-tuning pre-trained language models, using techniques such as knowledge distillation and model pruning.
   - helps reduce the memory requirements of large models during training, making it possible to train them on smaller machines or with less RAM.

- can compress large models into more compact forms, making them easier to store and transfer.
- allows users to combine multiple pre-trained models into a single ensemble model, which can improve performance on certain tasks.

5. **MLX framework**.

   MLX is a new player in the LLM community, offering an innovative array framework for machine learning research on Apple silicon. Developed by Apple's machine learning research team, MLX is primarily designed to leverage Apple silicon processors and GPUs for local training of LLMs.

   Key features:

   - supports composable function transformations for automatic differentiation, automatic vectorization, and computation graph optimization.
   - operations can run on any of the supported devices (currently the CPU and the GPU).
   - provides a Python API that closely resembles NumPy, allowing for seamless integration and compatibility.

> If you're an owner of a Mac device featuring Apple silicon processors (such as M1, M2, or M3) with sufficient memory, MLX is an excellent choice for fine-tuning LLMs.

Keep in mind that each framework described above has its own strengths and advantages, which depend on factors such as model scale, resource availability, and ease of use. Additionally, there are other frameworks like JAX, Flax, and Colossal-AI that cater to cutting-edge research and large-scale fine-tuning needs.

Step-by-step example of fine-tuning an LLM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Prerequisites

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Part 1. Analyze business requirements, choosing a base model and environment setup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Part 2. Exploring the training dataset

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Part 3. Dataset pre-processing and adapter configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Part 4. Train the model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Part 5. Evaluate the model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Part 6. Save & deploy the final model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Chapter 6: Image processing & generating with LLM

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Image visioning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Possibilities and Functionalities of LLaVA-v1.6

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

LLaVa architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Step-by-Step Example: Utilizing LLaVA-v1.6 for Image Visioning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Incorporating LLaVA into your application for image processing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Image processing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Tips for Better Results

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Chapter 7: Developing and utilizing AI agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

The future of AI agents

Looking ahead, AI agents will evolve through a series of progressive stages as they become more integrated into enterprise operations. By automating both simple and complex tasks, AI agents will free up human workers to focus on creative problem-solving and strategic decision-making.

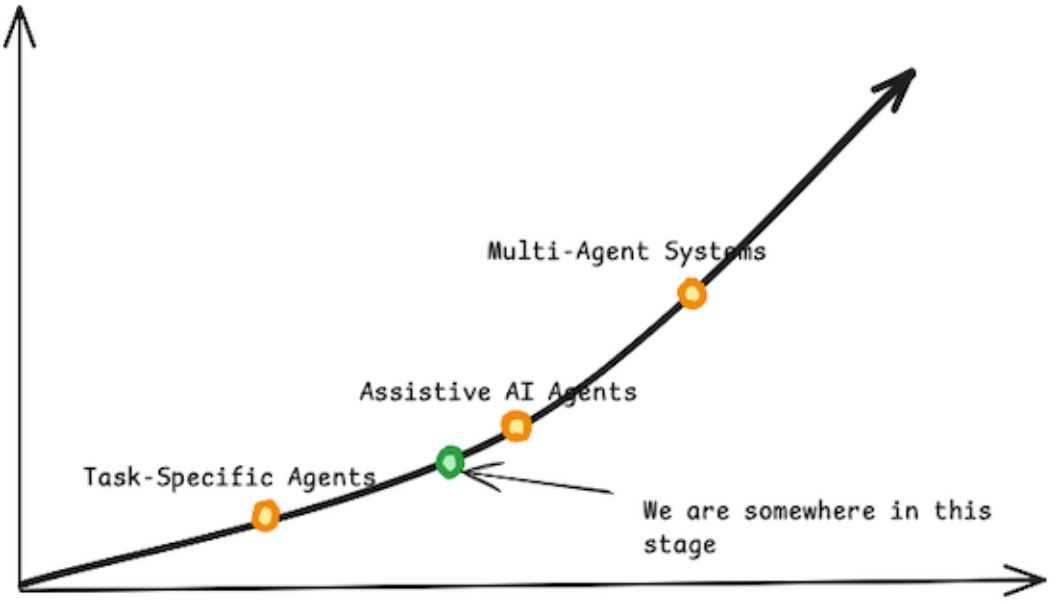


Figure 7.1

According to Google Cloud's observations, this transformation will occur in three key phases:

1. **Task-Specific Agents**. The first wave of AI agents will focus on specialized, well-defined tasks. These agents will be designed to handle repetitive or routine processes, significantly improving efficiency in specific areas. For example, an AI agent might handle customer inquiries in a call center, automate data entry, or generate reports based on structured data. These early AI agents will excel at doing one thing extremely well, allowing enterprises to automate smaller, isolated tasks.
2. **Assistive AI Agents**. The next step will involve the development of more versatile AI assistants, capable of working alongside humans to enhance productivity. These agents won't just complete tasks independently but will actively collaborate with human workers, offering suggestions, providing real-time insights, and automating parts of more complex tasks. For example, an AI assistant can help a marketing professional write campaign content or assist a software developer by generating code snippets, making their work faster to

complete.

3. **Multi-Agent Systems**. In the most advanced phase, we'll see the emergence of multi-agent systems. These systems will consist of multiple AI agents working together to tackle complex, end-to-end processes that span across departments or workflows. In this stage, AI agents will coordinate among themselves to execute large-scale operations, such as managing an entire supply chain or handling a multi-step approval process. The agents will be able to autonomously delegate tasks to one another and adapt to the specific requirements of different stages in the workflow.



Info

Don't be surprised if you encounter new BPMN activity types, such as `AI Agent`, integrated into workflows in the near future.

At these moments, several companies and IT giants are already working seriously on developing Assistive and Multi-agent AI systems. While these systems are still in their early stages, they are capable of solving specific tasks. In our view, we are at a turning point between the era of `Task-Specific Agents` and `Assistive AI Agents`. In the near future, AI agents are likely to become an essential part of business processes, with multi-agent systems playing a key role in optimizing workflows and driving innovation across industries.

Difference between AI Agents and AI Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Use cases of AI agents in Generative AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Use cases from a developer's perspective

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Use cases from a product manager's perspective

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Classification of AI Agents in Generative AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

AI agents architecture

AI agent architectures can vary depending on their classification or purpose. For instance, a fine-tuned agent might include components like learning layers that help it adapt over time. In contrast, a content creation agent might require additional components for tasks like tokenization or text embedding. In this section, we will cover a basic architecture of an AI agent that is commonly used for performing everyday tasks.

AI agents are complex systems built from different modules to perform tasks like understanding language, making decisions, and interacting with the environment. To explain the architecture of an AI, we can break down the components by its functionalities. Here's a detailed breakdown with a conceptual diagram.

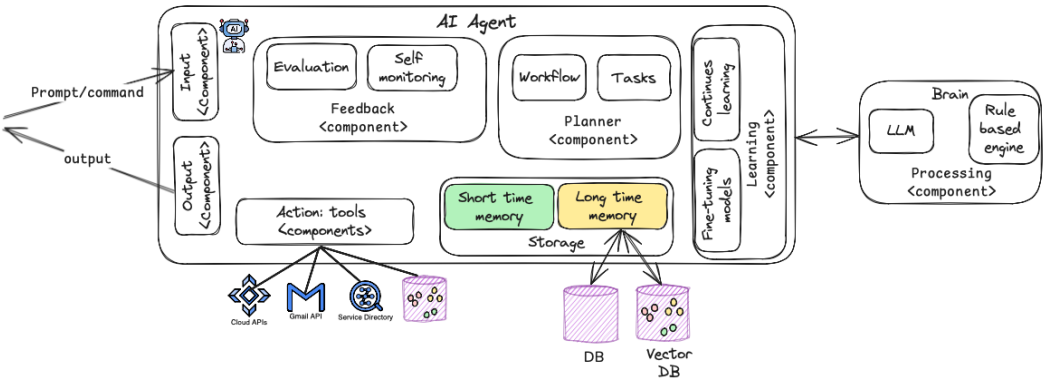


Figure 7.2

The architecture of an AI agent consists of the following components:

1. **Input**. This component takes input from the environment or user (e.g., text, images, or sensor data). For a conversational agent, it could include Natural Language Processing (NLP) components that convert speech to text and preprocess it.

    - Key elements:

        a. Sensors (for robotic AI agents)
        b. Text/Speech/Video inputs (for virtual agents)
        c. Preprocessing (tokenization, normalization, etc.)

2. **Brain or Processing**. This is the core logic where the agent processes input to understand and generate output. It involves various models and algorithms such as:

    - NLP Model: For understanding language in chatbots or virtual assistants.
    - Perception Models: In image or video agents (e.g., object recognition, image classification).
    - Decision-Making Models: For reasoning, planning, or predicting actions.
    - Key elements:

        a. Large Language Models (LLMs)
        b. Perception Models (e.g., CNNs, RNNs)

    c. Rule-based engines

3. **Storage**. This components stores the agent's knowledge, previous interactions, and any additional external information. Agents may access:

    - Short-term Memory: To remember recent interactions.
    - Long-term Knowledge Base: Which may contain facts, rules, or learned models.
    - Key elements:

        a. In-memory database
        b. Databases (structured and unstructured)
        c. Embedding Stores (Vector DB)

4. **Learning**. AI agents often adapt over time. This layer allows agents to update their models or learn from new data.

