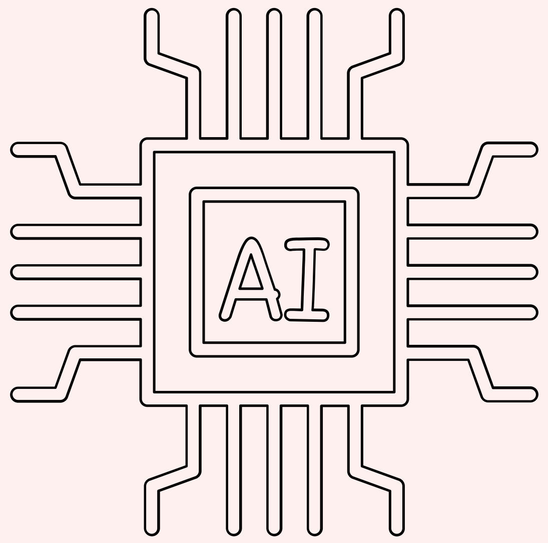
ARTIFICIAL INTELLIGENCE
DAY 2

Prepared by TARSOFT SDN BHD

TABLE OF CONTENTS

OBJECTIVE

Participants can integrate Ollama with code, understand prompt engineering, and apply AI in real systems.



Day 1 Recap

- AI vs traditional software (AI adapts, while traditional software follows fixed rules).
- How LLMs are trained on large datasets and can generate text similar to human writing.
- How Ollama allows you to run these models locally without relying on cloud systems.

Ollama API – Concept & Endpoint Structure

What is an API?

- An API (Application Programming Interface) is a way for different software systems to communicate. Imagine Ollama as the brain, and your application (whether a website or mobile app) as the body. The brain doesn't speak directly to the user — instead, it communicates through an API, which acts as a messenger.

- For Ollama, the API provides endpoints. Think of an endpoint as a specific "door" that you knock on to get information. The most important endpoint in Ollama is /api/generate. This is the door you use when you want Ollama to generate text.

Ollama API – Concept & Endpoint Structure

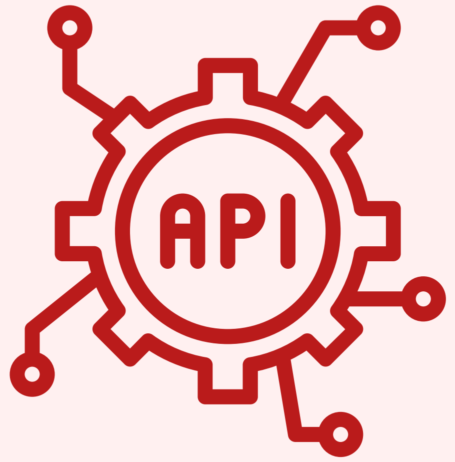
Ollama API?

When you want to interact with Ollama via the API, you send a request. The request tells Ollama which model to use and what the prompt is. The response is the generated text.

How Ollama Works in API Mode ?
For applications use (Python, JS, Laravel, Flutter, etc.), you need Ollama to behave as a server that listens for requests.

This is where **ollama serve** comes in.

- ollama run → for direct interaction in terminal
- ollama serve → for running Ollama as an API service that applications can call

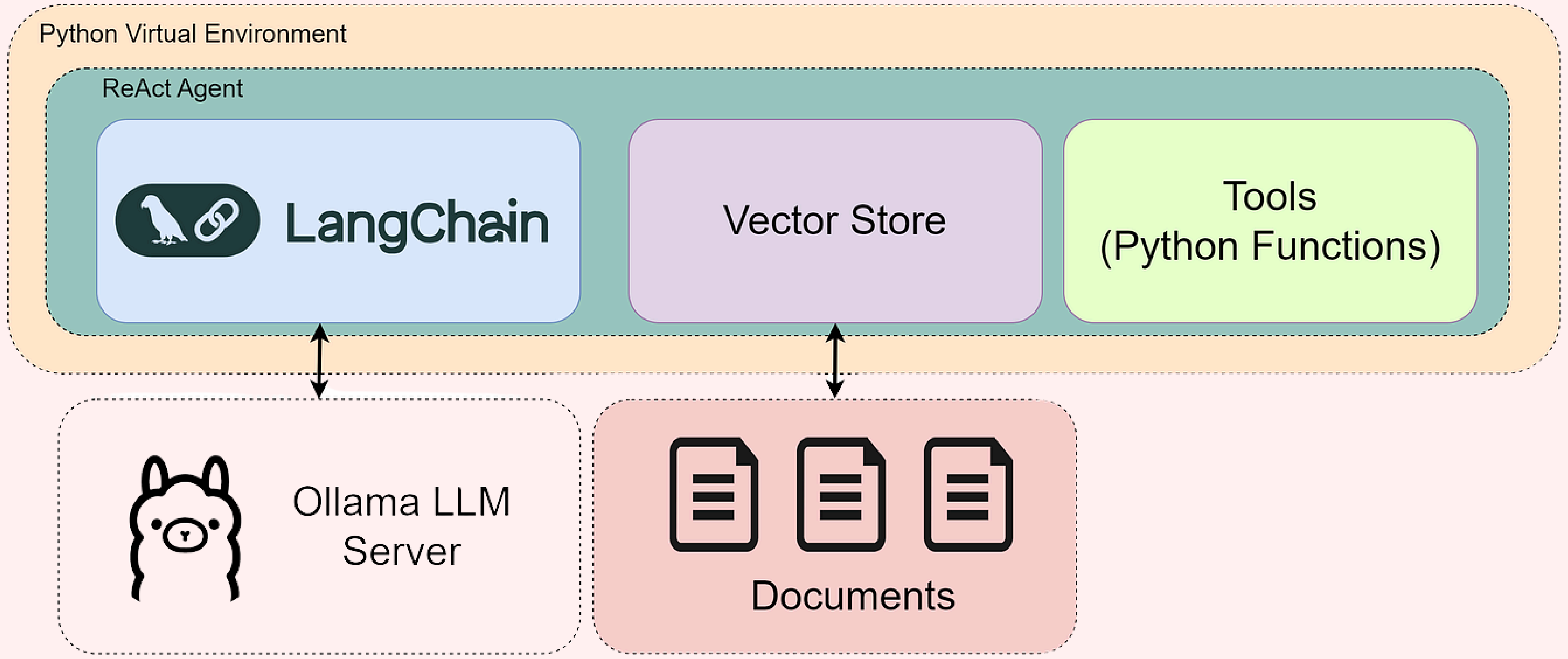


Ollama API – Concept & Endpoint Structure

Ollama API?

