Teaching 02 Handout

0. Introduction 🚀

In today's session, we will learn:

1. **Deploying Open-Source LLMs with Ollama –** Running models locally using Docker and Ollama (Optional to the Project).
2. **Prompts Engineering –** Introduce different ways to customize and control LLMs response. Learn different useful patterns in prompt engineering.
3. **Practical Applications –** Introduce prompt structure in code. How to do prompt engineering programmatically.

1. Deploying Open-Source LLMs with Ollama 🦙

Ollama is a tool that makes it easy to run large language models (LLMs) locally on your own machine. It supports a variety of open-source models like LLaMA, Mistral, and Qwen, and wraps everything into a simple command-line interface.

To get started with Ollama, follow these steps:

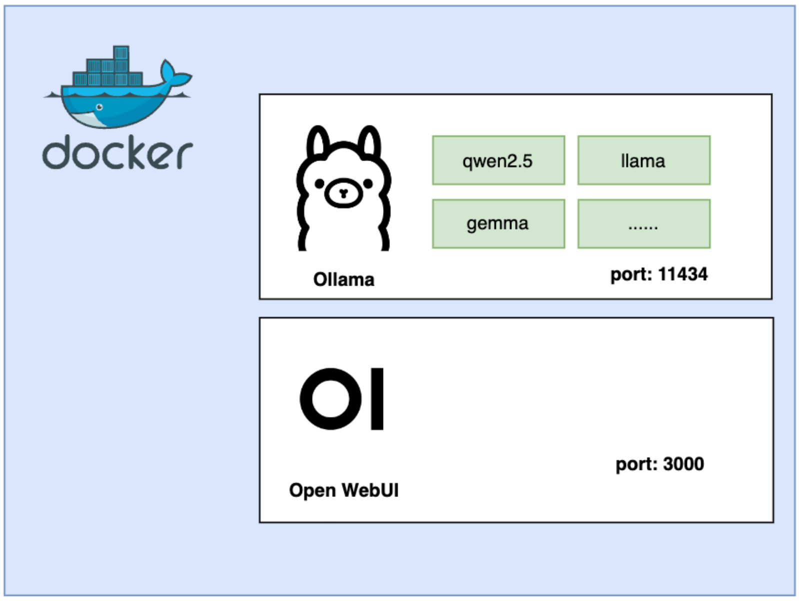
Step 1: Install Docker (Optional)

Ollama runs models using Docker containers. If you haven't installed Docker, follow the previous step on Teaching01.

To check if Ollama is installed successfully, you can run:

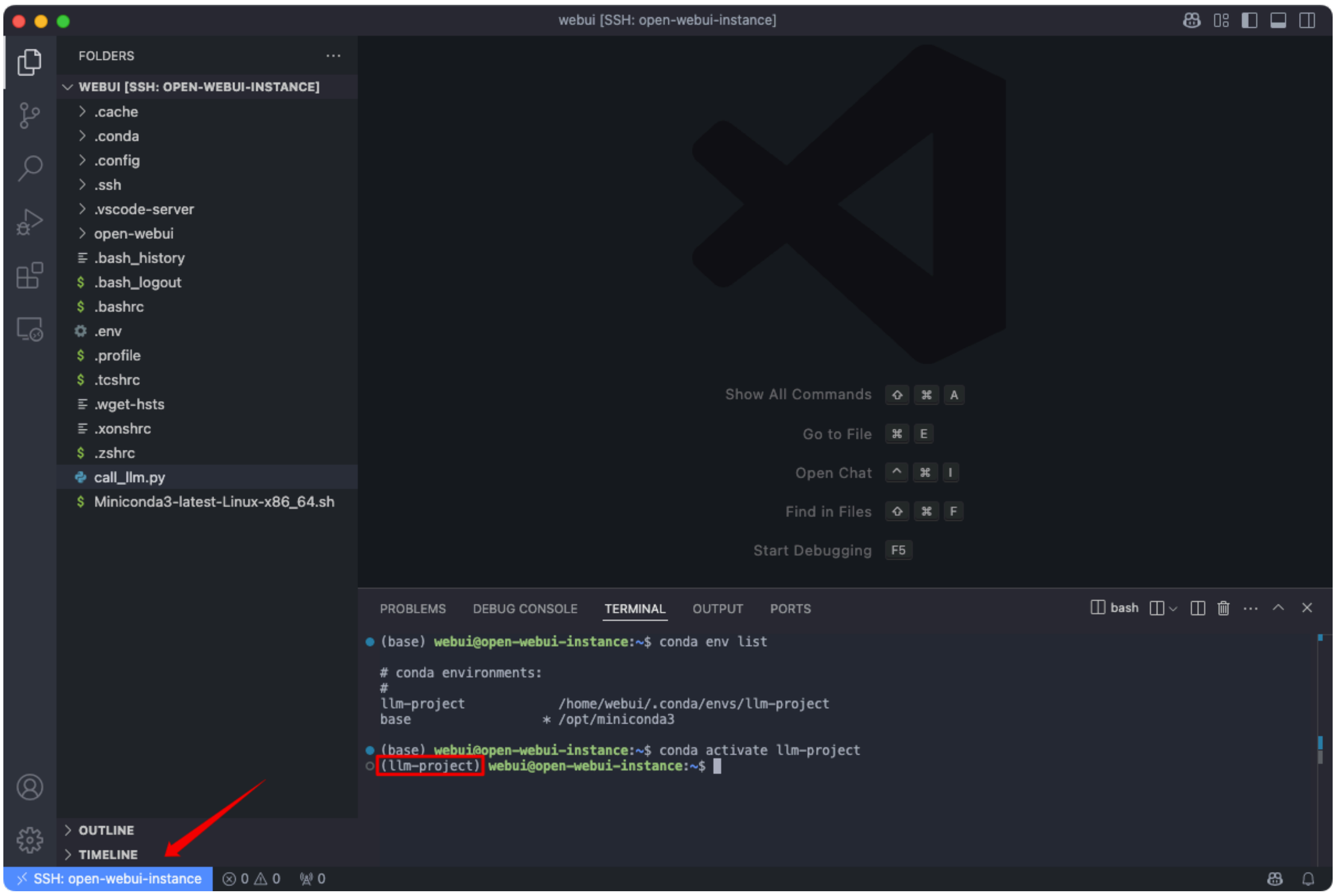
```
docker --version
```

For a better understanding, we can see the current structure in docker:



Step 2: Connect to VM instance

- Go to Google Cloud Console, and start your VM instance. 🔗 https://console.cloud.google.com/compute
- Open your VS Code platform and connect to your Google VM instance.



```
(base) webui@open-webui-instance:~$ conda env list

# conda environments:
#
llm-project            /home/webui/.conda/envs/llm-project
base                 * /opt/miniconda3

(base) webui@open-webui-instance:~$ conda activate llm-project
(llm-project) webui@open-webui-instance:~$
```

Step 3: Install Ollama

3.1 Give Your User Docker Permissions

By default, only root users can run Docker commands. To avoid typing sudo every time, add your user to the Docker group:

```
sudo usermod -aG docker ${USER}
```

> 💡 **Tip**
>
> If you are unsure of the current username ($USER is the variable for the currently logged-in user), you can first check who you are logged in to with the following command:
>
> ```
> whoami
> ```
>
> It will output your username, for example:

> ```
> webui
> ```
>
> Then you can replace $USER with this username and execute the command:
>
> ```
> sudo usermod -aG docker webui
> ```

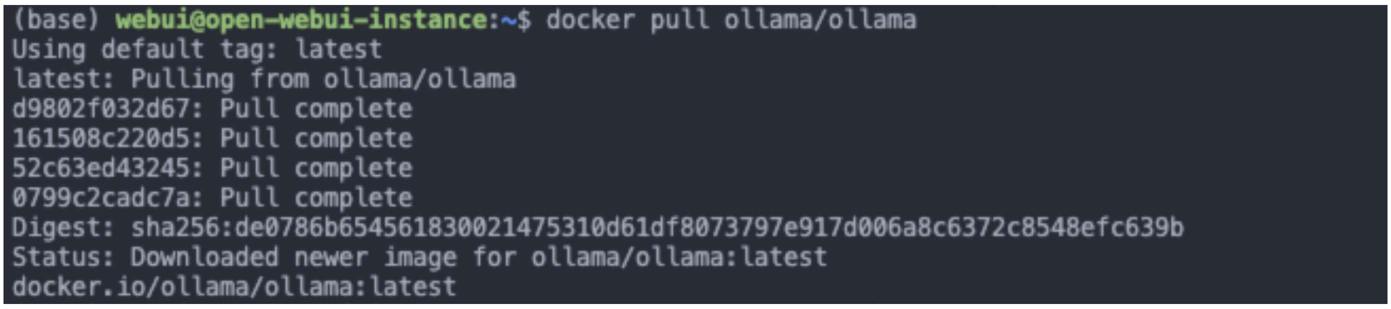
Then **log out and log back in**, or run:

```
newgrp docker
```