    - Key elements:

        – Supervised/Unsupervised Learning
        – Reinforcement Learning (updating policies based on feedback)
        – Fine-tuning Models
        – Continuous Learning

5. **Action**. Sometime this component also called tools. This component defines how the agent interacts with its environment or users, such as generating responses in text, executing commands, or invoking 3rd party API.

    - Key elements:

        – Text/Audio Generation (e.g., GPT or TTS models)
        – Actuators (for robotic agents)
        – APIs/Commands (to external systems)

6. **Feedback**. AI agents use this layer to evaluate their own performance and make adjustments, often feeding back into the learning layer. This component ensures the agent can evaluate its performance and learn from its mistakes.

    - Key elements:

  - Reward Functions (in RL)
  - Error Monitoring (to check for failures)
  - User Feedback (for improvement)

7. **Planner**. This component breaks down complex tasks into manageable steps. In some cases it also uses workflow engine or pipeline to execute tasks sequently.

   - Key elements:

     - Workflow engine
     - Tasks pipeline

Depending on the agent's framework or implementation, its architecture may vary. However, most AI agent frameworks include these core components, though they may be referred to by different names. Together, these components form a comprehensive agent capable of working with LLMs to complete complex user tasks.

Frameworks for developing AI Agents

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Langchain and LangGraph

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

CrewAI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

OpenAI Agents SDK

The OpenAI Agents SDK is a Python-based framework designed for building production-ready AI agent applications. It provides a streamlined approach to creating agentic systems where multiple AI agents collaborate through task delegation and tool usage.

Core Components:

- *Agent Loop*: Manages iterative task execution with integrated tool usage and result processing.
- *Handoff System*: Enables dynamic task delegation between specialized agents using triaging logic.
- *Guardrails*: Implements parallel validation pipelines for input sanitization and output safety checks.
- *Tracing*: Built-in observability with execution flow visualization and performance monitoring.



Info

The Large Language Model (LLM) is the core intelligence behind the OpenAI Agents SDK workflow. It powers the agent's decision-making, natural language understanding, and reasoning abilities.

Before diving into the technical details, let's explore its benefits and how it differs from other existing agentic frameworks.

Benefits:

1. *Production Readiness*

    - Simplified upgrade path from experimental Swarm framework
    - Built-in rate limiting and error handling
    - Enterprise-grade security controls for LLM interactions

2. *Developer Experience*

   - 83% reduction in boilerplate code compared to vanilla implementations
   - Integrated debugging through execution tracing dashboard
   - Python-native implementation using modern type hints

Framework Comparison:

There are many agentic frameworks available today. Let's take a quick look at how this protocol compares with other popular frameworks:

Feature	**OpenAI SDK**	**Pydantic AI**	**LangChain**	**CrewAI**
Core Paradigm	Agent orchestration	Structured I/O	Modular components	Role-based multi-agent collaboration
Validation System	Pydantic-based	Native Pydantic	Custom validators	Explicit task definitions
Tool Integration	Flexible, OpenAI tools	Custom + LangChain	Extensive ecosystem	Integrates with LangChain tools
Tracing	Built-in	Logfire integration	LangSmith	Limited built-in tracing
Learning Curve	Low	Moderate	Steep	Moderate
Best For	OpenAI-centric apps	Type-safe workflows	Complex RAG systems	Complex tasks requiring multiple specialists



Info

Pydantic validation is the process of ensuring that input data matches the expected structure, types, and constraints by using Pydantic models. It automates type-checking and data parsing, raising errors when data is invalid.

Workflow:

As I mentioned before that, the LLM is the core intelligence behind the OpenAI Agents SDK workflow. It powers the agent's decision-making, natural language understanding, and reasoning abilities. Here's how it fits into each stage of the workflow:

- Input Interpretation. The process starts when the agent receives an input. This could be a user query, a task to perform, or data from external sources.
- Decision-Making (Agent Logic). The LLM decides whether the agent should process the request internally or invoke external tools. It reasons through complex prompts and selects the best approach.
- Tool Invocation. When the agent receives input, it decides whether it needs external tools (e.g., web search, database queries) to fulfill the request. It calls the relevant tool and processes the output.
- Reasoning and Iteration (Agent Loop). The agent loop is powered by the LLM. It enables the agent to:
  - Reflect on intermediate outputs.
  - Determine if further actions are needed.
  - Continue processing until a valid response is generated.
- Task Delegation (Handoff Mechanism). It delegates tasks to specialized sub-agents. This is useful for breaking down complex workflows.
- Validation and Guardrails. Guardrails ensure that inputs and outputs meet specific validation and safety standards. They prevent the agent from executing unsafe or invalid actions.
- Response Generation. Finally, the LLM synthesizes all gathered information into a coherent, human-readable response.

The next image illustrates the entire workflow.

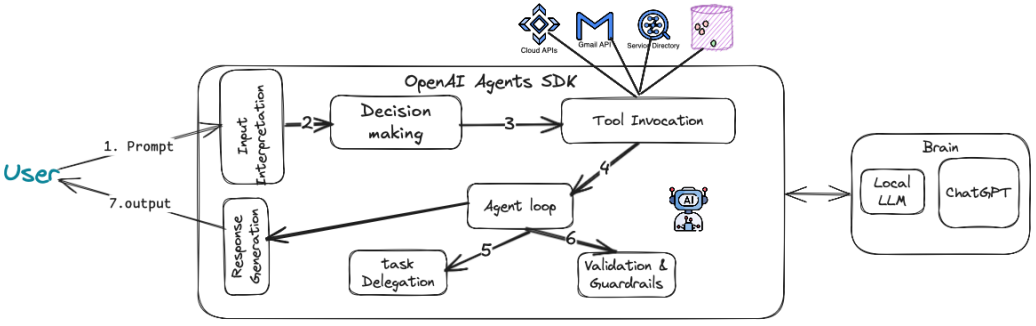


Figure 7.4

Let's summarize the workflow:

1. Input: User provides a request.
2. Agent: Receives and interprets the input.
3. Tools: Calls external tools (e.g., search, data retrieval).
4. Agent Loop: Iterates until the task is complete.
5. Handoff: Delegates tasks to specialized agents if needed.
6. Guardrails: Ensures input/output compliance.
7. Output: Delivers the final response to the user.

While we've only had space to cover a few popular frameworks in this book, however, there are several others worth noting: AutoGen and Vertex AI Agent Builder. Both AutoGen, developed by Microsoft, and Vertex AI Agent Builder, from Google, share a similar architecture and provide a robust platform for creating enterprise-grade generative AI applications and agents. What's more, these tools allow developers to build complex models without requiring expertise in deep machine learning.

In the next section, we'll take a hands-on approach to exploring AI Agents in action. We'll create an agent from scratch, integrating it with a LLM, and demonstrate how it can be used to perform specific tasks.

Developing AI Agents with different frameworks: a step-by-step guide

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Chapter 8: Advanced Tools and Techniques

The landscape of Generative AI is rapidly evolving, with new protocols and frameworks enhancing the efficiency and versatility of local and cloud-based models. As the demand for more flexible and scalable AI solutions grows, advanced tools and techniques play a crucial role in optimizing performance and enabling seamless integration between diverse environments.

This chapter describes cutting-edge protocols such as the Model Context Protocol (MCP), Minion(s): A Simple Protocol for Communicating with Local and Cloud LLMs and Google Agent-to-agent (A2A) protocol. These protocols address key challenges in model interoperability, context management, and efficient data exchange, allowing developers to leverage both local and remote Large Language Models (LLMs) effectively.

We will explore how these protocols facilitate smooth communication, enhance contextual understanding, and enable more sophisticated AI-driven applications. Additionally, this chapter covers practical implementation strategies to harness the full potential of advanced Generative AI tools. Whether you're developing for edge computing or integrating with cloud-based platforms, these innovations offer new pathways to maximize AI capabilities.

By the end of this chapter, you will gain a comprehensive understanding of how to implement and utilize these advanced protocols, empowering you to build more efficient and adaptive Generative AI solutions.



Info

In future sections, we will introduce additional protocols and frameworks that further extend the capabilities of Generative AI with local LLM.

A Simple Protocol for Communicating with Local and Cloud LLMs : Minion(s)

Minion(s) is a communication protocol developed by Stanford's Hazy Research that facilitates collaboration between local (on-device) LLMs and more powerful cloud-based models. Its core purpose is to maximize the use of local LLMs with local data, reducing cloud API costs while maintaining high-quality outputs.

Key components:

1. Local Model

    - Runs on-device (e.g., Llama3.2 via Ollama)
    - Handles context processing/understanding or job
    - Filters irrelevant information before cloud interaction.

2. Cloud Model

    - Uses high-powered models like GPT-4o
    - Receives distilled context from local model
    - Aggregates results
    - Provides sophisticated reasoning/analysis.

Key features of the protocol:

- *Hybrid Approach*: Combines local (on-device) models with powerful cloud-based models to optimize both cost and performance.
- *Local Processing*: Manages context understanding and filters out irrelevant information before interacting with the cloud.
- *Cloud Processing*: Receives the refined context and performs advanced analysis.
- *Iterative Refinement*: Supports multiple rounds of interaction between local and cloud models for improved accuracy.
- *Multi-Provider Support*: Compatible with various cloud providers, including OpenAI, AzurAI.

