…

Getting started with LLM prompt engineering

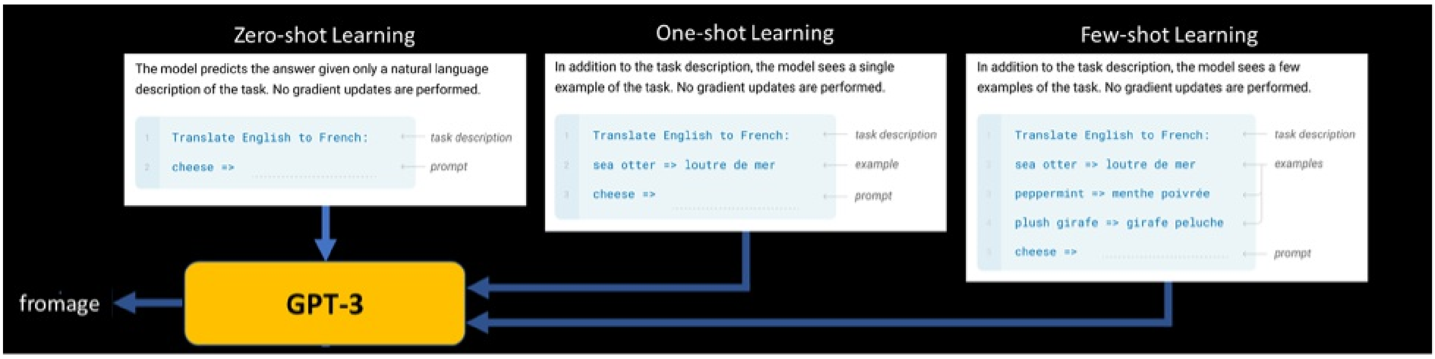
Article •

•

4 contributors

Feedback

Large Language Models (LLMs) have the ability to learn new tasks on the fly, without requiring any explicit training or parameter updates. This mode of using LLMs is called in-context learning. It relies on providing the model with a suitable input prompt that contains instructions and/or examples of the desired task. The input prompt serves as a form of conditioning that guides the model's output, but the model does not change its weights. In-context learning can be applied in different settings such as zero-shot, one shot, or few-shot learning. It depends on the amount of information that needs to be included in the input prompt.



Zero-shot Learning: The model predicts the answer given only a natural language description of the task. No gradient updates are performed.

```
Translate English to French:
cheese =>
```

One-shot Learning: In addition to the task description, the model sees a single example of the task. No gradient updates are performed.

```
Translate English to French:
sea otter => loutre de mer
cheese =>
```

Few-shot Learning: In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.

```
Translate English to French:
sea otter => loutre de mer
peppermint => menthe poivrée
plush girafe => girafe peluche
cheese =>
```

The process of designing and tuning the natural language prompts for specific tasks, with the goal of improving the performance of LLMs is called **prompt engineering**.

