ANONYMOUS AFFILIATIONS

- HANDS-ON TUTORIAL -

How to Build Impactful Applications using Large Language Models?

ANONYMOUS

Anonymous City, August 2025

Abstract

This tutorial offers a comprehensive guide on how to build impactful applications using Large Language Models (LLMs) in a multi-agent system setting, specifically tailored for high school students. With a strong emphasis on hands-on practice, the tutorial walks students through the complete development lifecycle—ranging from setting up a collaborative development environment, to designing a multi-agent system, constructing datasets, and finally deploying fully functional AI-driven applications. Divided into seven chapters, the guide introduces learners to both fundamental and advanced aspects of multi-agent architectures, showcasing how multiple intelligent agents can work together to build dynamic chatbots, educational assistants, AI games, and productivity tools. The tutorial not only nurtures technical proficiency but also encourages students to innovate with AI in meaningful and collaborative ways.

Contents

List of Figures

Chapter 1

Introduction

1.1 Goal and Motivation

Artificial Intelligence (AI) continues to transform how we solve problems, communicate, and automate real-world tasks. At the heart of these advancements are Large Language Models (LLMs), capable of generating human-like text and reasoning through complex scenarios. Recently, a new paradigm has emerged multi-agent systems where multiple LLM-powered agents collaborate, each with specific roles and responsibilities, to complete more sophisticated tasks than any single model could manage alone.

This book, titled ”How to Build Impactful Applications Using Multi-Agent”, is designed to introduce secondary education learners to this exciting field. This guide takes a practical, hands-on approach. We will skip the heavy theory and dive straight into building. Through step-by-step instructions and clear examples, students will immediately engage with the technology: setting up development tools, programming individual agents, building datasets, defining agent collaboration protocols, and finally deploying real-world applications.

Before we build our team of agents, let's zoom in on the star player: the Large Language Model, or LLM. What is it, and how does it work?

Under the Hood: What is a Large Language Model (LLM)?

A Large Language Model is the powerful "brain" that powers applications like ChatGPT, Google Gemini, and the very agents we will build in this book. Let's break down the name:

- **Large:** This is no exaggeration. An LLM is "large" because it has been trained on a massive, almost unimaginable, amount of text and code from the internet, books, articles, and more. Imagine reading a significant portion of the entire public web—that's the scale we're talking about. This vast library of information allows it to understand grammar, context, facts, different writing styles, and the subtle nuances of human communication.
- **Language:** Its entire world is built on words and sentences. It's an expert in understanding the patterns, structure, and relationships in language, whether it's English, Python code, or Spanish.
- **Model:** In AI, a "model" is a complex system (specifically, a massive neural network) that has been trained to make predictions. An LLM is a model of language itself; it has learned the probability of which words should follow other words to form coherent and meaningful sentences.

At its core, the way an LLM works is like a superpowered version of the autocomplete on your phone. When you type "The weather today is...", your phone might suggest "sunny" or "cloudy." An LLM does the same thing but on a superhuman scale. When you give it a prompt like, "Write a short story about a detective who is also a robot," it predicts the most likely first word, then the second based on the first, and so on. Because its knowledge is so vast, its predictions are incredibly sophisticated, allowing it to maintain a plot, create characters, and write in a specific style, resulting in what feels like a truly original piece of text.

Meet the Major Players: A Look at Popular LLMs

The field of AI is moving fast, and several major LLMs are leading the way. You might have heard of some of them:

- **The GPT Series (from OpenAI):** This family of models, including GPT-3.5 and GPT-4, powers the famous ChatGPT. They are known for their strong conversational and creative writing abilities.
- **The Llama Series (from Meta):** These models are very popular with developers and researchers, partly because Meta has made them more openly available, encouraging a lot of community-driven innovation.
- **The Claude Series (from Anthropic):** These models are known for their

large context windows (meaning they can remember longer conversations) and a strong focus on AI safety and producing helpful, harmless responses.

- **Gemini (from Google):** This is Google's latest and most capable family of models. Gemini was built from the ground up to be "multimodal," meaning it can understand and work with not just text, but also images, audio, and video, making it incredibly versatile.

Now that you understand the powerful engine we'll be working with, we are ready to give these engines specific jobs and make them work together as a team. This team-based approach is precisely what allows us to elevate a basic chatbot into a truly powerful personal assistant—one that can perform complex, multi-step tasks like managing your schedule and booking appointments,answering user's query. In this book, you will be equipped with the knowledge and confidence to build these very systems, turning your ideas for intelligent, helpful applications into reality.

1.2 Hands-on Summarization

This "Hands-on" tutorial is structured to guide you from foundational concepts to a fully functional multi-agent application across seven chapters. Our journey is designed to build your skills progressively, ensuring a solid understanding at each step.

We begin with the essentials: setting up a professional development environment and installing the specific software tools needed to orchestrate multiple AIs. This foundation is crucial for the complex and exciting work to come. From there, we will dive into the architecture of multi-agent systems. You will learn to design individual agents, define their unique roles, and manage the workflow that allows them to collaborate effectively. We will then explore how to give your agents specialized knowledge and tools. The final chapters focus on deploying your multi-agent application to make it accessible to others, and we will reflect on the incredible new possibilities and ethical considerations these advanced systems present.

Chapter 2: Environment Setup This chapter focuses on preparing a development environment suitable for building and orchestrating multi-agent systems. It introduces the essential tools, libraries, and software needed to build an effective workspace

Chapter 3: A simple multi-agent system implementation. This chapter demonstrates how to build a multi-agent chatbot using a Router/Dispatcher pattern. A central Router Agent directs user queries to specialized agents (e.g., HR, IT), which use their own dedicated knowledge bases stored in Milvus.

The system uses Chainlit for the user interface, resulting in a modular, scalable, and highly accurate conversational AI.

Chapter 4: Dataset Construction. The effectiveness of any multi-agent system, particularly one with specialized agents, is fundamentally dependent on the quality and structure of its underlying knowledge base. This chapter provides a detailed guide to the entire process of dataset construction, from initial creation to final integration.

First, we will detail the methodology for Dataset Creation and Format, outlining the steps required to gather and structure domain-specific information into a consistent and usable format. Following that, we will explain how to Integrate the Dataset into the Multi-agent System. This critical step involves making the structured data accessible to the agents, effectively transforming static information into an operational knowledge source.

By the end of this chapter, the reader will understand how to build and deploy a custom dataset, a crucial step in empowering agents with specialized expertise.

Chapter 5: Deploying Multi-agent System. The system is built on a modular, three-part architecture using a central proxy for secure communication.

A User Interface, built with Chainlit, captures user input. This input is managed by a Proxy Server, which routes requests to the core Multi-Agent System. This backend consists of specialized agents that collaborate to process the request and generate a response. This decoupled design ensures the system is scalable and maintainable.

Chapter 6: Multi-agent Applications. This chapter introduces the practical, real-world applications of Multi-Agent Systems (MAS). We will explore how autonomous, collaborating agents are used to solve complex problems in key industries.

The chapter will focus on three typical applications: optimizing Supply Chain Management, enhancing Healthcare Management, and securing Financial Markets, demonstrating the power and versatility of the MAS paradigm.

Chapter 2

Environment Setup

2.1 Introduction

Programing Enviroment: A programming environment is a collection of tools used in software development. It combines hardware and software, enabling developers to build applications efficiently. This environment encompasses everything from text editors and compilers to debuggers and integrated development tools necessary for writing and testing code.

Integrated Development Environments: An integrated development environment (IDE) is a software suite that consolidates the basic tools required for software development into a single, cohesive user interface. IDEs are designed to streamline the coding process by integrating common development tools within one application, providing a more efficient workflow for developers. An IDE typically consists of the following components:

- **File System:** Organizes and manages the files and directories involved in software development.
- **Text Editor:** A specialized editor designed for writing and editing source code.
- **Linker:** Combines various pieces of code and data together to create a single executable program.
- **Compiler:** Translates source code written in a programming language into machine code that can be executed by a computer.

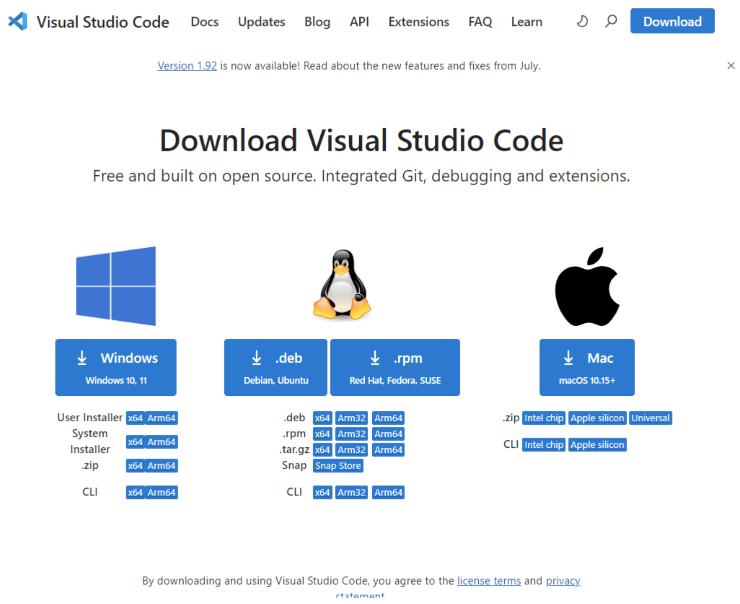
- **Integrated Tools:** Additional utilities that support the development process, such as debuggers, and version control

2.2 Visual Studio Code

Visual Studio Code is a lightweight but powerful source code editor that runs on your desktop and is available for Windows, macOS, and Linux. It comes with built-in support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages and runtimes (such as C++, C#, Java, Python, PHP, Go, .NET). Begin your journey with Visual Studio Code with these introductory videos (Visual Studio Code, 2021).

Setting up Visual Studio Code involves several steps:

1. Access the Official Website: Open any web browser, such as *Google Chrome*[1], *Microsoft Edge*[2], and navigate to the official *Visual Studio Code* installation website [3] (Figure 2.1).



Version 1.92 is now available! Read about the new features and fixes from July.

Download Visual Studio Code

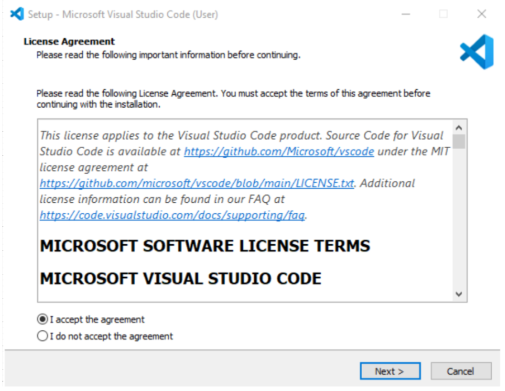
Free and built on open source. Integrated Git, debugging and extensions.

By downloading and using Visual Studio Code, you agree to the license terms and privacy statement

Figure 2.1: Visual Studio Code download page

2. Initiate Download: On the installation page (Figure 2.1), click on the button `Download for Windows` button to start downloading Visual Studio Code.

[1] `https://www.google.com/intl/en_us/chrome`
[2] `https://www.microsoft.com/en-us/edge`
[3] `https://code.visualstudio.com/download`


Setup - Microsoft Visual Studio Code (User)

License Agreement
Please read the following important information before continuing.

Please read the following License Agreement. You must accept the terms of this agreement before continuing with the installation.