The protocol is available in two versions: Minion and Minions. Let's begin with the simpler one—Minion.

Minion

A local LLM (such as Llama 3.2 with Ollama) running on a laptop or desktop processes local documents. It interacts with a cloud model like GPT-4o, determining which information to share and when to end the conversation. This selective communication reduces cloud API usage costs while maintaining the quality of the output.

Workflow:

1. Cloud initiates task without full context access
2. Issues targeted requests like: "Using the dealing with hotels dialog, find out how much the mini bar bill"
3. Local model processes full context (10K+ tokens) and responds
4. Iterates until cloud synthesizes final answer

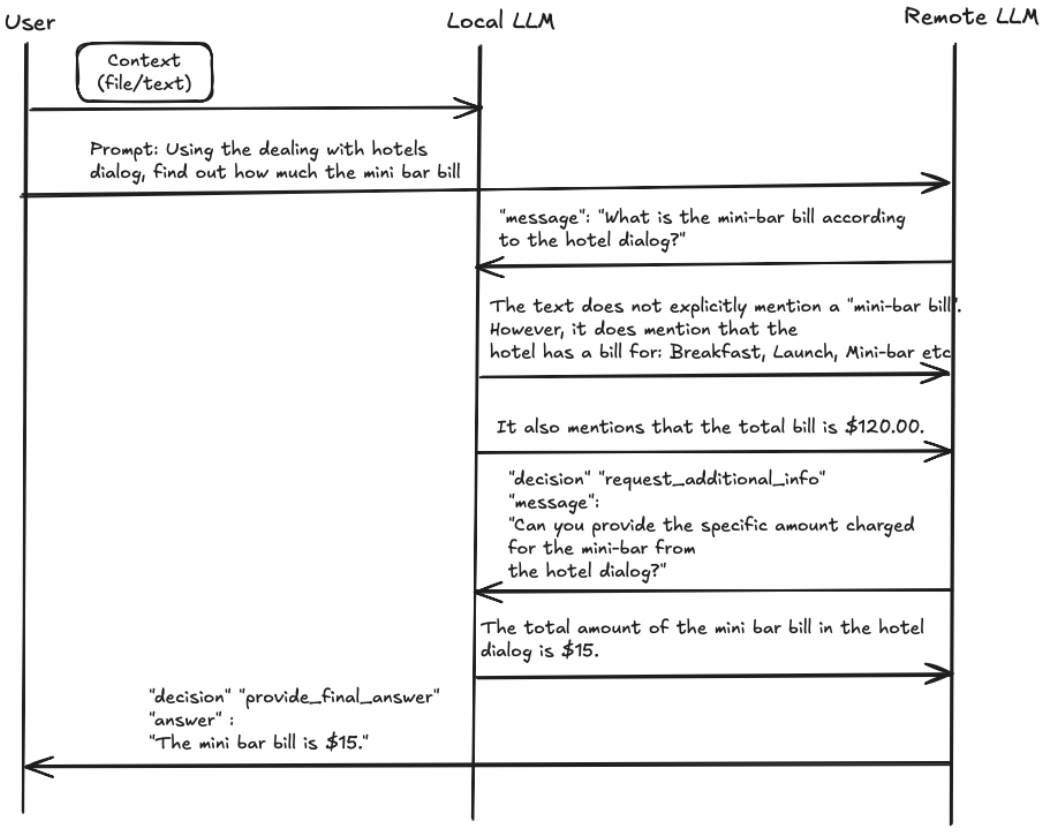


Figure 8.1

Advantages:

1. Sensitive data stays local
2. Avoids full context transmission
3. Customizable Workflows
4. Cost Savings, processes 1918 tokens locally vs. 50 tokens in cloud

Minions

The cloud model receives the prompt, breaks the task into smaller subtasks, and distributes them to the local LLM. The local model processes these subtasks in

parallel, while the cloud model aggregates the final result. This approach utilizes local resources for faster execution while minimizing cloud API costs.

Workflow:

The entire workflow divided into three-phase executions:

Phase	Actor	Action
Decompose	Cloud Model	Generates Python code to split task into parallel subtasks
Execute	Local Model	Runs subtasks concurrently across context chunks
Aggregate	Cloud Model	Synthesizes results, iterating if needed

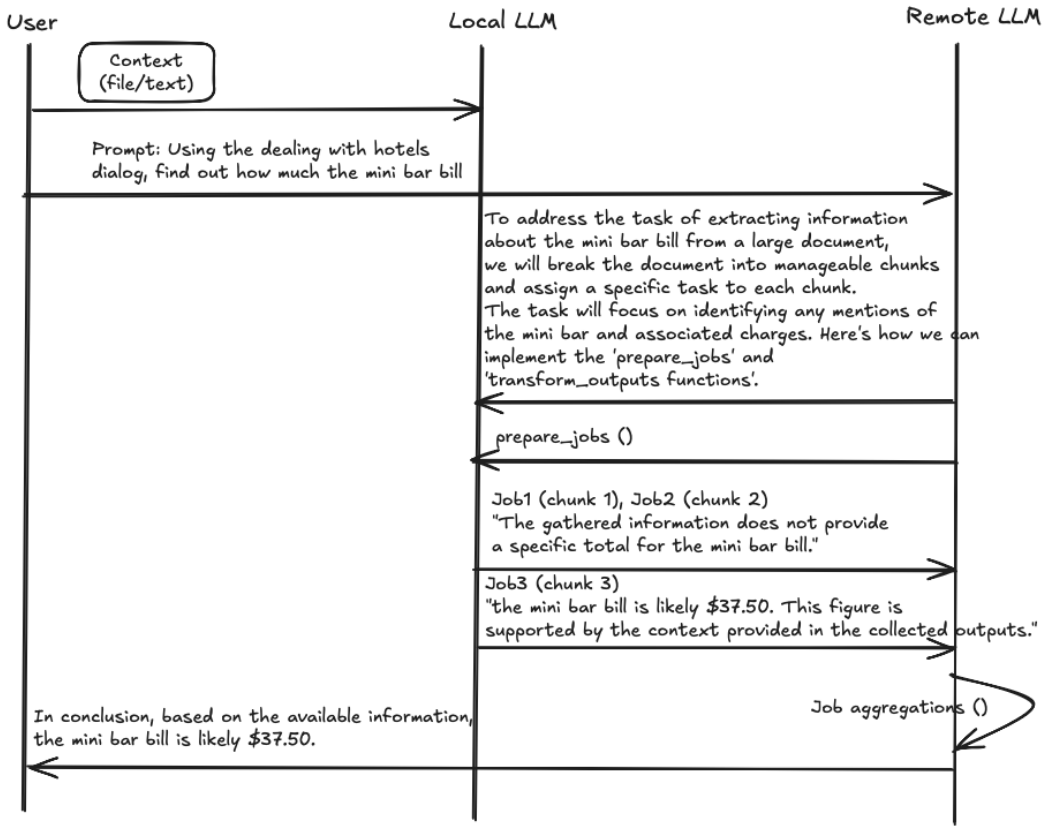


Figure 8.2

Here below are the subtasks created by the Remote LLM.

```
def prepare_jobs(
    context: List[str],
    prev_job_manifests: Optional[List[JobManifest]] = None,
    prev_job_outputs: Optional[List[JobOutput]] = None,
) -> List[JobManifest]:
    job_manifests = []

    # Define the maximum chunk size for this task
    max_chunk_size = 1000  # Smaller chunks for precise information ext\
```

```
raction

    # Chunk the document(s) using the provided chunking function
    for doc in context:
        chunks = chunk_by_section(doc, max_chunk_size=max_chunk_size)

        # Create a job for each chunk to extract mini bar billing infor\
mation
        for chunk in chunks:
            job_manifest = JobManifest(
                chunk=chunk,
                task="Extract any information related to the mini bar b\
ill, including charges, costs, or fees.",
                advice="Focus on identifying specific figures or amount\
s related to the mini bar.",
            )
            job_manifests.append(job_manifest)

    return job_manifests
```

The `prepare_jobs` function breaks down the document into smaller chunks of `1000` characters each, which is suitable for precise information extraction tasks. For each chunk, it creates a `JobManifest` with a task to extract any information related to the mini bar bill, including charges, costs, or fees.

```
def transform_outputs(
    jobs: List[Job],
) -> str:
    mini_bar_info = []

    # Process each job output
    for job in jobs:
        output = job.output
        if output.answer:  # Check if there is an answer related to the\
 mini bar
            mini_bar_info.append(f"Chunk ID {job.manifest.chunk_id}: {o\
```

```
utput.answer}")
            if output.citation:
                mini_bar_info.append(f"Citation: {output.citation}")
            if output.explanation:
                mini_bar_info.append(f"Explanation: {output.explanation\
}")

    # Aggregate all the mini bar information into a single string
    aggregated_info = "\n".join(mini_bar_info)

    return aggregated_info
```

The `transform_outputs` function processes the outputs from the worker model. It filters out any irrelevant or empty results and aggregates the relevant information about the mini bar bill, including any citations and explanations provided by the worker. The aggregated information is returned as a single string for further inspection.

