Language Models API

Lesson 10 & 11
RAG

Introduction to OpenAI APIs

OpenAI provides a collection of programmable interfaces that allow artificial intelligence features to be integrated into software applications. These interfaces expose core AI tasks such as text generation, speech processing, vision-based image generation, and vector embeddings.

In total, five categories of APIs are offered. Each category corresponds to a particular capability area, and understanding the purpose of each API helps in choosing the correct tool for chatbot development, content generation, recommendation tasks, or transcription systems.

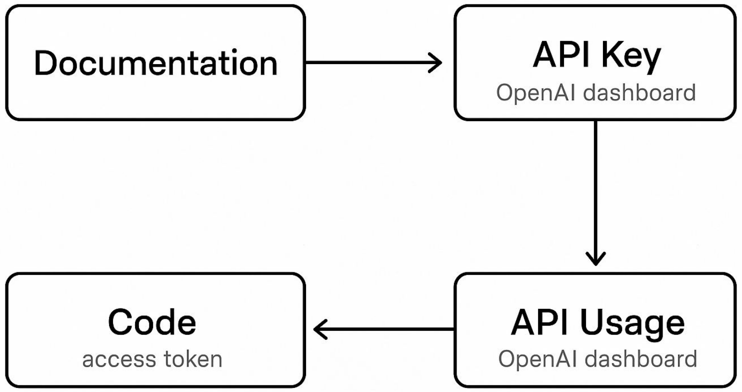
- The five API categories covered in the slides are:
  Chat or Completion API
- Embedding API
- DALL-E API
- Whisper API
- and Fine-tuning API.

Access Requirements for Using the APIs

Before initiating any interaction with the APIs, an access token is required. This token functions as an authentication key, permitting secure communication between a user's program and the OpenAI servers.

The key is created under the API Keys section of the OpenAI platform dashboard. The usage of the API is also viewed under the dashboard in the Usage section. These two dashboard areas are essential for both setup and monitoring.

The OpenAI documentation homepage provides the starting point for understanding how the APIs work.



Base Code Setup (Google Colab or VS Code)

A minimal Python setup is sufficient to start working with the APIs. The openai package is installed first. After installation, the access token is assigned to the openai.api_key attribute.

Once the key is set, the application is ready to send requests for any capability such as chat generation, embeddings, or image creation.

This foundational snippet serves as the starting point for all API interactions.

Next slide we will take a look at a sample code snippet for the same.

Sample Code Snippet

```python
# Install the package (run this in your terminal first):

# pip install openai

from openai import OpenAI

# Create a client using your API key

client = OpenAI(api_key="YOUR_API_KEY_HERE")

# Example: Send a simple chat request

response = client.chat.completions.create(

    model="gpt-4.1",

    messages=[

        {"role": "user", "content": "Hello, can you summarise the purpose of this API?"}

    ]

)

print(response.choices[0].message["content"])
```

This snippet:

1. Installs and imports the openai package.
2. Sets the API key directly when creating the OpenAI client.
3. Demonstrates a minimal chat completion request using a current model.

Chat API or Completion API

The Chat or Completion API enables conversational artificial intelligence. It allows natural language responses to be generated and supports conversations that continue across multiple turns.

Context from earlier messages in the same session can be recognised, enabling interactions that resemble human conversation.

This capability is commonly used in chatbot systems, customer support workflows, and virtual assistance tools.