By default:

- Ollama runs at **http://localhost:11434**
- The main endpoint is **/api/generate**
- Any app (Python, JavaScript, etc.) can now send HTTP requests to Ollama



Ollama API – Concept & Endpoint Structure

Ollama API?

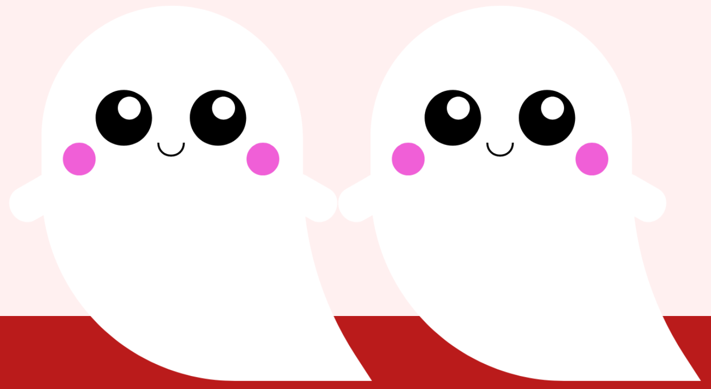
Once you start ollama serve, test if it works using curl:

```
curl http://localhost:11434/api/generate -d '{
"model": "llama3",
"prompt": "Hello, how are you?",
“Stream”: false }'
```

Expected Response (JSON):

```
{
 "model": "llama3",
 "created_at": "2025-09-07T10:00:00Z",
 "response": "I am doing well, thank you!
 How can I help you today?"
}
```

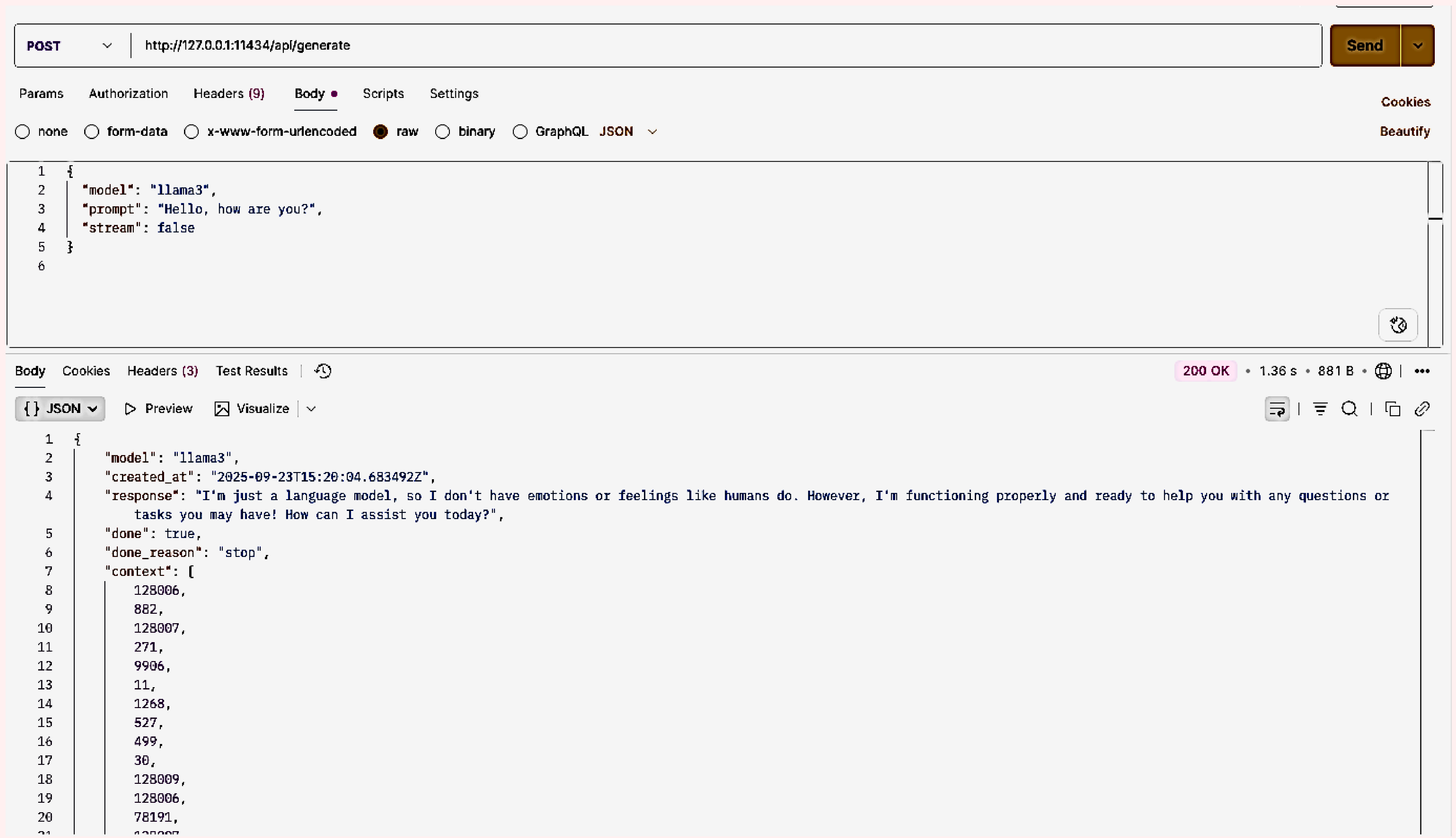
This proves the Ollama API server is active and ready for integration.



Ollama API – Concept & Endpoint Structure

Ollama API?