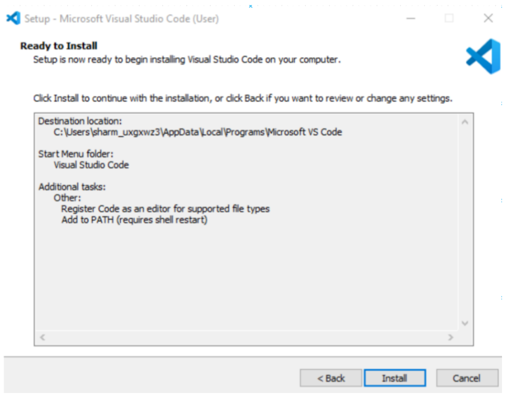
This license applies to the Visual Studio Code product. Source Code for Visual Studio Code is available at https://github.com/Microsoft/vscode under the MIT license agreement at https://github.com/microsoft/vscode/blob/main/LICENSE.txt. Additional license information can be found in our FAQ at https://code.visualstudio.com/docs/supporting/faq.

MICROSOFT SOFTWARE LICENSE TERMS

MICROSOFT VISUAL STUDIO CODE

[✓ I accept the agreement]
I do not accept the agreement

(a) Visual Studio Code License Agreement


Setup - Microsoft Visual Studio Code (User)

Ready to Install
Setup is now ready to begin installing Visual Studio Code on your computer.

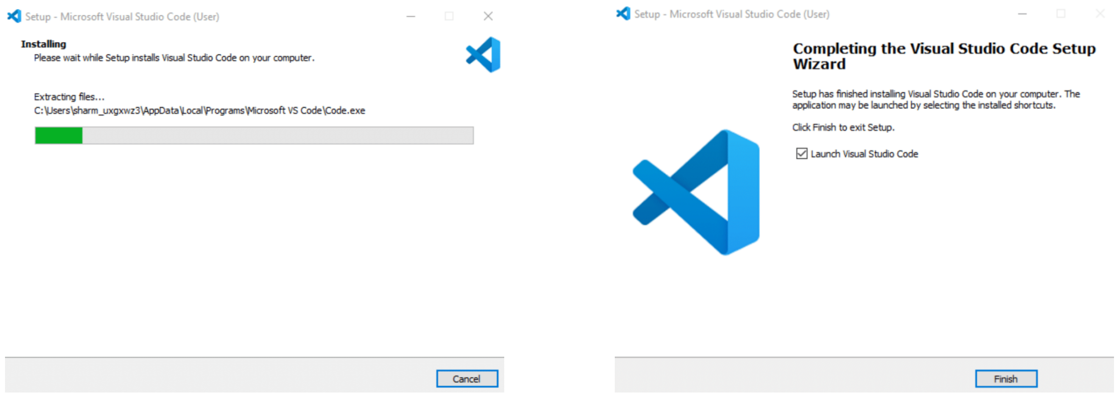
Click Install to continue with the installation, or click Back if you want to review or change any settings.

```
Destination location:
    C:\Users\sharm_uxgxwz3\AppData\Local\Programs\Microsoft VS Code

Start Menu folder:
    Visual Studio Code

Additional tasks:
    Other:
      Register Code as an editor for supported file types
      Add to PATH (requires shell restart)
```

(b) Allowing to process installation

Figure 2.2: Visual Studio Code Installation Process

3. Locate the Downloaded File: Once the download is complete, the Visual Studio Code icon will appear in your Downloads folder.
4. Run the Installer: Click on the installer icon to begin the installation process.
5. Accept the License Agreement: The installer will prompt you to accept the terms and conditions (Figure 2.2a). Please select `I accept the agreement` and click `Next`.
6. Specify Installation Directory: Choose the directory where you want to install Visual Studio Code. Browse to the desired location and click `Next`.
7. Commence Installation: The installer will prompt you to begin the installation (Figure 2.2b). Then you should click `Install` to proceed.
8. Installation Process: Allow time for the installation process to complete. The installation window will be shown as Figure 2.3a.
9. Launch Visual Studio Code: After installation, a screen will appear. Select the `Launch Visual Studio Code` option and click `Finish` (Figure 2.3b.
10. Complete Installation: Visual Studio Code will open successfully (Figure 2.4).


Setup - Microsoft Visual Studio Code (User)

Installing
Please wait while Setup installs Visual Studio Code on your computer.

Extracting files...
C:\Users\sharm_uxgxwz3\AppData\Local\Programs\Microsoft VS Code\Code.exe

Setup - Microsoft Visual Studio Code (User)

Completing the Visual Studio Code Setup Wizard

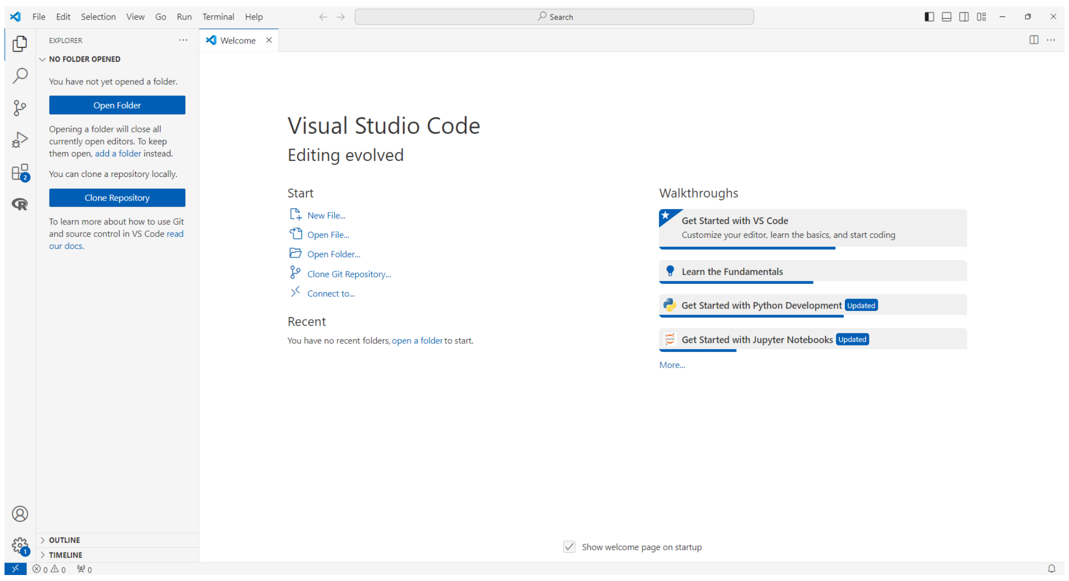
Setup has finished installing Visual Studio Code on your computer. The application may be launched by selecting the installed shortcuts.

Click Finish to exit Setup.

☑ Launch Visual Studio Code

(a) Installation Processing **(b)** Installation Completed

Figure 2.3: Visual Studio Code Installation Process



You have not yet opened a folder.

Opening a folder will close all currently open editors. To keep them open, add a folder instead.

You can clone a repository locally.

To learn more about how to use Git and source control in VS Code read our docs.

Customize your editor, learn the basics, and start coding

You have no recent folders, open a folder to start.

☑ Show welcome page on startup

Figure 2.4: Visual Studio Code Interface

2.3 Redis-server

For agents in a team to collaborate effectively, they must remember the ongoing conversation. Redis is an extremely fast, in-memory database that is perfect for this job. Think of it as the shared "short-term memory" for our agent system, allowing each agent to access the recent history of the conversation and stay in context. This ensures their collaborative dialogue is coherent and meaningful.

Since Redis is most commonly used on Linux, we will install it using the Windows Subsystem for Linux (WSL). This powerful feature lets you run a genuine

Linux environment directly on your Windows machine, giving you the best of both worlds.

1. **Install WSL and Redis Server.**

   The ideal installation process involves three main commands. First, open Windows PowerShell or Command Prompt **as an Administrator** and run the command to install WSL. After that, open your new Ubuntu terminal and run the next two commands to install and start the Redis server. The ‘sudo‘ command grants the necessary administrative permissions.

Standard Installation Commands

```
wsl --install
sudo apt install redis-server
sudo service redis-server start
```

Figure 2.5: The complete command sequence for a fresh installation of WSL and Redis.

2. **Troubleshooting: “Unable to locate package” Error.**

   Sometimes, the package manager’s list of available software is outdated. If you encounter the error `E: Unable to locate package redis-server`, it means ‘apt‘ doesn’t know where to find it.

   To fix this, first update the package list with ‘sudo apt update‘, and then retry the installation and start commands again.

Troubleshooting: If “Unable to locate package”

```
Step 1:  Update the package list :  sudo apt update
Step 2:  Retry the installation :  sudo apt install redis-server
Step 3:  Start the service :  sudo service redis-server start
```

Figure 2.6: Correct sequence of commands to fix the “Unable to locate” error.

3. **Verify that Redis is Running.**

   To confirm that your Redis server is active and listening for connections, use the Redis Command-Line Interface (‘redis-cli‘) to send a ‘ping‘ command.

   If the server responds with **PONG**, your Redis installation is successful and ready to provide memory for our AI agents!

Verify Server Status

```
redis-cli ping
```

Figure 2.7: Command to check the Redis server status.

4. **Install the Redis Library for Python.**
   With our Redis server running in the background, the final step is to give our Python code the ability to communicate with it. We do this by installing the official ‘redis-py‘ client library using ‘pip‘, Python’s standard package installer.

   Standard Python Library Installation

   ```
   pip install redis
   ```

   Pro Tip

   ```
   uv pip install redis
   ```

   The second command uses `uv`, an extremely fast, modern alternative to `pip`. If you have `uv` installed, it can make your package installations much quicker. For this tutorial, either command will work perfectly.

2.4 Chainlit

Once our AI agents are built, we need a way to interact with them. Chainlit is a fantastic Python library that lets you create a beautiful, functional chat interface with just a few lines of code. It saves us from the complexity of web development and lets us focus entirely on the AI logic.

2.4.1 Installation

You can install Chainlit using `pip`, Python's standard package installer. Open your terminal in VS Code and run:

```
pip install chainlit
```

2.4.2 A Simple Demo

Let's create a simple "echo" application to see how easy it is to get started. This app will simply take whatever we type and send it back to us with the word "Received:" in front.

1. **Create a Python File:** In your project folder, create a new file named `app.py`.
2. **Add the Code:** Open `app.py` and add the following code. The `@cl.on_message` decorator tells Chainlit to run this function every time a new message is sent from the user interface.

```python
import chainlit as cl

@cl.on_message
async def main(message: cl.Message):
    # This function is called every time a user sends a
      message.

    # We simply take the message content and send it back.
    await cl.Message(
        content=f"Received: {message.content}",
    ).send()
```

3. **Run the Application:** Now, go back to your terminal, make sure you are in the same directory as your `app.py` file, and run the following command:

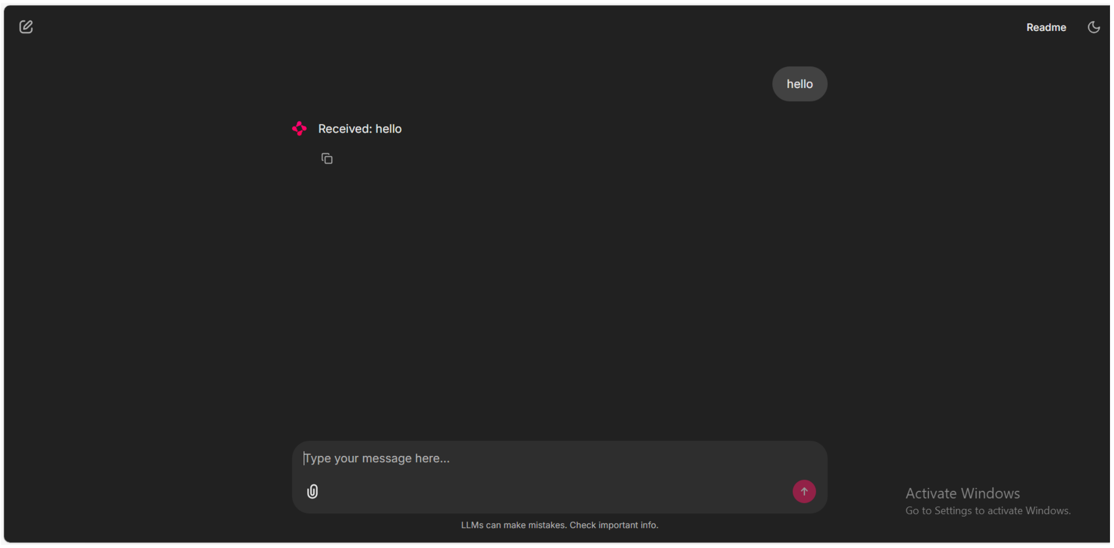
   Run Command

   ```
   chainlit run app.py -w
   ```

   The `-w` flag enables "watch mode", which automatically reloads the app whenever you save changes to your file, making development much faster.

