**Long Chain Prompt**

**What is a prompt?**

The first question that comes in mind is, what exactly is a prompt? Well prompts are basically the text input to the LLMs. Anything you are writing to an LLM is a prompt. We have done alot of prompting in the last article.

Infact prompting has a whole field dedicated to it called **“Prompt Engineering”**, where you learn an effective way to construct your prompts to LLM models to get the best out of them. More on that later.

**Prompt Template**

What is a prompt template in LangChain land? This is what the official documentation on LangChain says on it:

*“A prompt template refers to a reproducible way to generate a prompt”*

Prompt templates can contain the following:

*instructions to the language model,*

*a set of few shot examples to help the language model generate a better response,*

*a question to the language model.*

Here is a simple code demo of a simple prompt. If you have missed the [last article](https://medium.com/@princekrampah/langchain-building-language-model-applications-c54cfe7219cb), kindly go back to it and have the dev environment setup for this to work on your end.

from langchain.llms import OpenAI  
# decouple to read .env variables(OpenAI Key)  
from decouple import config  
# import openAI from langChain  
from langchain.llms import OpenAI  
# import prompt template  
from langchain import PromptTemplate  
  
# create the prompt  
prompt\_template: str = """/  
You are a vehicle mechanic, give responses to the following/   
question: {question}. Do not use technical words, give easy/  
to understand responses.  
"""  
  
prompt = PromptTemplate.from\_template(template=prompt\_template)  
  
# format the prompt to add variable values  
prompt\_formatted\_str: str = prompt.format(  
 question="Why won't a vehicle start on ignition?")  
  
# instantiate the OpenAI intance  
llm = OpenAI(openai\_api\_key=config("OPANAI\_API\_KEY"))  
  
# make a prediction  
prediction = llm.predict(prompt\_formatted\_str)  
  
# print the prediction  
print(prediction)

**Prompt Templates With Variable Inputs**

**Prompts with LLMs**

#### Generating Prompt templates Efficiently ####  
from langchain.llms import OpenAI  
# decouple to read .env variables(OpenAI Key)  
from decouple import config  
# import openAI from langChain  
from langchain.llms import OpenAI  
# import prompt template  
from langchain import PromptTemplate  
  
# create the prompt, here we use multiple inputs  
prompt = PromptTemplate(  
 template=""""/  
You are a vehicle mechanic, give responses to the following/   
question: {question}. Do not use technical words, give easy/  
to understand responses. Your response should be in {language}""",  
 input\_variables=["question", "language"]  
)  
  
# format the prompt to add variable values  
prompt\_formatted\_str: str = prompt.format(  
 question="Why won't a vehicle start on ignition?",  
 language="Spanish")  
  
# instantiate the OpenAI intance  
llm = OpenAI(openai\_api\_key=config("OPANAI\_API\_KEY"))  
  
# make a prediction  
prediction = llm.predict(prompt\_formatted\_str)  
  
# print the prediction  
print(prediction)

The two approaches above will give you almost the same results but, I personally prefer this approach.

How to use this with chat models. Here I simply modified the code from the last article section.

**Prompts with Chat Models**

#### Chat models with Prompts ####  
from decouple import config  
from langchain.prompts.chat import (  
 ChatPromptTemplate,  
 SystemMessagePromptTemplate,  
 HumanMessagePromptTemplate,  
)  
from langchain.chat\_models import ChatOpenAI  
from langchain.chains import LLMChain  
from langchain import PromptTemplate  
  
# Create the first prompt template  
sys\_prompt: PromptTemplate = PromptTemplate(  
 input\_variables=["original\_sentence", "desired\_language"],  
 template="""You are a language translater, an English speaker wants to translate/  
 {original\_sentence} to {desired\_language}. Tell him the corrent answer."""  
)  
  
system\_message\_prompt = SystemMessagePromptTemplate(prompt=sys\_prompt)  
  
student\_prompt: PromptTemplate = PromptTemplate(  
 input\_variables=["original\_sentence", "desired\_language"],  
 template="Translate {original\_sentence} to {desired\_language}"  
)  
student\_message\_prompt = HumanMessagePromptTemplate(prompt=student\_prompt)  
  
chat\_prompt = ChatPromptTemplate.from\_messages(  
 [system\_message\_prompt, student\_message\_prompt])  
  
# create the chat model  
chat\_model: ChatOpenAI = ChatOpenAI(openai\_api\_key=config("OPANAI\_API\_KEY"))  
  
  
# Create the LLM chain  
chain: LLMChain = LLMChain(llm=chat\_model, prompt=chat\_prompt)  
  
# make a call to the models  
prediction\_msg: dict = chain.run(  
 original\_sentence="I love Pizza!", desired\_language="French")  
  
print("#######################################")  
print(prediction\_msg)  
print("#######################################")

At its core, a **PromptTemplate** is just a string template we can pass variables to in order to generate our final string. And we can then pass these PromptTemplate’s to LLM’s in order to create consistently formatted prompts for the LLM.

**How to create a prompt template**

Usually we create the ***PromptTemplate*** using the ***from\_template*** method, and call the ***format*** method to generate the final string

prompt\_template = PromptTemplate.from\_template('my string ..')  
print(prompt\_template.format()) # -> 'my string ..'

More details below. But first ..

**Single v.s Double Quotes**

*There is no difference between single and double quotes in python so do not let that confuse you. They can be used interchangeably.*

One case it could make sense is if your string also includes a single or double quote inside it, in which case you may want to use the opposite kind to encapsulate the string literal, so the quote inside the string does not break the encapsulation. Of course you could alternatively just escape ( **\**) the quote that shows up inside the string.

In any case, **single and double quotes are basically the same**, so don’t let that part confuse you if you start seeing them in code samples and documentation. It is just a stylistic choice

**Prompt Template with pure strings**

Actually, you do not even need to have variables in a PromptTemplate for it work, though I am not sure there is a point using PromptTemplates for such a case. Might as well just pass a plain string to the LLM. This is how that looks

prompt\_template = PromptTemplate.from\_template(  
 'Tell me a funny joke about elephants.'  
)  
print(prompt\_template.format())  
# -> 'Tell me a funny joke about elephants.'

**Prompt Template with variables**

This is the much more common use case for prompt templates. We have a template string, which we want to dynamically be able to replace certain parts of that string only

The variable parts in the template are surround by curly brackets { }, and to fill these parts we pass in a list of key-value pairs (kwargs in python) with the variable name and text they should be filled with to the **format()**method on the Prompt Template.

prompt\_template = PromptTemplate.from\_template(  
 'Tell me a {adjective} joke about {content}'  
)  
print(prompt\_template.format(adjective='funny', content='chickens'))  
# -> 'Tell me a funny joke about chickens.'

**Creating Prompt Templates via the Constructor**

We can also create a PromptTemplate via the constructor, instead of via the **from\_template** method

prompt\_template = PromptTemplate(  
 template='Tell me a {adjective} joke about {content}',  
 input\_variables=['adjective', 'cotent']  
)  
print(prompt\_template.format(adjective='funny', content='chickens'))  
# -> 'Tell me a funny joke about chickens.'

**Prompt Templates with Multiline Strings and Variables**

This is actually just the same as a normal string. You just need to include triple quotes at the start and end and voila. Single or double quotes work the same

template = '''You are a joke generating chatbot.  
Provide funny jokes based on the themes requested.  
  
Question: Tell me a {adjective} joke about {content}  
  
Answer: '''  
  
prompt\_template = PromptTemplate.from\_template(template)  
print(prompt\_template.format(adjective='funny', content='chickens'))

**Prompt Templates with f-strings and Variables**

These come along all the time, and are probably the foundation for my confusion with template strings, which probably points at a lack of python knowledge, v.s anything actually do to with LangChain

prompt\_template = PromptTemplate.from\_template(  
 f'Tell me a {{adjective}} joke about {{content}}'  
)  
print(prompt\_template.format(adjective="funny", content="chickens"))

The trick with these is that the variables need to be surrounded with double quotes. The IDE does highlight these highlight these parts also, making them easier to spot out.

These reason for that is because f-strings parse anything inside curly brackets as python expressions. If you want to include curly brackets inside an f-string, as per the [python docs](https://docs.python.org/3/library/string.html#formatstrings):

*If you need to include a brace character in the literal text, it can be escaped by doubling: {{ and }}.*

A nice article with more details on f-strings is [here also](https://realpython.com/python-f-strings).

So the only reason to use f-strings in PromptTemplates would be if there is some external variable or python expression you would like to include in the PromptTemplate. This could be a legitimate use case based on whatever you are building at the time.

If you do not have a use for of those though, you are better off just using a normal string literal.

Just to demonstrates what that might look like, here is an example where we may want to include the date in our prompt template

from datetime import date  
  
today = date.today()  
  
prompt\_template = PromptTemplate.from\_template(  
 f'Todays Date: {today}: Tell me a {{adjective}} joke about {{content}}',  
 template\_format='f-string'  
)  
print(prompt\_template.format(adjective="funny", content="chickens"))  
# -> 'Todays Date: 2024-04-01:Tell me a funny joke about chickens.'

**Prompt Templates with Multiline f-strings and Variables**

This is just the same as the above, except again, we use triple quotes for the start and end of multiline strings, even if they are f-strings

template = f'''You are a joke generating chatbot.  
Provide funny jokes based on the themes requested.  
  
Question: Tell me a {{adjective}} joke about {{content}}'  
  
Answer: '''  
  
prompt\_template = PromptTemplate.from\_template(template)  
print(prompt\_template.format(adjective='funny', content='chickens'))

**Using a PromptTemplate in LLM Calls**

No guide is complete without making some calls to an LLM right?