Advantages:

1. Parallel job execution
2. Dynamic context chunking for memory constraints
3. 97.9% accuracy retention

A detailed explanation of the protocol can be found in this link.

So far, we've explored the protocol and how it works. Now, let's run some tests and examples to see it in action.

Examples

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Connecting Local LLMs to Various Data Sources: Model Context Protocol (MCP)

Anthropic launched the Model Context Protocol (MCP) in November 2024 — an open standard for data exchange between LLMs and various data sources. The protocol provides a simplified way for LLMs to integrate with tools and services to perform tasks such as searching files on local systems, accessing GitHub repositories to edit files, and streamlining interactions with external platforms.

From the beginning, the MCP protocol was supported by the Claude Desktop app, and within a few months, several applications like Cline and Cursor AI adopted it to integrate with popular enterprise systems such as Google Drive, Slack, GitHub, Git, Postgres, and Puppeteer. If you follow AI events or newsletters, you've likely come across numerous articles and Vlogs about the MCP protocol and its use with the Claude Desktop app.

One interesting aspect of the protocol is that any LLM with `function-calling` support such as `Llama` or `Qwen` can be used with it. This sparked my curiosity to experiment with the protocol on my local LLM running on home servers. There are also a few open-source projects on GitHub that enable the use of MCP with locally hosted LLMs.



Info

Function-calling feature for LLMs enables language models to interact with external systems by converting natural language requests into structured API calls or function invocations. This capability transforms LLMs from text generators into actionable systems that can execute real-world tasks.

Key Advantages of the MCP protocol:

- *Reduced Integration Complexity*: Converts M×N integration challenges to M+N through standardization
- *Dynamic Discovery*: LLMs automatically detect available tools/resources without pre-configuration

- *Real-Time Context*: Maintains live connections to data sources like:
- *Version control systems*: Git/GitHub
- *Cloud storage*: Google Drive
- *Messaging platforms*: Slack
- *Security*: Granular permission controls and isolated client-server connections

Core Architecture

MCP uses a client-server architecture with four key components:

Component	Role	Examples	
Host	Coordinates LLM interactions and manages multiple clients	Claude Desktop, Cursor IDE	
Client	Maintains 1:1 connection with servers, handles message routing and capability discovery	IDE plugins, chatbot adapters, Custom Agent	
Server	Provides access to tools/resources and manages data operations	GitHub, Slack, Postgres	
Base Protocol	Defines communication standards using JSON-RPC 2.0	RPC 2.0	Message formatting rules

Let's break down the architecture in more detail to explain all the nuts and bolts.

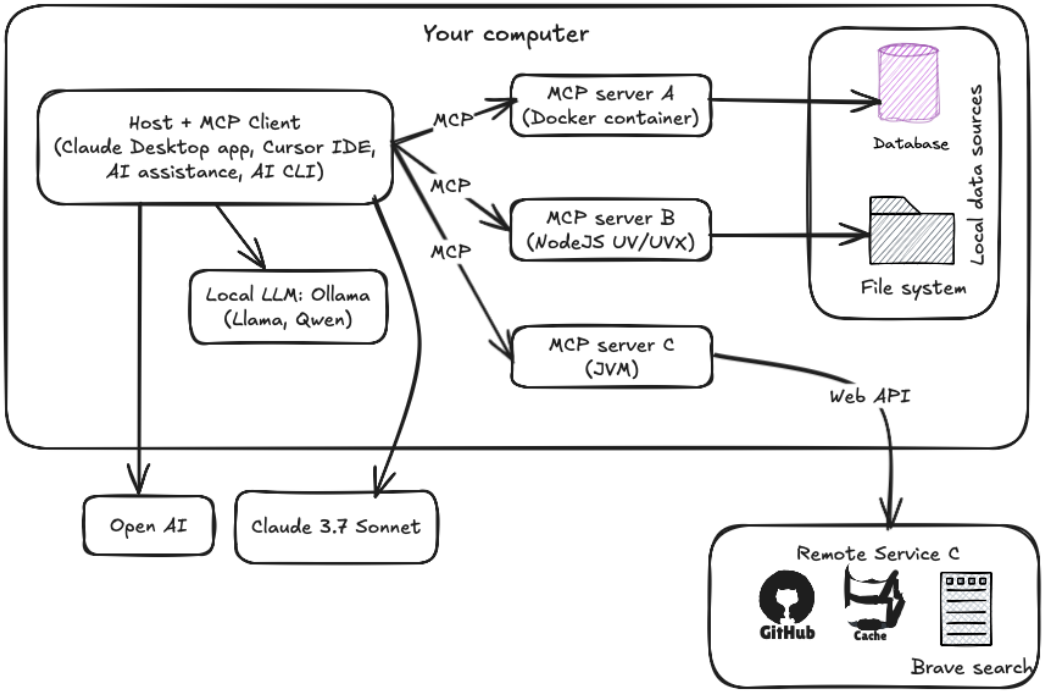


Figure 8.6

MCP Host:

An application that can access services through the MCP protocol. This could be the Claude Desktop app, an AI agent/CLI, or the Cursor IDE. Such applications also use an LLM (local or remote) to perform various tasks.

MCP Client:

A client integrated with the host application to connect with the MCP server.

MCP Server:

An application or program that exposes specific capabilities through the MCP protocol. The server can run on Docker containers, JVM, or Node.js (UV/UVX) processes. The MCP community also provides pre-built servers for performing different tasks.

Local Data Sources:

Databases or file systems located on your local machine.

Remote Services:

External resources, such as GitHub or Brave Search, accessible through web APIs.

Communication details

MCP enables bidirectional communication through:

- Transport Layer: Supports stdio (local processes) and HTTP with Server-Sent Events (remote)
- Message Types:
  - Requests/Responses (initialize, capabilities)
  - Notifications (real-time updates)
  - Error handling with standardized codes

The MCP Base Protocol establishes standardized communication between AI systems and external tools/data sources using JSON-RPC 2.0. The message format `JSON-RPC 2.0` with three core message types:

Message Type	Structure	Purpose
Requests	`{jsonrpc:"2.0", id, method, params}`	Initiate operations (e.g., initialize , capabilities)
Responses	`{jsonrpc:"2.0", id, result/error}`	Reply to requests with success/failure status
Notifications	`{jsonrpc:"2.0", method, params}`	Push real-time updates without response requirements

MCP supports multiple communication channels:

1. STDIO (Local):

- Uses standard input/output streams
- Ideal for local integrations like file systems
- Simple debugging through console logs.

2. HTTP with SSE (Remote):

- Combines HTTP POST with Server-Sent Events
- Enables real-time updates from cloud services
- Handles firewall-friendly bidirectional communication.

The protocol's design reduces integration complexity from `M×N` to `M+N` by standardizing:

- Message serialization
- Transport negotiation
- Capability discovery
- Error recovery patterns.

This foundation enables secure, context-aware AI interactions while maintaining compatibility with existing JSON-RPC tooling and infrastructure.

This is sufficient unless you plan to develop your own MCP client or server. Fortunately, the MCP community provides all the necessary documentation and SDKs to help you get started quickly with the protocol.

Using MCP: A Practical Walkthrough

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Extending the Practical Walkthrough: From CLI to MCP Agent based automation

In addition to previous tools like `mcp-cli` and `MCP-LLM Bridge`, a more advanced and modular integration is now possible using the MCP Agent. This approach doesn't just invoke tools — it turns your RESTful applications into intelligent, LLM-accessible services.

Why RESTful applications still matter

RESTful APIs remain the backbone of modern web and enterprise systems. They are simple, stateless, widely adopted, and easily integrated with databases, cloud functions, and microservices. While newer paradigms like gRPC or GraphQL offer performance and flexibility, REST APIs are often easier to document, secure, and expose to third parties. In most organizations, core business logic is already encapsulated in REST endpoints. By making these APIs LLM-accessible through MCP and agents, we don't need to rewrite or rethink existing infrastructure — we simply augment it with AI. This unlocks intelligent automation, natural language interfaces, and decision support for systems that were never built with LLMs in mind.

Application integration overview

In this section, we demonstrate how to connect an MCP Server with a REST API built using FastAPI and powered by a local language model (LLM) running on Ollama. The REST API manages a basic inventory system with CRUD operations backed by Redis for persistence.

This integration allows an LLM to:

- Discover available API endpoints (described via MCP metadata).
- Invoke them using natural language queries.
- Maintain multi-turn interactions via the agent layer.

It effectively turns traditional REST APIs into LLM-compatible tools — no frontend, no custom parser, just structured metadata and LLM reasoning.

What Is MCP Agent?

MCP Agent is a lightweight runtime framework that turns an LLM into a fully autonomous agent capable of:

- Discovering available MCP-compliant tools.
- Interacting with them using natural prompts.
- Supporting multi-turn interactions.
- Leveraging your preferred local model (via Ollama or OpenAI-compatible APIs).

How the entire work flow works?

The entire pipeline bridges the gap between traditional RESTful applications and modern AI-driven interfaces using the Model Context Protocol (MCP).