3.2 Pull the Ollama Docker Image

execute the following command:

```
docker pull ollama/ollama
```



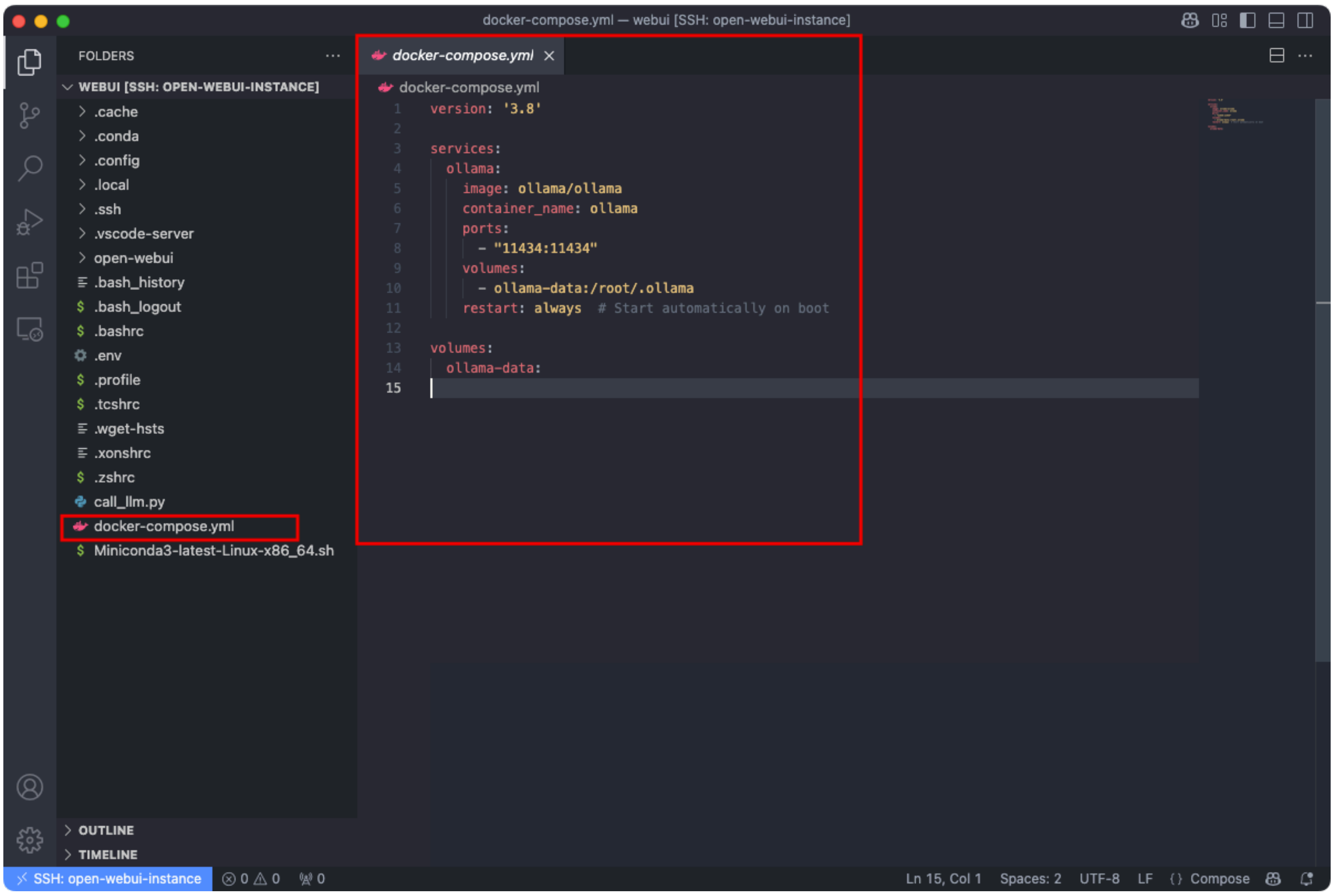
```
(base) webui@open-webui-instance:~$ docker pull ollama/ollama
Using default tag: latest
latest: Pulling from ollama/ollama
d9802f032d67: Pull complete
161508c220d5: Pull complete
52c63ed43245: Pull complete
0799c2cadc7a: Pull complete
Digest: sha256:de0786b6545618300214753 10d61df8073797e917d006a8c6372c8548efc639b
Status: Downloaded newer image for ollama/ollama:latest
docker.io/ollama/ollama:latest
```

3.3 Create docker-compose.yml to Manage Ollama

```
nano docker-compose.yml
```

You can install Ollama by running the following command:

```
sudo docker pull ollama/ollama
```



```yaml
version: '3.8'

services:
  ollama:
    image: ollama/ollama
    container_name: ollama
    ports:
      - "11434:11434"
    volumes:
      - ollama-data:/root/.ollama
    restart: always  # Start automatically on boot

volumes:
  ollama-data:

```

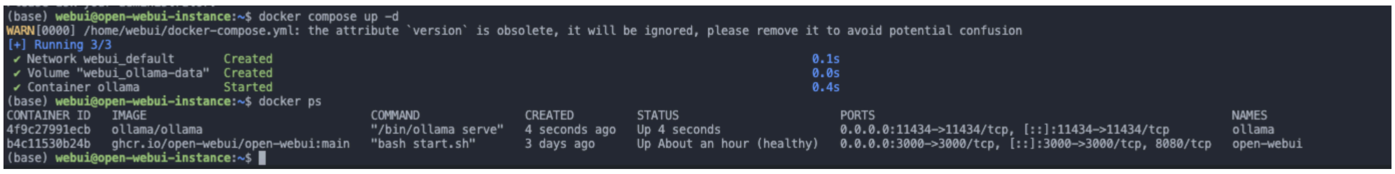
Step 4: Start Ollama

```
docker compose up -d
```

It starts the container and runs Ollama in the background. Then confirm:

```
docker ps
```

You should see a container called ollama running, listening on port 11434.



```
(base) webui@open-webui-instance:~$ docker compose up -d
WARN[0000] /home/webui/docker-compose.yml: the attribute `version` is obsolete, it will be ignored, please remove it to avoid potential confusion
[+] Running 3/3
 ✔ Network webui_default       Created                                                     0.1s
 ✔ Volume "webui_ollama-data"  Created                                                     0.0s
 ✔ Container ollama            Started                                                     0.4s
(base) webui@open-webui-instance:~$ docker ps
CONTAINER ID   IMAGE                                COMMAND               CREATED         STATUS                     PORTS                                            NAMES
4f9c27991ecb   ollama/ollama                        "/bin/ollama serve"   4 seconds ago   Up 4 seconds               0.0.0.0:11434->11434/tcp, [::]:11434->11434/tcp  ollama
b4c11530b24b   ghcr.io/open-webui/open-webui:main   "bash start.sh"       3 days ago      Up About an hour (healthy) 0.0.0.0:3000->3000/tcp, [::]:3000->3000/tcp, 8080/tcp open-webui
(base) webui@open-webui-instance:~$
```