One of the use cases for PromptTemplates in LangChain is that you can pass in the PromptTemplate as a parameter to an LLMChain, and on future calls to the chain, you only need to pass in the variables you want to substitute for.

prompt\_template = PromptTemplate.from\_template(  
 'Tell me a {adjective} joke about {content}'  
)  
  
openai = ChatOpenAI(  
 model\_name='gpt-3.5-turbo-16k',  
 openai\_api\_key='sk-'  
)  
  
chain = LLMChain(llm=openai, prompt=prompt\_template)  
  
response = chain.invoke(  
 input={'adjective': 'scary', 'content': 'French'}  
)  
  
print(response['text'])  
# -> Why did the French ghost refuse to go into haunted houses?  
# -> Because he had déjà boo!

**Partial with strings**

One common use case for wanting to partial a prompt template is if you get access to some of the variables in a prompt before others. For example, suppose you have a prompt template that requires two variables, foo and baz. If you get the foo value early on in your chain, but the baz value later, it can be inconvenient to pass both variables all the way through the chain. Instead, you can partial the prompt template with the foo value, and then pass the partialed prompt template along and just use that. Below is an example of doing this:

from langchain\_core.prompts import PromptTemplate  
  
prompt = PromptTemplate.from\_template("{foo}{bar}")  
partial\_prompt = prompt.partial(foo="foo")  
print(partial\_prompt.format(bar="baz"))

**API Reference:**[PromptTemplate](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.prompt.PromptTemplate.html)

You can also just initialize the prompt with the partialed variables.

prompt = PromptTemplate(  
 template="{foo}{bar}", input\_variables=["bar"], partial\_variables={"foo": "foo"}  
)  
print(prompt.format(bar="baz"))

foobaz

**Partial with functions**

The other common use is to partial with a function. The use case for this is when you have a variable you know that you always want to fetch in a common way. A prime example of this is with date or time. Imagine you have a prompt which you always want to have the current date. You can't hard code it in the prompt, and passing it along with the other input variables is inconvenient. In this case, it's handy to be able to partial the prompt with a function that always returns the current date.

from datetime import datetime  
  
  
def \_get\_datetime():  
 now = datetime.now()  
 return now.strftime("%m/%d/%Y, %H:%M:%S")  
  
  
prompt = PromptTemplate(  
 template="Tell me a {adjective} joke about the day {date}",  
 input\_variables=["adjective", "date"],  
)  
partial\_prompt = prompt.partial(date=\_get\_datetime)  
print(partial\_prompt.format(adjective="funny"))

Tell me a funny joke about the day 04/21/2024, 19:43:57

You can also just initialize the prompt with the partialed variables, which often makes more sense in this workflow.

prompt = PromptTemplate(  
 template="Tell me a {adjective} joke about the day {date}",  
 input\_variables=["adjective"],  
 partial\_variables={"date": \_get\_datetime},  
)  
print(prompt.format(adjective="funny"))

Tell me a funny joke about the day 04/21/2024, 19:43:57

**Prompt Templates**

Prompt templates help to translate user input and parameters into instructions for a language model. This can be used to guide a model's response, helping it understand the context and generate relevant and coherent language-based output.

Prompt Templates take as input an object, where each key represents a variable in the prompt template to fill in.

Prompt Templates output a PromptValue. This PromptValue can be passed to an LLM or a ChatModel, and can also be cast to a string or a list of messages. The reason this PromptValue exists is to make it easy to switch between strings and messages.

There are a few different types of prompt templates:

**String PromptTemplates**

These prompt templates are used to format a single string, and generally are used for simpler inputs. For example, a common way to construct and use a PromptTemplate is as follows:

import { PromptTemplate } from "@langchain/core/prompts";  
  
const promptTemplate = PromptTemplate.fromTemplate(  
 "Tell me a joke about {topic}"  
);  
  
await promptTemplate.invoke({ topic: "cats" });

StringPromptValue {  
 value: 'Tell me a joke about cats'  
}

**ChatPromptTemplates**

These prompt templates are used to format a list of messages. These "templates" consist of a list of templates themselves. For example, a common way to construct and use a ChatPromptTemplate is as follows:

from langchain.prompts.chat import ChatPromptTemplate, SystemMessagePromptTemplate, HumanMessagePromptTemplate

from langchain.schema import SystemMessage, HumanMessage

# 1️⃣ Create prompt template

system\_prompt = SystemMessagePromptTemplate.from\_template("You are a helpful assistant")

human\_prompt = HumanMessagePromptTemplate.from\_template("Tell me a joke about {topic}")

chat\_prompt\_template = ChatPromptTemplate.from\_messages([system\_prompt, human\_prompt])

# 2️⃣ Invoke with variables

prompt\_value = chat\_prompt\_template.format\_prompt(topic="cats")

# 3️⃣ Access messages

for msg in prompt\_value.to\_messages():

print(f"{type(msg).\_\_name\_\_}: {msg.content}")

In the above example, this ChatPromptTemplate will construct two messages when called. The first is a system message, that has no variables to format. The second is a HumanMessage, and will be formatted by the topic variable the user passes in.

**MessagesPlaceholder**

This prompt template is responsible for adding a list of messages in a particular place. In the above ChatPromptTemplate, we saw how we could format two messages, each one a string. But what if we wanted the user to pass in a list of messages that we would slot into a particular spot? This is how you use MessagesPlaceholder.

import {  
 ChatPromptTemplate,  
 MessagesPlaceholder,  
} from "@langchain/core/prompts";  
import { HumanMessage } from "@langchain/core/messages";  
  
const promptTemplate = ChatPromptTemplate.fromMessages([  
 ["system", "You are a helpful assistant"],  
 new MessagesPlaceholder("msgs"),  
]);  
  
await promptTemplate.invoke({ msgs: [new HumanMessage("hi!")] });

ChatPromptValue {  
 messages: [  
 SystemMessage {  
 "content": "You are a helpful assistant",  
 "additional\_kwargs": {},  
 "response\_metadata": {}  
 },  
 HumanMessage {  
 "content": "hi!",  
 "additional\_kwargs": {},  
 "response\_metadata": {}  
 }  
 ]  
}

This will produce a list of two messages, the first one being a system message, and the second one being the HumanMessage we passed in. If we had passed in 5 messages, then it would have produced 6 messages in total (the system message plus the 5 passed in). This is useful for letting a list of messages be slotted into a particular spot.

An alternative way to accomplish the same thing without using the MessagesPlaceholder class explicitly is:

const promptTemplate = ChatPromptTemplate.fromMessages([  
 ["system", "You are a helpful assistant"],  
 ["placeholder", "{msgs}"], // <-- This is the changed part  
]);

For specifics on how to use prompt templates,

**How to use few shot examples in chat models**

**Prerequisites**

This guide assumes familiarity with the following concepts:

* [Prompt templates](https://python.langchain.com/docs/concepts/prompt_templates/)
* [Example selectors](https://python.langchain.com/docs/concepts/example_selectors/)
* [Chat models](https://python.langchain.com/docs/concepts/chat_models/)
* [Vectorstores](https://python.langchain.com/docs/concepts/vectorstores/)

This guide covers how to prompt a chat model with example inputs and outputs. Providing the model with a few such examples is called [few-shotting](https://python.langchain.com/docs/concepts/few_shot_prompting/), and is a simple yet powerful way to guide generation and in some cases drastically improve model performance.

There does not appear to be solid consensus on how best to do few-shot prompting, and the optimal prompt compilation will likely vary by model. Because of this, we provide few-shot prompt templates like the [FewShotChatMessagePromptTemplate](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.few_shot.FewShotChatMessagePromptTemplate.html?highlight=fewshot" \l "langchain_core.prompts.few_shot.FewShotChatMessagePromptTemplate" \t "_blank) as a flexible starting point, and you can modify or replace them as you see fit.

The goal of few-shot prompt templates are to dynamically select examples based on an input, and then format the examples in a final prompt to provide for the model.

**Note:** The following code examples are for chat models only, since FewShotChatMessagePromptTemplates are designed to output formatted [chat messages](https://python.langchain.com/docs/concepts/messages/) rather than pure strings. For similar few-shot prompt examples for pure string templates compatible with completion models (LLMs), see the [few-shot prompt templates](https://python.langchain.com/docs/how_to/few_shot_examples/) guide.

**Fixed Examples**

The most basic (and common) few-shot prompting technique is to use fixed prompt examples. This way you can select a chain, evaluate it, and avoid worrying about additional moving parts in production.

The basic components of the template are:

* examples: A list of dictionary examples to include in the final prompt.
* example\_prompt: converts each example into 1 or more messages through its [format\_messages](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.chat.ChatPromptTemplate.html?highlight=format_messages" \l "langchain_core.prompts.chat.ChatPromptTemplate.format_messages" \t "_blank) method. A common example would be to convert each example into one human message and one AI message response, or a human message followed by a function call message.

Below is a simple demonstration. First, define the examples you'd like to include. Let's give the LLM an unfamiliar mathematical operator, denoted by the "🦜" emoji:

%pip install -qU langchain langchain-openai langchain-chroma  
  
import os  
from getpass import getpass  
  
if "OPENAI\_API\_KEY" not in os.environ:  
 os.environ["OPENAI\_API\_KEY"] = getpass()

If we try to ask the model what the result of this expression is, it will fail:

from langchain\_openai import ChatOpenAI  
  
model = ChatOpenAI(model="gpt-4o-mini", temperature=0.0)  
  
model.invoke("What is 2 🦜 9?")

AIMessage(content='The expression "2 🦜 9" is not a standard mathematical operation or equation. It appears to be a combination of the number 2 and the parrot emoji 🦜 followed by the number 9. It does not have a specific mathematical meaning.', response\_metadata={'token\_usage': {'completion\_tokens': 54, 'prompt\_tokens': 17, 'total\_tokens': 71}, 'model\_name': 'gpt-4o-mini', 'system\_fingerprint': None, 'finish\_reason': 'stop', 'logprobs': None}, id='run-aad12dda-5c47-4a1e-9949-6fe94e03242a-0', usage\_metadata={'input\_tokens': 17, 'output\_tokens': 54, 'total\_tokens': 71})

Now let's see what happens if we give the LLM some examples to work with. We'll define some below:

from langchain\_core.prompts import ChatPromptTemplate, FewShotChatMessagePromptTemplate  
  
examples = [  
 {"input": "2 🦜 2", "output": "4"},  
 {"input": "2 🦜 3", "output": "5"},  
]

**API Reference:**[ChatPromptTemplate](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.chat.ChatPromptTemplate.html) | [FewShotChatMessagePromptTemplate](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.few_shot.FewShotChatMessagePromptTemplate.html)

Next, assemble them into the few-shot prompt template.