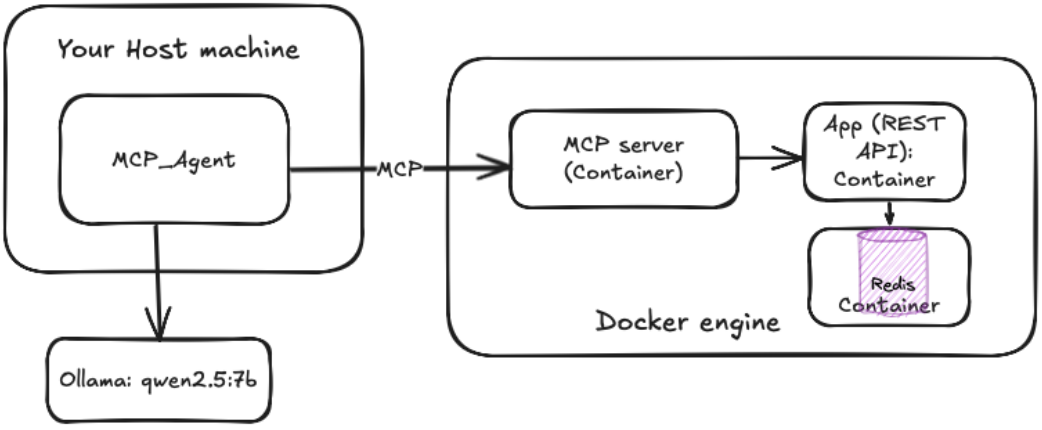


Figure 8.11

Here's how the components work together:

- **REST API backend (FastAPI + Redis)**:

  We start with a RESTful inventory service built with FastAPI. This backend exposes endpoints for managing warehouse orders — including listing, updating, and deleting order data. It uses Redis for persistence, ensuring lightweight and high-speed data operations.
- **MCP Server**:

  The REST API is wrapped inside an MCP-compliant server, exposing machine-readable metadata for LLM agents. This server describes available functions (tools) and their expected parameters, making it possible for an agent to discover and invoke these tools programmatically. It's modular, supports `OpenAPI/JSON` Schema, and runs seamlessly alongside your REST API.
- **Local LLM via Ollama**:

  A local language model like `qwen2.5:14b` is deployed via `Ollama`, offering both privacy and performance. The LLM handles user prompts, performs tool

selection, and generates actionable calls to the MCP Server. Since the entire stack is local, it's cost-effective and secure for internal deployments.

- **MCP Agent (lastmile-ai/mcp-agent)**:

  The MCP Agent is the orchestrator. It:

  1. Loads your MCP Server configuration.
  2. Lists all available tools via MCP metadata.
  3. Attaches the local LLM.
  4. Accepts natural language input from users.
  5. Converts the input into structured tool calls (e.g., hitting /list-orders).
  6. Handles multi-turn dialogue (e.g., summarizing output, generating tweets).

  It acts as a bridge between natural language understanding and structured API interaction — without writing prompt chains or managing HTTP calls manually.

- **Execution loop**:

  Once the user sends a prompt:

  1. The MCP Agent consults the MCP Server for available actions.
  2. The local LLM (via Ollama) generates a plan to call the API tools.
  3. The tools are invoked, results are collected.

- **Deployment with Docker compose**:

The full project (REST API, MCP server, Redis, Ollama, and MCP Agent) is dockerized using Docker Compose. This enables easy setup and consistent development across environments — whether you're running it locally, on a VM, or inside Kubernetes.

Once up, you can chat with your MCP Agent and ask questions like:

- `List all warehouse orders`
- `Update the status of order ID 1234` or `Change the order with ID 1234 status to delivered`

The agent will route these through the FastAPI app, using the MCP Server to understand the schema and behavior of available tools.

In a nutshell, we are going to do the following steps:

1. Develop an application that exposes a REST API for warehouse order automation.
2. Store order-related entities using Redis as the database.
3. Integrate and expose the application as an `MCP-compliant server`.
4. Validate MCP endpoints using tools like the `MCP Inspector`.
5. (Optional) Connect the MCP server with agent-based clients such as `Claude` or `Cursor`.
6. Use Ollama to perform local LLM inference.
7. Develop a `MCP Agent` to connect to the MCP server and orchestrate tasks using a local LLM backend.

Let's recap the benefits of using MCP server to use RESTfull api services.

Benefits of Integrating MCP Server with REST API

Integrating an existing REST API with an MCP server offers several advantages, particularly in modern AI-driven and microservice environments:

- **Discoverability**: MCP enables external tools and agents (including LLMs) to discover your API endpoints and understand their structure without prior knowledge.
- **Self-documentation**: MCP metadata automatically documents services, making them easier to onboard, share, and integrate with other systems.
- **Interoperability**: MCP provides a common interface for orchestrating diverse services, especially in multi-agent or local LLM-based environments.
- **No vendor lock**: Because MCP uses standardized metadata, your API can work across multiple platforms and AI agents without modification.
- **Dynamic invocation**: LLMs and automation tools can use MCP metadata to invoke your API intelligently and adaptively.

Let's explore a real world use case.

Use Case: AI-Driven warehouse automation

Imagine a local agent powered by a private LLM running on the edge or in an internal environment. The agent can use the MCP metadata of a REST API (like our warehouse order system which we will develop in this post) to:

- Query current orders.
- Auto-update order statuses based on email parsing or inventory checks.
- Generate summaries of pending shipments.
- Make decisions (e.g., cancel or expedite) using LLM logic.

This approach forms the backbone of smart, AI-enhanced applications without exposing your APIs or data to cloud-based models.

Why not integrate MCP server directly with the Database?

Although it might seem simpler to hook MCP directly to the database layer, this is discouraged for a few reasons:

- **Lack of business logic**: Databases don't encapsulate domain logic (e.g., validation, access control, formatting), which REST APIs often do.
- **Security risks**: Exposing a database directly increases the attack surface and can lead to data leaks or corruption.
- **Poor interoperability**: LLMs and external agents typically reason about actions, not raw data tables. REST APIs offer semantics that align better with AI workflows.
- **No versioning or abstraction**: REST APIs provide a stable interface, whereas raw DB schemas change more frequently and are harder to maintain in distributed settings.

REST APIs serve as the ideal bridge between your data and intelligent clients like LLMs. MCP adds a metadata layer that makes this bridge discoverable and usable by autonomous agents.

Now, let's get started with the example.

Step 1. Running your Redis instance locally on docker.

```
docker run --name redis -p 6379:6379 -d redis
```



Info

Note that by the end, we will have a Docker Compose file that deploys both Redis and the MCP server on the Docker engine. For now, we start the Redis server separately.

Step 2. Activate your Conda environment.

```
conda activate "YOUR Environment"
```

Step 3. Create a new directory in your local file system named `fastapi`

and clone the files from the Github repositories.

There should be two files:

1. requirements.txt
2. main.py

Step 4. Install the required Python packages.

```
pip install fastapi uvicorn redis fastapi-mcp
```

Alternatively, you can run the `requirements.txt` file to install all the necessary libraries.

```
pip install -r requirements.txt
```

Step 5. Create a new Python file named `main.py` or edit the file you cloned on `step 3`.

The `main.py` provides a simple warehouse order management system using Redis as the database.

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from typing import List
import redis

from fastapi_mcp import FastApiMCP
# Initialize FastAPI app
app = FastAPI(title="Warehouse Order System")
# Connect to Redis (localhost, port 6379, database 1)
redis_client = redis.StrictRedis(host="127.0.0.1", port=6379, db=1, dec\
ode_responses=True)
# Pydantic model for a complete order
class Order(BaseModel):
    order_id: int
    customer_name: str
    items: List[str]
    status: str  # Example statuses: 'pending', 'shipped', 'cancelled'
# Model used for creating new orders
class OrderCreate(BaseModel):
    customer_name: str
    items: List[str]
# Model used for updating existing orders
class OrderUpdate(BaseModel):
    customer_name: str | None = None
    items: List[str] | None = None
    status: str | None = None
# Endpoint to create a new order
@app.post("/orders")
def create_order(order: OrderCreate) -> dict:
    # Generate a new unique order ID
    order_id = redis_client.incr("order_id")
    # Prepare order data to store in Redis
    order_data = {
        "order_id": order_id,
        "customer_name": order.customer_name,
        "items": ",".join(order.items),  # Convert list to comma-separa\
```

```
ted string
        "status": "pending"  # Default status when order is created
    }
    # Store order hash in Redis
    redis_client.hset(f"order:{order_id}", mapping=order_data)
    # Track all order IDs in a Redis set
    redis_client.sadd("orders", order_id)
    return {"order_id": order_id, "message": "Order created successfull\
y."}
# Endpoint to get a specific order by ID
@app.get("/orders/{order_id}")
def get_order(order_id: int) -> dict:
    # Check if order exists
    if not redis_client.exists(f"order:{order_id}"):
        raise HTTPException(status_code=404, detail="Order not found.")
    # Retrieve order data from Redis
    data = redis_client.hgetall(f"order:{order_id}")
    data["items"] = data["items"].split(",")  # Convert string back to \
list
    return {"order": data}
# Endpoint to list all orders
@app.get("/orders")
def list_orders() -> dict:
    # Get all order IDs from the Redis set
    order_ids = redis_client.smembers("orders")
    orders = []
    for oid in order_ids:
        data = redis_client.hgetall(f"order:{oid}")
        if data:
            data["items"] = data["items"].split(",")  # Convert string \
to list
            orders.append(data)
    return {"orders": orders}
# Endpoint to update an existing order
@app.put("/orders/{order_id}")
def update_order(order_id: int, update: OrderUpdate) -> dict:
```