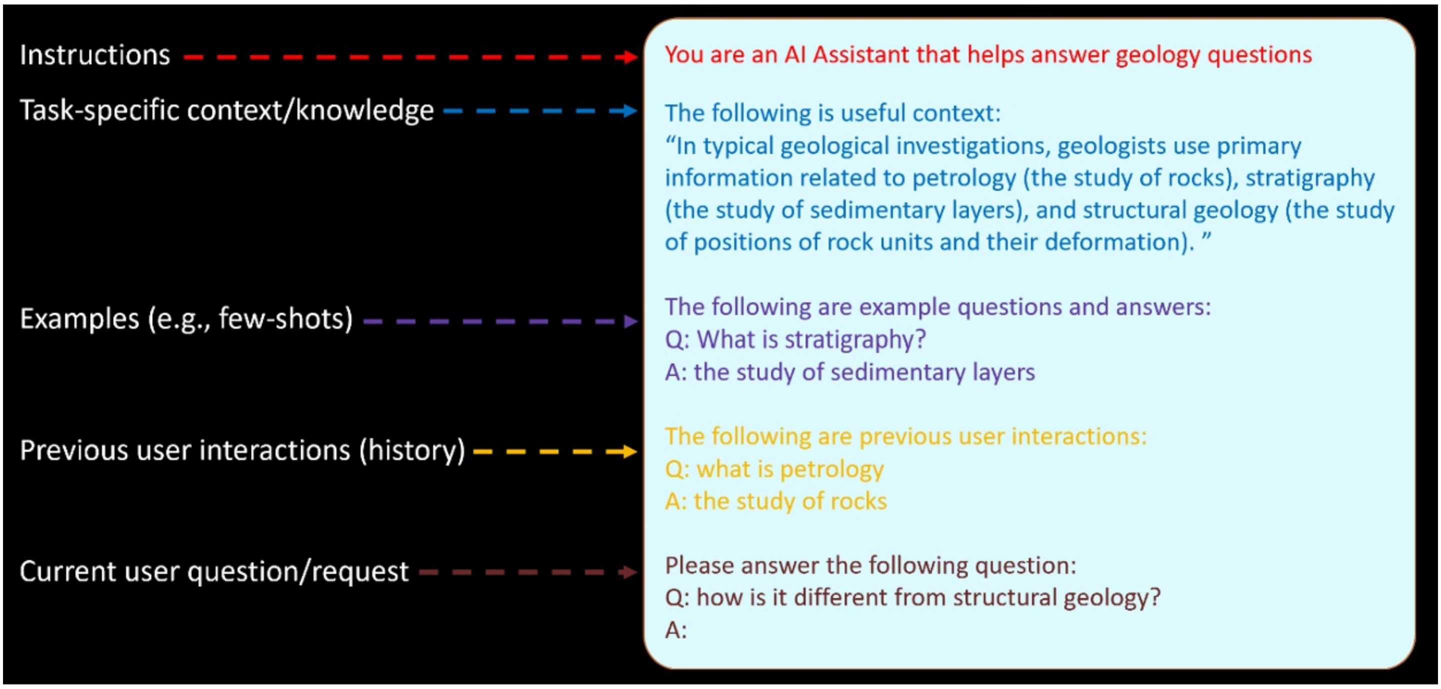
Effective prompt engineering can significantly improve the performance of LLMs on specific tasks. It is done by providing instructions and contextual information that help guide the model's output. By carefully designing prompts, researchers can steer the LLM's attention toward the most relevant information for a given task, leading to more accurate and reliable outputs.

Prompt engineering can also help mitigate the problem of "catastrophic forgetting," where an LLM may forget previously learned information during fine-tuning for a new task. By using carefully designed prompts, the model can retain relevant knowledge while still adapting to new tasks.

Prompt components

While it is considered a new field, rich literature is already available, including articles, blogs, research papers, repos, etc., about prompt engineering techniques.

A common technique is to construct prompts from a well-defined set of components, as shown in the following diagram.



Instructions

Task-specific context/knowledge

Examples (e.g., few-shots)

Previous user interactions (history)

Current user question/request

You are an AI Assistant that helps answer geology questions

The following is useful context:
"In typical geological investigations, geologists use primary information related to petrology (the study of rocks), stratigraphy (the study of sedimentary layers), and structural geology (the study of positions of rock units and their deformation). "

The following are example questions and answers:
Q: What is stratigraphy?
A: the study of sedimentary layers

The following are previous user interactions:
Q: what is petrology
A: the study of rocks

Please answer the following question:
Q: how is it different from structural geology?
A:

Instructions and other static context

Static context description refers to providing fixed information to the LLM. This information can include content and format instructions, database schema information, or any other contextual information that is relevant to the task. Here are some widely-used approaches that demonstrate using static context examples in prompt engineering:

Establish conversational or functional style with a system message

A system message can be used to inform the LLM about the context. The context may be the type of conversation it is engaging in, or the function it is supposed to perform. It helps the LLM generate more appropriate responses. For example, if the LLM is expected to perform a customer service function, the prompt could include a system message such as:

```
You are a friendly AI agent who can provide assistance to the customer regarding their recent order.
```