# This is a prompt template used to format each individual example.  
example\_prompt = ChatPromptTemplate.from\_messages(  
 [  
 ("human", "{input}"),  
 ("ai", "{output}"),  
 ]  
)  
few\_shot\_prompt = FewShotChatMessagePromptTemplate(  
 example\_prompt=example\_prompt,  
 examples=examples,  
)  
  
print(few\_shot\_prompt.invoke({}).to\_messages())

[HumanMessage(content='2 🦜 2'), AIMessage(content='4'), HumanMessage(content='2 🦜 3'), AIMessage(content='5')]

Finally, we assemble the final prompt as shown below, passing few\_shot\_prompt directly into the from\_messages factory method, and use it with a model:

final\_prompt = ChatPromptTemplate.from\_messages(  
 [  
 ("system", "You are a wondrous wizard of math."),  
 few\_shot\_prompt,  
 ("human", "{input}"),  
 ]  
)

And now let's ask the model the initial question and see how it does:

from langchain\_openai import ChatOpenAI  
  
chain = final\_prompt | model  
  
chain.invoke({"input": "What is 2 🦜 9?"})

AIMessage(content='11', response\_metadata={'token\_usage': {'completion\_tokens': 1, 'prompt\_tokens': 60, 'total\_tokens': 61}, 'model\_name': 'gpt-4o-mini', 'system\_fingerprint': None, 'finish\_reason': 'stop', 'logprobs': None}, id='run-5ec4e051-262f-408e-ad00-3f2ebeb561c3-0', usage\_metadata={'input\_tokens': 60, 'output\_tokens': 1, 'total\_tokens': 61})

And we can see that the model has now inferred that the parrot emoji means addition from the given few-shot examples!

**Dynamic few-shot prompting**

Sometimes you may want to select only a few examples from your overall set to show based on the input. For this, you can replace the examples passed into FewShotChatMessagePromptTemplate with an example\_selector. The other components remain the same as above! Our dynamic few-shot prompt template would look like:

* example\_selector: responsible for selecting few-shot examples (and the order in which they are returned) for a given input. These implement the [BaseExampleSelector](https://python.langchain.com/api_reference/core/example_selectors/langchain_core.example_selectors.base.BaseExampleSelector.html?highlight=baseexampleselector" \l "langchain_core.example_selectors.base.BaseExampleSelector" \t "_blank) interface. A common example is the vectorstore-backed [SemanticSimilarityExampleSelector](https://python.langchain.com/api_reference/core/example_selectors/langchain_core.example_selectors.semantic_similarity.SemanticSimilarityExampleSelector.html?highlight=semanticsimilarityexampleselector" \l "langchain_core.example_selectors.semantic_similarity.SemanticSimilarityExampleSelector" \t "_blank)
* example\_prompt: convert each example into 1 or more messages through its [format\_messages](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.chat.ChatPromptTemplate.html?highlight=chatprompttemplate" \l "langchain_core.prompts.chat.ChatPromptTemplate.format_messages" \t "_blank) method. A common example would be to convert each example into one human message and one AI message response, or a human message followed by a function call message.

These once again can be composed with other messages and chat templates to assemble your final prompt.

Let's walk through an example with the SemanticSimilarityExampleSelector. Since this implementation uses a vectorstore to select examples based on semantic similarity, we will want to first populate the store. Since the basic idea here is that we want to search for and return examples most similar to the text input, we embed the values of our prompt examples rather than considering the keys:

from langchain\_chroma import Chroma  
from langchain\_core.example\_selectors import SemanticSimilarityExampleSelector  
from langchain\_openai import OpenAIEmbeddings  
  
examples = [  
 {"input": "2 🦜 2", "output": "4"},  
 {"input": "2 🦜 3", "output": "5"},  
 {"input": "2 🦜 4", "output": "6"},  
 {"input": "What did the cow say to the moon?", "output": "nothing at all"},  
 {  
 "input": "Write me a poem about the moon",  
 "output": "One for the moon, and one for me, who are we to talk about the moon?",  
 },  
]  
  
to\_vectorize = [" ".join(example.values()) for example in examples]  
embeddings = OpenAIEmbeddings()  
vectorstore = Chroma.from\_texts(to\_vectorize, embeddings, metadatas=examples)

**API Reference:**[SemanticSimilarityExampleSelector](https://python.langchain.com/api_reference/core/example_selectors/langchain_core.example_selectors.semantic_similarity.SemanticSimilarityExampleSelector.html)

**Create the example\_selector**

With a vectorstore created, we can create the example\_selector. Here we will call it in isolation, and set k on it to only fetch the two example closest to the input.

example\_selector = SemanticSimilarityExampleSelector(  
 vectorstore=vectorstore,  
 k=2,  
)  
  
# The prompt template will load examples by passing the input do the `select\_examples` method  
example\_selector.select\_examples({"input": "horse"})

[{'input': 'What did the cow say to the moon?', 'output': 'nothing at all'},  
 {'input': '2 🦜 4', 'output': '6'}]

**Create prompt template**

We now assemble the prompt template, using the example\_selector created above.

from langchain\_core.prompts import ChatPromptTemplate, FewShotChatMessagePromptTemplate  
  
# Define the few-shot prompt.  
few\_shot\_prompt = FewShotChatMessagePromptTemplate(  
 # The input variables select the values to pass to the example\_selector  
 input\_variables=["input"],  
 example\_selector=example\_selector,  
 # Define how each example will be formatted.  
 # In this case, each example will become 2 messages:  
 # 1 human, and 1 AI  
 example\_prompt=ChatPromptTemplate.from\_messages(  
 [("human", "{input}"), ("ai", "{output}")]  
 ),  
)  
  
print(few\_shot\_prompt.invoke(input="What's 3 🦜 3?").to\_messages())

**API Reference:**[ChatPromptTemplate](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.chat.ChatPromptTemplate.html) | [FewShotChatMessagePromptTemplate](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.few_shot.FewShotChatMessagePromptTemplate.html)

[HumanMessage(content='2 🦜 3'), AIMessage(content='5'), HumanMessage(content='2 🦜 4'), AIMessage(content='6')]

And we can pass this few-shot chat message prompt template into another chat prompt template:

final\_prompt = ChatPromptTemplate.from\_messages(  
 [  
 ("system", "You are a wondrous wizard of math."),  
 few\_shot\_prompt,  
 ("human", "{input}"),  
 ]  
)  
  
print(few\_shot\_prompt.invoke(input="What's 3 🦜 3?"))

messages=[HumanMessage(content='2 🦜 3'), AIMessage(content='5'), HumanMessage(content='2 🦜 4'), AIMessage(content='6')]

**Use with an chat model**

Finally, you can connect your model to the few-shot prompt.

chain = final\_prompt | ChatOpenAI(model="gpt-4o-mini", temperature=0.0)  
  
chain.invoke({"input": "What's 3 🦜 3?"})

AIMessage(content='6', response\_metadata={'token\_usage': {'completion\_tokens': 1, 'prompt\_tokens': 60, 'total\_tokens': 61}, 'model\_name': 'gpt-4o-mini', 'system\_fingerprint': None, 'finish\_reason': 'stop', 'logprobs': None}, id='run-d1863e5e-17cd-4e9d-bf7a-b9f118747a65-0', usage\_metadata={'input\_tokens': 60, 'output\_tokens': 1, 'total\_tokens': 61})

**Next steps**

**How to compose prompts together**

**Prerequisites**

This guide assumes familiarity with the following concepts:

* [Prompt templates](https://python.langchain.com/docs/concepts/prompt_templates/)

LangChain provides a user friendly interface for composing different parts of [prompts](https://python.langchain.com/docs/concepts/prompt_templates/) together. You can do this with either string prompts or chat prompts. Constructing prompts this way allows for easy reuse of components.

**String prompt composition**

When working with string prompts, each template is joined together. You can work with either prompts directly or strings (the first element in the list needs to be a prompt).

from langchain\_core.prompts import PromptTemplate  
  
prompt = (  
 PromptTemplate.from\_template("Tell me a joke about {topic}")  
 + ", make it funny"  
 + "\n\nand in {language}"  
)  
  
prompt

**API Reference:**[PromptTemplate](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.prompt.PromptTemplate.html)

PromptTemplate(input\_variables=['language', 'topic'], template='Tell me a joke about {topic}, make it funny\n\nand in {language}')

prompt.format(topic="sports", language="spanish")

'Tell me a joke about sports, make it funny\n\nand in spanish'

**Chat prompt composition**

A chat prompt is made up of a list of messages. Similarly to the above example, we can concatenate chat prompt templates. Each new element is a new message in the final prompt.