Instead of using curl, we can use postman as alternative:



POST http://127.0.0.1:11434/api/generate

Body: none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
{
  "model": "llama3",
  "prompt": "Hello, how are you?",
  "stream": false
}
```

Body — 200 OK • 1.36 s • 881 B — { } JSON

```
{
    "model": "llama3",
    "created_at": "2025-09-23T15:20:04.683492Z",
    "response": "I'm just a language model, so I don't have emotions or feelings like humans do. However, I'm functioning properly and ready to help you with any questions or
        tasks you may have! How can I assist you today?",
    "done": true,
    "done_reason": "stop",
    "context": [
        128006,
        882,
        128007,
        271,
        9906,
        11,
        1268,
        527,
        499,
        30,
        128009,
        128006,
        78191,
```

Ollama API – Concept & Endpoint Structure

Ollama API?

COMMON ISSUES & FIXES

When running ollama serve, participants may face some issues:

- **Port already in use**
  - Error: "Address already in use: 11434"
    Fix: Stop the process using the port, or change the port. Example:

```
OLLAMA_HOST=127.0.0.1:12345 ollama serve
```

- **Firewall blocking requests**
  - Make sure firewall allows incoming requests to port 11434.
- **Running on remote server**
  - By default, Ollama binds to localhost.
    If you want remote access:

```
OLLAMA_HOST=0.0.0.0:11434 ollama serve
```

  - This makes Ollama accessible to other devices on the network.

Integrating Ollama with Code

Integration with Python

Python is one of the most popular languages for AI projects because of its simplicity and wide range of libraries. Using Python, we can call Ollama's API and display the response.

Example:

```python
import requests

url = "http://localhost:11434/api/generate"
data = {
    "model": "llama3",
    "prompt": "Explain quantum computing in simple terms."
}

response = requests.post(url, json=data)
print(response.json()["response"])
```

Integrating Ollama with Code

Integration with Python

Explanation:

1. We import the requests library, which allows us to send HTTP requests.
2. We define the Ollama API URL and specify the model and prompt.
3. We send a POST request with the prompt.
4. We print the AI’s response.

Integrating Ollama with Code

Integration with JS

JavaScript is widely used in web development, so integrating Ollama with JavaScript allows us to add AI features directly to websites or web apps.

Example :

```
const url = "http://localhost:11434/api/generate";

async function askOllama(prompt) {
  const response = await fetch(url, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ model: "llama3", prompt })
  });
  const data = await response.json();
  console.log(data.response);
}