Step 5: Download a New LLM Model

5.1 Access the Ollama Container

```
docker exec -ti ollama bash
```

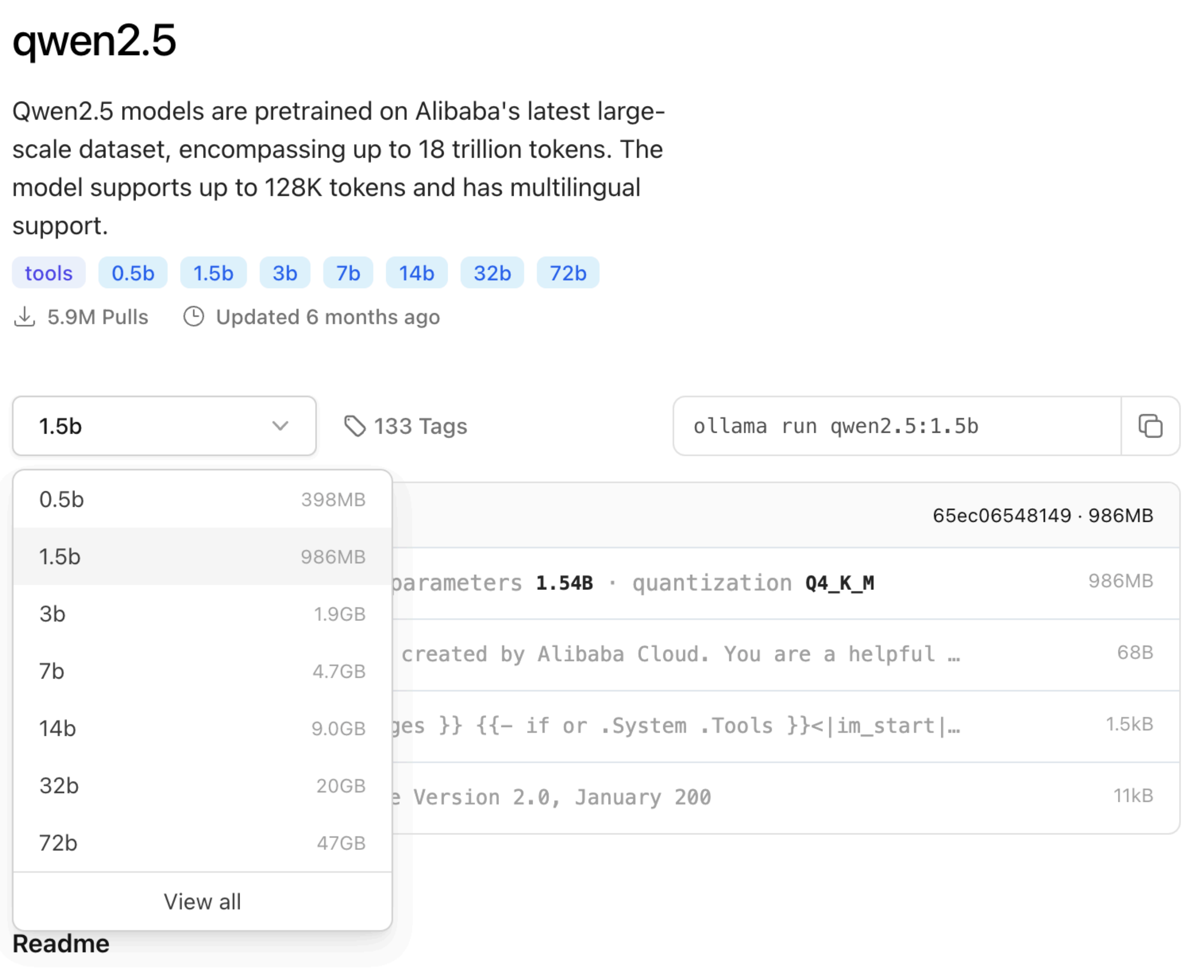
💡 **Tip**

> This command allow you enter the Ollama container Linux environment, like entering a normal terminal operation.
>
> Once you enter, you will see the prompt change to:
>
> ```
> root@<container-id>:/#
> ```

5.2 Download Models

To download a new model from the Ollama library, use the following command:

You can see what models are available by checking the Ollama website at: 🔗 https://ollama.com/library



qwen2.5

Qwen2.5 models are pretrained on Alibaba's latest large-scale dataset, encompassing up to 18 trillion tokens. The model supports up to 128K tokens and has multilingual support.

tools 0.5b 1.5b 3b 7b 14b 32b 72b

5.9M Pulls · Updated 6 months ago

1.5b · 133 Tags

```
ollama run qwen2.5:1.5b
```

0.5b	398MB
1.5b	986MB
3b	1.9GB
7b	4.7GB
14b	9.0GB
32b	20GB
72b	47GB

View all

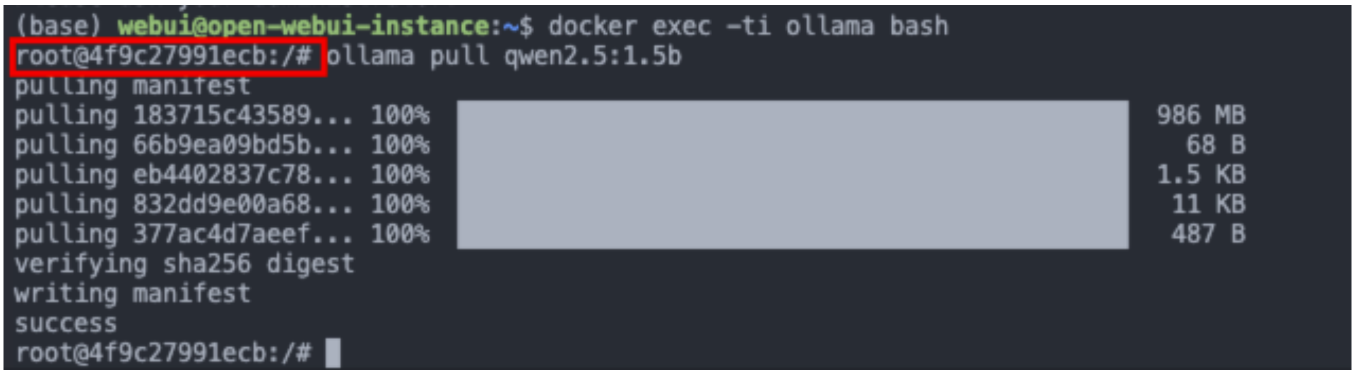
65ec06548149 · 986MB

```
parameters 1.54B · quantization Q4_K_M              986MB
created by Alibaba Cloud. You are a helpful …         68B
ges }} {{- if or .System .Tools }}<|im_start|…       1.5kB
e Version 2.0, January 200                           11kB
```

Readme

Install by following command:

```
ollama pull qwen2.5:1.5b
```

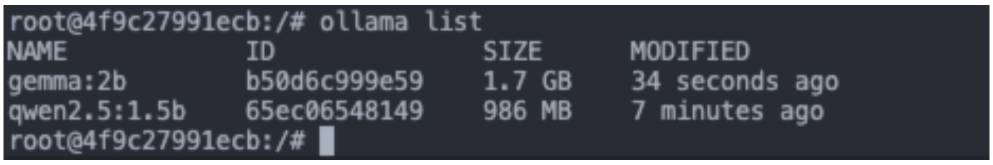


```
(base) webui@open-webui-instance:~$ docker exec -ti ollama bash
root@4f9c27991ecb:/# ollama pull qwen2.5:1.5b
pulling manifest
pulling 183715c43589... 100%                                   986 MB
pulling 66b9ea09bd5b... 100%                                    68 B
pulling eb4402837c78... 100%                                   1.5 KB
pulling 832dd9e00a68... 100%                                    11 KB
pulling 377ac4d7aeef... 100%                                   487 B
verifying sha256 digest
writing manifest
success
root@4f9c27991ecb:/#
```

5.3 List All Available LLM Models in Ollama

If you want to check the current list of downloaded models, you can run:

```
ollama list
```



```
root@4f9c27991ecb:/# ollama list
NAME            ID              SIZE      MODIFIED
gemma:2b        b50d6c999e59    1.7 GB    34 seconds ago
qwen2.5:1.5b    65ec06548149    986 MB    7 minutes ago
root@4f9c27991ecb:/#
```

5.4 Delete a Model (Optional)

If you need to remove an LLM model, you can do so with:

```
ollama delete ${model_name}
```

Step 6: Send a Model Call to Ollama

Activate the virtual environment we set up during Teaching01.

```
conda activate ${your env name}
```

Tip

If you don't remember the name of the environment from Teaching01, run the following command to list all available environments:

```
conda env list
```

Open the `call_llm.py` file, this time let's try to connect Ollama instead of

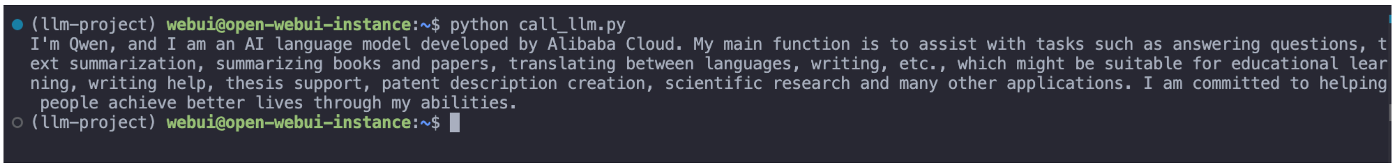
```
# from dotenv import load_dotenv
import os
import openai

# load_dotenv()
# API_KEY = os.getenv('QWEN_KEY')
# Initialize OpenAI client for OPEN AI
```

```python
client = openai.OpenAI(
    api_key='xxxxx',
    base_url="http://localhost:11434/v1"
)

# Make the first API call using the new syntax
response = client.chat.completions.create(
    model="qwen2.5:1.5b",
    messages=[{"role": "user", "content": "Hello, who are you?"}]
)
# Print the response
print(response.choices[0].message.content)
```

- You don't need an API key when using the local Ollama API.
- Make sure the base URL matches your running Ollama service, usually http://localhost:11434.
- Use a model name that has already been pulled into your instance, such as `qwen2.5:1.5b`