First, let's initialize the a [ChatPromptTemplate](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.chat.ChatPromptTemplate.html" \t "_blank) with a [SystemMessage](https://python.langchain.com/api_reference/core/messages/langchain_core.messages.system.SystemMessage.html" \t "_blank).

from langchain\_core.messages import AIMessage, HumanMessage, SystemMessage  
  
prompt = SystemMessage(content="You are a nice pirate")

**API Reference:**[AIMessage](https://python.langchain.com/api_reference/core/messages/langchain_core.messages.ai.AIMessage.html) | [HumanMessage](https://python.langchain.com/api_reference/core/messages/langchain_core.messages.human.HumanMessage.html) | [SystemMessage](https://python.langchain.com/api_reference/core/messages/langchain_core.messages.system.SystemMessage.html)

You can then easily create a pipeline combining it with other messages *or* message templates. Use a Message when there is no variables to be formatted, use a MessageTemplate when there are variables to be formatted. You can also use just a string (note: this will automatically get inferred as a [HumanMessagePromptTemplate](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.chat.HumanMessagePromptTemplate.html" \t "_blank).)

new\_prompt = (  
 prompt + HumanMessage(content="hi") + AIMessage(content="what?") + "{input}"  
)

Under the hood, this creates an instance of the ChatPromptTemplate class, so you can use it just as you did before!

new\_prompt.format\_messages(input="i said hi")

[SystemMessage(content='You are a nice pirate'),  
 HumanMessage(content='hi'),  
 AIMessage(content='what?'),  
 HumanMessage(content='i said hi')]

**Using PipelinePrompt**

**Deprecated**

PipelinePromptTemplate is deprecated; for more information, please refer to [PipelinePromptTemplate](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.pipeline.PipelinePromptTemplate.html" \t "_blank).

LangChain includes a class called [PipelinePromptTemplate](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.pipeline.PipelinePromptTemplate.html" \t "_blank), which can be useful when you want to reuse parts of prompts. A PipelinePrompt consists of two main parts:

* Final prompt: The final prompt that is returned
* Pipeline prompts: A list of tuples, consisting of a string name and a prompt template. Each prompt template will be formatted and then passed to future prompt templates as a variable with the same name.

from langchain\_core.prompts import PipelinePromptTemplate, PromptTemplate  
  
full\_template = """{introduction}  
  
{example}  
  
{start}"""  
full\_prompt = PromptTemplate.from\_template(full\_template)  
  
introduction\_template = """You are impersonating {person}."""  
introduction\_prompt = PromptTemplate.from\_template(introduction\_template)  
  
example\_template = """Here's an example of an interaction:  
  
Q: {example\_q}  
A: {example\_a}"""  
example\_prompt = PromptTemplate.from\_template(example\_template)  
  
start\_template = """Now, do this for real!  
  
Q: {input}  
A:"""  
start\_prompt = PromptTemplate.from\_template(start\_template)  
  
input\_prompts = [  
 ("introduction", introduction\_prompt),  
 ("example", example\_prompt),  
 ("start", start\_prompt),  
]  
pipeline\_prompt = PipelinePromptTemplate(  
 final\_prompt=full\_prompt, pipeline\_prompts=input\_prompts  
)  
  
pipeline\_prompt.input\_variables

**API Reference:**[PipelinePromptTemplate](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.pipeline.PipelinePromptTemplate.html) | [PromptTemplate](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.prompt.PromptTemplate.html)

['person', 'example\_a', 'example\_q', 'input']

print(  
 pipeline\_prompt.format(  
 person="Elon Musk",  
 example\_q="What's your favorite car?",  
 example\_a="Tesla",  
 input="What's your favorite social media site?",  
 )  
)

You are impersonating Elon Musk.  
  
Here's an example of an interaction:  
  
Q: What's your favorite car?  
A: Tesla  
  
Now, do this for real!  
  
Q: What's your favorite social media site?  
A:

**Next steps**

**How to use multimodal prompts**

Here we demonstrate how to use prompt templates to format [multimodal](https://python.langchain.com/docs/concepts/multimodality/) inputs to models.

To use prompt templates in the context of multimodal data, we can templatize elements of the corresponding content block. For example, below we define a prompt that takes a URL for an image as a parameter:

from langchain\_core.prompts import ChatPromptTemplate  
  
# Define prompt  
prompt = ChatPromptTemplate(  
 [  
 {  
 "role": "system",  
 "content": "Describe the image provided.",  
 },  
 {  
 "role": "user",  
 "content": [  
 {  
 "type": "image",  
 "source\_type": "url",  
 "url": "{image\_url}",  
 },  
 ],  
 },  
 ]  
)

**API Reference:**[ChatPromptTemplate](https://python.langchain.com/api_reference/core/prompts/langchain_core.prompts.chat.ChatPromptTemplate.html)

Let's use this prompt to pass an image to a [chat model](https://python.langchain.com/docs/concepts/chat_models/#multimodality):

from langchain.chat\_models import init\_chat\_model  
  
llm = init\_chat\_model("anthropic:claude-3-5-sonnet-latest")  
  
url = "https://upload.wikimedia.org/wikipedia/commons/thumb/d/dd/Gfp-wisconsin-madison-the-nature-boardwalk.jpg/2560px-Gfp-wisconsin-madison-the-nature-boardwalk.jpg"  
  
chain = prompt | llm  
response = chain.invoke({"image\_url": url})  
print(response.text())

This image shows a beautiful wooden boardwalk cutting through a lush green wetland or marsh area. The boardwalk extends straight ahead toward the horizon, creating a strong leading line through the composition. On either side, tall green grasses sway in what appears to be a summer or late spring setting. The sky is particularly striking, with wispy cirrus clouds streaking across a vibrant blue background. In the distance, you can see a tree line bordering the wetland area. The lighting suggests this may be during "golden hour" - either early morning or late afternoon - as there's a warm, gentle quality to the light that's illuminating the scene. The wooden planks of the boardwalk appear well-maintained and provide safe passage through what would otherwise be difficult terrain to traverse. It's the kind of scene you might find in a nature preserve or wildlife refuge designed to give visitors access to observe wetland ecosystems while protecting the natural environment.

Note that we can templatize arbitrary elements of the content block:

prompt = ChatPromptTemplate(  
 [  
 {  
 "role": "system",  
 "content": "Describe the image provided.",  
 },  
 {  
 "role": "user",  
 "content": [  
 {  
 "type": "image",  
 "source\_type": "base64",  
 "mime\_type": "{image\_mime\_type}",  
 "data": "{image\_data}",  
 "cache\_control": {"type": "{cache\_type}"},  
 },  
 ],  
 },  
 ]  
)

import base64  
  
import httpx  
  
image\_data = base64.b64encode(httpx.get(url).content).decode("utf-8")  
  
chain = prompt | llm  
response = chain.invoke(  
 {  
 "image\_data": image\_data,  
 "image\_mime\_type": "image/jpeg",  
 "cache\_type": "ephemeral",  
 }  
)  
print(response.text())

<https://www.newline.co/@zaoyang/dynamic-prompts-with-langchain-templates--71d0c244>

**Prompts — deep dive (step-by-step, one type per conversation)**

Great — we’ll go deep on **Prompts** (just this building block). I’ll cover three focused subtopics you listed, with clear, beginner-friendly code and line-by-line explanations:

1. **Templates** (PromptTemplate, ChatPromptTemplate) — how to build and render prompts
2. **Examples / Few-shot & Example Selectors** — static few-shot and a simple semantic selector example
3. **Output control / Parsers** — structured output via JSON/Pydantic and a simple regex fallback

All code examples use Python and LangChain-friendly patterns. You can run the prompt-formatting parts without calling an LLM; where I show sending to a model I’ll note the call so you can plug your choice of LLM (OpenAI, Hugging Face, etc.).

**1) Templates — PromptTemplate and ChatPromptTemplate**

**Why templates?**

A template turns a reusable instruction into a *programmatic prompt*, replacing variables safely and consistently. This prevents manual string concat mistakes and makes prompt testing repeatable.

**PromptTemplate (single-string prompts)**

python

CopyEdit

# 1) import the PromptTemplate class

from langchain.prompts import PromptTemplate

# 2) define a template with named variables

template\_text = (

"You are a helpful translator.\n\n"

"Task: Translate the following English text to French, keeping it concise.\n\n"

"Text:\n"

"{text}\n\n"

"Rules:\n"

"- Translate only; do not add commentary.\n"

"- Keep it under 30 words.\n"

)

# 3) create the PromptTemplate object that knows the variable names it needs

prompt = PromptTemplate(input\_variables=["text"], template=template\_text)

# 4) render/format the template with concrete content

formatted = prompt.format(text="Hello! Please translate this sentence into French.")

print(formatted)

**Line-by-line:**

* PromptTemplate — class that holds a template and a list of input\_variables.
* template\_text — the actual instruction; {text} is the placeholder.
* PromptTemplate(... ) — tells LangChain which variables will be supplied.
* prompt.format(...) — replaces {text} with provided text and returns the final prompt string you can send to an LLM.

Tip: keep templates *explicit* (system style rules, constraints, length limits). Always include the variable names in input\_variables.

**ChatPromptTemplate (role-based chat prompts)**

Chat models expect *messages with roles* (system, user, assistant). ChatPromptTemplate helps you build those messages programmatically.

python

CopyEdit

from langchain.prompts import ChatPromptTemplate

# define a role-based prompt with placeholders

chat\_prompt = ChatPromptTemplate.from\_messages([

("system", "You are a concise assistant that answers using only the provided facts."),

("human", "Use the facts below to answer the question.\n\nFACTS:\n{facts}\n\nQUESTION:\n{question}"),

])

# format into a list of messages ready to send to a chat LLM

messages = chat\_prompt.format\_messages({"facts": "Paris is the capital of France.", "question": "What is the capital of France?"})

print(messages)

**Notes:**

* from\_messages([...]) accepts tuples (role, template\_string). Roles can be "system", "human", "ai", etc.
* format\_messages(...) returns a list of message dicts like {"role":"system","content":"..."} — ready to hand to a chat LLM (or LangChain chat wrapper).

Use ChatPromptTemplate whenever you target a chat-style model — it keeps system/human/assistant roles explicit and avoids accidental role swaps.