askOllama("Give me 3 startup ideas in fintech.");
```

Integrating Ollama with Code

Integration with JS

Explanation:

1. fetch() sends a request to the Ollama API.
2. The request includes the model and the prompt.
3. The response is read in JSON format.
4. Finally, we print the AI’s response in the console.

Hands-on 1

Participants now practice calling Ollama from Python or JavaScript.

Task:

- Write a script that sends a question to Ollama and displays the response.
- Example prompts:
  - “Summarize today’s top 3 news headlines.”
  - “Explain blockchain in one paragraph.”

Expected Outcome: Participants will see answers printed on their terminal or browser console. This builds confidence in making API calls.

Prompt Engineering

What is Prompt Engineering

Prompt engineering is the art of writing better instructions for AI. A prompt is simply what you ask the AI. But the way you ask greatly affects the answer you get.

- Prompt engineering is the process of designing inputs (prompts) to guide AI models like LLaMA or Mistral to produce accurate, relevant, and useful outputs.
- Unlike traditional programming, where logic is explicitly coded, in AI we influence behavior by how we “ask” the question.

- Example:
  - Poor prompt: “Tell me about cats.”
  - Better prompt: “You are a veterinary expert. Explain the dietary needs of cats in simple language suitable for a pet owner.”

This difference leads to better control and reduced hallucinations.

Fine-Tuning Prompts

What is Fine-Tuning Prompts

There are different strategies to refine prompts:

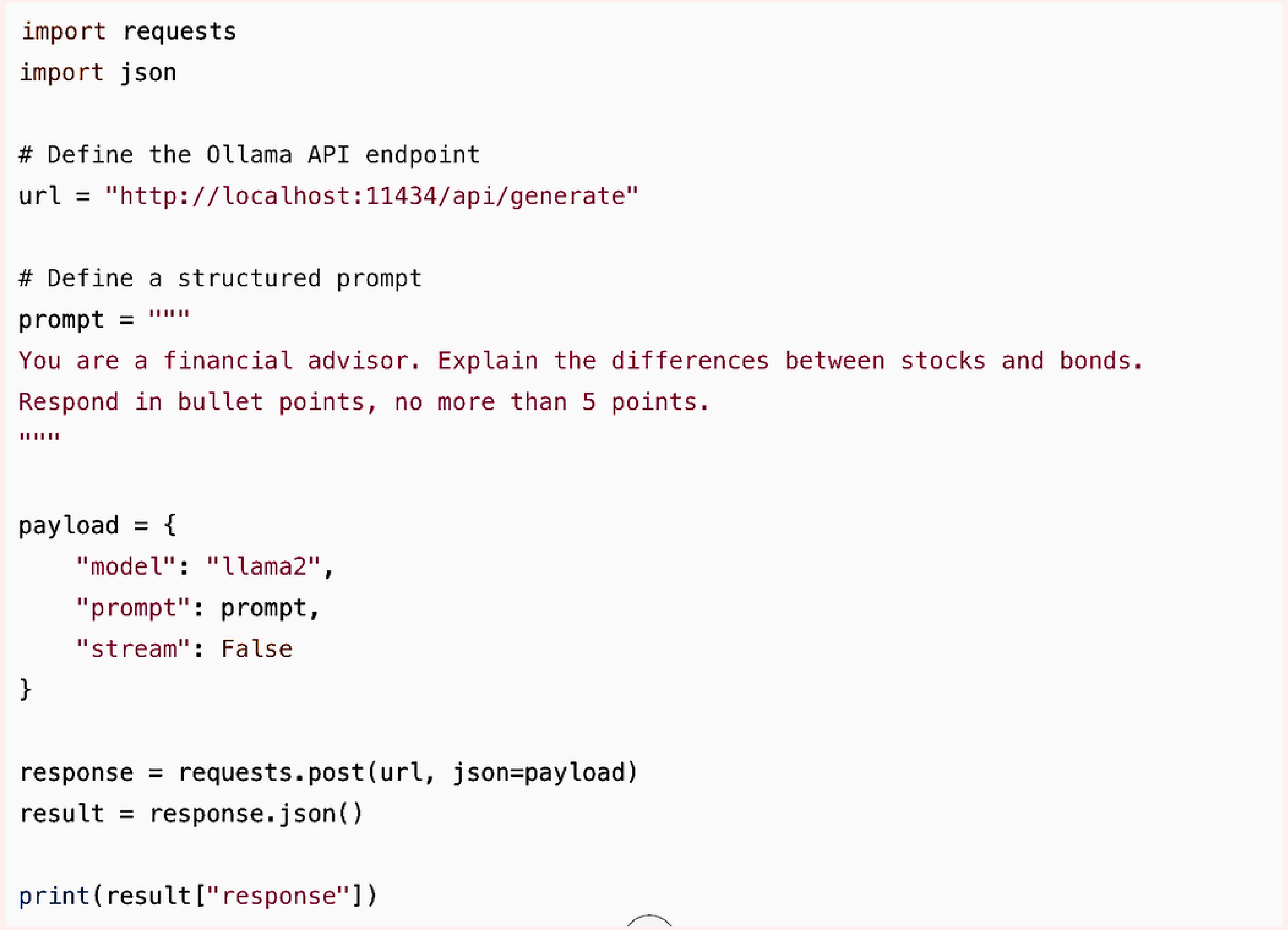
- **Role-based prompts:** “You are a doctor explaining medical advice.”
- **Format-specific prompts:** “Answer in JSON format with ‘title’ and ‘content’.”
- **Multi-step prompts:** “First explain the concept in simple terms. Then give a real-world example. Finally, summarize in 3 bullet points.”

These techniques help us build AI applications that are consistent and reliable.

Prompt Engineering

What is Prompt Engineering

Prompt Engineering with Ollama (Python Example)



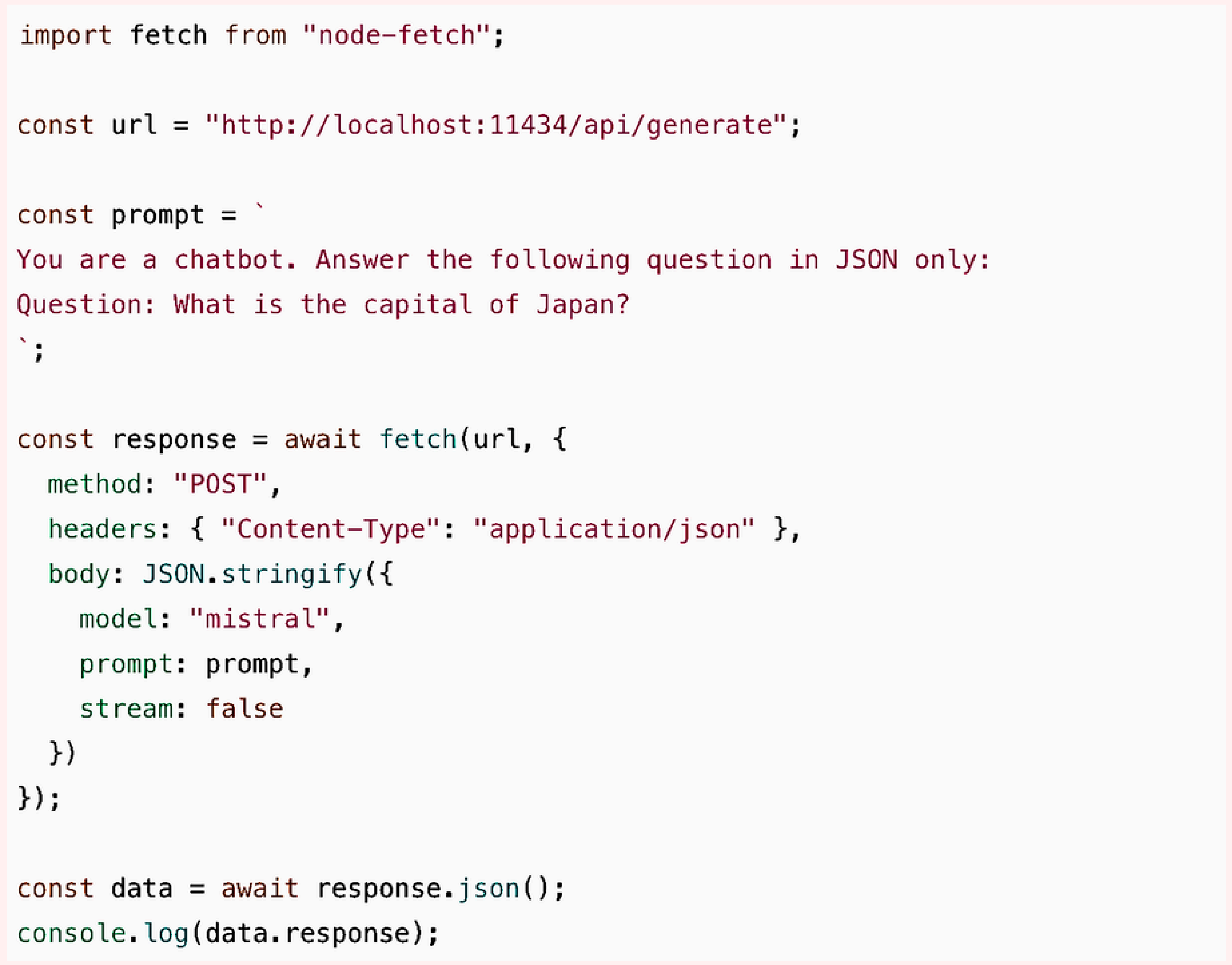
```python
import requests
import json

# Define the Ollama API endpoint
url = "http://localhost:11434/api/generate"

# Define a structured prompt
prompt = """
You are a financial advisor. Explain the differences between stocks and bonds.
Respond in bullet points, no more than 5 points.
"""

payload = {
    "model": "llama2",
    "prompt": prompt,
    "stream": False
}

response = requests.post(url, json=payload)
result = response.json()

print(result["response"])
```

Prompt Engineering

What is Prompt Engineering

Prompt Engineering with Ollama (JS Example)

- Ensures the model responds in structured JSON, useful for integration into apps or databases.