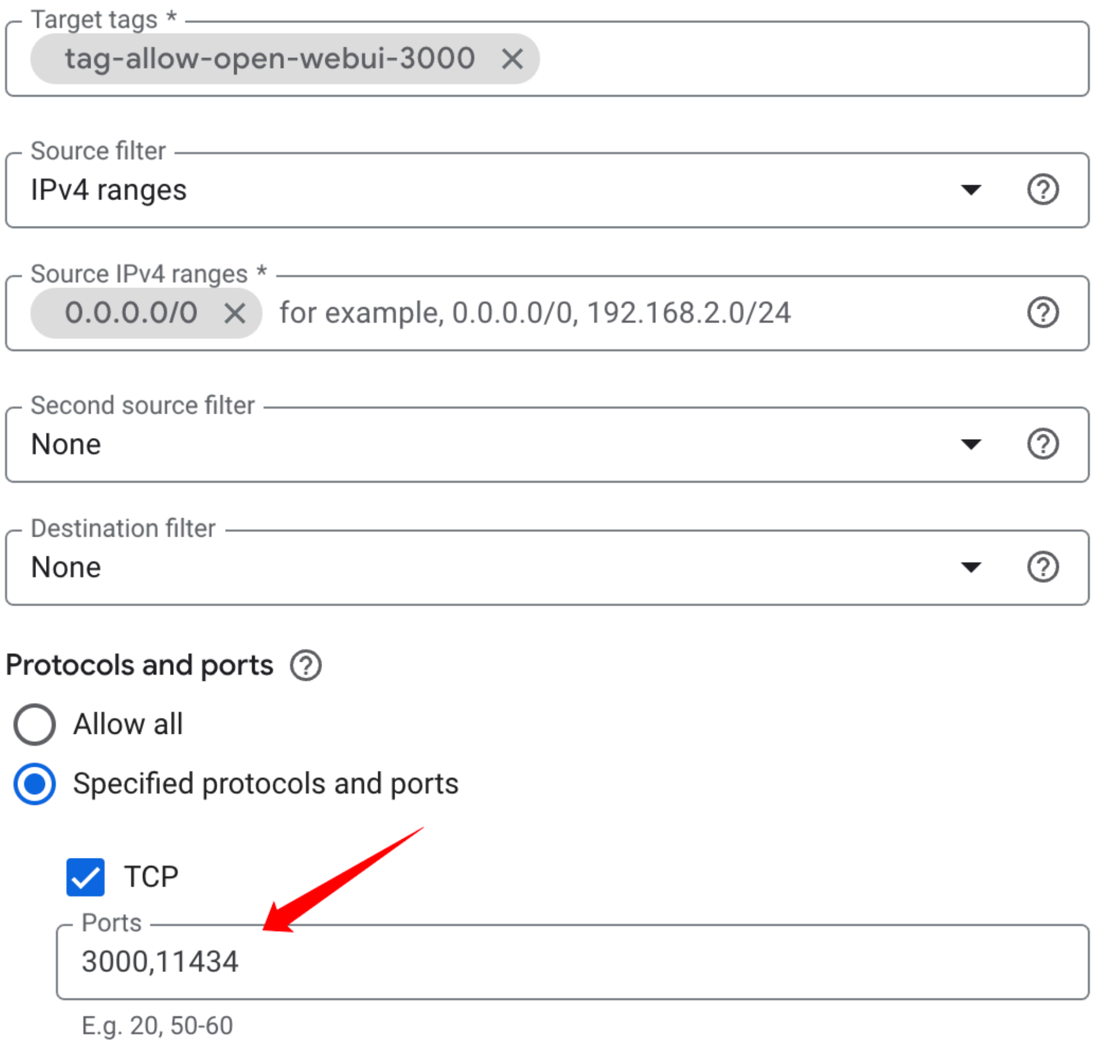


```
● (llm-project) webui@open-webui-instance:~$ python call_llm.py
  I'm Qwen, and I am an AI language model developed by Alibaba Cloud. My main function is to assist with tasks such as answering questions, t
  ext summarization, summarizing books and papers, translating between languages, writing, etc., which might be suitable for educational lear
  ning, writing help, thesis support, patent description creation, scientific research and many other applications. I am committed to helping
   people achieve better lives through my abilities.
○ (llm-project) webui@open-webui-instance:~$ 
```

Step 7: Connecting Ollama to Open WebUI

Before connecting Ollama to Open WebUI, we need to configure the firewall rules to allow internet access to the Ollama instance:

1. Go to **VPC Network** ➡️ **Firewall**.
2. Click on the firewall rule we previously created in **Teaching01**, named `allow-open-webui-3000`.
3. Click **Edit**, and add port `11434` after `3000` in the **Protocols and ports** field.
4. Save the changes.



Target tags *
tag-allow-open-webui-3000

Source filter
IPv4 ranges

Source IPv4 ranges *
0.0.0.0/0 — for example, 0.0.0.0/0, 192.168.2.0/24

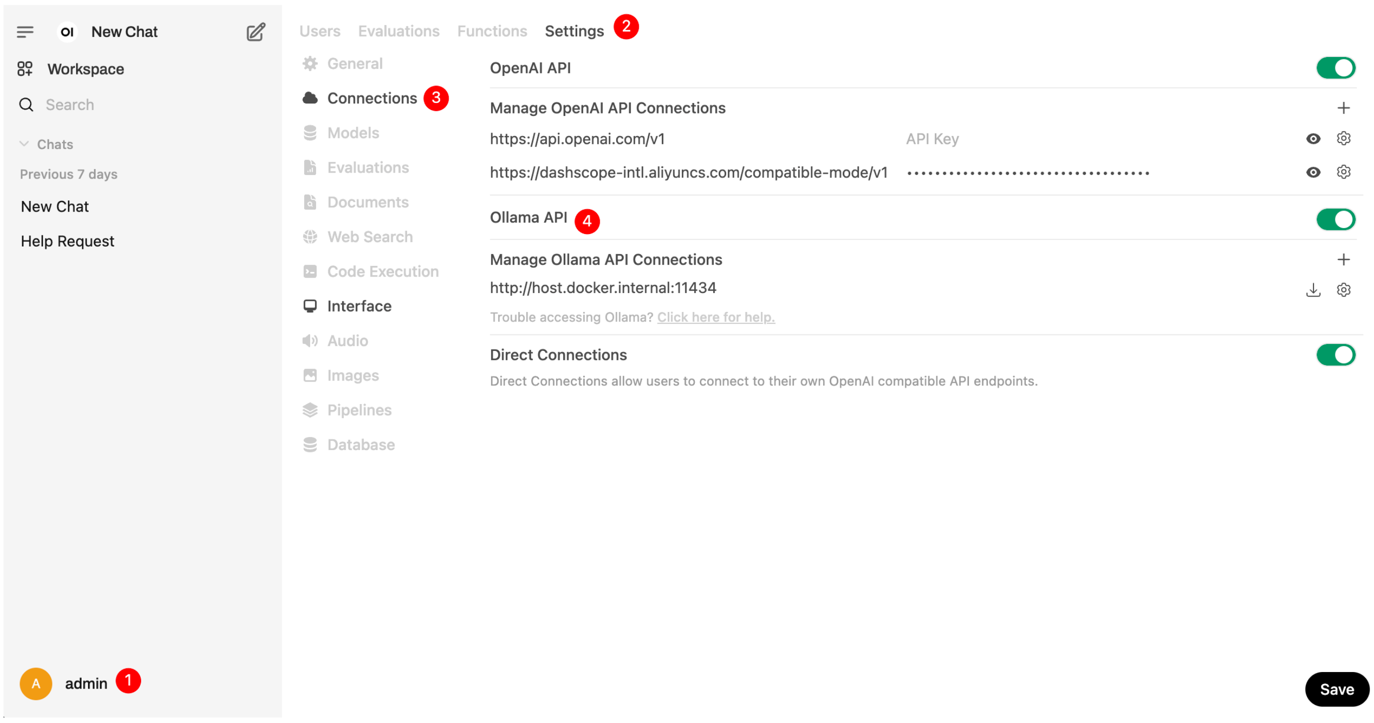
Second source filter
None

Destination filter
None

Protocols and ports
[ ] Allow all
[✓ Specified protocols and ports]