2.4.3 Expected Output

After running the command, your web browser should automatically open to a new tab showing the Chainlit interface. If you type "hello" and press enter, you will see the exact output shown in Figure 3.1.



hello

Received: hello

Type your message here...

LLMs can make mistakes. Check important info.

Figure 2.8: The Chainlit interface showing our simple echo bot in action.

We will use this powerful tool to build the user interface for our multi-agent system.

2.5 The .env File

Our application needs to connect to external services like Google Gemini (for its brain) and Milvus (for its long-term memory). These services require "secrets"—API keys, URIs, and tokens—to grant access. Think of these like the passwords to your online accounts.

A very common mistake for new programmers is to write these secrets directly in their code. This is called hard-coding and it is extremely insecure.

Warning: Extremely Bad Practice!

```
# NEVER DO THIS IN A REAL PROJECT!
gemini_key = ``MySuperSecretGoogleApiKeyGoesHere"
```

If you share this code or upload it to a public place like GitHub, you have just given your passwords to the entire world. The correct and professional way to handle this is by using an **environment file**, or `.env`.

2.5.1 What is a .env File?

A `.env` file is a simple, plain text file where you store your secrets as key-value pairs. This file lives on your computer but is **never** shared or uploaded with the rest of your code. Your application then reads the secrets from this file at runtime.

2.5.2 How to Create and Use Your .env File

Let's set it up for our project.

1. **Install the Helper Library.**
   Python needs a small library to read `.env` files. Install it with pip:

   ```
   pip install python-dotenv
   ```

2. **Create the `.env` File.**
   In the root folder of your project create a new file and name it exactly `.env` (starting with a dot and with no other name).

3. **Add Your Secrets.**
   Open the `.env` file and add your keys. Each line should be in the format `KEY="VALUE"`.

   Example .env File Content

   ```
   GEMINI_API_KEY="YourSecretGoogleApiKeyGoesHere"
   MILVUS_URI="https://your-milvus-uri"
   MILVUS_TOKEN="YourMilvusPasswordOrToken"
   ```

   Note: Replace the placeholder values with your actual credentials.

4. **Load the Secrets in Your Python Code.**
   At the very top of your main Python script, add these two lines. `load_dotenv()` reads the `.env` file, and `os.getenv()` fetches the specific secret you need.

```python
from dotenv import load_dotenv
import os

# This line reads your .env file and makes the variables
   available
load_dotenv()

# Now you can safely access your secrets
gemini_key = os.getenv("GEMINI_API_KEY")
milvus_uri = os.getenv("MILVUS_URI")
milvus_token = os.getenv("MILVUS_TOKEN")

# Your application can now use these variables
```

2.5.3 The Benefits of Using a .env File

This approach might seem like a little extra work, but it is the professional standard for several critical reasons:

- **Security:** Your secrets are never exposed in your source code. You can share your code freely without worrying about leaking your credentials.
- **Collaboration:** Each developer on a team can have their own `.env` file with their own keys, but the main application code remains unchanged for everyone.
- **Flexibility:** You can easily have different `.env` files for different environments (e.g., one for testing on your computer, another for the final deployed application) without changing a single line of your program logic.

2.6 Pydantic-AI

Large Language Models can be creatively unpredictable. If you ask an LLM to "summarize a user's request", it might give you a paragraph, a bulleted list, or something else entirely. For a program—especially an AI agent that needs to pass information to another agent—this unpredictability is a problem. We need data in a consistent, machine-readable format.

This is where `pydantic-ai` comes in. It's a library built on top of Pydantic that forces an LLM to provide its output in a specific structure that we define. Think of it as giving the AI a strict form to fill out, rather than a blank piece of paper.

2.6.1 Installation

Like our other packages, `pydantic-ai` is a simple pip install. We also need to install the library for the specific LLM we want to use, in this case, Google Gemini.

```
pip install pydantic-ai google-generativeai
```

2.6.2 How It Works: Building a Simple Extractor Agent

Let's build an agent whose entire job is to read a sentence, identify the user's name and the task they want to do, and report back in a clean, bulleted list.

Code Breakdown

We'll build the code step-by-step to understand each part.

Step 1: Import the necessary classes.

We need to import the components to build our agent: the `Agent` itself, the specific `GeminiModel` we want to use, and the `GoogleGLAProvider` which handles the connection to Google's API.

```python
from pydantic_ai.models.gemini import GeminiModel
from pydantic_ai.providers.google_gla import GoogleGLAProvider
from pydantic_ai.agent import Agent
import os
from dotenv import load_dotenv
load_dotenv()
import asyncio
```

Step 2: Configure the LLM Model.

Here, we define which “brain” our agent will use. We specify the model name (e.g., `"gemini-2.0-flash"`) and provide the “provider,” which is our authenticated connection to the service using our API key. To ensure security, we use the `os` module to load environment variables instead of hard-coding sensitive information.

```
# Create the model object, specifying the model name and the
   API provider
model = GeminiModel(
   ``gemini-2.0-flash", # A fast and capable model
   provider=GoogleGLAProvider(api_key=os.getenv("GEMINI_API_KEY
      "))
)
```

Step 3: Create and Instruct the Agent.

This is the most important part. We create an instance of the `Agent` class. We pass it the `model` we just configured and, crucially, a `system_prompt`. This prompt acts as the agent’s permanent set of instructions, defining its personality and primary goal.

```
agent = Agent(
   model=model,
   system_prompt="You are a helpful assistant. Extract the user
      's name and the task they want to perform from the
      sentence. Present the result as a simple bulleted list
      with 'Name' and 'Task'."
)
```

Step 4: Run the Agent.

Because communicating with an AI can take time, we use an `async` function. The `async with agent:` block opens a communication channel, and `await agent.run(...)` sends our specific user request and waits for the agent to follow its instructions.

```
async def run():
   async with agent:
      result = await agent.run('Alice wants to book a flight to
          Paris')
```

```
    print(result.output)
```

Full Code and Expected Output

Here is the complete, runnable script. Remember to run it in an environment that supports `asyncio`.

```
from pydantic_ai.models.gemini import GeminiModel
from pydantic_ai.providers.google_gla import GoogleGLAProvider
from pydantic_ai.agent import Agent
import os
from dotenv import load_dotenv
load_dotenv()
import asyncio # Library to run async code

# Configure the LLM model
model = GeminiModel(
    ``gemini-1.5-flash-latest",
    provider=GoogleGLAProvider(api_key=os.getenv("GEMINI_API_KEY
        "))
)

# Create the Agent with its core instructions
agent = Agent(
    model=model,
    system_prompt="You are a helpful assistant. Extract the user
        's name and the task they want to perform from the
        sentence. Present the result as a simple bulleted list
        with 'Name' and 'Task'."
)

# Define the main async function to run the agent
async def main():
    async with agent:
        result = await agent.run('Alice wants to book a flight to
            Paris')
    print(result.output)
```

```
# Run the async function
if __name__ == ``__main__":
   asyncio.run(main())
```

When you run this script by command : **python your-file.py**, the agent will follow its system prompt precisely, providing the following clean output:

Agent Output

```
• Name:  Alice
• Task:  Book a flight to Paris
```

This ability to create specialized, instruction-driven agents is the fundamental building block for the powerful multi-agent systems we will build next.

2.7 MCP-server

2.7.1 Introduction to MCP: The Universal Adapter for AI

In the world of AI, a Large Language Model (LLM) is like a brilliant brain in a jar. It has immense knowledge, but to perform tasks in the real world, it needs hands, eyes, and ears. It needs to connect to other tools and data sources, like calendars, calculators, databases, or even the internet. This is where the Model-Context Protocol (MCP) comes in.

MCP is an open protocol that standardizes how applications provide this crucial context to LLMs. Think of MCP like a USB-C port for AI applications. Before USB-C, you needed a different cable for your phone, your camera, and your hard drive. It was a mess. USB-C created a single, universal standard that just works.

Similarly, without a standard like MCP, every time you want your LLM to use a new tool (like a calculator), you might have to write custom, one-off code. MCP provides that standardized, ”plug-and-play” way to connect AI models to a universe of different data sources and tools. In short, MCP enables you to build powerful agents and complex workflows on top of LLMs, connecting your models with the world.

2.7.2 The Core Concepts of MCP

At its heart, MCP works on a classic client-server architecture. To make this easy to understand, let's use a restaurant analogy:

- You, the AI application, are the **Diner** with a goal (e.g., "I want to calculate 5 times 10").
- The tool provider is the **Kitchen** (e.g., a calculator service that knows how to do math).
- The MCP components act as the **Waiter** who facilitates the interaction.

Participants

The MCP architecture has three key participants:

- **MCP Host:** This is the main AI application, the "Diner". It's the system that has a high-level goal.The Host coordinates and manages one or more MCP clients to get the job done.
- **MCP Client:** This is the "Waiter". The Host creates one Client for each Server it needs to talk to. The Client's job is to maintain that dedicated connection, take the Host's request, send it to the Server, and bring back the result.
- **MCP Server:** This is the "Kitchen". It's a program that provides specific tools or context. It listens for requests from a Client and provides the service. In our example, this will be a simple calculator that offers math functions.

  MCP Host (Agent) → *MCP Client* → *MCP Server (Tools)*

2.7.3 Example: Using MCP with PydanticAI

Theory is great, but let's see this in action. We will build a simple agent that can solve math problems. The agent itself won't know how to do math, but it will use MCP to connect to a "Calculator Server" that provides the necessary tools.

Installation

First, we need to install the necessary library. PydanticAI is a framework for building LLM-powered applications, and it has built-in support for acting as an MCP client.

You will need Python 3.10 or higher. You can install the library using pip:

```
- Install the lightweight version of PydanticAI with MCP support
pip install "pydantic-ai-slim[mcp]"
- Or, install the full version which also includes MCP
pip install pydantic-ai
```

Part 1: Create a Simple MCP Tool Server

This is our “Kitchen”. We will create a Python script that acts as an MCP server and offers four simple tools: add, multiply, square root, and power.

```
# mcp_calculator_server.py
from mcp.server.fastmcp import FastMCP
import math

# Initialize the MCP server application
app = FastMCP()

# Define a tool for addition
@app.tool()
def add(a: float, b: float) -> float:
    """Add two numbers"""
    return a + b

# Define a tool for multiplication
@app.tool()
def multiply(a: float, b: float) -> float:
    """Multiply two numbers"""
    return a * b

# Define a tool for square root
@app.tool()
def square_root(number: float) -> float:
    """Calculate square root"""
    if number < 0:
        raise ValueError("Cannot calculate square root of
            negative numbers")
    return math.sqrt(number)
```

```python
# Define a tool for powers
@app.tool()
def power(base: float, exponent: float) -> float:
   """Calculate power"""
   return base ** exponent

# Run the server when the script is executed
if __name__ == '__main__':
   print("Calculator MCP Server running on http://localhost
      :8000")
   app.run(transport='streamable-http', port=8000)
```

Part 2: Create the MCP Client (Our Agent)

Now we'll create our "Diner" and "Waiter". This script will set up a Agent (the Host), connect it to our running server (via a Client), and ask it to solve some math problems.

```python
# mcp_calculator_client.py
from pydantic_ai import Agent
from pydantic_ai.mcp import MCPServerStreamableHTTP
import asyncio

async def main():
   # 1. Connect to our running calculator server
   # This creates the MCP Client that talks to the server
   server = MCPServerStreamableHTTP('http://localhost:8000/mcp'
      )

   # 2. Create the Agent (the MCP Host) and give it the server
      connection
   agent = Agent('openai:gpt-4o', mcp_servers=[server])

   # 3. Start the server connection and ask questions
   async with agent.run_mcp_servers():
      questions = [
         "What is 15 + 27?",
            "Calculate the square root of 144",
            "What is 2 to the power of 8?",
            "Multiply 7 by 9"
        ]

        for question in questions:
            print(f"Q: {question}")
            result = await agent.run(question)
            print(f"A: {result.output}\n")

if __name__ == '__main__':
    # First run `python mcp_calculator_server.py` in one
        terminal.
    # Then run this script in another terminal.
    asyncio.run(main())
```

When you run the client, the agent analyzes each question, realizes it needs a specific tool, and uses the MCP connection to ask the server to execute the correct function. The server does the calculation and returns the answer, which the agent then presents to you. This elegant separation of concerns is the power of MCP.