**2) Examples — few-shot + example selectors**

**Static few-shot (manual)**

Few-shot means you give the model a handful of *example input→output pairs* inside the prompt so it sees the expected behavior.

python

CopyEdit

from langchain.prompts import PromptTemplate, FewShotPromptTemplate

# 1) define the example template (how each example will be formatted)

example\_prompt = PromptTemplate(

input\_variables=["q", "a"],

template="Q: {q}\nA: {a}\n"

)

# 2) prepare a small list of examples (could be strings or dicts)

examples = [

{"q": "Capital of France", "a": "Paris"},

{"q": "Capital of Japan", "a": "Tokyo"},

]

# 3) create the FewShotPromptTemplate: it will inject the examples into the prompt

few\_shot = FewShotPromptTemplate(

examples=examples,

example\_prompt=example\_prompt,

prefix="Answer the questions concisely.",

suffix="Q: {user\_question}\nA:",

input\_variables=["user\_question"],

example\_separator="\n"

)

# 4) format the final prompt for a new question

prompt\_text = few\_shot.format(user\_question="Capital of India")

print(prompt\_text)

**Why this helps:** The model sees a pattern (Q: ... → A: ...) and is much more likely to produce the same pattern for new inputs.

**Dynamic example selection (semantic selector)**

Static few-shot is fine for a few examples, but it’s better to **pick examples similar to the current query** (so the LLM sees the most relevant demonstrations). Below is a simple *semantic* selector using sentence-transformers (you can use OpenAI embeddings + a vector DB instead).

python

# simple semantic example selector (not LangChain built-in: DIY)

from sentence\_transformers import SentenceTransformer

import numpy as np

# example set

examples = [

{"q": "How to boil an egg?", "a": "Place eggs in boiling water for 6-7 minutes for a soft center."},

{"q": "How to bake a cake?", "a": "Preheat oven to 350°F, mix batter, bake 30-35 minutes."},

{"q": "How to roast chicken?", "a": "Season bird, roast at 425°F until 165°F internal temp."},

]

# embedder

embedder = SentenceTransformer("all-MiniLM-L6-v2")

example\_texts = [ex["q"] for ex in examples]

example\_vecs = embedder.encode(example\_texts, convert\_to\_numpy=True)

def select\_examples(query, k=2):

qv = embedder.encode([query], convert\_to\_numpy=True)[0]

sims = (example\_vecs @ qv) / (np.linalg.norm(example\_vecs, axis=1) \* np.linalg.norm(qv) + 1e-12)

top\_idx = np.argsort(-sims)[:k]

return [examples[i] for i in top\_idx]

# usage

query = "What's the best temperature and time to bake a chocolate cake?"

selected = select\_examples(query, k=2)

print("Selected examples:", selected)

**How to use selected examples in a few-shot prompt:** format the selected examples with your example template, then inject them into a PromptTemplate/FewShotPromptTemplate suffix.

Production note: LangChain has ExampleSelector classes (semantic & length-based) you can use that integrate with vectorstores — the concept above is the core: compute similarity and pick top K.

We’ll keep it step-by-step and only cover **PromptTemplate** and **ChatPromptTemplate** today.

**2) Prompts → Templates**

LangChain prompt templates let you define *structured and reusable* ways to pass instructions to an LLM instead of hardcoding raw strings.  
Think of them as **blueprints** for your prompts.

**1. PromptTemplate**

**Purpose:**  
A PromptTemplate is used for **single-turn prompts** (non-chat) — you create a template with placeholders, and fill them with dynamic data before sending to the LLM.

**Structure:**

python

CopyEdit

from langchain.prompts import PromptTemplate

template = """

You are a helpful assistant.

Translate the following text to {language}:

{text}

"""

prompt = PromptTemplate(

input\_variables=["language", "text"],

template=template

)

final\_prompt = prompt.format(language="French", text="Hello, how are you?")

print(final\_prompt)

**Output:**

css

CopyEdit

You are a helpful assistant.

Translate the following text to French:

Hello, how are you?

**Key points:**

* input\_variables must match placeholders in your template.
* You can reuse the same template for different inputs.
* Useful when you need **consistent formatting** for similar tasks.

**2. ChatPromptTemplate**

**Purpose:**  
Used for **multi-turn or role-based prompts** — for example, conversations between a *system*, *human*, and *AI*.  
You can specify **roles** so the model understands *context + speaker*.

**Structure:**

python

CopyEdit

from langchain.prompts import ChatPromptTemplate

chat\_template = ChatPromptTemplate.from\_messages([

("system", "You are a professional translator."),

("human", "Translate the following text to {language}: {text}")

])

final\_chat\_prompt = chat\_template.format\_messages(

language="Spanish", text="Good morning"

)

for message in final\_chat\_prompt:

print(message)

**Output:**

scss

CopyEdit

SystemMessage(content='You are a professional translator.')

HumanMessage(content='Translate the following text to Spanish: Good morning')

**Key points:**

* .from\_messages() takes a list of (role, content) tuples.
* Roles: "system", "human", "ai" (or SystemMessage, HumanMessage, AIMessage classes).
* .format\_messages() returns structured Message objects ready for chat models.
* Ensures **role separation** and avoids mixing instruction text with conversation history.

**PromptTemplate vs ChatPromptTemplate**

| **Feature** | **PromptTemplate** | **ChatPromptTemplate** |
| --- | --- | --- |
| Best for | Single-turn instructions | Multi-turn chat / role-based messages |
| Output | String | List of message objects |
| Roles support | ❌ No | ✅ Yes |
| Example use | Summarizing text | Customer support chatbot |

✅ **Summary so far:**  
We’ve covered **Templates** as the first type of Prompts in LangChain:

1. **PromptTemplate** → for string-based, single-turn prompts.
2. **ChatPromptTemplate** → for role-based, multi-turn prompts.

If you want, in the **next step** we can move to **Few-shot + Example Selectors**, which is the second type under **Prompts**. That’s where we start teaching the LLM with *examples inside the prompt* for better accuracy.

Do you want me to continue to that part next?

**You said:**

yes please

**ChatGPT said:**

Alright Rajeev,  
let’s go step-by-step into the **second building block — Prompts**, and today we’ll focus only on **Prompt Templates** (the first sub-type under Prompts).

**2.1 Prompt Templates**

**What is a Prompt Template?**

In LangChain, a **PromptTemplate** is like a blueprint for prompts.  
Instead of hardcoding prompts as plain strings, you create a reusable template where **variables** can be dynamically inserted at runtime.

**Why use them?**

* **Reusability:** Same structure, different inputs.
* **Consistency:** Ensures the same style and tone across prompts.
* **Parameterization:** Dynamically fill placeholders from your code.
* **Reduced human error:** No manual copy-pasting prompts.

**Basic Structure**

A PromptTemplate has:

1. **Template text** → a string with placeholders {variable\_name}.
2. **Input variables** → list of names you’ll replace at runtime.

**Example 1 — Without Template**

python

CopyEdit

from langchain.prompts import PromptTemplate

prompt = "Translate the following text to French: Hello, how are you?"

This is static — you’d need to rewrite it for every sentence.

**Example 2 — With PromptTemplate**

python

CopyEdit

from langchain.prompts import PromptTemplate

template = "Translate the following text to French: {sentence}"

prompt = PromptTemplate(

input\_variables=["sentence"],

template=template

)

# Fill the variable at runtime

final\_prompt = prompt.format(sentence="Hello, how are you?")

print(final\_prompt)

**Output:**

pgsql

CopyEdit

Translate the following text to French: Hello, how are you?

**Key Parameters**

* input\_variables → list of variable names expected.
* template → the prompt text with placeholders.
* partial\_variables → pre-fill some variables in advance.

**Example 3 — Partial Variables**

Sometimes you want **part** of the prompt fixed:

python

CopyEdit

template = "Translate the following text to {language}: {sentence}"

prompt = PromptTemplate(

input\_variables=["sentence"],

template=template,

partial\_variables={"language": "French"} # pre-fill

)

print(prompt.format(sentence="Good morning"))

**Output:**

pgsql

CopyEdit

Translate the following text to French: Good morning

**Real-World Usage**

PromptTemplates are very useful when:

* You’re calling the same model many times with different data.
* You want a **consistent style** across multiple runs (e.g., customer support responses, summarization, translation).
* You’re building **multi-step chains** where the output of one step becomes the input to another.

**Special Case: ChatPromptTemplate**

This is used for **conversational models** (like ChatGPT) where messages have roles (system, human, ai).

Example:

python

CopyEdit

from langchain.prompts import ChatPromptTemplate

chat\_template = ChatPromptTemplate.from\_messages([

("system", "You are a helpful assistant that translates {language}."),

("human", "{text}")

])

final\_prompt = chat\_template.format\_messages(

language="French",

text="Hello, how are you?"

)

print(final\_prompt)

**Output:**

vbnet

CopyEdit

[SystemMessage(content='You are a helpful assistant that translates French.'),

HumanMessage(content='Hello, how are you?')]

These message objects are directly compatible with chat models in LangChain.

✅ **In short:**  
PromptTemplate = dynamic string templates for LLM prompts.  
ChatPromptTemplate = structured templates for multi-turn chat messages.

Awesome — you said **yes**, so here’s a single, **runnable Python script** that demonstrates **few-shot prompting with a semantic example selector** + **PromptTemplate** + **Pydantic output parser** — all wired together step-by-step so you (as a beginner) can run it and inspect every part.

Two modes:

* **live** mode (calls a real LLM via LangChain ChatOpenAI) — requires OPENAI\_API\_KEY.
* **mock** mode (no API key required) — shows how formatting + parsing works with a fake model output.

**What it does (high level)**

1. Defines a small set of example Q→A pairs.
2. Embeds those examples with sentence-transformers.
3. Given a user question, selects the **k most semantically similar examples**.
4. Builds a PromptTemplate that **injects the selected examples** and **parser format instructions**.
5. Calls an LLM (or uses mock output) to get a response.
6. Parses the response with a **PydanticOutputParser** into a typed object.

This shows the real workflow of *semantic example selection → few-shot prompt → constrained structured output*.

**Requirements**

Install the packages used:

bash

CopyEdit

python -m venv venv

source venv/bin/activate # Windows: venv\Scripts\activate

pip install langchain sentence-transformers openai python-dotenv pydantic

(If you don’t want to call OpenAI, you can run the script in **mock** mode.)

Create a .env (optional) with:

ini

CopyEdit

OPENAI\_API\_KEY=sk-...