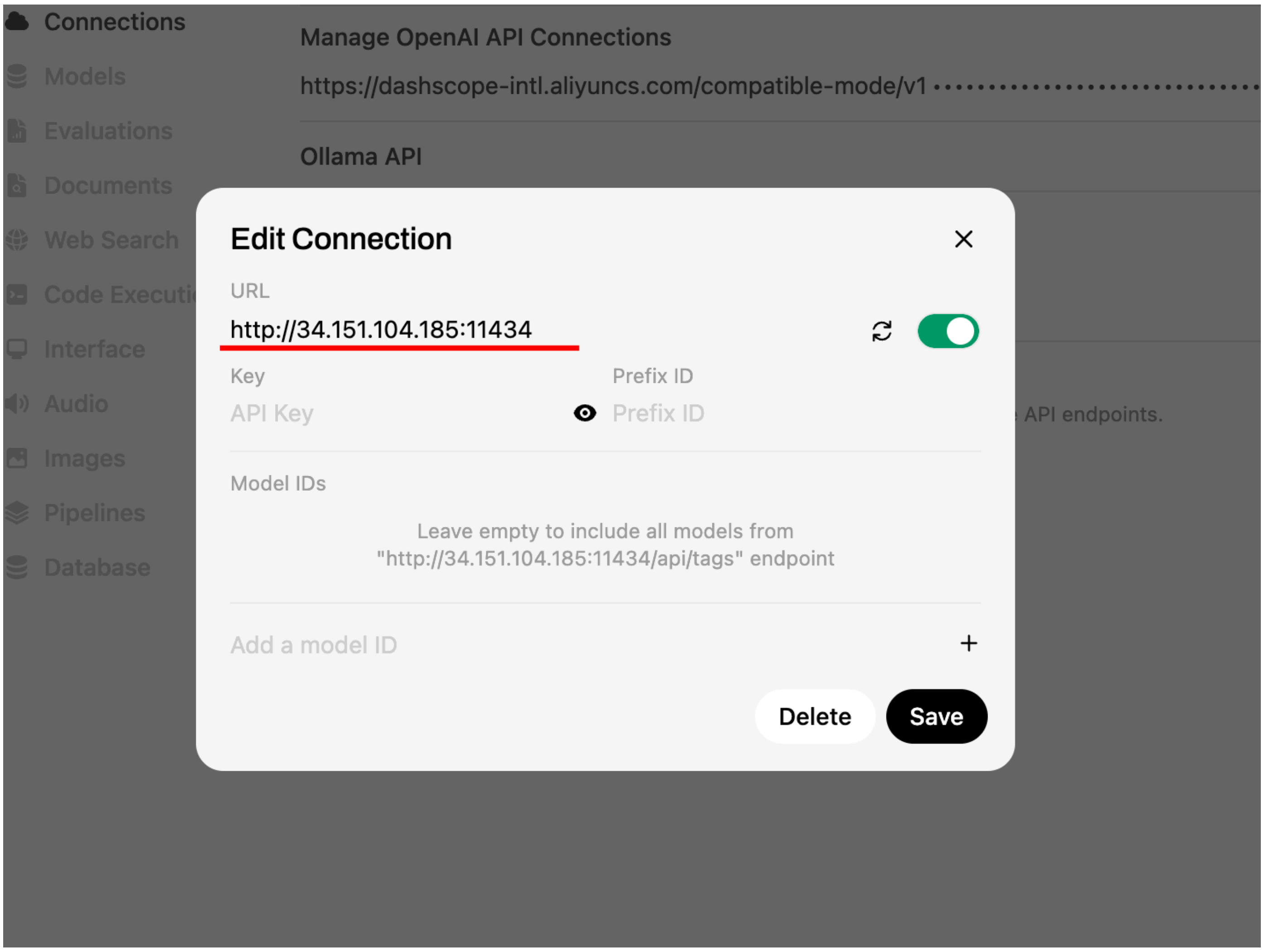
[✓] TCP
Ports
3000,11434
E.g. 20, 50-60

Next, open your **Open WebUI** interface by visiting `http://${your external ip}:3000`



OpenAI API
Manage OpenAI API Connections
https://api.openai.com/v1 — API Key
https://dashscope-intl.aliyuncs.com/compatible-mode/v1 — ••••••••••••••••••••••••••••••••••
Ollama API 4
Manage Ollama API Connections
http://host.docker.internal:11434
Trouble accessing Ollama? Click here for help.
Direct Connections
Direct Connections allow users to connect to their own OpenAI compatible API endpoints.

In the WebUI, Enter your Ollama address as `http://${your external ip}:11434`, Leave the **API Key** field empty.


Connections

Models

Evaluations

Documents

Web Search

Code Executi

Interface

Audio

Images

Pipelines

Database

Manage OpenAI API Connections

https://dashscope-intl.aliyuncs.com/compatible-mode/v1 ••••••••••••••••••••••••••••••

Ollama API

API endpoints.

Edit Connection

URL

http://34.151.104.185:11434

Key

API Key

Prefix ID

Prefix ID

Model IDs

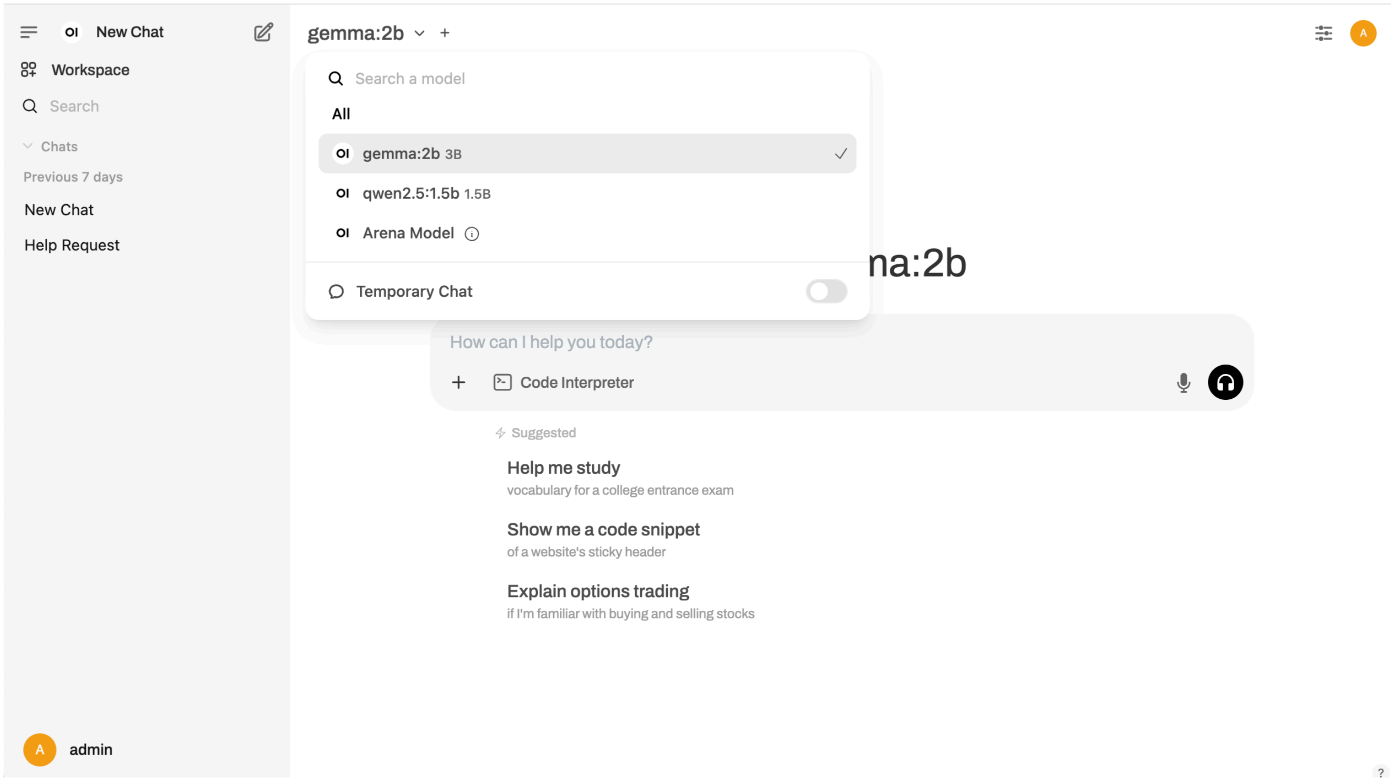
Leave empty to include all models from "http://34.151.104.185:11434/api/tags" endpoint

Add a model ID

Delete

Save

If the connection to Ollama is successful, the LLM models you've downloaded in Ollama will appear in the interface.


New Chat

Workspace

Search

Chats

Previous 7 days

New Chat

Help Request

admin

gemma:2b

Search a model

All

gemma:2b 3B

qwen2.5:1.5b 1.5B

Arena Model

Temporary Chat

na:2b

How can I help you today?

Code Interpreter

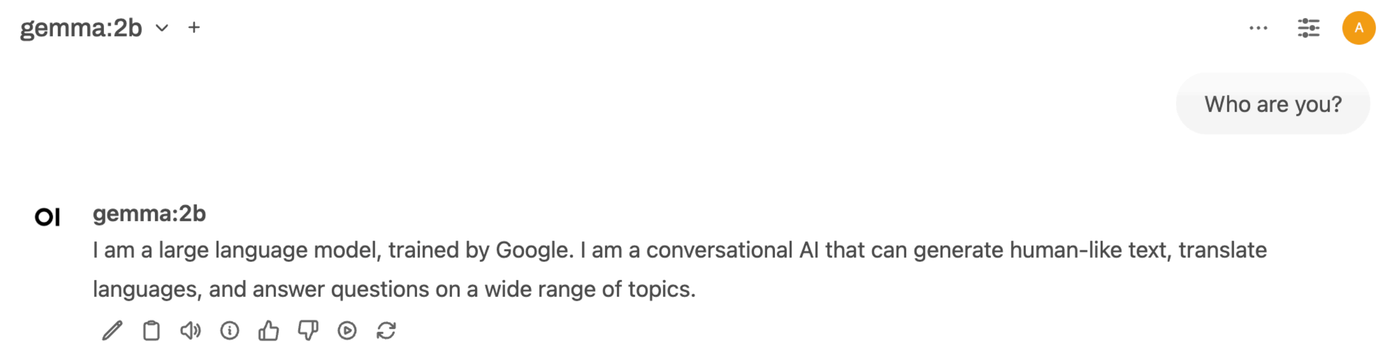
Suggested

Help me study
vocabulary for a college entrance exam

Show me a code snippet
of a website's sticky header

Explain options trading
if I'm familiar with buying and selling stocks

Start a conversation:


gemma:2b

Who are you?