```
import fetch from "node-fetch";

const url = "http://localhost:11434/api/generate";

const prompt = `
You are a chatbot. Answer the following question in JSON only:
Question: What is the capital of Japan?
`;

const response = await fetch(url, {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({
    model: "mistral",
    prompt: prompt,
    stream: false
  })
});

const data = await response.json();
console.log(data.response);
```

Prompt Engineering

What is Prompt Engineering

Using System Prompts for Consistency

System prompts = permanent instructions at the **start of every interaction.**

Example:

```
system_prompt = """
You are an English teacher. Always explain in simple language suitable for beginners.
Correct grammar mistakes and give examples.
"""


payload = {
    "model": "gemma",
    "prompt": system_prompt + "\nUser: Can you help me write better English?",
    "stream": False
}
```

Hands-On 2

Create **3 different prompts** for a **Customer Support AI.**

Examples:

1. "Answer like a customer service agent in a polite and formal tone."
2. "Respond in casual language, limited to 2 sentences."
3. "Provide a troubleshooting guide in numbered steps."

Goal: Compare the different outputs and see how prompt structure changes the response.

RAG (Retrieval Augmented Generation)

What is RAG(Retrieval Augmented Generation)?

Definition:

RAG combines **retrieval-based methods** (fetching relevant documents/data) with **generative models** (LLMs) to provide more accurate and context-aware answers.

Why RAG?

- LLMs have knowledge limits (cut-off dates).
- Reduces hallucination by grounding answers with real data.
- Allows private/custom knowledge integration.

RAG (Retrieval Augmented Generation)

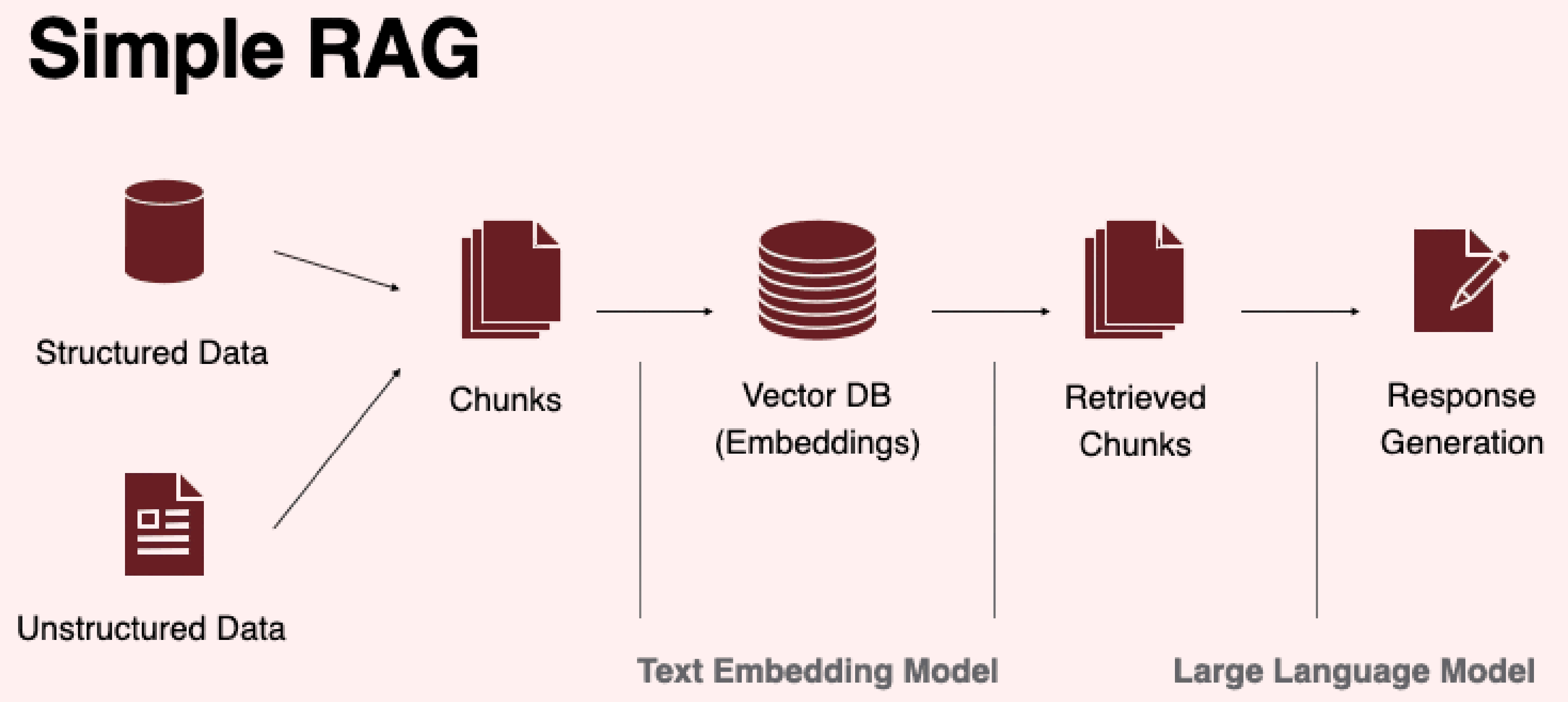
How RAG Works

Step 1: User sends query.

Step 2: System retrieves relevant documents from database/vector store.

Step 3: Retrieved context is added into the LLM prompt.

Step 4: LLM generates answer using both context + pre-trained knowledge.



RAG (Retrieval Augmented Generation)

Key Components of RAG

Retriever

- Finds relevant documents from dataset.
- Uses embeddings + vector similarity search.

Vector Store

- Stores documents as embeddings.
- Examples: **FAISS, Chroma, Pinecone.**

Generator (LLM)

- Uses retrieved docs to generate natural language output.

RAG (Retrieval Augmented Generation)

Example Use Cases

- **FAQ Bot:** Answer customer queries with internal knowledge.
- **Document Search:** Summarize and explain PDFs.
- **Data Analysis Assistant:** Retrieve statistics or KPIs from reports.
- **E-Government System:** Answer policy or regulation queries using government documents.

RAG (Retrieval Augmented Generation)

Example: Python + Ollama