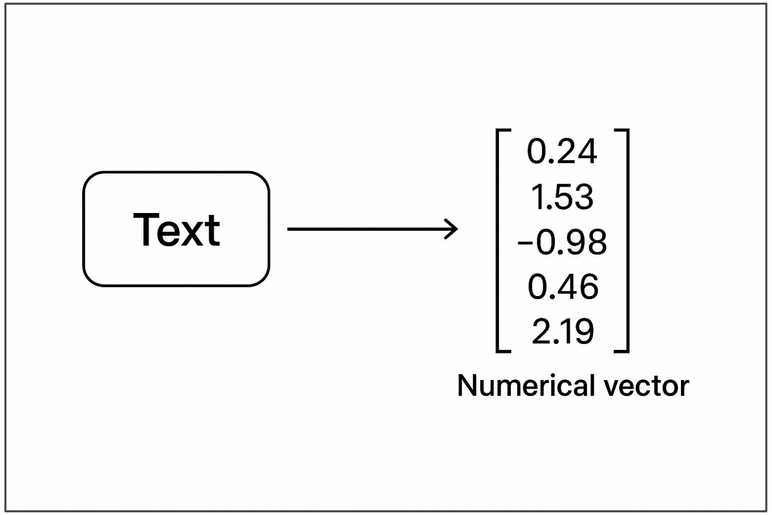
Models such as gpt-3.5-turbo, gpt-4, and gpt-4o-mini are examples of model families that can be invoked for such tasks.

A short scenario: When a customer asks about an order, receives an answer, and then asks a related question, the follow-up can be understood within the same context rather than as an unrelated message.

Embedding API

The Embedding API converts text into high-dimensional vectors that represent meaning. These vectors make similarity matching and semantic comparison possible, since related sentences or documents tend to appear close to each other in vector space.

This capability supports information retrieval, sentiment classification, recommendation systems, and search applications that rely on meaning rather than simple keywords.



DALL-E API

The DALL-E API generates images from textual descriptions. When a description such as “a sunset behind a modern city skyline” is provided, the system produces a visual representation of that description.

This is applied in creative content generation, advertising material development, and early prototyping where rapid illustration is useful.

The focus here is that text is interpreted to produce images, enabling visual output directly from descriptive language.

Whisper API

Whisper enables the conversion of spoken audio into text. It performs transcription and can also support translation across multiple languages.

High accuracy is achieved for clear audio, making it suitable for lecture transcription, meeting documentation, or accessibility-oriented captioning.

For example, an audio recording of a classroom explanation can be processed to produce a readable transcript.

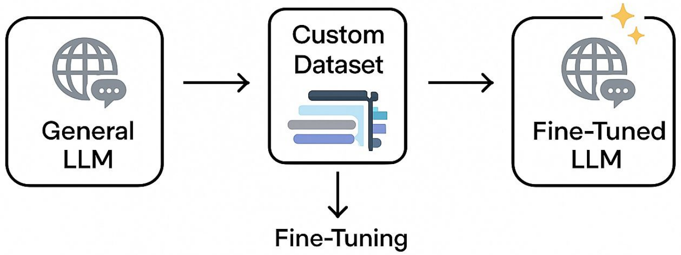
Fine Tuning API

Purpose of Fine-tuning

Fine-tuning is used to adapt an existing model such as GPT-4o-mini using a custom dataset so that the model performs specialised tasks more effectively.

The process allows domain-specific behaviour to be added, making the resulting model more suitable for focused applications.

Examples mentioned include customer service bots designed for a specific organisation, educational systems shaped for a particular curriculum, and specialised AI tools created for industry-specific requirements.



Pre-requisite for Fine-tuning

A dataset must be prepared before fine-tuning begins.
The dataset is required in JSONL (JSON Lines) format where each line is an object containing two fields:

- **prompt** containing the input text
- and **completion** containing the expected output text.

An example:

```
{"prompt": "Is the dissertation project a group project or an individual project",

"completion": "It has to be an individual project. Group projects are not allowed."}
```

This format ensures that the fine-tuning process can clearly identify what response should be produced for each input pattern.

Steps in Fine-tuning an LLM

Fine-tuning follows a sequence of steps. These steps involve:

1. A pre-trained model such as GPT-4o-mini is selected.
2. A relevant dataset is gathered in JSONL format.
3. The dataset is pre-processed to remove errors or inconsistencies.
4. The fine-tuning procedure is run to train the selected model on the prepared dataset.
5. The resulting model adapts its behaviour to the target task while retaining the language knowledge learned during its original training.

This sequence produces a focused model that performs reliably within its intended domain.

Rivet

Rivet functions as an open source visual programming environment designed for building AI agents powered by large language models. It provides a space where prompt logic can be organised visually, making it easier to understand how different parts of an AI agent interact.