```
    # Check if the order exists
    if not redis_client.exists(f"order:{order_id}"):
        raise HTTPException(status_code=404, detail="Order not found.")
    # Update customer name if provided
    if update.customer_name:
        redis_client.hset(f"order:{order_id}", "customer_name", update.\
customer_name)
    # Update items list if provided
    if update.items:
        redis_client.hset(f"order:{order_id}", "items", ",".join(update\
.items))
    # Update status if provided
    if update.status:
        redis_client.hset(f"order:{order_id}", "status", update.status)
    return {"message": "Order updated successfully."}
# Endpoint to delete an order by ID
@app.delete("/orders/{order_id}")
def delete_order(order_id: int) -> dict:
    # Check if the order exists
    if not redis_client.exists(f"order:{order_id}"):
        raise HTTPException(status_code=404, detail="Order not found.")
    # Remove order data and ID reference
    redis_client.delete(f"order:{order_id}")
    redis_client.srem("orders", order_id)
    return {"message": "Order deleted successfully."}
```

The code enables basic *CRUD* operations on warehouse orders with Redis used as a fast, in-memory store for persistence. Here's a breakdown of the codebase:

- Setup and Initialization:
  - `FastAPI()` creates the main app instance.
  - `redis.StrictRedis()` connects to the local Redis server.
- Data Models:

  - `OrderCreate`: Used when creating a new order (excludes order_id and status).
  - `OrderUpdate`: Used for partial updates; all fields are optional.

- API Endpoints:

  - `POST /orders`: Creates a new order.
  - `GET /orders/{order_id}`: Retrieves a single order by ID.
  - `GET /orders`: Lists all orders.
  - `PUT /orders/{order_id}`: Updates an existing order.
  - `DELETE /orders/{order_id}`: Deletes an order by ID.

- Redis Schema:

  - `order:{id}`: Hash storing order details.
  - `orders`: Set of all active order IDs.
  - `order_id`: Auto-incrementing counter for generating new order IDs.

Step 6. MCP Integration

Attach the MCP server with one line of code at the end of the `main.py` file:

```
mcp = FastApiMCP(
    app,
    name="Warehouse MCP",
    description="MCP server for managing warehouse orders."
)
mcp.mount(app)
```

The above code will create a MCP server named `Warehouse MCP` and mount on default port.

Step 7. Run the App

```
python3 -m uvicorn main:app --reload
```

After a while, if everything goes fine, you should get a similar output in your console:

```
INFO:     Will watch for changes in these directories: ['/Users/shamim/\
../fastapi']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to qui\
t)
INFO:     Started reloader process [42288] using WatchFiles
INFO:     Started server process [42290]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Now, you have the following resources available in your system:

- `http://127.0.0.1:8000/docs` — Swagger UI
- `http://127.0.0.1:8000/mcp` — MCP metadata
- `http://127.0.0.1:8000/redoc` — API documentation

Step 8. Add some items into the system through Swagger UI `http://127.0.0.1:8000/docs` as shown below.

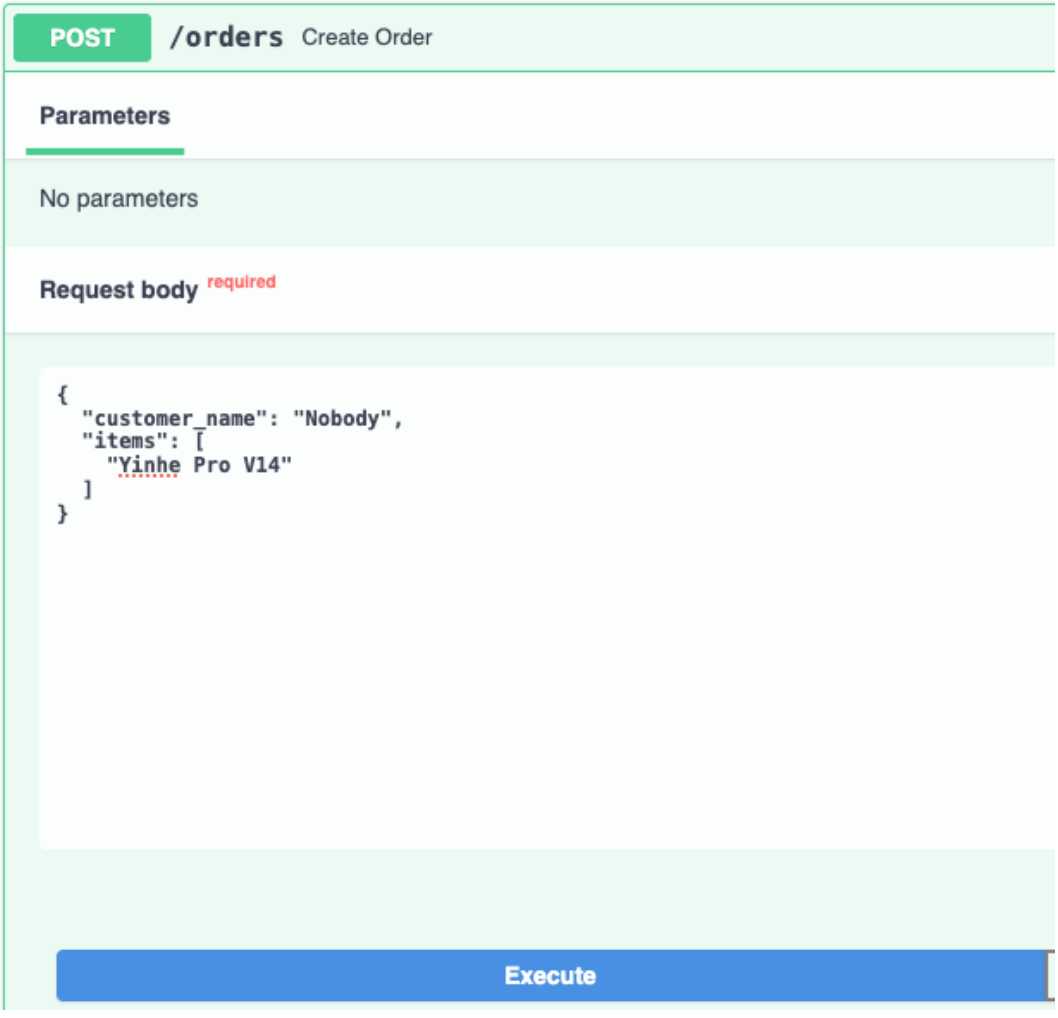


Figure 8.12

Now, invoke the `/orders` List orders REST API to get the list of all the items you entered into the database.

```
curl -X 'GET' \
  'http://127.0.0.1:8000/orders' \
  -H 'accept: application/json'
```

It should return something like as shown below (I have entered 4 items):


```
{
  "orders": [
    {
      "order_id": "3",
      "customer_name": "Timur",
      "items": [
        "Butterfly Timoball"
      ],
      "status": "delivered"
    },
    {
      "order_id": "2",
      "customer_name": "Mishel",
      "items": [
        "Yinhe Pro V1"
      ],
      "status": "pending"
    },
    {
      "order_id": "4",
      "customer_name": "Nobody",
      "items": [
        "Yinhe Pro V14"
      ],
      "status": "pending"
    },
    {
      "order_id": "1",
      "customer_name": "Shamim",
      "items": [
        "Butterfly Viscari"
      ],
      "status": "pending"
    }
```

```
  ]
}
```

The application works as we expected. Now let's check it's MCP protocol.

Step 9. Use `MCP Inspector` to test the MCP server.

The MCP inspector is a developer tool for testing and debugging MCP servers. It's an opensource project which is very helpful to test the functionalities of the MCP server before connecting to any MCP Client.

You don't need to clone of install any source code. Just Install `NodeJS` and run the following command:

```
npx @modelcontextprotocol/inspector
```

It will be available by the address `http://127.0.0.1:6274`. Note that, the MCP Inspector is not working well on MacOS `Safari` browser, use `Google chrome` or some other browser to use it.

So, select the following parameters:

- **Transport type**: SSE
- **URL**: `http://localhost:8000/mcp` (or your URL to the MCP Server)

and hit the button `Connect`. After successfully connected to the MCP server, click on the `List tools` button on page `tools` as shown below:

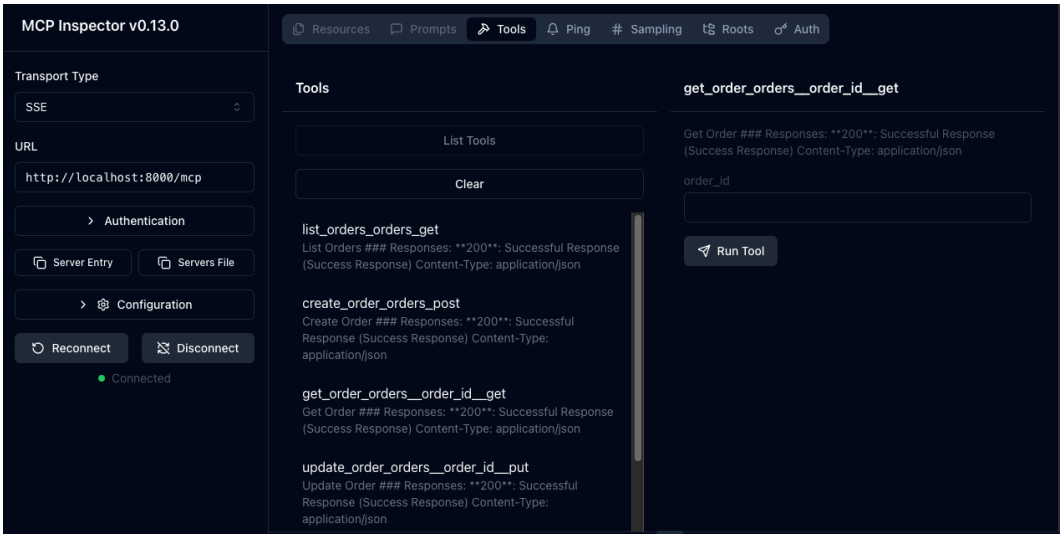


Figure 8.13

Choose any tool such as `list_orders_orders_get` and click `Run tool`. It will invoke the REST API endpoint list all orders `/orders` and return the result.