Similarly, if the LLM is engaging in a casual conversation, the prompt could include a message such as

```
Let's have a friendly chat about your favorite hobbies.
```

Use input text to guide completion formatting

By formatting input text in a specific way, we can guide the LLMs towards generating more relevant responses. For example, if the LLM is trained to generate recipe instructions, the input text could be formatted as "First, [action], then [action], and finally [action]." This formatting guides the LLM towards generating instructions in a sequential manner. Similarly, if the LLM is trained to generate summaries of news articles, the input text could be formatted as "The article talks about [topic]. It discusses [point 1], [point 2], and [point 3]." This formatting guides the LLM towards generating a summary that covers the main points of the article. Here we can see the importance of using the correct comment character for the language that you want to generate.

Prompt: /* Create a javascript dictionary with 5 cities and their location */ *Completion*:

```
  var cities = {
'Los Angeles': [34.0522, -118.2437],
'New York': [40.7128, -74.0059],
'San Francisco': [37.7749, -122.4194],
'Chicago': [41.8781, -87.6298],
'Houston': [29.7604, -95.3698]
};
```

In this case, we get a properly formatted JavaScript dictionary with the requested information. Now, if you change "JavaScript" to python and leave the comment characters the same you get: *Prompt*: /* Create a python dictionary with 5 cities and their location */ *Completion*:

```
PyObject *pDict = PyDict_New();
PyDict_SetItemString(pDict, "San Francisco", Py_BuildValue("(f,f)", 37.77493, -122.419416));
PyDict_SetItemString(pDict, "New York", Py_BuildValue("(f,f)", 40.7127, -74.0059));
PyDict_SetItemString(pDict, "London", Py_BuildValue("(f,f)", 51.507351, -0.127758));
PyDict_SetItemString(pDict, "Paris", Py_BuildValue("(f,f)", 48.856614, 2.352222));
PyDict_SetItemString(pDict, "Tokyo", Py_BuildValue("(f
f)", 35.689487, 139.691706));
```

Here we get C++ code making a python dictionary, because of the comment character being /* instead of #.

Add supporting information

Adding supporting information to the prompt can provide more context for the LLM, which can improve the accuracy of its responses. For example, if the prompt asks the LLM to generate a response to a specific question, providing additional information can help the LLM generate a more relevant response. A good example of this is:

```
Please explain transformer language model to a 15-year-old student.
```

Similarly, the LLM generates a more accurate and persuasive description when provided with additional information if asked to generate a product description. A good example is:

```
Write a witty product description in a conversational style so young adult shoppers understand
what this product does and how it benefits them.

Use the following product details to summarize your description:

Title: {{shopify.title}}
Type: {{shopify.type}}
Vendor: {{shopify.vendor}}
Tags: {{shopify.tags}}
```

Task-specific knowledge enrichment

Prompt engineering by task-specific knowledge enrichment involves retrieving relevant knowledge from a large corpus of text and incorporating that into the prompt to improve the performance of language models. This enrichment is also known as data augmented generation. One way to achieve this enrichment is through a knowledge retrieval strategy. This type of strategy involves chunking and indexing bulk knowledge context, followed by embedding similarity context selection. Here's how this approach works:

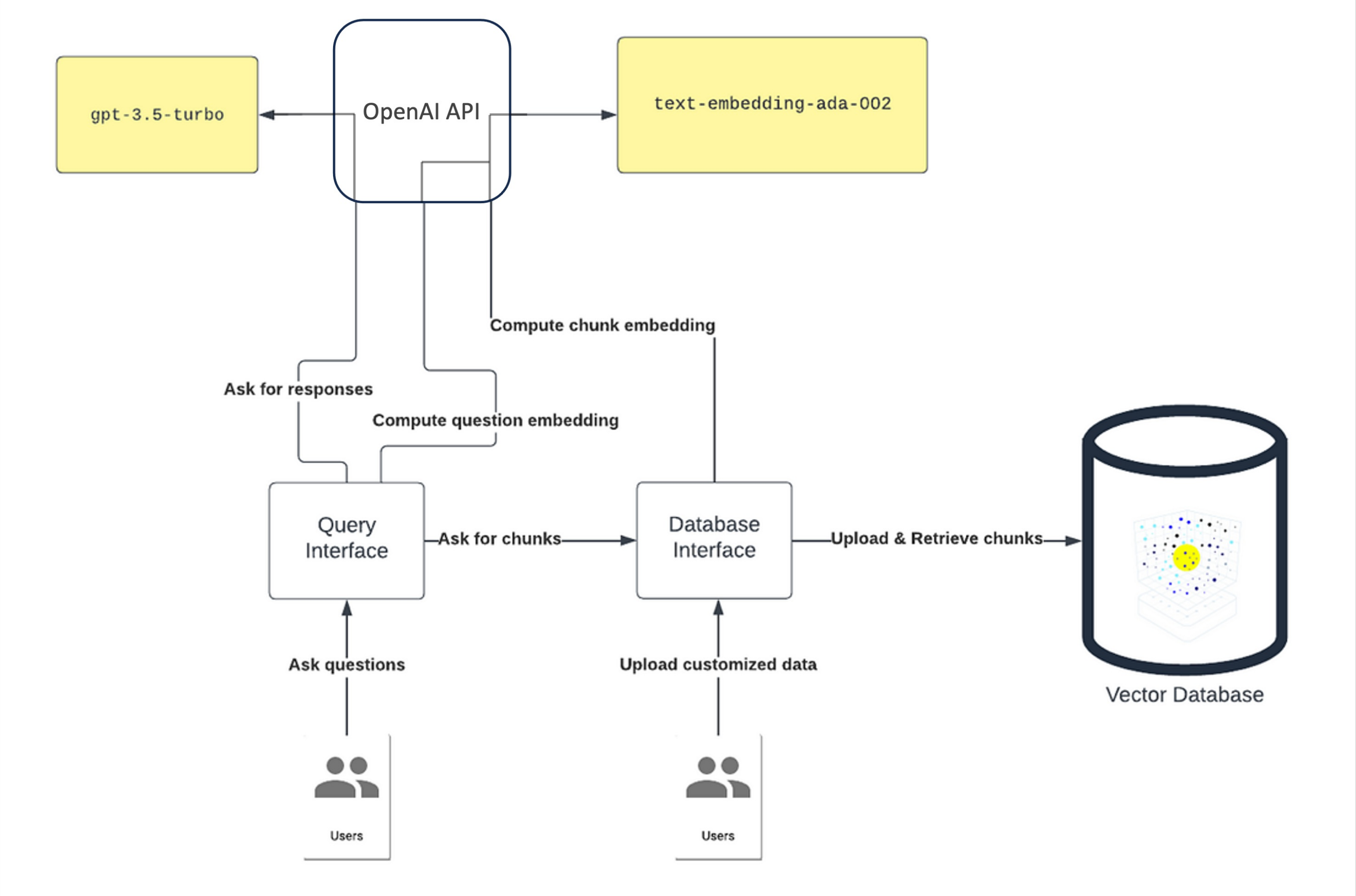
- **Importing knowledge data into a document store.** Multiple data sources could be used to build the knowledge document store. For example, you could import the following data:
    - Car reviews from websites such as Edmunds and Consumer Reports
    - Technical manuals in .pdf format from car manufacturers
    - News articles from sources such as Reuters and CNN

    Each of these data sources would be represented as a separate document within the document store.
- **Chunking the text into smaller, possibly overlapping and more manageable segments.** The chunk can be implemented by a static size of token (for example, 1000). The chunk size is defined by considering token size limit of LLMs' model inference. It can also be carried out using embedded titles, topics, or natural paragraph styles in the knowledge data.
- **Embedding generation/indexing for efficient retrieval.** The final step is to generate

embeddings and index the segmented text for efficient retrieval. This generation involves representing each segment as a vector or a set of features that capture its key semantic attributes. For example, you could generate embeddings using a pretrained language model (like, BERT, text_embedding-ada-002) and save it as a vector store. Next, an index is created that maps each embedding to its corresponding document. This index allows you to quickly retrieve all the documents that contain relevant information based on their similarity to the query embedding.