Prompt graphs can be created, modified, and refined directly inside Rivet. Once these prompt graphs reach the desired structure, they can be executed within an application without rewriting the logic elsewhere. This creates a smoother workflow for experimenting with prompt structures and testing their behaviour.

Rivet supports teamwork by enabling groups to design, debug, and collaborate on complex prompt graphs. This is valuable when multiple contributors need to refine the behaviour of an AI agent or inspect why a particular prompt chain produces a certain output.

After the design process, these prompt graphs can be deployed in an organisation's own environment, keeping the system consistent with internal requirements.

A simple diagram showing interconnected prompt nodes flowing into an AI agent would help to visualise how Rivet manages complex LLM prompt logic.

Retrieval Augmented Generation (RAG)

Introduction and Core Idea

Retrieval Augmented Generation represents an approach in which retrieval mechanisms and generative models are combined. The intention is to improve how AI systems understand queries and produce human-like text.

RAG relies on the strengths of two components. Retrieval mechanisms bring relevant information from stored sources, and generative models then use that information to form coherent responses.

This combination enables responses that stay grounded in external knowledge rather than relying only on the model’s internal training.

Why is Retrieval-Augmented Generation important?

LLMs are a key **artificial intelligence (AI)** technology powering intelligent chatbots and other **natural language processing (NLP)** applications. The goal is to create bots that can answer user questions in various contexts by cross-referencing authoritative knowledge sources. Unfortunately, the nature of LLM technology introduces unpredictability in LLM responses. Additionally, LLM training data is static and introduces a cut-off date on the knowledge it has.

Known challenges of LLMs include:

- Presenting false information when it does not have the answer.
- Presenting out-of-date or generic information when the user expects a specific, current response.
- Creating a response from non-authoritative sources.
- Creating inaccurate responses due to terminology confusion, wherein different training sources use the same terminology to talk about different things.

You can think of the **Large Language Model** as an over-enthusiastic new employee who refuses to stay informed with current events but will always answer every question with absolute confidence.

Unfortunately, such an attitude can negatively impact user trust and is not something you want your chatbots to emulate!

RAG is one approach to solving some of these challenges. It redirects the LLM to retrieve relevant information from authoritative, pre-determined knowledge sources. Organizations have greater control over the generated text output, and users gain insights into how the LLM generates the response.

Key Terminologies in RAG

Prompt

A prompt refers to the user’s query. It initiates the entire process and guides the model’s behaviour.

Context

Context refers to internal or external information used to construct the response. Internal context corresponds to the model’s built-in knowledge. External context refers to documents or data retrieved at runtime.

LLM

The term LLM refers to large language models such as GPT-3.5, GPT-4, GPT-4o, GPT-4o-mini, or LLama3. These models generate the final answers once context is supplied.

Retriever

A retriever searches a knowledge base and selects information relevant to the user’s query. This step ensures the LLM receives only the most suitable context.

Embedding

An embedding is a numerical representation of text. It captures meaning in a vector form so that related pieces of text appear closer together. A diagram of vectors distributed in a space would assist understanding.

Vector Store

A vector store is a database that stores embeddings. It supports similarity search so that relevant chunks can be located efficiently when queries arrive.

What are the benefits of Retrieval-Augmented Generation?

RAG technology brings several benefits to an organization's generative AI efforts.

Cost-effective implementation

Chatbot development typically begins using a foundation model. **Foundation models** (FMs) are API-accessible LLMs trained on a broad spectrum of generalized and unlabeled data. The computational and financial costs of retraining FMs for organization or domain-specific information are high. RAG is a more cost-effective approach to introducing new data to the LLM. It makes generative artificial intelligence (generative AI) technology more broadly accessible and usable.

Current information

Even if the original training data sources for an LLM are suitable for your needs, it is challenging to maintain relevancy. RAG allows developers to provide the latest research, statistics, or news to the generative models. They can use RAG to connect the LLM directly to live social media feeds, news sites, or other frequently-updated information sources. The LLM can then provide the latest information to the users.