gemma:2b

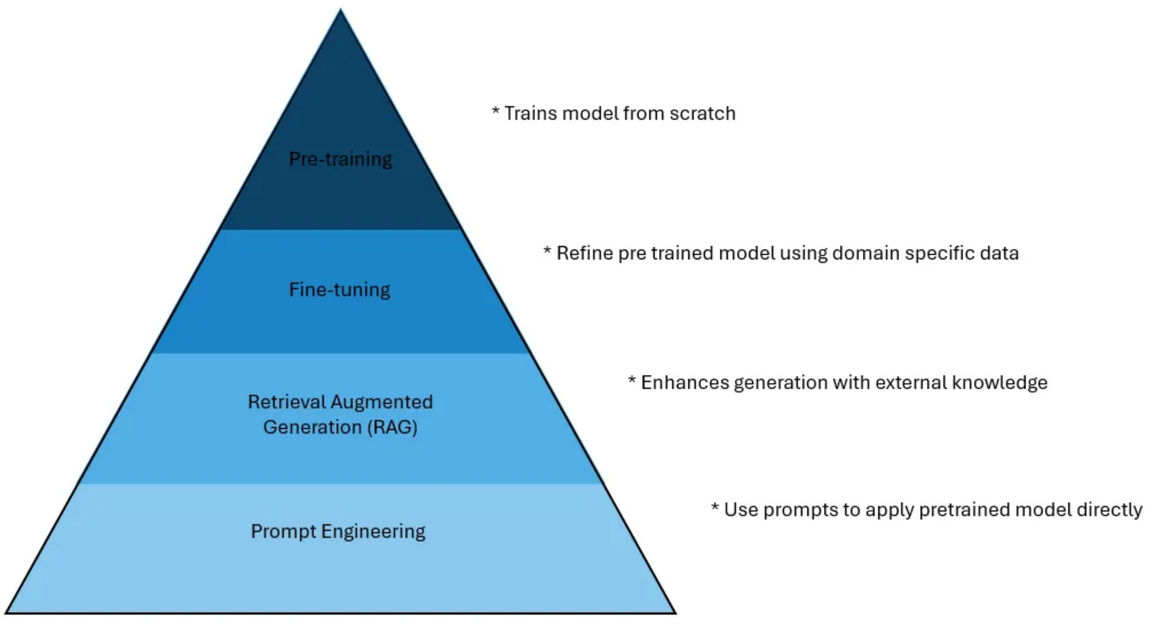
I am a large language model, trained by Google. I am a conversational AI that can generate human-like text, translate languages, and answer questions on a wide range of topics.

2. Prompt Engineering 🏗️

Large Language Models (LLMs) like GPT, Claude, and Qwen are typically **pre-trained as general-purpose models**.
 This means: without any customization, they produce **generic**, **broad**, and sometimes **unfocused** responses.

To make them work well for **specific tasks** or **domains**, we need ways to adapt them.

There are **four major ways** to customize and control LLMs:


Pre-training
* Trains model from scratch
Fine-tuning
* Refine pre trained model using domain specific data
Retrieval Augmented Generation (RAG)
* Enhances generation with external knowledge
Prompt Engineering
* Use prompts to apply pretrained model directly

💡 **Tip**

To see more differences, you can read:

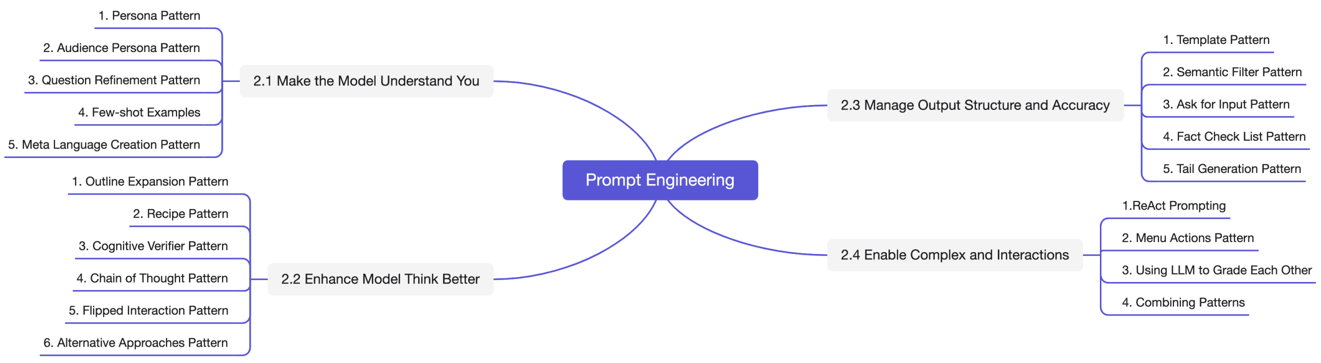
🔗 https://medium.com/@jainpalak9509/use-llms-pre-training-fine-tuning-rag-and-prompt-engineering-564d5670f44d

Among these, **Prompt Engineering** stands out as the **simplest, most flexible, and most accessible method** — no retraining, no infrastructure, just better instructions.

In this module, we'll introduce a set of **prompt design patterns** — proven techniques for crafting prompts that produce better, more reliable results.

We'll organize them by **purpose**:

- 🤔 Understand you better (personas, few-shot, etc.)
- 🧠 Think better (reasoning, CoT, refinement)
- ✍️ Output better (structure, filters, fact-check)
- 🔁 Interact better (tools, workflows, evolution)


Prompt Engineering
2.1 Make the Model Understand You
1. Persona Pattern
2. Audience Persona Pattern
3. Question Refinement Pattern
4. Few-shot Examples
5. Meta Language Creation Pattern
2.2 Enhance Model Think Better
1. Outline Expansion Pattern
2. Recipe Pattern
3. Cognitive Verifier Pattern
4. Chain of Thought Pattern
5. Flipped Interaction Pattern
6. Alternative Approaches Pattern
2.3 Manage Output Structure and Accuracy
1. Template Pattern
2. Semantic Filter Pattern
3. Ask for Input Pattern
4. Fact Check List Pattern
5. Tail Generation Pattern
2.4 Enable Complex and Interactions
1.ReAct Prompting
2. Menu Actions Pattern
3. Using LLM to Grade Each Other
4. Combining Patterns

2.1 Make the Model Understand You 🤔

1. Persona Pattern

The **Persona Pattern** helps you get more realistic and useful responses by telling the model to **"act as"** a specific role.

It packs a lot of meaning into just one sentence. Instead of writing long instructions, you don't need to explain every detail about how that role should respond. Just say:

> 👉 **"Act as a [role]..."**

and the model will take on that identity using its built-in knowledge.

💡 **Example prompt:**

```
- Act as Persona X
- Perform task Y

Examples:

- Act as a speech language pathologist. Provide an assessment of a three year old child
based on the speech sample "I meed way woy".
- Act as a computer that has been the victim of a cyber attack. Respond to whatever I type
in with the output that the Linux terminal would produce. Ask me for the first command.
- Act as a the lamb from the Mary had a little lamb nursery rhyme. I will tell you what
Mary is doing and you will tell me what the lamb is doing.
- Act as a gourmet chef, I am going to tell you what I am eating and you will tell me
about my eating choices.
```

2. Audience Persona Pattern

The **Audience Persona Pattern** helps you tell the model **who the output is for** — not who the model is, but who it's talking to.

This pattern is powerful when you want the same content to be explained in different ways, depending on the listener's knowledge, age, role, or personality.

The format is simple:

> 👉 **"Assume the audience is [someone]..."**

This helps the model adjust its explanation, tone, and word choices based on the audience's background.

💡 **Example prompt:**

```
Explain X to me.
Assume that I am Persona Y.

Examples:

- Explain large language models to me. Assume that I am a bird.
- Explain how the supply chains for US grocery stores work to me. Assume that I am Ghengis
Khan.
```

You don't need to define special rules. Just set the audience, and the model will adjust its answer based on what that audience might understand or care about.

3. Question Refinement Pattern

The Question Refinement Pattern helps you ask **better questions** by letting the model rewrite your input to make it clearer, deeper, or more useful.

> 👉 **"Whenever I ask a question, suggest a better version and ask me if I'd like to use it instead."**

This pattern works well because large language models have seen **many ways** of asking similar questions.

This process will help you:

- Makes your prompts more thoughtful
- Helps you reflect on what you **actually** want to ask
- Surfaces missing details (e.g. "I want to study computer science")

💡 **Example prompt:**

```
Examples:

- From now on, whenever I ask a question, suggest a better version of the question and ask
me if I would like to use it instead.

Tailored Examples:

- Whenever I ask a question about dieting, suggest a better version of the question that
emphasizes healthy eating habits and sound nutrition. Ask me for the first question to
refine.
```

Use this pattern when:

- Your initial question feels too general
- You want to improve prompt quality
- You're not sure how to phrase something clearly

It's a small change that often leads to much better answers.

4. Few-shot Examples

The Few-shot Examples pattern is a way to teach the model by example — instead of writing rules, you show what to do.

The basic idea is:

> 👉 **Give the model a few input–output examples, then ask it to continue the pattern.**

💡 **Example prompt:**

```
- example 1
Input: I love this book.
Sentiment: Positive

Input: The movie was okay but a bit long.
Sentiment: Neutral

Input: The food was terrible.
Sentiment:
```

This method works because language models are good at recognizing patterns. If your examples are clear, consistent, and focused, the model will likely follow them.

🔍 **Tips for writing effective few-shot examples:**

- Make sure your examples are **detailed** and **unambiguous**.
- Use **descriptive prefixes** (like "Input", "Sentiment") instead of vague ones ("Input", "Output" alone may be too generic).
- You can include **intermediate steps** like "Think → Action" to guide multi-step reasoning.