**The runnable script**

python

CopyEdit

"""

few\_shot\_semantic\_selector.py

Run:

- optionally set OPENAI\_API\_KEY in env

- python few\_shot\_semantic\_selector.py

Two modes:

- If OPENAI\_API\_KEY is present, script tries to call ChatOpenAI (live).

- Otherwise it runs in mock mode (no API calls).

"""

import os

from dotenv import load\_dotenv

load\_dotenv()

from typing import List, Dict

import numpy as np

# Embeddings

from sentence\_transformers import SentenceTransformer

# LangChain prompt + LLM integration

from langchain.prompts import PromptTemplate

from langchain.chat\_models import ChatOpenAI

from langchain.chains import LLMChain

# Output parsing with Pydantic

from pydantic import BaseModel

from langchain.output\_parsers import PydanticOutputParser

# ---------------------------

# 0) Configuration

# ---------------------------

EMBED\_MODEL\_NAME = "all-MiniLM-L6-v2" # small, fast embedder

N\_EXAMPLES\_TO\_SELECT = 2 # k for semantic selector

USE\_LIVE\_LLM = bool(os.getenv("OPENAI\_API\_KEY")) # auto-mode if key present

# ---------------------------

# 1) Example data (small demo set)

# ---------------------------

examples = [

{"q": "Capital of France", "a": "Paris"},

{"q": "Capital of India", "a": "New Delhi"},

{"q": "How to boil an egg?", "a": "Place eggs in boiling water for 6-7 minutes for soft yolk, 9-12 minutes for hard."},

{"q": "How to make tea?", "a": "Boil water, steep tea leaves for 3-5 minutes, strain, serve."},

{"q": "What is Python used for?", "a": "Python is used for web development, data science, scripting, automation, and more."}

]

# ---------------------------

# 2) Create an embedder and precompute example embeddings

# ---------------------------

print("Loading embedder and encoding examples... (this may take a few seconds)")

embedder = SentenceTransformer(EMBED\_MODEL\_NAME)

example\_texts = [e["q"] for e in examples]

example\_vecs = embedder.encode(example\_texts, convert\_to\_numpy=True)

example\_norms = np.linalg.norm(example\_vecs, axis=1, keepdims=True) # for cosine

# ---------------------------

# 3) Semantic example selector (simple cosine)

# ---------------------------

def select\_examples\_semantic(query: str, k: int = N\_EXAMPLES\_TO\_SELECT) -> List[Dict]:

"""Return top-k example dicts most similar to query."""

qv = embedder.encode([query], convert\_to\_numpy=True)[0]

q\_norm = np.linalg.norm(qv) + 1e-12

sims = (example\_vecs @ qv) / (example\_norms.flatten() \* q\_norm)

top\_idx = np.argsort(-sims)[:k]

return [examples[int(i)] for i in top\_idx]

# ---------------------------

# 4) Example formatter (how examples appear inside the prompt)

# ---------------------------

EXAMPLE\_FORMAT = "Q: {q}\nA: {a}\n"

def format\_examples\_block(selected\_examples: List[Dict]) -> str:

return "\n".join(EXAMPLE\_FORMAT.format(q=e["q"], a=e["a"]) for e in selected\_examples)

# ---------------------------

# 5) Pydantic schema + parser for structured output

# ---------------------------

class MyOutput(BaseModel):

summary: str

sentiment: str # "positive" | "neutral" | "negative"

parser = PydanticOutputParser(pydantic\_object=MyOutput)

format\_instructions = parser.get\_format\_instructions() # text to include in prompt

# ---------------------------

# 6) PromptTemplate that accepts examples\_block + question + format\_instructions

# ---------------------------

prompt\_template = PromptTemplate(

input\_variables=["examples\_block", "question", "format\_instructions"],

template=(

"You are a concise assistant. Use the following examples to match style and brevity.\n\n"

"EXAMPLES:\n{examples\_block}\n"

"INSTRUCTION: Answer the question below based on your knowledge and the example style.\n\n"

"Please follow this output format exactly:\n{format\_instructions}\n\n"

"QUESTION:\n{question}\n\n"

"Provide the JSON only (no extra commentary)."

)

)

# ---------------------------

# 7) LLM setup (LangChain ChatOpenAI) - optional live mode

# ---------------------------

llm = None

if USE\_LIVE\_LLM:

print("OPENAI\_API\_KEY detected — using live ChatOpenAI (make sure you have access to the model).")

try:

llm = ChatOpenAI(model\_name="gpt-4o-mini", temperature=0.0) # adjust model\_name if needed

except Exception as e:

print("Warning: ChatOpenAI initialization failed (you may need different LangChain version).", e)

llm = None

# ---------------------------

# 8) Helper: call model (or mock)

# ---------------------------

def call\_model(prompt\_text: str) -> str:

"""

If llm is available, call it via LLMChain. Otherwise return a mock JSON string.

"""

if llm is not None:

chain = LLMChain(llm=llm, prompt=prompt\_template)

# we call chain.predict with all variables; LLMChain will replace template

return chain.predict(examples\_block=current\_examples\_block, question=current\_question, format\_instructions=format\_instructions)

else:

# MOCK: create a plausible JSON response using simple heuristics

# NOTE: this mock follows the Pydantic schema so parser.parse(...) will succeed

mock\_summary = f"Short answer about: {current\_question[:80]}"

mock\_sentiment = "neutral"

import json

return json.dumps({"summary": mock\_summary, "sentiment": mock\_sentiment})

# ---------------------------

# 9) Interactive loop: ask a question, pick examples, build prompt, call LLM, parse

# ---------------------------

if \_\_name\_\_ == "\_\_main\_\_":

print("\n=== Few-shot + Semantic Example Selector demo ===")

print("Enter a question (or 'exit'):\n")

while True:

current\_question = input("Q> ").strip()

if current\_question.lower() in ("exit", "quit"):

print("Goodbye!")

break

if not current\_question:

continue

# select examples dynamically

selected = select\_examples\_semantic(current\_question, k=N\_EXAMPLES\_TO\_SELECT)

current\_examples\_block = format\_examples\_block(selected)

print("\n[Selected examples to include in prompt:]")

for idx, ex in enumerate(selected):

print(f" ({idx+1}) Q: {ex['q']} -> A: {ex['a']}")

# build final prompt text (for debug + optional inspection)

final\_prompt\_text = prompt\_template.format(

examples\_block=current\_examples\_block,

question=current\_question,

format\_instructions=format\_instructions

)

print("\n[Final prompt preview — the text that will be sent to the LLM:]\n")

print(final\_prompt\_text)

print("\n[Calling model...]\n")

raw\_output = call\_model(final\_prompt\_text)

print("Raw model output:\n", raw\_output, "\n")

# parse using the Pydantic parser

try:

parsed = parser.parse(raw\_output)

print("Parsed (typed) result:")

print(parsed.json(indent=2))

except Exception as e:

print("Parsing failed:", e)

print("You can try to re-run with a different prompt or inspect raw output above.")

**Step-by-step explanation (what every major block does)**

1. **Example dataset** — small static list of {"q","a"} that act as few-shot demonstrations.
2. **Embedder & precompute** — we use sentence-transformers to embed all example questions once so semantic comparisons are fast.
3. **Selector** — select\_examples\_semantic() computes cosine similarity between the query and each example question, returns top-k examples. This is the **semantic example selector**.
4. **Format examples** — we convert the selected examples into the textual block that will be inserted in the prompt. (You can change format easily to show more/less context.)
5. **PydanticOutputParser** — defines the output schema (summary, sentiment) and gives format\_instructions() which we inject into the prompt to instruct the LLM how to produce machine-readable JSON.
6. **PromptTemplate** — defines the final prompt blueprint with three dynamic variables: the example block, the question, and the format instructions from the parser.
7. **LLM call** — if OPENAI\_API\_KEY is found, we try to instantiate ChatOpenAI and call LLMChain.predict(...). Otherwise the script uses a deterministic mock JSON so you can test parsing locally without API calls.
8. **Parse** — parser.parse(raw\_output) returns a typed MyOutput Pydantic model (or raises if the output wasn't valid JSON / didn't match schema). This enforces structured output.

**Notes, tips & next steps**

* Tune N\_EXAMPLES\_TO\_SELECT and chunk size for your use case. Here we select examples by question similarity; in production you might select examples by *quality*, *length*, or *relevance to user domain*.
* If you prefer LangChain’s built-in ExampleSelector classes (they exist in some versions), you can plug them into a FewShotPromptTemplate. The manual approach here is straightforward and version-robust.
* For higher relevance, store many examples in a vector DB (Chroma, Pinecone) and retrieve them via a vector retriever.
* Use **low temperature (0–0.2)** for structured outputs to minimize variability. We set temperature=0.0 in the example.
* If parsing fails in live runs, log the **raw LLM output** and add a small "repair prompt" to ask the LLM to reformat exactly to the schema.

If you'd like I can:

* Replace the simple mock with a **local LLM** (e.g., using Hugging Face/transformers) for offline generation, or
* Convert this to a **FewShotPromptTemplate** that is populated at runtime (I can show the exact LangChain API for dynamic examples, depending on your installed version), or
* Build a small FastAPI endpoint around this flow so the frontend can request select+prompt+parse from a server.

Got it — you want to take a **LangChain flow** (select → prompt → parse) and wrap it inside a **FastAPI** endpoint so that a frontend can hit one endpoint and get the result.

Here’s a clean, working example:

**1️⃣ Install dependencies**

bash

CopyEdit

pip install fastapi uvicorn langchain langchain-openai pydantic

*(Replace langchain-openai with the provider you want; here I’ll use OpenAI’s API for simplicity.)*

**2️⃣ FastAPI + LangChain flow**

python

CopyEdit

# app.py

from fastapi import FastAPI

from pydantic import BaseModel

from langchain.prompts import PromptTemplate

from langchain\_openai import ChatOpenAI

from langchain.schema import StrOutputParser

app = FastAPI()

# ----- LangChain pipeline -----

def build\_chain():

# Step 1: Select (in this example, we just accept user input from request)

# Step 2: Prompt

prompt = PromptTemplate(

input\_variables=["topic"],

template="You are a helpful assistant. Summarize the topic: {topic} in 2 sentences."