Play around the Inspector with other endpoints. You can also add other functionalities in the FastAPI application like Authentification and test through the MCP inspector.

For now, everything is up and running. At this point, you can use any MCP client—such as `Cursor`, `Claude Desktop`, or `MCP_CLI` (as introduced in the previous section) to interact with the MCP server.

Note, to keep this book concise, we'll skip detailed setup instructions for MCP clients.

- `Claude Desktop` users can simply add the following configuration and restart the application:

```
{
  "mcpServers": {
    "fastapi-mcp": {
      "command": "npx",
      "args": ["mcp-remote", "http://localhost:8000/mcp"]
    }
  }
}
```

- For `MCP_CLI` setup, please refer to the blog section on our book's website.

Now, let's dive into the final part of the project. Before moving forward, let's first explore the architecture of the MCP Agent.

Understanding MCP Agent architecture

The architecture of the MCP Agent framework is designed for modularity, composability, and extensibility, with the following key building blocks:

- `MCPApp`: Acts as the root application. It initializes configuration, logging, and runtime context for the entire agent lifecycle. It's the main entry point for bootstrapping and running the agent system.
- `Agent`: Represents an intelligent autonomous unit capable of using tools from MCP servers, attaching LLMs, and running workflows. An agent manages:
  - LLM integration (OpenAI, Claude, Ollama, etc.)
  - Tool discovery via registered MCP servers
  - Multi-turn conversations and memory (if enabled)
- `Tool`: A callable REST API operation defined and served by an MCP server. Tools are automatically discovered and registered using OpenAPI metadata provided by your MCP server.
- `Workflow`: Reusable logic steps that LLMs and tools execute as part of a reasoning chain. Workflows help compose advanced tasks, e.g., combining multiple API calls and summarizing results.

- `LLM Wrapper`: Standardizes how local or remote LLMs are integrated. For example:
  - OpenAIAugmentedLLM wraps GPT models.
  - You can implement your own wrapper to support Ollama or other providers.

Now, let's move on to the next step. Before getting started, make sure that:

1. Your FastAPI app MCP Server are running.
2. Redis running as your database.

Step 10. Install required dependencies (mcp agent)

```
pip install mcp-agent
```

Step 11. MCP Agent Implementation Walkthrough

Here's the full code for your MCP Agent (also available on GitHub book repository)

```
import asyncio
import os

from mcp_agent.app import MCPApp
from mcp_agent.agents.agent import Agent
from mcp_agent.workflows.llm.augmented_llm_openai import OpenAIAugmente\
dLLM
from mcp_agent.workflows.llm.augmented_llm import RequestParams
app = MCPApp(name="Inventory Agent")
async def example_usage():
    async with app.run() as mcp_agent_app:
        logger = mcp_agent_app.logger
        context = mcp_agent_app.context
        model="qwen2.5:14b"
        # This agent will use the invemtory api to communicate with the\
 FastAPI REST application
```

```
        inventory_agent = Agent(
            name="inventory",
            instruction="""You can use API to working with orders from \
the warehouse.
                Return the requested information when asked.""",
            server_names=["fastapi-mcp"], # MCP servers this Agent can \
use
        )
        async with inventory_agent:
            # Automatically initializes the MCP servers and adds their \
tools for LLM use
            tools = await inventory_agent.list_tools()
            logger.info(f"Tools available:", data=tools)
            # Attach an OpenAI LLM to the agent (defaults to GPT-4o)
            llm = await inventory_agent.attach_llm(OpenAIAugmentedLLM)
            # invoke fast-api MCP Server
            result = await llm.generate_str(
                message="Let me know all the orders from the warehouse",
                request_params=RequestParams(model=model),
            )
            logger.info(f"Available orders: {result}")
            # Multi-turn interactions by default
            result = await llm.generate_str(
                "Summarize the result in human language",
                request_params=RequestParams(model=model),
                )
            logger.info(f"Result: {result}")
if __name__ == "__main__":
    asyncio.run(example_usage())
```

Let's break down what this agent does:

- `Initializes` a custom MCP Agent with a defined purpose (inventory).
- `Registers` with the FastAPI-based MCP Server (fastapi-mcp) and lists its tools.
- `Attaches` to a local LLM (e.g., Qwen2.5).
- `Executes` LLM prompts that utilize API calls behind the scenes.

- `Follows` up with chained prompts (multi-turn interaction).

Step 12. MCP Agent Configuration

To bring the MCP Agent to life in our project, a `YAML` configuration file is used to define how the agent runs, which servers it communicates with, and how logs and LLMs are handled. Here's a breakdown of the configuration used in our example (`mcp-agent.config.yaml`).

```yaml
$schema: ../../../schema/mcp-agent.config.schema.json

execution_engine: asyncio
logger:
  type: console
  level: debug
  batch_size: 100
  flush_interval: 2
  max_queue_size: 2048
  http_endpoint:
  http_headers:
  http_timeout: 5
mcp:
  servers:
    fastapi-mcp:
      command: "npx"
      args: ["mcp-remote", "http://localhost:8000/mcp"]
openai:
  base_url: "http://localhost:11434/v1"
  api_key: ollama
```

Here is the key points of the config file:

1. The entire agent runtime is built on Python's asyncio to support non-blocking I/O operations — ideal for fast and scalable tool invocations over HTTP.
2. This config sets up a console-based logger in debug mode. It supports batching and async flushing, which helps reduce latency and improve visibility during agent runs. Here, you can change it to level INFO.

3. The section servers This section registers the MCP Server (running at `http://localhost:8000/mcp`) under the name fastapi-mcp. It's invoked using the mcp-remote utility from npx, allowing dynamic tool registration via OpenAPI spec discovery. You can add as much mcp server you want.
4. The LLM backend is configured to use Ollama (running on `localhost:11434`) to simulate an `OpenAI-compatible API`. This allows us to swap in powerful local models like `Qwen 2.5` without relying on external cloud-based APIs.

Step 13. Run your Ollama inference.

```
ollama run qwen2.5:7b
```



Warning

Here, carefully choose your LLM which supports tools functions, otherwise you will get error like `Error during conversation processing: Ollama API Error: registry.ollama.ai/library/gemma3:12b does not support tools`.

Step 14. Run the Agent.

Execute the following command into the terminal.

```
uv run main.py
```

You should get a lot of debug/info informations into the console as shown below:

```
{
  "data": {
    "progress_action": "Running",
    "target": "Inventory Agent",
    "agent_name": "mcp_application_loop",
    "session_id": "c5dd5517-79e7-4e3c-a6d6-f0398f9e7c0b"
  }
}
[INFO] 2025-06-03T17:08:12 mcp_agent.mcp.mcp_connection_manager -
fastapi-mcp: Up and running with a persistent connection!
[700] Using automatically selected callback port: 16442
[700] [700] Connecting to remote server: http://localhost:8000/mcp
....
[700] [Remote□Local] 0
[700] [Local□Remote] notifications/initialized
[700] [Local□Remote] tools/list
[700] [Remote□Local] 1
[INFO] 2025-06-03T17:08:13 mcp_agent.Inventory Agent - Tools available:
{
  "data": {
    "meta": null,
    "nextCursor": null,
    "tools": [
      {
        "name": "fastapi-mcp_list_orders_orders_get",
        "description": "List Orders\n\n### Responses:\n\n**200**:
Successful Response (Success Response)\nContent-Type: application/json",
        "inputSchema": {
          "type": "object",
          "properties": {},
          "title": "list_orders_orders_getArguments"
        },
        "annotations": null
      },

[1455] [Local□Remote] tools/call
```

```
[1455] [Remote□Local] 2
[INFO] 2025-06-03T17:14:35 mcp_agent.Inventory Agent - Available orders\
:
.didReceiveMemoryWarning
```

And the end answer from the MCP Agent:

```
Here are the current orders in the warehouse:
1. Order ID: 1, Customer Name: Shamim, Items: Yinhe pro v1, Status: pen\
ding
2. Order ID: 2, Customer Name: Mishel, Items: Butterfly Viscary, Status\
:
pending
[INFO] 2025-06-03T17:14:38 mcp_agent.Inventory Agent - Result: Currentl\
y,
there are two orders pending in the warehouse:
- **Order 1**: Ordered by Shamim for "Yinhe pro v1".
- **Order 2**: Ordered by Mishel for "Butterfly Viscary".
```

That what we expected. At these moments, you can switch to any LLM through the config file.