- Format matters! Keep the pattern clean and easy to follow (e.g. one step per line).

5. Meta Language Creation Pattern

The Meta Language Creation Pattern lets you teach the model a new shorthand or custom "language" that you define — and then use it in your prompts. The reason why we do this is because writing everything in full sentences can be slow, repetitive, or unclear. A well-designed shorthand is faster and more consistent.

> 👉 **First, explain the pattern, then use your new meta-language in prompts**

💡 **Example prompt:**

```
- example 1
When I say Sydney,3 -> Melbourne,2.
I mean: I will go from Sydney to Melbourne, staying 3 days in Sydney and 2 in Melbourne.
Sydney,0 -> Melbourne,1 -> Brisbane,4
```

or

```
- example 2
When I say Task X [Task Y], I mean Task X depends on Task Y being completed first.
Usage: "Describe the steps for building a house using my task dependency language."

Usage: "Provide an ordering for the steps: Boil Water [Turn on Stove], Cook Pasta [Boil
Water], Make Marinara [Turn on Stove], Turn on Stove [Go Into Kitchen]"
```

The model now understands your format and can generate the trip plan accordingly.

This is especially useful when:

- You do a **repetitive task** (like trip planning or classification).
- You want to define a **standard input format** for yourself or your team.
- You want the model to **interpret inputs consistently**, without needing full natural language.

2.2 Enhance Model Think Better 🧠

1. Outline Expansion Pattern

The **Outline Expansion Pattern** helps you work around the size limits of large language models by generating content **step by step, based on an outline**.

Instead of asking the model to write a full document all at once, you first ask it to build a **structured outline**. Then, you ask it to expand one bullet point at a time.

💡 **Example prompt:**

```
- example 1
Act as an outline expander. Generate a bullet point outline based on the input that I give
you and then ask me for which bullet point you should expand on.
Each bullet can have at most 3-5 sub bullets. The bullets should be numbered using the
pattern [A-Z].[i-v]. Create a new outline for the bullet point that I select.
At the end, ask me for what bullet point to expand next. Ask me for what to outline.
```

This pattern is useful for:

- Writing long documents (e.g. articles, handbooks, books)
- Creating structured code or project plans
- Generating consistent pieces that fit together

This pattern gives you a map to follow — and lets the model fill in the pieces as you go.

2. Recipe Pattern

The Recipe Pattern is used when you **know part of a solution**, but not the whole thing. You give the model the **goal** and a few **known steps**, and ask it to **fill in the missing parts**.

Think of it like making a recipe: You have a few ingredients or steps, and the model completes the rest.

> 👉 **I would like to achieve X**
>
> **I know that I need to perform steps A,B,C**
>
> **Provide a complete sequence of steps for me**
>
> **Fill in any missing steps**

💡 **Example prompt:**

```
- I would like to  purchase a house. I know that I need to perform steps make an offer and
close on the house. Provide a complete sequence of steps for me. Fill in any missing
steps.

- I would like to drive to NYC from Nashville. I know that I want to go through Asheville,
NC on the way and that I don't want to drive more than 300 miles per day. Provide a
complete sequence of steps for me. Fill in any missing steps.
```

✅ You can use this pattern when:

- You know **some** of the steps, but not all.
- You want the model to **fill in a process**, **suggest missing pieces**, or **bridge gaps**.

You can combine it with other patterns, like **Meta Language Creation**, to express the known and unknown steps in a compact way.

3. Cognitive Verifier Pattern

The Cognitive Verifier Pattern helps the model answer complex questions more accurately by **breaking them into smaller sub-questions**, answering each one, and combining the results.

> 👉 **When asked a question, generate helpful sub-questions. Then combine their answers into a final answer.**

This works well because:

- Language models have seen many **related questions** in their training.
- Sub-questions help the model **think more clearly** and reduce mistakes.
- You get **more transparency** into how the answer was produced.

💡 **Example prompt:**

```
Examples:

- When you are asked a question, follow these rules. Generate a number of additional
questions that would help you more accurately answer the question. Combine the answers to
the individual questions to produce the final answer to the overall question.

Tailored Examples:

- When you are asked to plan a trip, follow these rules. Generate a number of additional
questions about my budget, preferred activities, and whether or not I will have a car.
Combine the answers to these questions to better plan my itinerary.
```

Use this pattern when:

- The original question is **broad or vague**
- You want **better logic and transparency**
- You want to **guide the model through a reasoning process**

This pattern is great for learning, planning, and answering open-ended or fuzzy questions.

4. Chain of Thought Pattern

Chain of Thought (CoT) Prompting improves the model's reasoning by asking it to **show its thinking step by step**, before giving an answer. This is similar to how we were taught in school to "show your work." When a model explains its reasoning, it's more likely to reach a correct and logical answer.

> 👉 Give examples that include both the **reasoning** and the **final answer**.
>
> Ask the model to follow the same format when answering your new question.

💡 **Example comparison:**

```
❌ Without CoT:

Is there anything on the floor in a zero-gravity spaceship?
Answer: Yes — the cup and needle fell to the floor.

✅ With CoT:

Reasoning: In zero gravity, objects don't fall. They float.
Answer: No — there's nothing "on the floor" in a zero-gravity environment.
```

That's a better answer, because the model followed a reasoning chain.

🧠 Why it works:

- Step-by-step reasoning helps the model **stay on track**.
- If it gets the early steps right, the later steps often follow correctly.
- This leads to more accurate answers in tasks that require **logic, calculation, or common sense**.

Use this pattern when:

- The task involves **math, logic, physics**, or multiple steps.
- You want the model to be **transparent** and avoid guessing.
- You want to teach the model a **thinking process**, not just an answer.

Even just **asking for reasoning first** can greatly improve the quality of output.

> 💡 **Tip**
>
> For more details, you can read papers 🔗: https://arxiv.org/abs/2201.11903

5. Flipped Interaction Pattern

The **Flipped Interaction Pattern** tells the model to **ask you questions**, instead of you asking it.

This is helpful when:

- You don't know what to ask.
- You want the model to gather details from you before giving a result.
- You want to be quizzed or guided through a process.

> 👉 **Ask me questions about [topic] until you have enough information to [goal]. Ask me the first question.**

💡 **Example Prompt:**

```
- I would like you to ask me questions to help me create variations of my marketing
materials.  You should ask questions until you have sufficient information about my
current draft messages, audience, and goals. Ask me the first question.

- I would like you to ask me questions to help me diagnose a problem with my Internet. Ask
me questions until you have enough information to identify the two most likely causes. Ask
me one question at a time. Ask me the first question.
```

After collecting enough answers, the model provides the full result — in this case, a **personalized training plan**.

Use this pattern when you need **smart question-asking** before getting to a final answer.

6. Alternative Approaches Pattern

The Alternative Approaches Pattern is used to help the model **brainstorm multiple ways** to solve a problem or complete a task — not just give one answer. Instead of saying “do this,” you say:

> 👉 **List alternative ways to achieve this goal.**

💡 **Example Prompt:**

```
- For every prompt I give you, If there are alternative ways to word a prompt that I give
you, list the best alternate wordings . Compare/contrast the pros and cons of each
wording.

- For anything that I ask you to write, determine the underlying problem that I am trying
to solve and how I am trying to solve it. List at least one alternative approach to solve
the problem and compare / contrast the approach with the original approach implied by my
request to you.
```

Use this pattern when:

- You want to explore ideas
- You're unsure which prompt design will work best

You can also combine this pattern with others (like **Ask for Input**) to make the interaction smoother and more controlled.

2.3 Manage Output Structure and Accuracy ✍️

1. Template Pattern

The **Template Pattern** helps you get model outputs in the **exact format you want**, by giving it a **template with placeholders** to fill in. Think of it like giving instructions to an assistant:

> 👉 **Here’s the format — now fill it in.**

💡 **Example Prompt:**

```
example 1:

- Create a random strength workout for me today with complementary exercises. I am going
to provide a template for your output . CAPITALIZED WORDS are my placeholders for content.
Try to fit the output into one or more of the placeholders that I list. Please preserve
the formatting and overall template that I provide. This is the template: NAME, REPS @
SETS, MUSCLE GROUPS WORKED, DIFFICULTY SCALE 1-5, FORM NOTES

example 2:

- Please create a grocery list for me to cook macaroni and cheese from scratch, garlic
bread, and marinara sauce from scratch. I am going to provide a template for your output .
<placeholder> are my placeholders for content. Try to fit the output into one or more of
the placeholders that I list. Please preserve the formatting and overall template that I
provide.

This is the template:
Aisle <name of aisle>:
<item needed from aisle>, <qty> (<dish(es) used in>
```

Use this pattern when:

- You need **clean, consistent formatting**
- You're building tools, content, forms, or structured data
- You want precise control over output style

It's one of the most reliable ways to **get what you want, where you want it** in the output.