)

# Step 3: Model

llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)

# Step 4: Parse output

parser = StrOutputParser()

# Chain: prompt → model → parser

return prompt | llm | parser

# ----- Request body schema -----

class RequestBody(BaseModel):

topic: str

@app.post("/generate")

async def generate\_text(request: RequestBody):

chain = build\_chain()

result = chain.invoke({"topic": request.topic})

return {"summary": result}

**3️⃣ Run the API**

bash

CopyEdit

uvicorn app:app --reload

**4️⃣ Example frontend call**

If your frontend is JavaScript:

javascript

CopyEdit

fetch("http://127.0.0.1:8000/generate", {

method: "POST",

headers: { "Content-Type": "application/json" },

body: JSON.stringify({ topic: "Quantum Computing" })

})

.then(res => res.json())

.then(data => console.log(data.summary));

✅ **What this does**

* The frontend sends a topic (select step is simply choosing the input).
* LangChain builds a prompt (prompt step).
* OpenAI LLM generates a response.
* The parser (parse step) returns a clean string.
* FastAPI wraps the whole flow so the frontend only needs to hit /generate.

If you want, I can now **extend this** so that the same FastAPI endpoint can handle **few-shot examples** in LangChain as well, so your LLM gets context before answering. That way you can keep your *select → prompt → parse* but with **smarter context injection**.

Do you want me to extend it that way?

**You said:**

go with - Replace the simple mock with a local LLM (e.g., using Hugging Face/transformers) for offline generation

**ChatGPT said:**

Got it — you want to replace the mock text generation in the FastAPI + LangChain flow with a **local LLM** using Hugging Face's transformers, so the server can run offline without relying on OpenAI or other hosted APIs.

Here’s how we can do that step-by-step.

**1. Install dependencies**

bash

CopyEdit

pip install fastapi uvicorn transformers torch langchain

*(If you have a GPU, also install the right CUDA version of PyTorch for speed.)*

**2. Example folder structure**

graphql

CopyEdit

.

├── app.py # FastAPI server

├── model\_loader.py # Handles local LLM loading

└── requirements.txt

**3. Load a local LLM (model\_loader.py)**

Here we’ll use a small, CPU-friendly model (e.g., tiiuae/falcon-7b-instruct or gpt2 for quick testing).  
You can replace it with a larger one if you have enough VRAM.

python

CopyEdit

# model\_loader.py

from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline

MODEL\_NAME = "tiiuae/falcon-7b-instruct" # change to your preferred model

print(f"Loading model: {MODEL\_NAME}...")

tokenizer = AutoTokenizer.from\_pretrained(MODEL\_NAME)

model = AutoModelForCausalLM.from\_pretrained(

MODEL\_NAME,

torch\_dtype="auto",

device\_map="auto" # Uses GPU if available

)

llm\_pipeline = pipeline(

"text-generation",

model=model,

tokenizer=tokenizer,

max\_length=512,

temperature=0.7,

do\_sample=True

)

def generate\_text(prompt: str) -> str:

result = llm\_pipeline(prompt, num\_return\_sequences=1)

return result[0]["generated\_text"]

**4. FastAPI endpoint with LangChain integration (app.py)**

Here, instead of using an API call to OpenAI, we use our **local LLM pipeline**.

python

CopyEdit

# app.py

from fastapi import FastAPI

from pydantic import BaseModel

from model\_loader import generate\_text

app = FastAPI()

class PromptRequest(BaseModel):

prompt: str

@app.post("/generate")

async def generate\_endpoint(request: PromptRequest):

output = generate\_text(request.prompt)

return {"prompt": request.prompt, "output": output}

@app.get("/")

async def root():

return {"message": "Local LLM FastAPI server is running"}

**5. Run the server**

bash

CopyEdit

uvicorn app:app --reload --port 8000

**6. Test with curl**

bash

CopyEdit

curl -X POST "http://127.0.0.1:8000/generate" \

-H "Content-Type: application/json" \

-d '{"prompt":"Write a haiku about the moon"}'

**7. Notes**

* **Offline:** As long as you have downloaded the Hugging Face model beforehand, this runs entirely offline.
* **LangChain integration:**  
  If you want LangChain’s prompt templates and parsing, you can wrap the generate\_text() call in a langchain.llms.HuggingFacePipeline object:

python

CopyEdit

from langchain.llms import HuggingFacePipeline

from model\_loader import llm\_pipeline

llm = HuggingFacePipeline(pipeline=llm\_pipeline)

# Example:

from langchain.prompts import PromptTemplate

template = PromptTemplate.from\_template("Summarize this: {text}")

prompt = template.format(text="LangChain makes chaining LLM calls easier.")

print(llm(prompt))

If you want, I can **extend this** so that your FastAPI endpoint runs a **select → prompt → parse** flow **entirely offline** with LangChain and the local model. That way the frontend just calls the endpoint and gets structured results.

Do you want me to go ahead with that version?

**You said:**

go with - Convert this to a FewShotPromptTemplate that is populated at runtime (I can show the exact LangChain API for dynamic examples, depending on your installed version)

**ChatGPT said:**

Got it — we’ll take your current prompt logic and wrap it in a **LangChain FewShotPromptTemplate**, so we can dynamically inject examples at runtime before sending it to the LLM.

Here’s a **step-by-step** conversion:

**1. Install dependencies**

If you haven’t yet:

bash

CopyEdit

pip install langchain langchain-core

(If you’re on an older LangChain version, the imports may vary slightly — I’ll note that.)

**2. Example Data Structure**

We’ll keep examples separate, so they can be built dynamically in Python:

python

CopyEdit

examples = [

{"input": "Translate 'Hello' to French", "output": "Bonjour"},

{"input": "Translate 'Goodbye' to Spanish", "output": "Adiós"},

]

**3. Create Example & Prefix/Suffix Templates**

python

CopyEdit

from langchain.prompts import PromptTemplate, FewShotPromptTemplate

# This defines how a \*single\* example will look inside the prompt

example\_prompt = PromptTemplate(

input\_variables=["input", "output"],

template="Input: {input}\nOutput: {output}"

)

# Prefix and suffix to wrap around the few-shot examples

prefix = """You are a translation assistant.

Here are some examples:"""

suffix = """Now, please provide the translation for the following:

Input: {user\_input}

Output:"""

**4. Combine into FewShotPromptTemplate**

python

CopyEdit

few\_shot\_prompt = FewShotPromptTemplate(

examples=examples, # runtime-populated list

example\_prompt=example\_prompt, # how to render each example

prefix=prefix,

suffix=suffix,

input\_variables=["user\_input"], # variable expected from final call

example\_separator="\n\n" # optional formatting

)

**5. Using it with an LLM**

python

CopyEdit

from langchain.llms import HuggingFacePipeline

from transformers import pipeline

# Local model (can replace with any HF model)

hf\_pipeline = pipeline("text-generation", model="gpt2", max\_new\_tokens=100)

llm = HuggingFacePipeline(pipeline=hf\_pipeline)

# Final prompt text

final\_prompt = few\_shot\_prompt.format(user\_input="Translate 'Thank you' to German")

print("Prompt sent to LLM:\n", final\_prompt)

# Generate output

result = llm(final\_prompt)

print("Model Output:", result)

**6. Adding Examples Dynamically at Runtime**

Because examples is just a Python list, you can build it dynamically before calling FewShotPromptTemplate.format():

python

CopyEdit

runtime\_examples = []

for (inp, outp) in [("How are you?", "Comment ça va?"), ("Good morning", "Buenos días")]:

runtime\_examples.append({"input": inp, "output": outp})

few\_shot\_prompt.examples = runtime\_examples

If you want, I can **merge this with your earlier FastAPI endpoint** so the frontend sends:

* examples list
* user\_input string  
  And the backend builds the few-shot prompt + runs it locally on a Hugging Face model.

That would give you **offline few-shot prompting as an API**.

Do you want me to build that FastAPI + FewShotPromptTemplate backend version next?

**You said:**

go with - Do you want me to also show how PromptTemplate works inside a LangChain Chain in the same run? That will make it more practical.

**ChatGPT said:**

Here’s how we can **go with that idea** and show you a PromptTemplate inside a LangChain chain in the **same run** so you see it in a practical, end-to-end flow.

**1. Install Dependencies**

bash

CopyEdit

pip install langchain langchain-community langchain-core transformers accelerate

**2. Example with FewShotPromptTemplate + LLM Chain**

This example:

* Uses FewShotPromptTemplate populated **at runtime**.
* Runs a Hugging Face local LLM (offline capable).
* Executes the prompt inside an LLMChain.

python

CopyEdit

from langchain.prompts import PromptTemplate, FewShotPromptTemplate

from langchain.chains import LLMChain

from langchain\_community.llms import HuggingFacePipeline

from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline

# -------------------------

# 1️⃣ Load Local Model (Offline)

# -------------------------

model\_name = "tiiuae/falcon-7b-instruct" # Example, use smaller if limited resources

tokenizer = AutoTokenizer.from\_pretrained(model\_name)

model = AutoModelForCausalLM.from\_pretrained(model\_name, device\_map="auto")

local\_pipeline = pipeline(

"text-generation",

model=model,

tokenizer=tokenizer,

max\_length=512,

temperature=0.7,

top\_p=0.95

)

llm = HuggingFacePipeline(pipeline=local\_pipeline)

# -------------------------

# 2️⃣ FewShot Examples (Runtime)

# -------------------------

examples = [

{"question": "What is the capital of France?", "answer": "Paris"},

{"question": "Who wrote Hamlet?", "answer": "William Shakespeare"},

]

example\_prompt = PromptTemplate(

input\_variables=["question", "answer"],

template="Q: {question}\nA: {answer}"

)

# -------------------------

# 3️⃣ FewShotPromptTemplate

# -------------------------

dynamic\_fewshot = FewShotPromptTemplate(

examples=examples,

example\_prompt=example\_prompt,

suffix="Q: {user\_question}\nA:",

input\_variables=["user\_question"]

)

# -------------------------

# 4️⃣ LLM Chain

# -------------------------

chain = LLMChain(

llm=llm,

prompt=dynamic\_fewshot

)

# -------------------------

# 5️⃣ Run Prompt Inside Chain

# -------------------------

response = chain.run(user\_question="What is the tallest mountain in the world?")

print(response)

**How it works**

1. **Model** – Uses HuggingFace’s Falcon-7B-Instruct as an offline local LLM.
2. **FewShotPromptTemplate** – Injects example Q&A pairs **dynamically** at runtime.
3. **LLMChain** – Runs the generated prompt with the local LLM.
4. **Output** – You get a complete end-to-end prompt → generation flow.