- **Retrieval of relevant context (for example, chunks).** Once the knowledge base has been chunked and indexed, the next step is to select the most relevant pieces of information to incorporate into the prompt. One approach for doing that is by semantic search. Specifically, embeddings of the indexed knowledge segments are compared with the embeddings of the input prompt to identify the most similar pieces of information.

This method of prompt engineering by task-specific knowledge enrichment can be highly effective in improving the accuracy and relevance of LLM responses. By incorporating relevant knowledge into the prompt, the LLM can generate more informed and accurate responses. These responses can enhance the user's experience and increase the effectiveness of the system. The following figure provides one example of task-specific knowledge enrichment architecture design.



Few-shot examples

Few-shot examples involve including a few input and output examples (input-output pairs) in the LLM to guide its completions in both content and format. The following example is a simple few-shot classification:

```
apple: fruit
orange: fruit
zucchini: vegetable
```

Now, if we want to know if a tomato is a fruit or vegetable, we include this few-shot example prior to input:

```
apple: fruit
orange: fruit
zucchini: vegetable
tomato:

Complete this list
```

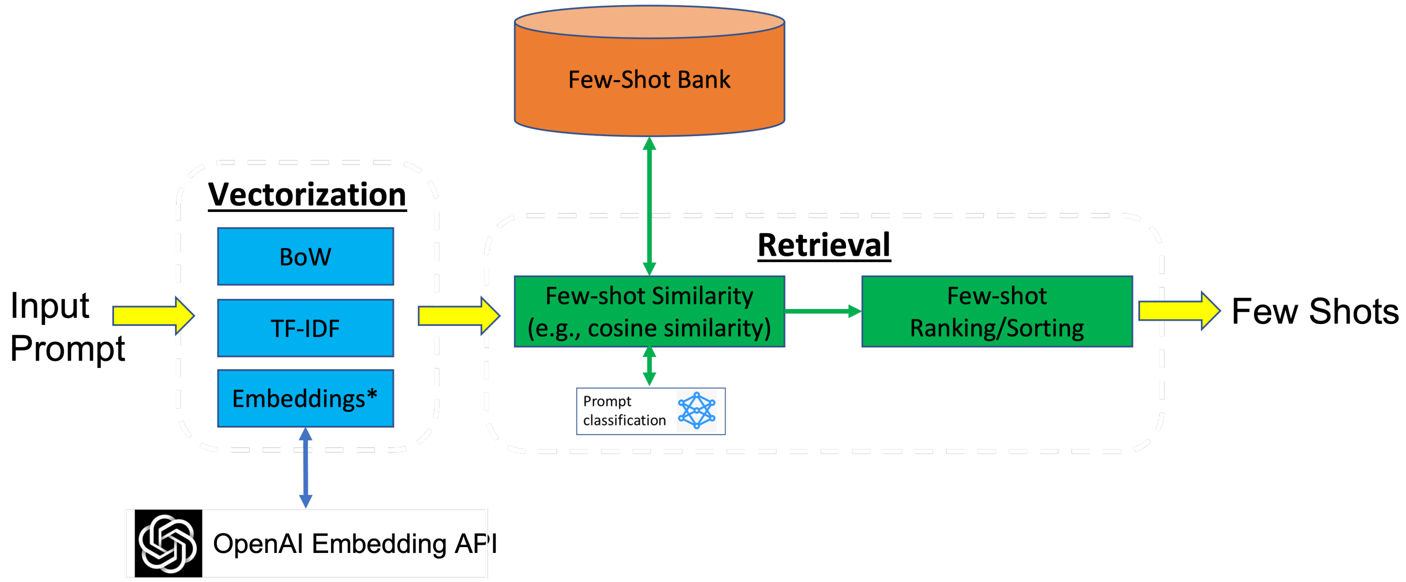
To which the GPT-3.5 responds with "tomato: fruit (botanically), vegetable (culinarily)".