2. Semantic Filter Pattern

The Semantic Filter Pattern helps you use a language model to **remove or keep certain types of information** from a piece of text — based on meaning, not just keywords.It's like saying:

> 👉 **Act as a filter. Remove anything that fits this description.**

This is powerful for cleaning up sensitive or unwanted content, such as:

- Removing **dates** from a public version of a document
- Stripping out **personal or medical details**
- Keeping only parts of text that match a specific meaning

💡 Examples:

```
- Filter this information to remove any personally identifying information or information
that could potentially be used to re-identify the person.

- Filter this email to remove redundant information.
```

⚠️ Note: This is a powerful tool, but not perfect. For privacy or security use cases, always have a **human double-check the output.**

3. Ask for Input Pattern

The **Ask for Input Pattern** is a simple way to tell the model:

> 👉 **Wait for my instruction before doing anything.**

Use this when:

- You want to stay in control of the session
- You're designing tools, games, or multi-turn workflows
- You want the model to wait for **user input**

Without this pattern, the model might **invent its own task** and generate an answer immediately — even if you haven't given it anything yet.

💡 **Example Prompt:**

```
- From now on, I am going to cut/paste email chains into our conversation. You will
summarize what each person's points are in the email chain. You will provide your summary
as a series of sequential bullet points. At the end, list any open questions or action
items directly addressed to me. My name is Jill Smith. Ask me for the first email chain.
```

✅ With this pattern:

> Model replies:
> *"What is the first email you'd like me to summarize for?"*

❌ Without this pattern:

> Model might make something up:
> *"Sure! Let's start with summarising email chain…"*

This small trick gives you clean, predictable results — and avoids wasted output.

4. Fact Check List Pattern

The **Fact Check List Pattern** helps you identify and verify the key facts a language model includes in its output — instead of just assuming the text is correct. The idea is simple:

> 👉 **After you generate a response, list the core facts it contains.**

These are the facts you can then check yourself or with other tools.

💡 **Prompt format:**

```
- Whenever you output text, generate a set of facts that are contained in the output. The
set of facts should be inserted at the end of the output. The set of facts should be the
fundamental facts that could undermine the veracity of the output if any of them are
incorrect.
```

You can then review the facts, compare them to the full text or follow up on any that seem doubtful.

This pattern **doesn't guarantee correctness**, but it:

- Helps you **spot questionable claims**
- Builds a habit of **active reading and verification**
- Makes it easier to **review AI-generated content responsibly**

You can use this pattern when accuracy matters — like in education, journalism, decision-making, or planning.

5. Tail Generation Pattern

The Tail Generation Pattern helps the model **remember the goal of a task across long conversations**. It works by attaching a short reminder — a "tail" — at the **end of every output**.

💡 **Example prompt:**

```
- Act as an outline expander. When I give you a bullet point, expand it into sub-points.
At the end of your output, ask me which bullet point you should expand next.
```

"ask me which bullet point you should expand next" is the `tail`.

This prompt will remind the model to keep output `which bullet point you should expand next ?`That ensures that it doesn't forget the task and will keep going.

You can use this pattern when:

- You have a **repeating structure** across turns
- You want to avoid the model "drifting" from the task
- You need to **remind the model what to expect next**
- Reinforces **instructions without repeating full prompts**

The tail is like a **smart sticky note** — small, but keeps everything on track.

2.4 Enable Complex and Interactions 🔁

1. ReAct Prompting

ReAct Prompting (short for **Reasoning + Acting**) teaches the model to solve a problem by combining **step-by-step thinking** with **external actions**, like web searches or tool calls.

It’s useful when a task:

- The model needs fresh data or real-time info
- You want to simulate a multi-step reasoning process
- You're integrating LLMs with external tools or APIs

> 👉 **How it works:**
>
> 1. The model **thinks** about what it needs
> 2. It **calls an action** (like `SEARCH`, `VIDEO`, or `CALCULATE`)
> 3. It uses the **result** to keep reasoning
> 4. It repeats until the task is done

💡 **Prompt format:**

```
- example 1

Task: Figure out when my son’s BMX race starts.

Think: I need to find the race start time.
Action: SEARCH https://musiccitybmx.com
Result: All races start at 9:00 AM

Think: I need to know how many races come before his.
Action: SEARCH https://musiccitybmx.com
Result: 30 motos before his

Think: I need to know how long each moto takes.
Action: VIDEO https://usabmx.com/stream
Result: First 10 motos took 5 minutes

Think: 30 motos = 15 mins → Arrive by 8:45 AM
```

Even if the Action is not actually done by LLMs, you can do those actions in other pipelines and give feedback on the result.

> **Tip**
>
> For more details, you can read papers 🔗: https://arxiv.org/abs/2210.03629

2. Menu Actions Pattern

The Menu Actions Pattern lets you create a custom set of **reusable commands** (like a menu) for a language model to follow — just like clicking options in software.

Instead of repeating long instructions every time, you define a format once, and reuse it with short commands.

> 👉 How it works:

> 1. You define a menu of actions.
> 2. Each action follows a **specific keyword + structure**.
> 3. The model understands what to do when it sees your custom command.

💡 **Example Prompt:**

```
- example 1

Whenever I type: "add FOOD", you will add FOOD to my grocery list and update my estimated
grocery bill.
Whenever I type "remove FOOD", you will remove FOOD from my grocery list and update my
estimated grocery bill.
Whenever I type "save" you will list alternatives to my added FOOD to save money. At the
end, you will ask me for the next action.
Ask me for the first action.
```

You can use this pattern when:

- You perform **repetitive prompt tasks**
- You want to build a **tool-like workflow** using prompts
- You need consistent behavior across a process or team

Think of it as building your own "program" inside a conversation — one keyword at a time.

3. Using LLM to Grade Each Other

This pattern uses large language models to **evaluate and score** the output of prompts — either their own output or another model's. It solves a real problem: You need a way to **check if the outputs still meet your standards**, especially at scale.

> 👉 **How it works:**
>
> 1. Give the model an input + output pair.
> 2. Provide a few examples of what makes an answer good or bad.
> 3. Ask the model to grade future outputs using that same structure.

💡 **Example Prompt:**

```
- example 1

Input: A paragraph about Vanderbilt University
Output: "Vanderbilt University, 1873"
Explanation: Correctly extracts the event and date
Grade: 10/10

Output: "1873"
Explanation: Date is correct, but missing the event
Grade: 3/10
```

You can use this pattern, especially if you want to:

- Reduces the need for human reviewers
- Works across different models or changing data
- Helps track prompt quality over time
- Can be combined with retry logic or thresholds (e.g., only retry if score < 7)

This pattern is like building a **feedback loop** into your prompt engineering workflow — making the system more self-aware and maintainable.

4. Combining Patterns

Combining Patterns means using two or more prompt patterns together to solve more complex tasks. This is one of the most important skills in advanced prompt engineering.

💡 **Example combos:**

Case 1: For example when using LLM to grade each other, you can combine with:

- **Few-shot Examples** (to teach scoring)
- **Persona Pattern** (e.g., "Act as a prompt critic", make it more critical thinking.)

Case 2: When you have no idea which options should be asked, you can design as:

- **Alternative Approaches** (gives multiple options)
- **Ask for Input** (asks you to pick)

Case 3: You want LLM to assist you in writing a paper outlines

- **Outline Expansion** (Build outline for you)
- **Menu Actions** (define the action to expand or write each section)

> 💡 **Tip**
>
> If you are insteresting, you can read relevant papers
>
> 🔗: https://arxiv.org/abs/2302.11382
>
> 🔗: https://arxiv.org/pdf/2303.07839

3. Prompt Structure in Code 👨🏻‍💻

When using a language model in code (e.g., with `Ollama` or `OpenAI`), prompts are passed in using a **list of messages**. Each message has a **role** and **content**.

In OpenAI SDK, Each message follows this format:

```
messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Tell me a joke."}
]
```

There are three roles:

- `"system"` — sets the behavior and personality of the assistant
- `"user"` — the actual input/question from the user
- `"assistant"` — can show prior model responses if you want to simulate chat history

Here's a complete example showing how to structure and manage prompts in code using a multi-turn conversation.

```
# from dotenv import load_dotenv
import os
import openai

# load_dotenv()
# API_KEY = os.getenv('QWEN_KEY')
# Initialize OpenAI client for OPEN AI

client = openai.OpenAI(
    api_key='xxxxx',
    base_url="http://localhost:11434/v1"
)

messages = [
    {
        "role": "system",
        "content": "You are Qwen2.5, created by Alibaba Cloud. You are a helpful assistant who always responds concisely and clearly."
    },
    {
        "role": "user",
        "content": "Can you explain what a large language model is, like I'm five years old?"
    }
  # ......
]

# Make the first API call using the new syntax
response = client.chat.completions.create(
    model="qwen2.5:1.5b",
    messages=messages
)

reply = response.choices[0].message.content
print("Assistant:", reply)
```

```python
# append assistant's answers to messages to form a conversation history
messages.append({"role": "assistant", "content": reply})

# then do the second round of questions ...
user_input = "Can you tell me a fun fact about that city?"
messages.append({"role": "user", "content": user_input})

#Second invocation of the model (the context already contains the historical dialogue)
response = client.chat.completions.create(
    model="qwen2.5:1.5b",
    messages=messages
)

# The second round of answers
reply = response.choices[0].message.content
print("Assistant:", reply)

# append assistant's answers again
messages.append({"role": "assistant", "content": reply})
```