Enhanced user trust

RAG allows the LLM to present accurate information with source attribution. The output can include citations or references to sources. Users can also look up source documents themselves if they require further clarification or more detail. This can increase trust and confidence in your generative AI solution.

More developer control

With RAG, developers can test and improve their chat applications more efficiently. They can control and change the LLM's information sources to adapt to changing requirements or cross-functional usage. Developers can also restrict sensitive information retrieval to different authorization levels and ensure the LLM generates appropriate responses. In addition, they can also troubleshoot and make fixes if the LLM references incorrect information sources for specific questions. Organizations can implement generative AI technology more confidently for a broader range of applications.

How does Retrieval-Augmented Generation work?

Without RAG, the LLM takes the user input and creates a response based on information it was trained on or what it already knows. With RAG, an information retrieval component is introduced that utilizes the user input to first pull information from a new data source. The user query and the relevant information are both given to the LLM. The LLM uses the new knowledge and its training data to create better responses. The following sections provide an overview of the process.

Create external data

The new data outside of the LLM's original training data set is called *external data*. It can come from multiple data sources, such as a APIs, databases, or document repositories. The data may exist in various formats like files, database records, or long-form text. Another AI technique, called *embedding language models*, converts data into numerical representations and stores it in a vector database. This process creates a knowledge library that the generative AI models can understand.

Retrieve relevant information

The next step is to perform a relevancy search. The user query is converted to a vector representation and matched with the vector databases. For example, consider a smart chatbot that can answer human resource questions for an organization. If an employee searches, *"How much annual leave do I have?"* the system will retrieve annual leave policy documents alongside the individual employee's past leave record. These specific documents will be returned because they are highly-relevant to what the employee has input. The relevancy was calculated and established using mathematical vector calculations and representations.