```python
from ollama import chat
from chromadb import Client

# Step 1: Connect to Chroma (vector DB)
chroma = Client()

# Step 2: Retrieve relevant documents
query = "What are the requirements for cat hotel booking?"
results = chroma.query(query_texts=[query], n_results=3)

# Step 3: Build context
context = " ".join([doc['text'] for doc in results['documents']])

# Step 4: Pass query + context into Ollama
response = chat(model="llama3", messages=[
    {"role": "system", "content": "You are a cat booking assistant."},
    {"role": "user", "content": f"Answer based on context: {context}\n\nQuestion: {query}"
])

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

RAG (Retrieval Augmented Generation)

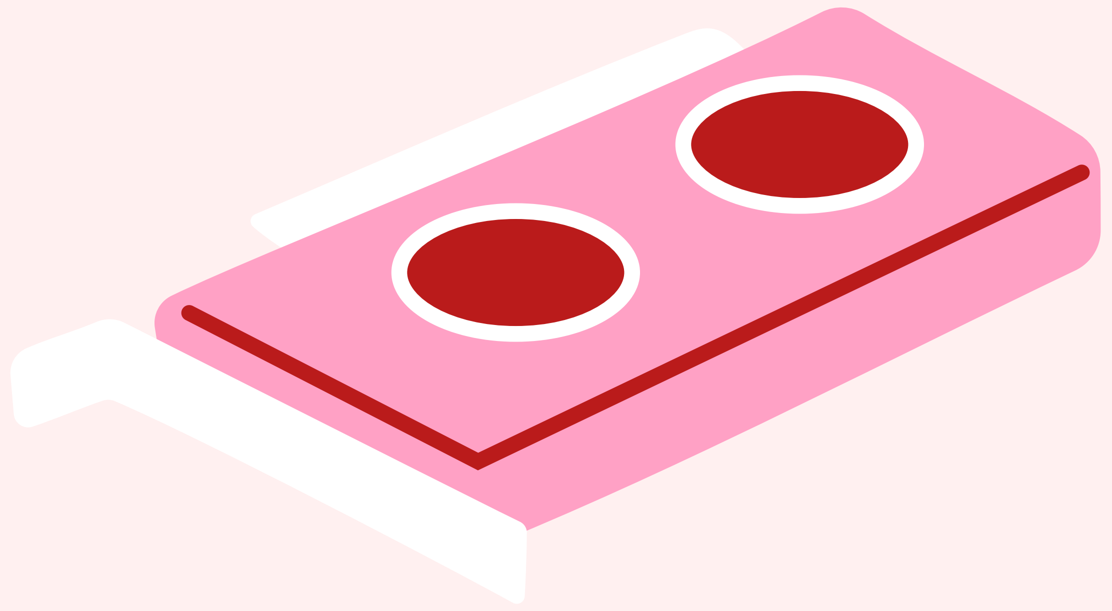
Why RAG is Better than Fine-Tuning

Fine-tuning

- Expensive, requires GPUs.
- Must retrain for each new dataset.

RAG

- No retraining needed.
- Update knowledge instantly.
- Works with small hardware (even laptops).



Ollama Embeddings

What are Embeddings?

Embeddings are **numbers that represent meaning** of words, sentences, or documents.

They let computers understand **similarity** between texts (e.g., “dog” is closer to “puppy” than “car”).

Think of it as **putting text on a map**, where similar things are near each other.

Ollama Embeddings

Why Use Embeddings in RAG?

RAG needs a way to **find the most relevant documents** from your knowledge base.

Embeddings make it possible to:

- Compare questions with stored documents.
- Retrieve only the **most related chunks.**
- Improve accuracy of AI answers.

Ollama Embeddings

Ollama Embeddings

Ollama supports generating embeddings with many models.

You can use them to:

- Build **semantic search** (find info by meaning, not exact keywords).
- Power **recommendation systems.**
- Connect RAG pipelines easily.

Ollama will:

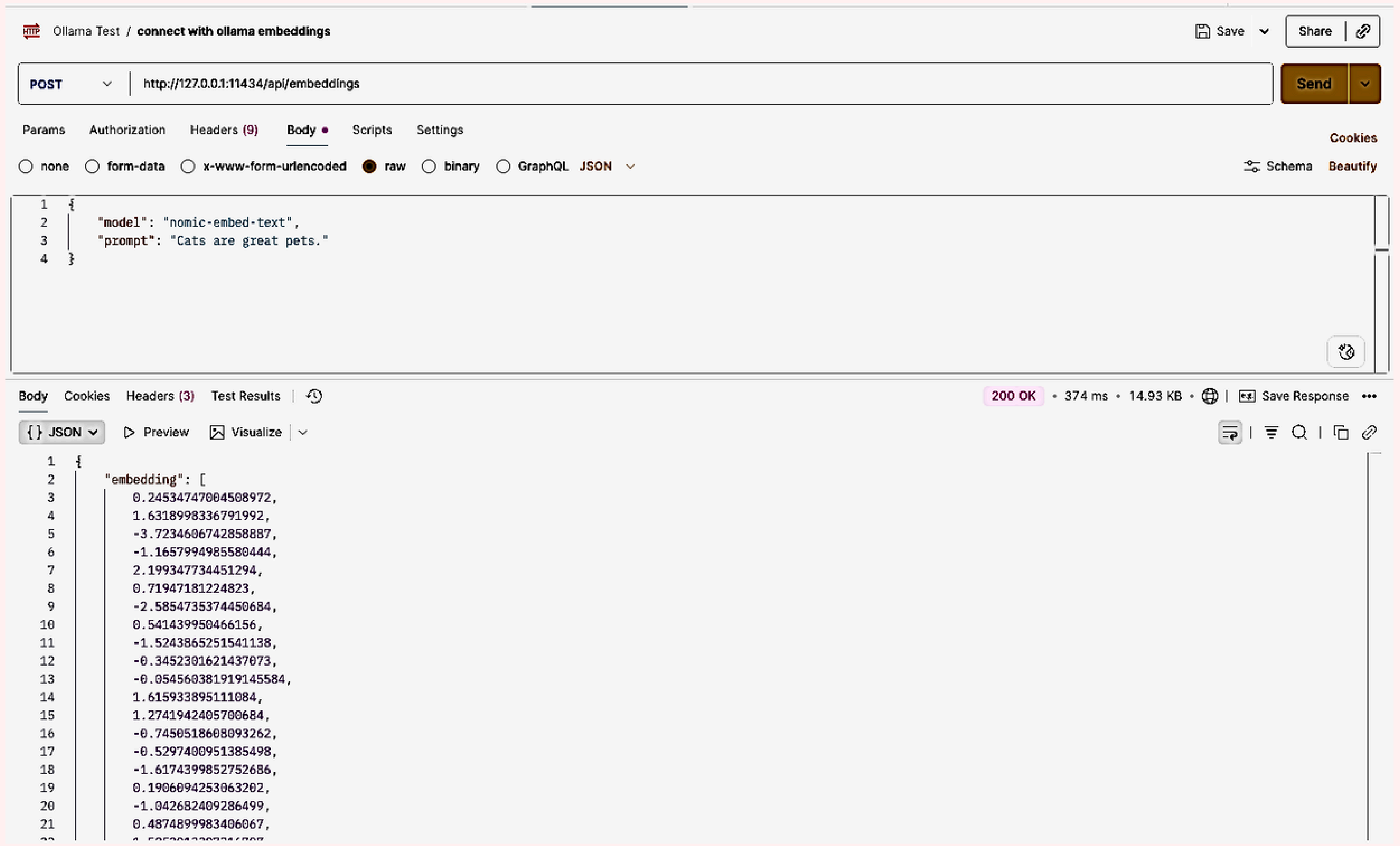
1. Use the nomic-embed-text model.
2. Generate an embedding vector for the input "Cats are great pets.".
3. Return the result as a JSON response with a long list of floating-point numbers (the embedding).

Ollama Embeddings

Ollama Embeddings

Output:

- Returns a long list of numbers (vector).
- These numbers represent the meaning of the text.


Ollama Test / connect with ollama embeddings

POST http://127.0.0.1:11434/api/embeddings

```
{
    "model": "nomic-embed-text",
    "prompt": "Cats are great pets."
}
```

200 OK • 374 ms • 14.93 KB

```
{
    "embedding": [
        0.24534747004508972,
        1.6318998336791992,
        -3.7234606742858887,
        -1.1657994985580444,
        2.199347734451294,
        0.71947181224823,
        -2.5854735374450684,
        0.541439950466156,
        -1.5243865251541138,
        -0.3452301621437073,
        -0.054560381919145584,
        1.615933895111084,
        1.2741942405700684,
        -0.7450518608093262,
        -0.5297400951385498,
        -1.6174399852752686,
        0.1906094253063202,
        -1.042682409286499,
        0.4874899983406067,
```

Ollama Embeddings

Why we use nomic-embed-text in Ollama?

1. Purpose of embeddings

- Embeddings convert text into **vectors (lists of numbers)** that capture the *meaning* of the text.
- Example:
  - "cat" and "kitten" will have vectors that are *close together*.
  - "cat" and "car" will be farther apart.
- This is the foundation for **search, clustering, recommendation, FAQ retrieval, and RAG (Retrieval Augmented Generation)**.

2. **nomic-embed-text specifically**

- It's a lightweight open-source embedding model included with Ollama.
- It is optimized for **text similarity** tasks like:
  - FAQ bots (find closest answer).
  - Semantic search (search based on meaning, not exact words).
  - Document chunking for RAG.
- Output: usually **768-dimensional vectors** (size depends on the embedding model).

3. **Why not use a generation model (llama3, mistral) for embeddings?**

- Generation models are trained to **generate text**, not produce good embeddings.
- They *can* produce embeddings with special techniques, but they are slower, heavier, and not optimized for this.
- nomic-embed-text is small, fast, and specifically designed for embeddings.

Ollama Embeddings

Alternatives to embeddings

Model Name	Dimension Size	Speed (relative)	Strengths	Weaknesses	Best Use Case
nomic-embed-text	~768	⚡ Fast (lightweight)	Default model in Ollama, simple setup, good semantic similarity	Not state-of-the-art accuracy, English-focused	FAQ bots, small RAG projects, quick demos
all-minilm	~384	⚡⚡ Very Fast	Great for short sentences, low memory usage	Slightly weaker with long documents	Semantic search, chatbot retrieval
gte-base / gte-large	768–1024	⚡ Medium	Higher accuracy, well-balanced for RAG	Heavier, slower on laptops	Document retrieval, enterprise RAG
bge-m3	1024	⚡ Medium	Multilingual support (works in English, Malay, Chinese, etc.), strong semantic	Higher resource needs	Multilingual FAQ bots, global apps
OpenAI text-embedding-3-small/large	1536	Cloud (depends on API)	State-of-the-art, very accurate, large dataset	Requires internet, $$$ cost per use	Production-level search, recommendation, enterprise RAG

Ollama Embeddings

Which one should you choose?

It depends on your **use case**:

- **FAQ chatbot / small app:** nomic-embed-text (fast + good enough).
- **Search in large document sets:** gte-base or bge-m3.
  Multilingual support: bge-m3.
- **If you're resource-limited** (running on laptop): stick with nomic-embed-text.
- **If you want best accuracy & don't mind cost** external API like OpenAI text-embedding-3-large.

Data Analysis with AI

What is Data Analysis with AI?

Traditional data analysis:

- Uses SQL, Excel, or BI tools (Power BI, Tableau).
- Requires predefined queries and dashboards.

AI-powered data analysis:

- Can **understand natural language** queries.
- Can **summarize insights** automatically.
- Example: Instead of writing SQL → ask AI “Show me sales trends by region.”