Step 15. Deploy into Docker.

As promised at the beginning, we've provided both the `Dockerfile` and `docker-compose.yml` files in the book's GitHub repository. These files allow you to deploy the MCP server and Redis database using the Docker engine with minimal effort.

To launch the entire setup with a single command, run:

```
docker-compose up --build
```

You should see logs from both Redis and FastAPI app in the terminal.

Once running, as usual you can access the app at:

- `http://localhost:8000`

- `http://localhost:8000/docs` # Swagger UI
- `http://localhost:8000/mcp` #MCP

It's been a 15-step journey, so let's take a moment to recap what we've accomplished so far:

1. REST API that manages inventory orders using Redis.
2. MCP-compliant FastAPI server that exposes metadata.
3. LLM-powered MCP agents that can discover and invoke those APIs with natural prompts.
4. Real-world example of the MCP Agent interacting with your system, all in Python.

This pattern is a solid starting point for internal tools, enterprise assistants, or advanced automation in your AI infrastructure. You've now seen MCP in action — modular, secure, and LLM-friendly.

Potential Improvements to the Pipeline

While this end-to-end pipeline successfully demonstrates how to invoke REST APIs using local LLMs through the MCP Server and Agent, it's far from the real word. There are several areas where we can enhance scalability, reliability, and the overall developer experience:

- **Better Tool Descriptions & Metadata**. Currently, tool descriptions exposed by the MCP Server may be minimal or overly generic. Enhancing these with structured metadata and usage examples can help LLMs reason more effectively about when and how to call a tool.
- **Persistent Agent Memory**. The current agent is stateless between runs. Integrating a memory backend (e.g. Redis or SQLite) could allow agents to recall past interactions or maintain task progress over time, making them more context-aware.
- **Orchestration with MCP**. By leveraging the MCP Agent's ability to discover and reason about tools from multiple MCP Servers, we can build centralized orchestration logic — either declarative (YAML-based plans) or procedural (Python-based workflows).

- **Security**. By default, MCP Servers (such as those built with fastapi-mcp) expose their metadata and tools without any access control. While this is great for local testing and prototyping, it becomes risky in shared or production environments. However, you can leverage FastAPI's powerful dependency injection system to add authentication via API Key Headers or by using OAuth2/JWT.

Google's Agent-to-Agent Protocol: enabling cross-agent communication

Google's Agent-to-Agent (A2A) protocol is an open specification that allows AI agents to:

- Expose capabilities (like "get weather" or "search the web"),
- Advertise themselves on a shared network,
- and receive and handle tasks using standard HTTP-based JSON APIs.

Think of it like giving each agent its own little *resume*, *email address*, and *inbox* — so that other agents can contact it and ask for help.

Let's first understand in plain language what the Google Agent-to-Agent (A2A) protocol is all about. Think of it as a set of rules that let AI agents talk to each other smoothly over the web.



Info

Refer to the official protocol documentation for in-depth technical details.

Here are the main building blocks you need to know:

- **Agent Card**. Every agent has a public *business card* — a small file (usually found at `/.well-known/agent.json`) that tells the world:

  - What the agent can do (its skills)?
  - Where to find it (the URL)?
  - How to talk to it (authentication and more).

Clients read this card to discover and connect to agents.

- **A2A Server**. This is the agent that listens for requests. It exposes an HTTP API following the A2A protocol. You can think of it like a smart bot that's ready to accept tasks and do something useful — like fetching weather, searching the web, or generating text.
- **A2A Client**. This is any app (or another agent) that sends tasks to the A2A server. It knows how to talk to the agent using the protocol, and it's the one that says: `Hey agent, I need you to do this for me.`
- **Task**. A task is the actual *job* or *command* you want the agent to perform. It could be anything — `What's the weather in Paris?` or `Summarize this article.` Each task has a *unique ID* and goes through stages like:

  - submitted → just sent
  - working → the agent is doing its job
  - input-required → the agent is waiting for more input
  - completed → all done
  - failed or canceled → something went wrong or it was stopped

- **Message**. Messages are the back-and-forth communication between the client and the agent during a task. Each message has a role:

  - user → from the client
  - agent → from the agent

Messages carry actual content (called `Parts`) — like text, images, or structured data — that help the agent understand what to do.

In summary, the A2A protocol just defines a clear way for two smart AI Agents to collaborate over HTTP — one asks for something, the other does it, and they keep track of their conversation along the way.

Pros and cons of the A2A Protocol

While the A2A Protocol provides several strengths that make it well-suited for building agent-based systems, it also comes with certain drawbacks.

Good parts of the A2A Protocol:

- *JSON-RPC schema*: Provides a standardized and lightweight communication format between agents.
- *Agent marketplace and discovery*: Enables agents to find and interact with one another dynamically.
- *Agent card concept*: Offers a structured way to define agent capabilities and metadata.
- *Built-in authentication*: Ensures secure interactions from the ground up.
- *Support for MCP (Model Context Protocol)*: Enhances compatibility with other agent-based systems and tooling.

Bad parts of the A2A Protocol:

- *Complex codebase*: Difficult to read and understand, especially for beginners.
- *Challenging testing process*: Writing tests for agent interactions can be non-trivial.
- *Multi-agent coordination issues*: Managing communication and logic across multiple agents is often error-prone.
- *Library conflicts*: Integration can be problematic due to clashes with various Python libraries.

A2A vs. MCP — not a battle, but a partnership

One more thing to clarify: *there's no competition between A2A and the MCP (Model Context Protocol).* They serve different but complementary purposes:

- MCP is great for low-level operations like accessing local files, querying databases, or running tools and functions on the same machine.

- A2A, on the other hand, shines in communication and collaboration between different AI agents, even if they live on separate servers.

In real-world AI applications, you often need both:

- Use MCP when an agent needs to use a tool.
- Use the A2A protocol when multiple agents need to collaborate as a team whether through *orchestration* or *choreography*.

In the next section, we'll walk through building a real-world example: a Travel Planner AI that uses multiple A2A agents to provide a personalized itinerary.

However, the official GitHub repository provides good examples to get started, but for *Python beginners, it's not so straightforward to dive into right away*. That's why, we focuses on a human-friendly, practical approach — showing you exactly what to write and how it works, one line at a time.

Therefore, I explored several third-party community implementations that help reduce boilerplate code and offer a much smoother entry point into working with A2A agents. These tools allow you to focus more on logic and behavior rather than wiring up the protocol manually.

On GitHub, I found two popular high-level Python libraries that stand out:

1. python-a2a — a simple abstraction layer that makes agent creation and communication intuitive.
2. a2a-server — another community package that wraps around the A2A protocol and integrates nicely with FastAPI and LangChain.

These libraries are great for getting your hands dirty quickly, especially if you're building practical use cases like agent orchestration or work-flow. And that's exactly what we'll use in the upcoming examples.

From these, `python-a2a` appears to be the more advanced option. It requires minimal external dependencies, making it easier to use in lightweight setups or small projects. With `python-a2a`, you can spin up agents and connect them with just a few lines of code — perfect for quick experimentation or educational use cases.

Let's Build: a travel planner AI Agent using A2A protocol

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.

Conclusion

This chapter has provided a comprehensive exploration of advanced tools and techniques for enhancing AI applications using the Model Context Protocol (MCP) framework and the Minion(s) protocol.

The introduction of Minion(s) added a powerful abstraction for enabling seamless communication between local LLMs and remote services, helping to reduce infrastructure costs while preserving performance and scalability.

We demonstrated how to integrate RESTful applications—such as a FastAPI-based inventory system backed by Redis—directly into an MCP Server. This shows how existing systems and data sources can be made accessible to LLMs via standardized protocols, transforming traditional APIs into intelligent interfaces driven by natural language.

Furthermore, we introduced the MCP Agent—a higher-level orchestration tool capable of discovering, invoking, and summarizing REST APIs using LLMs with multi-turn reasoning. By combining local models via Ollama, standardized MCP-compatible tools, and real-world data sources, developers can create robust and flexible AI-driven pipelines.

We also discussed how developers can interact with MCP Servers using various clients such as Claude Desktop, Cursor, or the Python-based mcp-cli. This modularity allows teams to prototype and scale AI integrations with minimal setup, leveraging Docker Compose for fast deployment.

In addition to MCP, we explored Google's Agent-to-Agent (A2A) protocol, which offers a complementary approach to agent communication and interoperability. A2A introduces a discovery layer, authentication, and an agent marketplace backed by JSON-RPC—allowing agents to collaborate either through orchestration or choreography. This protocol is especially valuable when multiple autonomous agents must dynamically coordinate their actions across distributed systems.

By leveraging tools like mcp-cli, MCP Agents, and integrating RESTful services, you can build internal tools, enterprise AI assistants, or intelligent automation pipelines—securely and efficiently.

As you continue your AI development journey, we encourage you to push further: build custom MCP servers, enhance orchestration with Minion(s), experiment with A2A-based agent teams, and deploy real-world applications. The future of AI lies in combining interoperable protocols like MCP, Minion, and A2A to create intelligent, modular, and collaborative systems that are both powerful and adaptable.

Final words

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/quickstartwithai.