2.8 API

2.8.1 What is an API? The Language of Software

At the heart of the modern internet is a concept that allows countless different applications to work together seamlessly: the API.

API stands for Application Programming Interface. That might sound technical, but the idea is surprisingly simple. An API is a set of rules and definitions that allows one software application to talk to another. It acts as an official contract that says, "If you send me a request in this specific format, I will give you back a response in a predictable format."

The Restaurant Analogy: The Best Way to Understand APIs

To understand APIs, let's use a simple analogy: ordering food at a restaurant.

Imagine you are sitting at a table. You want food, and you know the kitchen in the back is where the food is made. However, you can't just walk into the kitchen and start grabbing ingredients or shouting orders at the chefs. There is a system, a set of rules, and that system is the restaurant's API.

- **You (The Client):** You are the application that needs a service or data. You are hungry and want to order a burger.
- **The Kitchen (The Server):** This is the other application that has the data or functionality you need. The kitchen has the ingredients and chefs to prepare your burger.
- **The Menu (The API Documentation):** The menu is critical. It tells you exactly what you can order (the available functions), what options you have (e.g., "add cheese," "no onions"), and what you'll get in return. It defines the valid requests you can make.
- **The Waiter (The API):** The waiter is the crucial intermediary. You give your structured order to the waiter based on the menu. The waiter takes your request to the kitchen, and then brings the finished burger back to you. You don't need to know *how* the kitchen works, who the chefs are, or what their recipes are; you only need to know how to talk to the waiter. The waiter is the API.

2.8.2 A Real-World Example: The Weather API

Let's move from food to data. Imagine you are building a mobile app that needs to display the current weather in London. You are not going to launch your own weather satellite! Instead, you use a weather service's API.

1. The Request (Ordering from the Menu) Your application (the client) sends a structured request to the weather service's server. This request usually includes:

- **An Endpoint:** This is the specific URL for the data you want. It's like telling the waiter you're ordering from the "Main Courses" section of the menu.

  ```
  https://api.weatherservice.com/v1/current
  ```

- **Parameters:** These are the options for your order, refining your request.

```
?city=London&units=celsius
```

- **An API Key (Authentication):** This is a unique code that proves you have permission to make a request. It's like your library card or a reservation name, identifying you as a legitimate customer.

  ```
  &appid=YOUR_API_KEY
  ```

2. The Response (Getting Your Food) The weather server's kitchen processes your request and sends the data back, often in a format called JSON (JavaScript Object Notation). JSON is extremely popular because it is lightweight and easy for both humans to read and computers to parse.

A response from the weather API might look like this:

```
{
  "location": "London",
  "temperature": 15,
  "unit": "celsius",
  "condition": "Cloudy",
  "humidity": "72%"
}
```

Your application receives this neat, structured data and can then easily pull out the values it needs to display a user-friendly message like: "It's currently 15°C and Cloudy in London."

2.8.3 Why APIs are So Important

APIs are the essential glue of the modern digital world. They are the reason your phone's map app can show you restaurant locations, why you can log in to a new website using your Google account, and why online stores can process payments securely.

- **They save time and effort:** Developers don't have to build every single feature from scratch.
- **They allow for specialization:** Companies can focus on what they do best (like weather data, map information, or payment processing) and provide that service to the world through an API.

- **They enable connectivity:** They allow thousands of different systems, built by different people in different programming languages, to connect and share information seamlessly.

Understanding APIs is a fundamental skill for building almost any modern software application.

Chapter 3

A Simple Multi-agent System Implementation

3.1 Introduction

As LLMs become more powerful, developers are moving beyond single, monolithic agent architectures towards more sophisticated **multi-agent systems (MAS)**. A single agent tasked with handling every possible user request can become overly complex, difficult to maintain, and prone to errors as its knowledge and toolset grow. It may confuse context between different domains, leading to less accurate and less reliable responses.

The multi-agent paradigm addresses these challenges by applying a "divide and conquer" strategy. Instead of one agent doing everything, a multi-agent system is a society of collaborating agents, each with a specialized role and skillset. This approach offers several distinct advantages:

- **Modularity and Maintainability:** Each agent is a self-contained unit that can be developed, tested, and updated independently. Fixing a bug in the "HR Agent" does not risk breaking the "IT Agent".
- **Scalability:** Adding new capabilities to the system is as simple as creating a new specialist agent and registering it with the system. This is far cleaner than trying to merge new tools and knowledge into a single, massive agent.
- **Expertise and Accuracy:** Specialist agents, equipped with focused prompts, limited tools, and domain-specific knowledge bases, provide more precise and

contextually relevant answers. This reduces the chance of hallucination or providing information from the wrong domain.

This guide provides a practical, step-by-step walkthrough for implementing a foundational multi-agent pattern: the **Router/Dispatcher**. We will construct a system where a central "Router Agent" receives all user queries and delegates them to the appropriate "Specialist Agent". You will learn how to define agent roles, create specialized tools backed by distinct knowledge bases, use effective prompting techniques, and integrate the entire system into a functional application using Chainlit.

3.2 A Simple Multi-Agent System Implementation

A multi-agent system comprises multiple autonomous agents collaborating to solve a problem that is typically beyond the capabilities of a single agent. The Router/Dispatcher pattern is an effective architecture that we will implement here. It involves a central agent routing user requests to one of several specialized agents.

Core Architecture Components

This architecture consists of two primary types of agents:

- **Router Agent**: This is the main entry point for all user requests. Its sole responsibility is to analyze the user's intent and delegate the task to the most appropriate specialist. It does not answer questions itself.
- **Specialist Agents**: These agents are experts in a specific domain (e.g., Human Resources, IT Support, Finance). Each is equipped with its own dedicated knowledge base and a limited set of tools relevant to its domain.

3.2.1 The Agent

In the context of this project, an agent is defined as an instance of the `AgentClient` class. Each agent is uniquely configured by its language model, a guiding system prompt, and a specific set of tools it is authorized to use.

Tools: The Key to Specialization

Tools are functions that an agent can invoke to perform actions or retrieve information. In a multi-agent system, the primary method of creating "specialists" is by providing them with a highly specific and limited set of tools.

For example, an HR agent should only have access to tools that search HR-related documents, while an IT agent should only be able to query IT knowledge bases. The `create_faq_tool(collection_name: str)` factory function is perfectly suited for this purpose, as it allows us to create distinct retrieval tools, each pointing to a different Milvus collection.

Listing 3.1: Defining specialized tools for different agents.

```
# Tool for the HR Agent, searching the 'hr_policies' collection
# This tool can only access HR-related information.
hr_faq_tool = create_faq_tool(collection_name="hr_policies")

# Tool for the IT Agent, searching the 'it_kb' collection
# This tool is restricted to the IT knowledge base.
it_faq_tool = create_faq_tool(collection_name="it_kb")

# A non-retrieval tool for a potential Finance Agent
# This demonstrates that tools can be any function.
from some_calculator_library import calculate
calculator_tool = calculate
```

Memory: Enabling Contextual Conversations

The `MessageMemoryHandler` provides agents with short-term memory, allowing them to recall previous turns in a conversation. For a multi-agent system, a **shared memory** architecture is a simple yet powerful approach.

Advisory: The Importance of Shared Memory

By instantiating a single `MessageMemoryHandler` and passing it to all agents, the entire conversation history is accessible to both the router and any specialist that gets called. This shared context is crucial. For instance, if a user first asks "What's the policy on vacation?" (routed to HR) and then asks "How do I request it?", the specialist needs the history to understand what "it" refers to.

Retrieval: Accessing the Knowledge Base

Retrieval is the process of fetching relevant information from a knowledge base in response to a query. This is accomplished via tools, specifically the `faq_tool`. To implement effective retrieval in our multi-agent system, we must first partition our data into domain-specific collections.

Step 1: Index Data into Separate Collections. Each specialist agent requires its own isolated knowledge base. We use the `MilvusIndexer` to ingest data from different source files into distinct Milvus collections.

Listing 3.2: Indexing data into domain-specific Milvus collections.

```
# Index HR data into the 'hr_policies' collection
hr_indexer = MilvusIndexer(
   collection_name="hr_policies",
   faq_file="src/data/mock_data/HR_FAQ.xlsx"
)
hr_indexer.run()

# Index IT data into the 'it_kb' collection
it_indexer = MilvusIndexer(
   collection_name="it_kb",
   faq_file="src/data/mock_data/company4/company_policy_it.csv"
)
it_indexer.run()
```

Step 2: Create a Specialized Retrieval Tool for Each Agent. As demonstrated in the "Tools" section (Listing 3.1), we then create a unique `faq_tool` for each specialist, configuring it with the appropriate `collection_name`.

3.2.2 Effective Prompting Techniques

The system prompt is the most critical element in directing an agent's behavior, personality, and function.

1. **Role-Playing**: Assign a clear and specific persona to each agent. This narrows its focus and prevents it from answering off-topic questions.
   - **HR Agent Prompt**: '"You are an expert Human Resources assistant. Your knowledge is strictly limited to the company's HR policies. You must answer questions about benefits, leave, and company culture."'
   - **IT Agent Prompt**: '"You are a technical support specialist. You must only answer questions related to IT policies, software, and hardware issues."'
2. **Router/Dispatcher Prompt**: The router's prompt is paramount. It must explicitly define the available specialists and their capabilities, and instruct the router to delegate, not answer.

Warning: The Router's Prompt is Critical!

The prompt for your Router Agent is the most important piece of the puzzle. It must be explicitly instructed to delegate tasks and forbidden from answering questions itself. Vague instructions will cause the Router to fail its one job, which is to route the request to the correct specialist.

```
You are a master dispatcher. Your primary function is to
analyze the user's query and delegate it to the correct
specialist agent. You have the following specialists available:
1. `hr_agent`: Use this agent for any questions related to
Human Resources, company policies, vacation, benefits, or
payroll.
2. `it_agent`: Use this agent for any questions related to IT
support, software issues, hardware problems, or network access.
Based on the user's question, you must call the appropriate
agent's tool. Do not attempt to answer the question yourself.
Your only job is to route the request.
```

3.2.3 A Complete Implementation Example

The following script provides a complete, runnable example of the multi-agent system, integrated with the Chainlit UI. We'll build it step-by-step.

Step 1: Setup and Data Indexing First, we import the necessary libraries. Then, we prepare our specialized knowledge bases by creating two distinct Milvus collections, 'hr_domain' and 'it_domain', and indexing the relevant documents into each. This step ensures that each specialist agent will have its own isolated data source.