Data Analysis with AI

AI Use Cases in Data Analysis

Exploratory Data Analysis (EDA):

- AI can describe dataset, detect missing values, highlight patterns.

Report Automation:

- Automatically generate executive summaries of datasets.

Prediction & Forecasting:

- AI can suggest trends (e.g., “Sales are likely to dip next quarter”).

Local Malaysian Example:

- Finance: AI explains anomalies in monthly transactions.
- Government: AI highlights suspicious activity in subsidy claims.
- Healthcare: AI auto-summarizes patient data for doctors.

Data Analysis with AI

Summarizing Data with Ollama

sales.csv

Region	Month	Sales
KL	Jan	12000
Penang	Jan	8000
Johor	Jan	9500

```python
import pandas as pd
import requests

# Load dataset
df = pd.read_csv("sales.csv")

# Convert dataframe to string for AI
data_text = df.to_string(index=False)

OLLAMA_URL = "http://localhost:11434/api/generate"

def analyze_data(data_text):
    prompt = f"Analyze this sales data and summarize key insights:\n\n{data_text}"
    payload = {"model": "llama2", "prompt": prompt, "stream": False}
    response = requests.post(OLLAMA_URL, json=payload)
    return response.json()["response"]

summary = analyze_data(data_text)
print("AI Summary:\n", summary)
```

In January, KL achieved the highest sales (12,000), followed by Johor (9,500). Penang showed the lowest sales (8,000). KL contributes 40% of the total sales.

Data Analysis with AI

Visual Analysis with AI Assistance

- **Traditional:** Use matplotlib/Power BI to make charts.
- **AI-assisted:** AI can suggest the right type of chart.

Example prompt:
“Given this dataset, suggest the best visualization and explain why.”

AI might answer:

- “Use a bar chart to compare sales by region.”
- “Use a line chart to track monthly growth.”

Automating Reports with AI

Automation with Ollama

Instead of manually writing monthly reports → AI can generate them.

```
def generate_report(df):
    data_text = df.describe().to_string()
    prompt = f"Write a business report based on this summary statistics:\n\n{data_text}"
    payload = {"model": "mistral", "prompt": prompt, "stream": False}
    response = requests.post(OLLAMA_URL, json=payload)
    return response.json()["response"]

report = generate_report(df)
print("Generated Report:\n", report)
```

Example Output:
"The dataset shows strong sales performance with an average of RM10,500. KL consistently outperforms other regions, suggesting high market demand. Penang shows potential for improvement."

Python Backend API Connecting to Ollama

Let’s say we want to build a **backend service** that frontend apps (React, Flutter, Vue, etc.) can call.

We’ll use **FastAPI** as the backend framework.

Need to Install Requirements

```
pip install fastapi uvicorn requests
```

AI in Web & Mobile Systems

```python
from fastapi import FastAPI
from pydantic import BaseModel
import requests

# Define the FastAPI app
app = FastAPI()

# Define request body for incoming JSON
class ChatRequest(BaseModel):
    user_input: str

# Ollama API endpoint (local running server)
OLLAMA_URL = "http://localhost:11434/api/generate"

@app.post("/chat")
def chat_with_ollama(request: ChatRequest):
    # Construct payload for Ollama
    payload = {
        "model": "llama2",   # can be mistral, gemma, etc.
        "prompt": request.user_input,
        "stream": False
    }

    # Send request to Ollama
    response = requests.post(OLLAMA_URL, json=payload)
    result = response.json()

    # Return the model's response to frontend
    return {"response": result["response"]}
```

How It Works

1. **Frontend (React / Flutter / Mobile App)**
   - Sends a POST request with user input → POST /chat

Example payload:

```
{ "user_input": "Explain quantum computing in simple terms" }
```

2. **Backend (FastAPI)**
   - Receives the request.
   - Calls **Ollama API** running locally (http://localhost:11434/api/generate).
   - Passes the user's input as the prompt.

3. **Ollama Server**
   - Runs the model (e.g., LLaMA 2, Mistral).
   - Returns AI-generated response.

4. **Backend → Frontend**
   - The backend **formats and sends back** the response in JSON.
   - **Example:**

```
{ "response": "Quantum computing uses qubits..." }
```

AI in Web & Mobile Systems

Why Use a Backend API for Frontend Apps?

Security

- If frontend apps call Ollama directly, they expose the **model endpoint** and potentially sensitive prompts.
- Backend acts as a **secure layer** between the model and users.

Scalability

- You can deploy Ollama on a **powerful server with GPU/CPU**, while frontend users just need internet access.
- Multiple apps (web, iOS, Android) can share the same backend.

Control & Customization

- Backend can add:
  - **Authentication** (only logged-in users can use AI).
  - **Rate limiting** (avoid overload).
  - **Custom RAG** pipelines (inject knowledge base).
  - **Logging** (store prompts & responses for analysis).

Consistency Across Platforms

- Web app, mobile app, and admin dashboard all connect to **one backend API.**
- Ensures all use the **same AI behavior and model.**

AI in Web & Mobile Systems

Frontend Calling the API

```javascript
const response = await fetch("http://localhost:8000/chat", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ user_input: "Tell me a joke" })
});

const data = await response.json();
console.log(data.response);
```

React (JavaScript)

```dart
final response = await http.post(
  Uri.parse("http://localhost:8000/chat"),
  headers: {"Content-Type": "application/json"},
  body: jsonEncode({"user_input": "Tell me a joke"}),
);

if (response.statusCode == 200) {
  final result = jsonDecode(response.body);
  print(result["response"]);
}
```

Flutter (Dart)

With this setup:

- **Frontend stays lightweight** → only sends/receives JSON.
- **Backend handles AI logic** → prompts, RAG, formatting, restrictions.
- **Ollama runs on server** → heavy computation happens in one place.

Q&A & Day 2 Recap

API basics (concept, request, response).

Integration with Python & JavaScript.

Prompt engineering & fine-tuning.

Real-world integration (Web + Mobile).

Hands-on projects.

TARSOFT SDN BHD

Thank You

Any Enquiries?

tarsoft.com.my