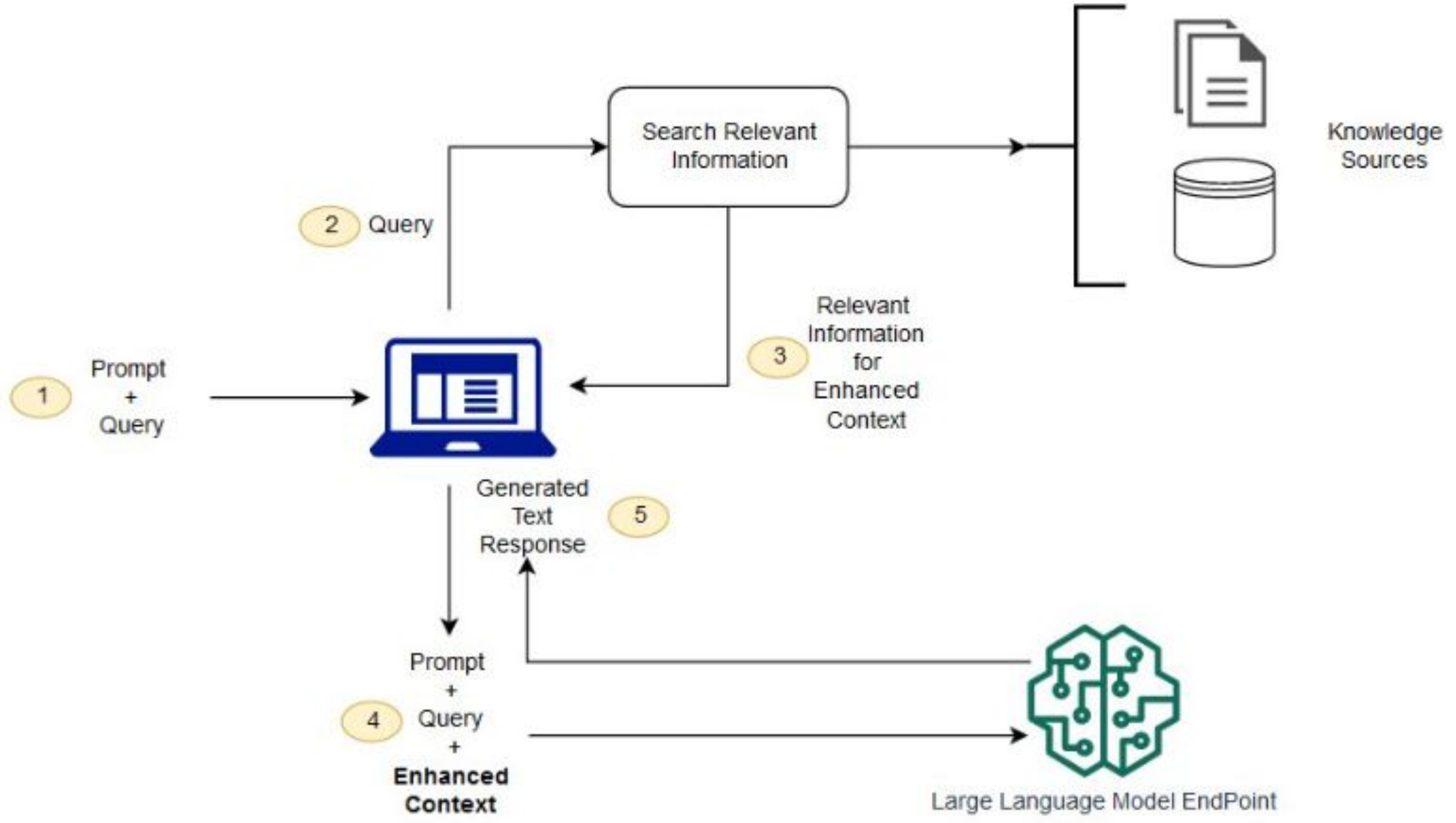
Augment the LLM prompt

Next, the RAG model augments the user input (or prompts) by adding the relevant retrieved data in context. This step uses prompt engineering techniques to communicate effectively with the LLM. The augmented prompt allows the large language models to generate an accurate answer to user queries.

Update external data

The next question may be—what if the external data becomes stale? To maintain current information for retrieval, asynchronously update the documents and update embedding representation of the documents. You can do this through automated real-time processes or periodic batch processing. This is a common challenge in data analytics—different data-science approaches to change management can be used.

The following diagram shows the conceptual flow of using RAG with LLMs.


Search Relevant Information
Knowledge Sources
2 Query
1 Prompt + Query
3 Relevant Information for Enhanced Context
Generated Text Response 5
4 Prompt + Query + Enhanced Context
Large Language Model EndPoint

What is the difference between Retrieval-Augmented Generation and semantic search?

Semantic search enhances RAG results for organizations wanting to add vast external knowledge sources to their LLM applications. Modern enterprises store vast amounts of information like manuals, FAQs, research reports, customer service guides, and human resource document repositories across various systems. Context retrieval is challenging at scale and consequently lowers generative output quality.

Semantic search technologies can scan large databases of disparate information and retrieve data more accurately. For example, they can answer questions such as, *"How much was spent on machinery repairs last year?"* by mapping the question to the relevant documents and returning specific text instead of search results. Developers can then use that answer to provide more context to the LLM.

Conventional or keyword search solutions in RAG produce limited results for knowledge-intensive tasks. Developers must also deal with word embeddings, document chunking, and other complexities as they manually prepare their data. In contrast, semantic search technologies do all the work of knowledge base preparation so developers don't have to. They also generate semantically relevant passages and token words ordered by relevance to maximize the quality of the RAG payload.

Types of Embeddings

There are several embedding approaches available such as:

- OpenAI embeddings suitable for general NLP tasks
- BERT embeddings that capture semantic meaning
- Sentence Transformer embeddings optimised for sentence and document similarity
- DPR (Dense Passage Retriever) embeddings used frequently in open-domain question answering
- GloVe embeddings that capture semantic relationships but lack contextual awareness

Vector Stores

There are several options used in RAG systems:

- Pinecone, a managed vector database for scalable similarity search
- FAISS, an open-source library designed for fast large-scale vector search
- Weaviate, a vector database with semantic search capabilities
- Milvus, a high-performance open-source vector database
- ElasticSearch with KNN plugin for vector-based similarity inside text indices