Listing 3.3: Imports and indexing into separate collections.

```
# multi_agent_workflow.py
import os
import chainlit as cl
from llm.base import AgentClient
from pydantic_ai.models.gemini import GeminiModel
from pydantic_ai.providers.google_gla import GoogleGLAProvider
from utils.basetools import create_faq_tool
from data.milvus.indexing import MilvusIndexer
from pydantic import BaseModel, Field

# This setup block should ideally be run once, not every time
    the chat starts.
hr_indexer = MilvusIndexer(collection_name="hr_domain",
    faq_file="src/data/mock_data/HR_FAQ.xlsx")
hr_indexer.run()

it_indexer = MilvusIndexer(collection_name="it_domain",
    faq_file="src/data/mock_data/company4/company_policy_it.csv"
    )
it_indexer.run()
```

Step 2: Model Initialization Next, we initialize the language model that all agents will share. We'll use Gemini via the 'GoogleGLAProvider'.

Listing 3.4: Initializing the shared language model.

```
# Initialize Model
```

```
provider = GoogleGLAProvider(api_key=os.getenv("GEMINI_API_KEY"
   ))
model = GeminiModel('gemini-2.0-flash', provider=provider)
```

Step 3: Define Specialist Agents Now we create our specialists. The ‘hr_agent‘ is given a tool that can only search the ‘hr_domain‘ collection, along with a system prompt that defines its role as an HR assistant. The ‘it_agent‘ is configured similarly, but with a tool and prompt focused entirely on IT.

Listing 3.5: Defining the HR and IT specialist agents.

```
# HR Specialist Agent
hr_faq_tool = create_faq_tool(collection_name="hr_domain")
hr_agent = AgentClient(
   model=model,
   system_prompt="You are an expert Human Resources assistant.
      Use the faq_tool to find information about HR policies,
      benefits, and leave.",
   tools=[hr_faq_tool]
).create_agent()

# IT Specialist Agent
it_faq_tool = create_faq_tool(collection_name="it_domain")
it_agent = AgentClient(
   model=model,
   system_prompt="You are a technical support specialist. Use
      the faq_tool to answer questions about IT policies,
      software, and hardware.",
   tools=[it_faq_tool]
).create_agent()
```

Step 4: Create Router's Tools for Delegation This is a critical step. The Router Agent does not answer questions; it delegates them. To do this, we create wrapper functions (‘ask_hr_agent‘, ‘ask_it_agent‘) that the Router will use as its ”tools.” When the Router calls one of these tools, it is actually invoking the corresponding specialist agent. We use a Pydantic model ‘AgentInput‘ to define a clear and strongly-typed input for these functions.

Listing 3.6: Creating the delegation functions for the Router Agent.

```
class AgentInput(BaseModel):
   query: str = Field(..., description="The user's question to
      be passed to the specialist agent.")

async def ask_hr_agent(input: AgentInput) -> str:
   """Delegates the user's query to the HR specialist agent.
      Use for HR-related questions."""
   response = await hr_agent.run(input.query)
   return str(response.output)

async def ask_it_agent(input: AgentInput) -> str:
   """Delegates the user's query to the IT specialist agent.
      Use for IT-related questions."""
   response = await it_agent.run(input.query)
   return str(response.output)
```

Step 5: Define the Router Agent With the delegation tools ready, we can now define the Router Agent itself. We provide it with a very specific system prompt instructing it to only delegate tasks by calling the appropriate tool. Its toolset consists of the 'ask_hr_agent' and 'ask_it_agent' functions we just created.

Listing 3.7: Defining the main Router Agent.

```
router_prompt = """
You are a master dispatcher. Your primary function is to
   analyze the user's query and delegate it to the correct
   specialist agent by calling the appropriate tool. You have
   these specialists:

1. `ask_hr_agent`: For questions about Human Resources, company
    policies, vacation, benefits, or payroll.
2. `ask_it_agent`: For questions about IT support, software,
   hardware, or network access.

Do not answer the question yourself. Your only job is to call
   the correct tool.
"""
```

```
router_agent = AgentClient(
   model=model,
   system_prompt=router_prompt,
   tools=[ask_hr_agent, ask_it_agent]
).create_agent()
```

Step 6: Set up the Chainlit UI Finally, we integrate our system with a Chainlit user interface. The '@cl.on_chat_start' function sends a welcome message. The '@cl.on_message' function is the main entry point for user interaction. Every message from the user is passed to the 'router_agent'. The router analyzes the message, calls the correct specialist tool (which in turn calls the specialist agent), and the final response is sent back to the user.

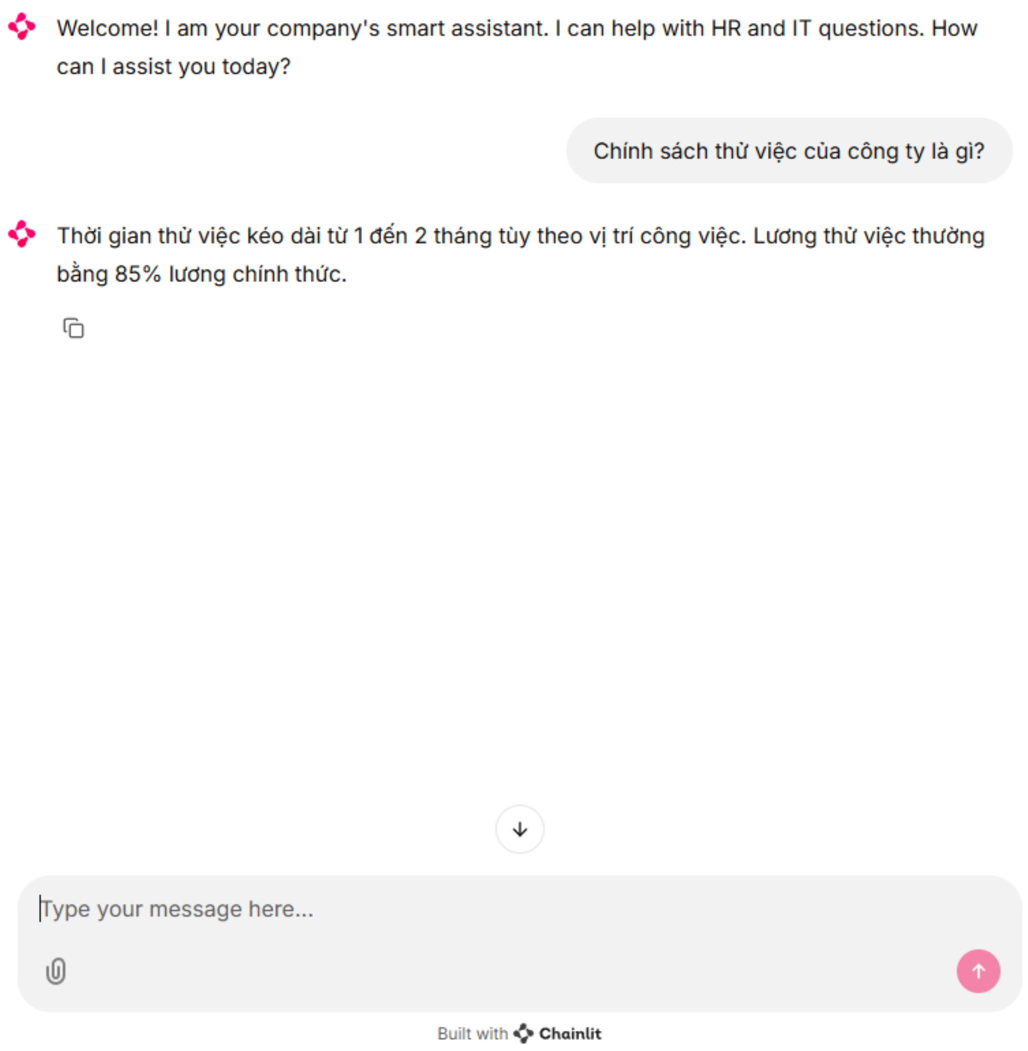
Listing 3.8: Integrating the multi-agent system with Chainlit.

```
@cl.on_chat_start
async def start():
   await cl.Message(content="Welcome! I am your company's smart
       assistant. I can help with HR and IT questions. How can
      I assist you today?").send()

@cl.on_message
async def main(message: cl.Message):
   # Every user message is first sent to the router agent.
   # The router decides which specialist to call, and the
      specialist generates the final answer.
   response = await router_agent.run(message.content)
   await cl.Message(content=str(response.output)).send()
```

Running the Application. To run this multi-agent system, save the code above as a Python file (e.g., `multi_agent_workflow.py`) and execute it with Chainlit from your terminal:

Run Application

```
chainlit run multi_agent_workflow.py
```


Welcome! I am your company's smart assistant. I can help with HR and IT questions. How can I assist you today?

Chính sách thử việc của công ty là gì?

Thời gian thử việc kéo dài từ 1 đến 2 tháng tùy theo vị trí công việc. Lương thử việc thường bằng 85% lương chính thức.

Figure 3.1: The Chainlit interface showing our multi-agent application in action.

Chapter 4

Dataset Construction

4.1 Introduction

The performance and reliability of an AI Agent are fundamentally dependent on the quality of its underlying knowledge base. For agents designed for information retrieval, such as those powering FAQ chatbots or support systems, a well-structured Question & Answer (Q&A) dataset is paramount. Such a dataset enables efficient **indexing** and accurate **retrieval**, allowing the agent to quickly find and deliver relevant answers to user queries.

This guide provides detailed, technical instructions on how to construct a Q&A dataset from raw documents, process it into a structured format, and prepare it for use in an AI Agent's knowledge base. The process outlined ensures the data is clean, accurate, and formatted correctly for maximum compatibility with modern data indexing systems.

4.2 Dataset Creation

4.2.1 Data Sources - Q&A Pairs Collection

The initial phase involves the identification and collection of raw data, which will be transformed into Q&A pairs.

Recommended method: Finding Documents on the Web

This method involves using public search engines to find authoritative documents, which will serve as the source material for the knowledge base.

Step 1. Search for documents online. Utilize a search engine like Google to find documents pertaining to the desired domain. It is critical to use precise keywords to locate high-quality sources. For instance, to build a knowledge base on mental health, a suggested keyword is: "Frequently asked questions about autism spectrum disorder".

Step 2. Select and copy content from a reliable source. From the search results, select a website from a trusted entity (e.g., a national health institute, a university, or a reputable hospital). Copy a section of the document relevant to the topic.

Step 3. Use a Large Language Model (LLM) to generate Q&A pairs. The copied text can be processed by an LLM (e.g., GPT-4, Gemini) to automatically generate Q&A pairs. Use a precise prompt to ensure the output is correctly formatted.

Recommended prompt for generating a Q&A table

```
From the information provided below, create a table of Q&A pairs
suitable for an FAQ chatbot.  The table must be in a two-column
format with the headers 'Question' and 'Answer'.
[Paste the copied content from the website here]
```

The resulting table format simplifies the process of transferring the data into a spreadsheet for final processing.

Advisory: Best Practices for Data Quality

To ensure the integrity and quality of the dataset, rigorous verification is essential.

- **Verify Content Accuracy:** Carefully review all generated questions and answers to correct any errors in meaning, spelling, or factual content. The AI may misinterpret nuanced information, and manual oversight is crucial.
- **Use Authoritative Sources:** When gathering material from the internet, exclusively use reliable sources. Official hospital websites, peer-reviewed scientific articles, and government publications are examples of trustworthy references. This practice prevents the inclusion of misinformation in your dataset.
- **Maintain Consistency:** Ensure that the tone and terminology are consistent throughout the dataset. This is particularly important if data is gathered from multiple sources.

4.2.2 Process of Creating a Dataset File

Once the Q&A pairs have been collected and verified, they must be formatted into a structured file.

Steps to Create the Dataset in a Spreadsheet

1. **Utilize a Spreadsheet Application.** Use Google Sheets or Microsoft Excel to create a new sheet. For optimal CSV compatibility, Google Sheets is often recommended.
2. **Create Column Headers.** In the first row, create two columns. The first column (A1) should be named `Question`, and the second column (B1) should be named `Answer`.
3. **Populate the Data.** Paste the collected Q&A pairs into the appropriate columns. Each row should represent a single, self-contained Q&A pair.