The previous example is a case of a static few shot example. No matter what object we are trying to classify, the same examples are used. However, there are cases where we may want to pick different few shot examples dynamically based on the input prompt. To do that, a library/bank of few-shot examples is created manually. Each example is represented in a feature space (for example, embeddings using a pretrained model). Then, when a new prompt is presented, the few-shot examples that are most similar in that feature space are selected to guide the language model. This method is useful when the following statements are true:

- The few-shot bank data is large and diverse
- The examples share a common underlying pattern

For example, if the few-shot bank data consists of various examples of restaurant reviews, embeddings can capture similarities in the language used to describe the quality of food, service, and atmosphere. The most similar examples can be used to optimize the language model inference.

There are different techniques to improve/optimize the dynamic few-shot selection further. One approach is to filter or categorize the examples in the few-shot for faster retrieval of more relevant examples. To do that, the few-shot bank examples are labeled using intentions or tasks. A custom model can be trained to classify those examples (for example, sports, entertainment, politics, etc.). When a new prompt is presented, the classifier is used to predict the task or intention of the prompt. Then, the few-shot examples that are most relevant to the predicted task are selected to instruct the language model inference. The following figure illustrates the architecture design for dynamic few-shot example retrieval. This retrieval uses the embedding similarity or intention prediction classifier method.



Using session history

Prompt engineering using session history involves tracking the history of a conversation between the user and the language model. This method can help the language model generate more accurate responses by taking into account the context of the conversation. Here's an example of how LLMs track the history of conversation to help generate accurate responses:

```
User: The capital of India?
LLM: The capital of India is New Delhi.
User: What is the population of this city?
LLM: As of 2021, the estimated population of New Delhi is around 31.8 million people.
```

In this scenario, the LLM is able to use the history of the conversation to understand "this city" refers to "New Delhi".

Another example takes the following multi-turn NL2Code interaction where the user's requests follow the # comment character and the model's code follows.

```
# Add a cube named "myCube"
cube(name="myCube")

# Move it up three units
move(0, 5, 0, "myCube")
```

For the second request ("Move it up three units") the model is only able to get the correct completion because the previous interaction is included.

One helpful trick for deciding if you need session history is to put yourself in the place of the model, and ask yourself "Do I have all of the information I need to do what the user wants?"

Challenges and limitations of prompt engineering

While prompt engineering can be useful for improving the accuracy and effectiveness of LLMs inference results, it has substantial challenges and limitations.

Here we summarize some major challenges when using prompt engineering.

- **Token size limit for prompt input:** Most LLMs have a limit on the number of tokens that can be used as input to generate a completion. This limit can be as low as a few dozen tokens. This limit can restrict the amount of context that can be used to generate accurate completions.
- **Data for prompt engineering are not always available:** For example, prompts may require domain-specific knowledge or language that is not commonly used in everyday communication. In such cases, it may be challenging to find suitable data to use for prompt engineering. Additionally, the quality of the data used for prompt engineering affects the quality of the prompts.
- **Evaluation becomes extremely complex as prompt volume grows:** As the number of prompts increases, it becomes more difficult to keep track of the various experiments and to isolate the effect of the prompts on the final output. This tracking difficulty can lead to confusion and make it more challenging to draw meaningful conclusions from the experiments.
- **Complex prompts add latency and costs:** LLMs require time and resources to process and respond to complex prompts. It also adds latency that can slow down the overall process of model development and deployment. More complex prompts also increase the prompt token size in each LLM call, increasing the cost of running experiments.
- **Small prompt changes can have a large impact:** It makes it difficult to predict how the model will behave with even small prompt changes. This can lead to unexpected results. This becomes problematic in applications where accuracy and consistency are critical, such as in automated customer service or medical diagnosis.