Exporting the File to CSV Format

Exporting data from standard spreadsheet formats (like `.xlsx`) can sometimes introduce encoding or formatting errors. The following procedure is recommended to ensure a clean CSV export.

Recommended Procedure (Using Google Sheets): If your data is currently in an Excel file, it is advisable to first import it into Google Sheets before exporting to CSV.

1. **Open Google Sheets.** Navigate to the Google Sheets web interface.
2. **Import the Excel File.** Go to `File` - `Open`. In the dialog box, select the `Upload` tab. Browse for and select the Excel file from your computer to transfer it to Google Sheets.
3. **Export as CSV.** Once the data is loaded and verified in Google Sheets, go to `File` - `Download` - `Comma-separated values (.csv)`.

This process will download a `.csv` file to your computer. This file is now properly formatted and ready to be used by an indexer or ingested into a platform for managing knowledge bases.

Chapter 5

Deploying Multi-agent System

To ensure scalability, modularity, and secure communication between agents, we deploy the multi-agent system using a proxy-based architecture. Each agent operates as an independent service and communicates through a central proxy, which acts as a message broker and access controller.

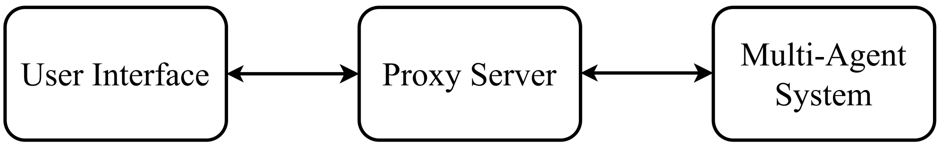


Figure 5.1: System Pipeline

The overall architecture of our system follows a modular and scalable pipeline composed of three main components: the **User Interface**, the **Proxy Server**, and the **Multi-Agent System**.

1. **User Interface (UI):** This module serves as the entry point for user interaction. It collects user input and displays the final response returned by the system. The UI is designed to be intuitive and accessible, ensuring seamless communication with the backend.
2. **Proxy Server:** Acting as a communication hub, the proxy server receives input requests from the UI, forwards them to the appropriate agent within the multi-agent system, and routes the responses back. It also manages session information and orchestrates task distribution to the agents based on the input type.
3. **Multi-Agent System:** This is the core decision-making and processing

layer of the system. It consists of specialized agents designed to handle specific subtasks (e.g., symptom analysis, care recommendation, vaccine suggestion). Each agent operates independently but collaboratively, contributing to a holistic and accurate system response.

This pipeline ensures flexibility, real-time responsiveness, and the potential for future expansion by adding more specialized agents.

5.1 User Interface

The User Interface (UI) serves as the primary interaction layer between the end user and the system. Its main role is to facilitate seamless communication by allowing users to input queries, receive responses, and interact with the system's functionalities in a user-friendly manner. Designed with accessibility and clarity in mind, the UI abstracts the complexity of the underlying multi-agent architecture and ensures that the user experience remains intuitive and efficient.

To implement the UI, we adopt the open-source framework **Chainlit**, which is tailored for building conversational applications powered by language models. Chainlit provides a lightweight web interface out of the box and supports real-time communication with the backend system via Python scripts.

5.2 Directory Structure and Modification of the User Interface

The project follows a clear and modular directory structure that separates configuration files, frontend assets, and the application logic. Below is an overview of the main components from the material repository[1]:

```
your_project/
|
+-- .chainlit/
| +-- config.toml # Main configuration file for Chainlit
| +-- translations/
| | +-- vi-VN.json # Vietnamese translation for UI elements
| +-- public/
```

[1]https://github.com/waanney/summerschool_workshop.git

```
| +-- your_logo.png # Custom logo for chat interface
| +-- favicon.png # Favicon displayed in the browser tab
| +-- stylesheet.css # Custom CSS file for styling UI
|
+-- app.py # Main backend script defining chat logic
+-- chainlit.md # Optional Markdown file for app documentation
```

The `.chainlit/` directory holds all UI-related configuration and assets. The `config.toml` file is used to define the application name, language settings, and custom styles. The `public/` folder stores all static files such as images and stylesheets. The main application logic resides in `app.py`, where we define event handlers and integrate the system with the backend agents. An example of User Interface with Chainlit shown as Figure 3.1

This organized structure makes it straightforward to maintain and extend the user interface while keeping it visually coherent and functionally integrated with the backend logic.

Our Chainlit-based implementation includes several key customizations:

- **Application Branding:** We personalize the app name by modifying the `config.toml` file located under the `.chainlit/` directory using the `name = "Your App Name"` field.
- **Welcome Message:** A customized welcome message is defined in the `@cl.on_chat_start` function to greet users upon session initiation.
- **Logo and Favicon Customization:** Custom images (e.g., university logos) are added to the `.chainlit/public/` folder as `logo_dark.png` and `logo_light.png`, and `favicon.png`. These assets are automatically used in both the chat screen and browser tab.
- **Background Styling:** We customize the UI background by modifying the CSS file (`stylesheet.css`) and linking it via the `custom_css` field in `config.toml`. This allows us to apply custom images, overlays, and visual enhancements for improved readability and visual coherence.

  ```
  [UI]
     custom_css = "/public/custom.css"
  ```

This setup enables flexible and modular customization of the UI, allowing the system to provide a visually appealing, brand-consistent, and functional user experience tailored to the target audience.

5.3 Database Setup with Milvus

In a multi-agent system, agents often need to access and retrieve high-dimensional vector representations of data such as text, images, embeddings, or knowledge chunks. **Milvus** serves as a *vector database* that enables efficient storage, indexing, and similarity search over these embeddings at scale.

By integrating Milvus into the multi-agent architecture, the system gains the following benefits:

- **Centralized Knowledge Retrieval**: Agents (e.g., Retrieval Agent, Memory Agent, or Knowledge Agent) can perform fast vector similarity search over a shared vector space to retrieve semantically relevant information in real time.
- **Scalable Embedding Storage**: Milvus can store millions to billions of embeddings generated from multiple agents, enabling large-scale memory or context persistence.
- **Agent Collaboration through Shared Memory**: Milvus acts as a shared memory backend, allowing agents to query past contexts, user interactions, or external documents using dense embeddings.
- **Fast and Flexible Search**: Agents can execute approximate nearest neighbor (ANN) queries with sub-second latency, supporting interactive or streaming use cases.

For example, a *Symptom Agent* in a healthcare system can encode user input into an embedding, query Milvus for similar known cases or documents, and pass the result to a *Care Agent* or *Vaccine Agent* for further analysis.

To connect your application to Milvus via Zilliz Cloud, you will need both the URI (endpoint) of your deployed Milvus instance and a valid API Key.

1. Access Zilliz Cloud Dashboard via: `https://cloud.zilliz.com/login` and create your account, login if you already have one.
2. After you successfully created an account or login, next step is to select your milvus instance by clicking create cluster button. and following the setup

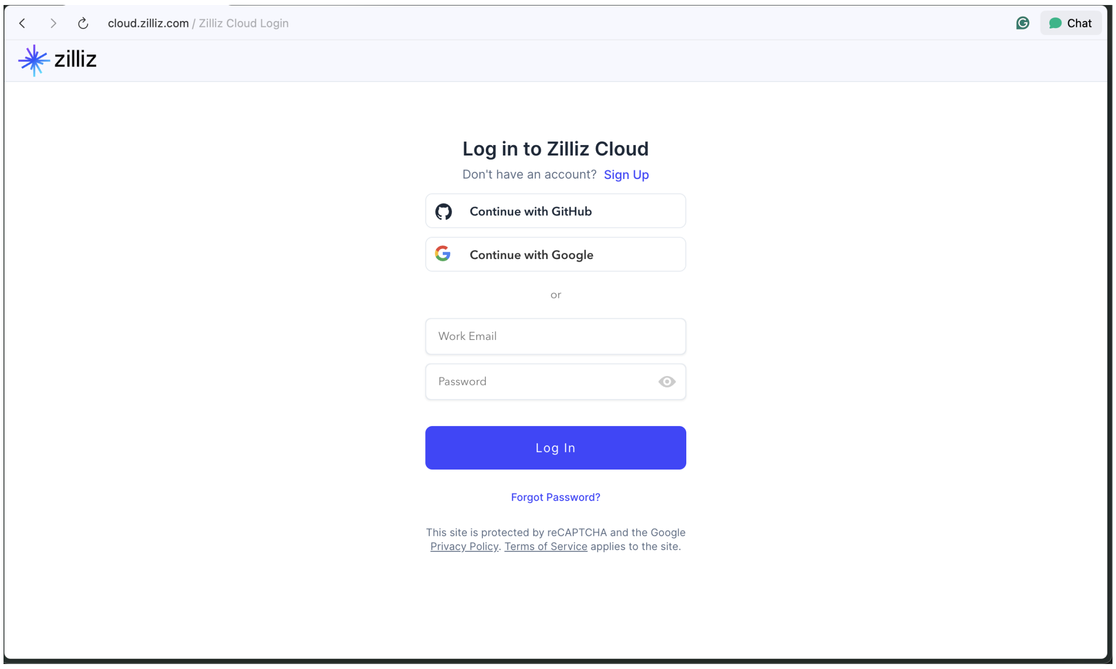

cloud.zilliz.com / Zilliz Cloud Login
Chat
zilliz
Log in to Zilliz Cloud
Don't have an account? Sign Up
Continue with GitHub
Continue with Google
or
Work Email
Password
Log In
Forgot Password?
This site is protected by reCAPTCHA and the Google Privacy Policy, Terms of Service applies to the site.

Figure 5.2: Zilliz Interface

instructions.

3. Locate the URI: Once inside the instance detail page, scroll to the Connection Info or Public Endpoint section as shown in Figure 5.4. Copy the URI and TOKEN, which typically follow this format:

```
https://<your-instance-id>.api.zillizcloud.com
```

```
service.zillizcloud.com<project-id>|<long-random-base64-
   like-string>
```

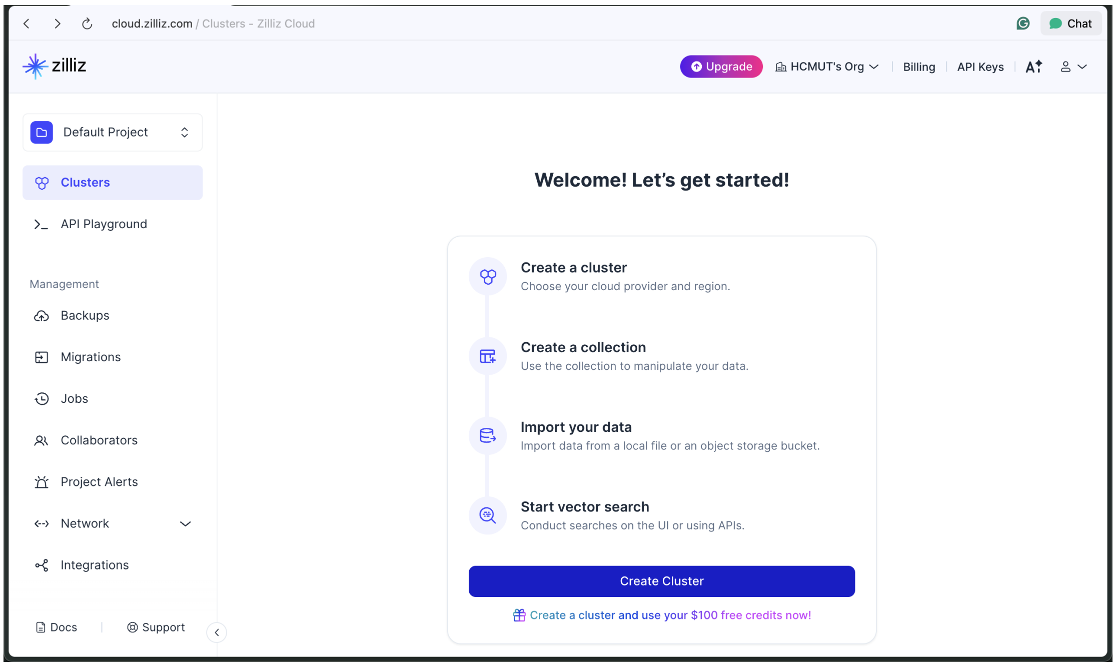
   You will use this URI as the endpoint when connecting via SDK or REST API.

4. Store Securely: Store the URI and your Token in a secure place (e.g., environment variables or secret manager) to use them in your application. In our material, we suggest you to use .env file to store your API keys.

Example .env File Content

```
MILVUS_URI="https://your-milvus-uri"
MILVUS_TOKEN="YourMilvusPasswordOrToken"
```

You are now ready to connect to your Milvus instance using the SDK or REST



Welcome! Let's get started!

Create a cluster
Choose your cloud provider and region.

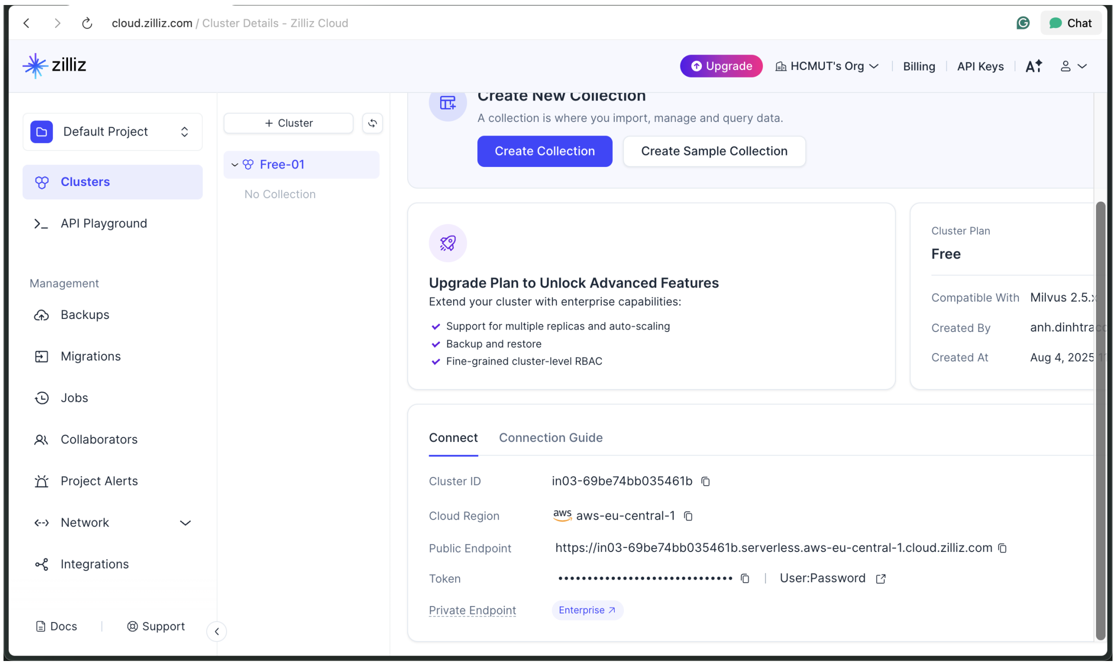
Create a collection
Use the collection to manipulate your data.

Import your data
Import data from a local file or an object storage bucket.

Start vector search
Conduct searches on the UI or using APIs.

Create a cluster and use your $100 free credits now!

Figure 5.3: Create Milvus Instance



Create New Collection
A collection is where you import, manage and query data.

Upgrade Plan to Unlock Advanced Features
Extend your cluster with enterprise capabilities:
- Support for multiple replicas and auto-scaling
- Backup and restore
- Fine-grained cluster-level RBAC

Cluster Plan
Free

Compatible With	Milvus 2.5.
Created By	anh.dinhtra
Created At	Aug 4, 2025

Cluster ID	in03-69be74bb035461b
Cloud Region	aws aws-eu-central-1
Public Endpoint	https://in03-69be74bb035461b.serverless.aws-eu-central-1.cloud.zilliz.com
Token	•••••••••••••••••••••••••••••• | User:Password
Private Endpoint	Enterprise

Figure 5.4: Example of a public endpoint section

client of your choice.

5.4 Pydantic AI for Multi-Agent System (MAS) Server

At the heart of the system lies the Multi-Agent System (MAS), a collection of autonomous agents, each responsible for a distinct subtask. These agents collaborate to analyze user input, retrieve relevant information, and construct comprehensive responses. The agent-based architecture promotes specialization, parallel processing, and ease of integration for new functionalities.

This section outlines the structure of the MAS, the roles and responsibilities of individual agents, and the inter-agent communication protocols that drive coordinated decision-making.

5.4.1 What is it?

Pydantic AI is a Python-based framework designed to streamline the development of multi-agent systems powered by large language models (LLMs). Built on the foundation of Pydantic, a widely-used library for data validation and settings management, Pydantic AI integrates Pydantic's robust type-checking capabilities with LLMs to create type-safe, structured, and maintainable AI applications. The framework is model-agnostic, supporting a variety of LLMs such as OpenAI, Anthropic, Gemini, Ollama, Groq, and Mistral, enabling developers to switch between models without significant code refactoring.

The framework is particularly suited for building complex workflows where multiple agents collaborate to perform specialized tasks, such as processing user queries, managing schedules, or retrieving data from external sources. This collaborative approach allows for the creation of sophisticated systems that can handle diverse and complex user requirements through the coordinated efforts of specialized agents.

The main features of Pydantic AI include type safety, which utilizes Python's type hints to ensure code reliability and maintainability, reducing errors in LLM outputs. The framework employs structured responses through Pydantic models to validate and structure LLM outputs, ensuring consistency across runs. Additionally, it supports streamed responses with real-time streaming of LLM outputs and immediate validation for rapid and accurate results.

Pydantic AI is ideal for applications requiring coordinated AI-driven workflows, such as intelligent assistants, customer support systems, or health management platforms like VitalFit Persona. However, its setup and configuration may require careful tuning to achieve production-grade performance, particularly for scalability and robustness. The framework's flexibility and comprehensive feature set make it an excellent choice for developers looking to build sophisticated multi-agent systems that can adapt to various business requirements and use cases.

5.4.2 Create Tools

Tools in Pydantic AI are Python functions that agents use to perform specific actions, such as retrieving data, interacting with external services, or delegating tasks to other agents. These tools represent the core functional capabilities that agents can leverage to accomplish their designated tasks within the multi-agent system. Tools are defined as regular Python functions and then attached to agents, creating a modular and extensible architecture that allows for easy maintenance and enhancement of system capabilities.

Pydantic AI ensures that tool inputs and outputs are validated using Pydantic models, enhancing reliability and type safety throughout the system. This validation mechanism prevents common errors and ensures that data flowing between different components of the system maintains its integrity and expected format. Tools can range from simple functions, such as retrieving the current date, to complex operations, such as querying a database or calling another agent for specialized processing.

The process of creating a tool involves several well-defined steps that ensure consistency and reliability across the system. First, developers must define the function by writing a Python function that performs the desired task, specifying input and output types using Pydantic models for validation. This approach ensures that all data entering and leaving the tool is properly structured and validated according to predefined schemas.

For example, when designing an input format for setting up a meeting in a calendar system, developers can create a comprehensive data model that captures all necessary information:

Listing 5.1: Calendar Event Input Model

```
class CalendarEventInput(BaseModel):
   summary: str = Field(..., description="Event title")
   description: Optional[str] = Field(None, description="Event
      description")
   start_time: str = Field(..., description="Start time in
      RFC3339 format.")
   end_time: str = Field(..., description="End time in RFC3339
      format.")
   calendar_id: str = Field("primary", description="Google
      Calendar ID")
```

Similarly, the output format can be defined to provide consistent response structure:

Listing 5.2: Calendar Event Output Model

```
class CalendarEventOutput(BaseModel):
   success: bool
   event_id: Optional[str] = None
   message: str
```

The second step involves attaching the tool to an agent. Once the tool function is defined and validated using Pydantic, it can be seamlessly attached to an agent through the system's architecture. In this framework, each tool is exposed through a "factory" pattern, which returns a ready-to-use callable interface. This factory pattern provides several advantages, including encapsulation of configuration details, consistent initialization procedures, and easy testing and mocking capabilities.

Listing 5.3: Tool Factory Example

```
def create_send_email_tool():
   """
   Create a send email tool function with pre-configured
      recipient emails.

   Args:
        to_emails: List of recipient email addresses (default:
            None)

    Returns:
        A function that sends emails with the specified
            recipients
    """

    def configured_send_email_tool(input_data: EmailToolInput)
        -> EmailToolOutput:
        return send_email_tool(
            input_data,
            sender_email=os.environ.get("SENDER_EMAIL"),
            sender_password=os.environ.get("SENDER_PASSWORD"),
        )

    return configured_send_email_tool
```

This factory approach allows for flexible configuration of tools while maintaining a consistent interface that agents can rely upon. The factory can encapsulate environment-specific configurations, authentication details, and other setup requirements, presenting a clean and simple interface to the agents that will use these tools.

5.4.3 Create Agents

In Pydantic AI, agents are specialized entities within multi-agent systems designed to handle specific requests and perform particular functions using LLMs and associated tools. Each agent is defined with a specific role, system prompt, and output type, which Pydantic validates to ensure consistency and reliability. The agent creation process requires developers to specify critical components including model specification (such as openai:gpt-4o or google-gla:gemini-1.5-flash), system prompts that define the agent's behavior, and tools that extend functionality beyond text generation. Agents can be equipped with tools for calling external APIs, accessing databases, or delegating tasks to other agents, making them highly flexible and powerful. This tool integration capability allows agents to perform complex operations and collaborate effectively within larger systems, enabling sophisticated workflows that leverage the strengths of specialized components working together to accomplish tasks that would be difficult for single agents to manage.

Listing 5.4: Model Configuration

```
provider = GoogleGLAProvider(api_key=os.getenv("GEMINI_API_KEY"
   ))
model = GeminiModel('gemini-2.0-flash', provider=provider)
```

The second crucial component is the system prompt, which is a string defining the agent's role and behavior. This prompt serves as the foundational instruction that guides how the agent interprets and responds to various inputs. For example, a system prompt might specify "You are a nutrition expert" for an agent designed to provide dietary advice and health recommendations.

The third component involves specifying the tools that the agent can access and utilize. These are optional functions that the agent can call to perform specific tasks beyond text generation. The tool selection should align with the agent's intended role and the types of tasks it is expected to handle.

Here's an example of how an agent might be created with these components:

Listing 5.5: Agent Creation Example

```
schedule_agent = AgentClient(
   model=model,
   system_prompt=SCHEDULE_PROMPT,
   tools=[email_tool, calendar_tool]
).create_agent()
```

This example demonstrates the creation of a scheduling agent that has access to both email and calendar tools, enabling it to manage appointments and send notifications as needed. The agent's behavior is guided by the SCHEDULE_PROMPT, which would contain specific instructions about how to handle scheduling requests and interact with the available tools.

Agents can operate independently or as part of a multi-agent system, where they collaborate by calling each other as tools. This collaborative approach enables the creation of sophisticated systems where different agents can specialize in different domains while working together to accomplish complex tasks that require multiple areas of expertise.

5.4.4 Multi-agent Systems

Pydantic AI excels in building multi-agent systems, where multiple agents collaborate to handle complex tasks that would be difficult for a single agent to manage effectively. Each agent specializes in specific domains such as nutrition, scheduling, validation, or data analysis, interacting with others by calling them as tools. This specialization enables modular and scalable workflows, with a top-level orchestrator agent delegating tasks to specialized agents based on user input and system requirements.

The multi-agent approach provides significant advantages over single-agent systems: domain expertise through fine-tuned optimization for specific tasks, fault isolation where problems in one agent don't affect the entire system, and scalability benefits allowing new agents to be added without modifying existing ones.

The orchestration mechanism is critical to multi-agent systems. The orchestrator serves as the central coordinator, receiving user requests, analyzing their intent and content, and determining which specialized agents should handle different aspects of the request. It performs initial processing, routes requests to appropriate agents, collects and synthesizes responses from multiple agents, and presents a coherent final response.

For example, in an e-commerce platform, a Product Agent handles searches and recommendations, an Order Agent manages processing and payments, a Customer Service Agent handles inquiries and returns, while an Orchestrator Agent analyzes queries and delegates tasks appropriately, managing the overall workflow and user interaction seamlessly.

Listing 5.6: Multi-Agent System Configuration Example

```
# Agent configurations
agents_config = {
   "nutrition_agent": {
      "role": "Provides nutritional advice and health
         information",
      "tools": ["health_tool", "web_tool"],
      "specialization": "Health and dietary planning"
   },
   "schedule_agent": {
      "role": "Manages appointments and notifications",
      "tools": ["calendar_tool", "email_tool"],
      "specialization": "Scheduling and communication"
   },
   "validator_agent": {
      "role": "Validates user inputs",
      "tools": [],
      "specialization": "Data validation and quality assurance"
   },
   "orchestrator_agent": {
      "role": "Delegates tasks based on user input",
      "tools": [],
      "specialization": "Request routing and coordination"
   }
}
```

The interaction patterns between agents can be quite sophisticated, involving both direct tool calls and indirect communication through shared data structures. Agents might pass context information, intermediate results, and status updates between each other to ensure that complex multi-step processes are handled correctly and efficiently.

This multi-agent architecture also supports advanced features like parallel processing, where multiple agents can work on different aspects of a complex request simultaneously, and conditional workflows, where the results from one agent determine which other agents should be involved in processing the request. The system can also implement priority-based scheduling, error recovery mechanisms, and load balancing across multiple instances of the same agent type to ensure robust and efficient operation under various conditions.

5.5 Proxy Server

The Proxy Server acts as an intermediary layer between the user interface and the backend processing system. Its primary responsibilities include managing communication flow, routing user requests to the appropriate agents, and aggregating responses for delivery back to the user interface. By decoupling the UI from the internal logic of the multi-agent system, the proxy server ensures modularity, scalability, and better maintainability of the entire architecture.

This section describes the communication protocols, routing mechanisms, and session management strategies implemented within the proxy server module.

5.5.1 Installing the Multi Agents Proxy Server Project and Dependencies

Virtual Environment Setup

It is strongly recommended to create and use a virtual environment before installing project dependencies. Virtual environments provide isolation for your project's dependencies, preventing conflicts with other Python projects and ensuring consistent package versions across different development setups.

First, navigate to your project directory and create a new virtual environment:

```
python -m venv venv
```

After creating the virtual environment, activate it using the appropriate command for your operating system:

```
source venv/bin/activate # On macOS/Linux
# or
venv\Scripts\activate # On Windows
```

Once activated, your command prompt should display the virtual environment name (typically `(venv)`), indicating that you are now working within the isolated environment. This ensures that all subsequent package installations will be contained within this virtual environment rather than affecting your system-wide Python installation.

Project Setup and Dependency Installation

With your virtual environment activated, open the project folder using Visual Studio Code or your preferred integrated development environment. Navigate to the project root directory where the `requirements.txt` file is located, and install the necessary Python packages by running:

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

This command will install all required dependencies for the multi-agent proxy server within your virtual environment, including the Pydantic AI framework, Chainlit for user interface management, Redis client libraries for session management, and any additional packages specified in the requirements file. The virtual environment ensures that these packages are installed in isolation and won't interfere with other Python projects on your system.

5.5.2 Running the Multi-Agent System

Once you have completed the installation of dependencies and configuration of the proxy server, you can deploy and run the entire multi-agent system through the proxy server interface. The proxy server will handle all communication between the user interface and the backend agents, ensuring proper request routing and response aggregation.

Before launching the multi-agent system through the proxy server, ensure that all required services and configurations are properly established. This verification step is crucial for preventing runtime errors and ensuring seamless operation from system startup.

First, verify that your Redis instances are operational on the designated ports (typically 6379 and 6380 for dual-instance configurations). The proxy server relies on Redis for session management and inter-agent communication caching. Test the Redis connections using the ping command on both instances to confirm they are responding correctly and accepting connections from the proxy server.

Next, confirm that all environment variables are properly configured in your `.env` file. These critical parameters include API keys for your chosen language model providers (Gemini, OpenAI, Anthropic), database connection strings for persistent data storage, authentication credentials for external services, and proxy server specific configurations such as port settings and routing rules.

Running the Application. To run this multi-agent system, save the code above as a Python file (e.g., app.py) and execute it with Chainlit from your terminal:

Run Application

```
chainlit run app.py
```

Chapter 6

Multi-agent Applications

6.1 Introduction

Multi-Agent Systems (MAS) have emerged as a powerful paradigm in artificial intelligence and distributed computing, enabling the development of systems that are decentralized, scalable, robust, and capable of intelligent coordination. Each agent in a MAS operates autonomously, yet interacts with other agents and the environment to achieve both individual and collective objectives. This chapter presents a systematic overview of the major application domains of MAS, showcasing its versatility across industries such as telecommunications, robotics, healthcare, economics, and cybersecurity.

6.2 Typical Applications of Multi-Agent Systems

6.2.1 Supply Chain Management

Multi-agent systems play a critical role in optimizing supply chain operations by distributing tasks among agents representing suppliers, manufacturers, distributors, and retailers. Each agent specializes in a specific function such as inventory management, logistics coordination, or order processing. They communicate and collaborate asynchronously to synchronize activities, optimize resource usage, reduce operational costs, and respond swiftly to changes in demand or disruptions. This decentralized approach improves scalability and robustness, as individual agents can adapt independently while the system as a whole achieves efficient supply chain management.

6.2.2 Healthcare Management

In healthcare, MAS are applied to enhance patient care and streamline hospital operations. Intelligent agents can monitor patient conditions, coordinate diagnostic procedures, and optimize treatment schedules by collaborating with doctors, medical staff, and connected medical devices. Systems based on MAS have shown high accuracy in tasks such as infectious disease detection and outbreak modeling, enabling faster diagnosis and effective response strategies. The ability of agents to specialize and share information in real-time improves the quality of healthcare services while reducing the workload on human practitioners.

6.2.3 Financial Market and Fraud Detection

Multi-agent systems are utilized in financial sectors for automated trading, portfolio management, and fraud detection. Agents continuously analyze market data in real-time to identify trends and execute trades autonomously based on predefined strategies. Additionally, collaborative agents monitor transactions and network activities to detect anomalies indicative of fraudulent behavior, thus enhancing security and decision-making speed. MAS offers the advantage of high adaptability and fault tolerance, allowing financial institutions to operate efficiently in dynamic and complex market environments.

6.3 Conlusion

These applications illustrate the flexibility and power of multi-agent systems across different domains. Each use case benefits from agents' autonomy, collaboration, asynchronous communication, and specialization, which collectively improve task efficiency, scalability, and robustness crucial for solving complex real-world problems.

Chapter 7

Summerization

This comprehensive tutorial has guided high school students through the complete journey of building impactful applications using Large Language Models (LLMs) in multi-agent system architectures. Throughout seven detailed chapters, we have explored the fundamental concepts, practical implementations, and real-world applications of collaborative AI systems that demonstrate the power of distributed artificial intelligence.

7.1 Key Learning Outcomes

The tutorial began with establishing a solid foundation through **Environment Setup** (Chapter 2), where students learned to configure professional development tools including Visual Studio Code, Redis server for agent memory management, Chainlit for user interface development, and Pydantic-AI for structured agent interactions. This foundational knowledge proved essential for all subsequent development work, emphasizing the importance of proper tooling in modern AI application development. In **Chapter 3**, we explored the core architecture of multi-agent systems through a practical Router/Dispatcher pattern implementation. Students discovered how to create specialized agents with distinct roles, implement effective prompting techniques, and design systems where a central Router Agent intelligently delegates user queries to appropriate Specialist Agents. This modular approach demonstrated significant advantages over monolithic single-agent systems, including improved maintainability, enhanced scalability, and increased accuracy through domain specialization. **Dataset Construction** (Chapter 4) addressed the critical foundation of any knowledge-driven AI system. Students learned systematic approaches to gathering high-quality data from authoritative sources, transforming raw documents into structured Q&A pairs using LLMs, and creating clean, well-formatted datasets. The emphasis on data quality, source reliability, and consistent formatting highlighted how the effectiveness of multi-agent systems fundamentally depends on their underlying knowledge bases.

7.2 System Architecture and Deployment

The **Deployment** chapter (Chapter 5) introduced students to production-ready system architecture through a three-tier design: User Interface built with Chainlit, Proxy Server for secure communication management, and the core Multi-Agent System backend. This modular architecture demonstrated professional software development practices while ensuring scalability and maintainability. The integration with Milvus vector database showcased how modern AI systems leverage high-dimensional embeddings for efficient knowledge retrieval and semantic search capabilities. Students gained hands-on experience with Pydantic-AI's powerful features, including type-safe tool creation, structured agent interactions, and collaborative multi-agent workflows. The tutorial emphasized the factory pattern for tool creation, proper input/output validation using Pydantic models, and effective orchestration strategies for coordinating multiple specialized agents.

7.3 Practical Applications and Impact

Chapter 6 expanded students' understanding by exploring real-world applications across diverse industries. The examples of Supply Chain Management, Healthcare Management, and Financial Market systems demonstrated how multi-agent architectures address complex challenges requiring distributed decision-making, real-time coordination, and adaptive responses to dynamic environments. These case studies illustrated the versatility and practical impact of the technologies students had learned to implement.

7.4 Technical Skills and Professional Development

Throughout this tutorial, students developed both technical proficiency and professional development practices. They learned to work with modern development environments, implement secure credential management through `.env` files, design modular and maintainable code architectures, and create user-friendly interfaces for complex AI systems. The emphasis on proper documentation, code organization, and system architecture reflects industry best practices.

7.5 Innovation and Future Possibilities

This tutorial has equipped students with the knowledge and confidence to innovate with AI in meaningful ways. The multi-agent paradigm opens numerous possibilities for creative applications, from intelligent educational assistants and AI-powered games to productivity tools and specialized domain experts. Students now understand how to approach complex problems by decomposing them into specialized agent roles, enabling the creation of sophisticated systems that leverage the strengths of collaborative AI.

7.6 Ethical Considerations and Responsible Development

While building these powerful systems, the tutorial emphasized the importance of responsible AI development. Students learned to prioritize data quality and source reliability, implement secure handling of sensitive information, and design systems that are transparent and maintainable. These considerations prepare students to be thoughtful developers who consider the broader implications of their AI applications.

7.7 Conclusion

The journey from environment setup to deployed multi-agent applications represents a comprehensive introduction to modern AI development. Students have gained practical experience with cutting-edge technologies while developing a deep understanding of how collaborative AI systems can solve complex real-world problems. This foundation prepares them for continued exploration and innovation in the rapidly evolving field of artificial intelligence, equipped with both technical skills and the conceptual framework needed to build impactful, responsible AI applications. The skills and knowledge gained through this tutorial extend far beyond the specific technologies covered, providing students with a robust foundation for lifelong learning in AI and software development. As they continue their educational and professional journeys, students can build upon these fundamentals to create increasingly sophisticated and impactful AI-driven solutions.

Bibliography

Visual Studio Code (2021). Documentation for Visual Studio Code 2021.