If you want, I can now **extend this so that you don’t hardcode the examples but load them dynamically from a CSV/DB**, so the chain automatically adapts without code changes. That’s useful when building offline apps that update prompts on the fly.

Do you want me to extend it that way?

**You said:**

go with this in details deep dive - I can take you through Few-shot Prompts & Example Selectors — the second sub-type under Prompts — where the prompt will include examples to improve accuracy. That’s where things start getting smart.

**ChatGPT said:**

Alright Rajeev, let’s roll up our sleeves — we’re going for a **deep dive** into **Few-shot Prompts & Example Selectors** in LangChain.  
This is one of those areas where your LLM stops being “just a clever autocomplete” and starts acting like it has been *tutored for your exact problem*.

**1. Recap: Where Few-shot Prompts Fit in LangChain**

LangChain categorizes *Prompts* into subtypes:

1. **Prompt Templates** → Fixed structure with placeholders ({variables})
2. **Few-shot Prompts** → Adds **examples** to teach the model patterns before asking for the answer
3. **Example Selectors** → Dynamically decide which examples to use in the prompt

We’re now focusing on **#2 and #3**.

**2. Why Few-shot Prompts?**

In zero-shot prompting (just the question), the model may guess formats inconsistently.  
Few-shot prompting solves this by:

* Giving the model **contextual examples** of the **task and output style**
* Reducing hallucinations and irrelevant outputs
* Forcing output into a *learned structure*

💡 Think of it like giving the model a **mini training set** inside the prompt.

**Example Problem**

We want our LLM to convert English sentences into **active voice**.

**Zero-shot:**

kotlin

CopyEdit

Convert this to active voice: The ball was thrown by John.

Might give:

nginx

CopyEdit

John threw the ball.

…or sometimes it may paraphrase in a weird style.

**Few-shot:**

vbnet

CopyEdit

Task: Convert the following sentences to active voice.

Example 1:

Input: The cake was eaten by Mary.

Output: Mary ate the cake.

Example 2:

Input: The homework was completed by the students.

Output: The students completed the homework.

Now, do the same for this sentence:

Input: The ball was thrown by John.

Output:

Now the model **almost always** follows the same structure.

**3. FewShotPromptTemplate in LangChain**

LangChain gives a dedicated class:

python

CopyEdit

from langchain.prompts import FewShotPromptTemplate, PromptTemplate

# 1. Define example format

example\_template = """

Input: {sentence}

Output: {active\_voice}

"""

example\_prompt = PromptTemplate(

input\_variables=["sentence", "active\_voice"],

template=example\_template

)

# 2. Provide example data

examples = [

{"sentence": "The cake was eaten by Mary.", "active\_voice": "Mary ate the cake."},

{"sentence": "The homework was completed by the students.", "active\_voice": "The students completed the homework."}

]

# 3. Create Few-shot template

few\_shot\_prompt = FewShotPromptTemplate(

examples=examples,

example\_prompt=example\_prompt,

suffix="Input: {sentence}\nOutput:",

input\_variables=["sentence"]

)

# 4. Format the final prompt

final\_prompt = few\_shot\_prompt.format(sentence="The ball was thrown by John.")

print(final\_prompt)

📌 Output prompt will look like:

vbnet

CopyEdit

Input: The cake was eaten by Mary.

Output: Mary ate the cake.

Input: The homework was completed by the students.

Output: The students completed the homework.

Input: The ball was thrown by John.

Output:

This final string is sent to the LLM.

**4. Example Selectors — Making Few-shot Smart**

Static examples are fine, but sometimes:

* Input is **domain-specific** (medical, legal, casual speech)
* We have **hundreds of examples**
* We want only the **most relevant** examples for the query

**LangChain’s ExampleSelector** picks examples **on the fly**.

**SemanticSimilarityExampleSelector**

* Stores all examples in a **vector database**
* Picks the ones most similar to the input query
* Ensures prompt stays **small** and **context-relevant**

python

CopyEdit

from langchain.prompts.example\_selector import SemanticSimilarityExampleSelector

from langchain.vectorstores import FAISS

from langchain.embeddings import OpenAIEmbeddings

# Example data

examples = [

{"sentence": "The cake was eaten by Mary.", "active\_voice": "Mary ate the cake."},

{"sentence": "The homework was completed by the students.", "active\_voice": "The students completed the homework."},

{"sentence": "The letter was written by Alice.", "active\_voice": "Alice wrote the letter."},

{"sentence": "The meeting was attended by the managers.", "active\_voice": "The managers attended the meeting."}

]

# Create selector

example\_selector = SemanticSimilarityExampleSelector.from\_examples(

examples=examples,

embedding=OpenAIEmbeddings(),

vectorstore\_cls=FAISS,

k=2 # Pick top 2 relevant examples

)

# Use selector inside FewShotPromptTemplate

dynamic\_prompt = FewShotPromptTemplate(

example\_selector=example\_selector,

example\_prompt=example\_prompt,

suffix="Input: {sentence}\nOutput:",

input\_variables=["sentence"]

)

print(dynamic\_prompt.format(sentence="The book was read by the children."))

💡 Here, LangChain will:

1. Embed "The book was read by the children."
2. Compare it with embeddings of stored examples
3. Pick **2 most similar** examples
4. Create a **short, relevant** prompt for the LLM

**5. Inside a LangChain Chain**

We can plug this directly into an LLM chain:

python

CopyEdit

from langchain.llms import OpenAI

from langchain.chains import LLMChain

llm = OpenAI(temperature=0)

chain = LLMChain(llm=llm, prompt=dynamic\_prompt)

result = chain.run("The ball was thrown by John.")

print(result)

**6. Best Practices for Few-shot Prompts**

1. **Keep examples minimal but diverse** — avoid overloading the prompt
2. **Maintain consistent structure** — same format for each example
3. **Use ExampleSelector for scalability** — prevents huge prompts
4. **Domain-specific examples** improve accuracy drastically

If you want, I can **extend this** into a **single runnable script** where:

* Few-shot prompt
* ExampleSelector
* LLM chain
* One execution run

That would make it **fully practical** so you can run it without modification.

Do you want me to prepare that full runnable demo?

**You said:**

yes please - If you want, I can extend this into a single runnable script where: Few-shot prompt ExampleSelector LLM chain One execution run

**ChatGPT said:**

Alright Rajeev — let’s do a **deep-dive** and make a **single runnable script** that demonstrates:

1. **Few-shot Prompting** – we give the model examples to improve accuracy.
2. **ExampleSelector** – instead of hardcoding examples, we let the system dynamically choose the most relevant ones based on input.
3. **LLMChain** – combine the prompt and model into an execution pipeline.
4. **One Execution Run** – a single script you can run and see output immediately.

**Detailed Flow**

We’ll build a **movie genre guesser**.

* Few-shot examples: movie descriptions → genre.
* ExampleSelector: picks the most relevant examples for a new movie description.
* LLMChain: runs prompt with those examples.

**Code**

python

CopyEdit

from langchain\_openai import ChatOpenAI

from langchain.prompts import FewShotPromptTemplate, PromptTemplate

from langchain.prompts.example\_selector import SemanticSimilarityExampleSelector

from langchain\_community.vectorstores import FAISS

from langchain\_community.embeddings import OpenAIEmbeddings

from langchain.chains import LLMChain

# ---------------------------

# 1. Few-shot Examples

# ---------------------------

examples = [

{"description": "A young wizard attends a school of magic and battles dark forces.",

"genre": "Fantasy"},

{"description": "A group of astronauts travel through a wormhole in search of a new home.",

"genre": "Science Fiction"},

{"description": "A detective investigates a series of murders in a sleepy small town.",

"genre": "Mystery"},

{"description": "A couple falls in love on a doomed ocean liner.",

"genre": "Romance"},

{"description": "A superhero defends his city from a criminal mastermind.",

"genre": "Action"}

]

# ---------------------------

# 2. ExampleSelector with Semantic Search

# ---------------------------

embedding\_model = OpenAIEmbeddings(model="text-embedding-ada-002")

example\_selector = SemanticSimilarityExampleSelector.from\_examples(

examples,

embedding\_model,

FAISS,

k=3 # Pick top 3 most similar examples for each query

)

# ---------------------------

# 3. Create FewShotPromptTemplate

# ---------------------------

example\_prompt = PromptTemplate(

input\_variables=["description", "genre"],

template="Description: {description}\nGenre: {genre}"

)

dynamic\_few\_shot\_prompt = FewShotPromptTemplate(

example\_selector=example\_selector,

example\_prompt=example\_prompt,

suffix="Description: {input}\nGenre:",

input\_variables=["input"]

)

# ---------------------------

# 4. Create LLM Chain

# ---------------------------

llm = ChatOpenAI(model\_name="gpt-4o-mini", temperature=0)

chain = LLMChain(prompt=dynamic\_few\_shot\_prompt, llm=llm)

# ---------------------------

# 5. Execution Run

# ---------------------------

test\_description = "A team of scientists battle dinosaurs on a remote island."

result = chain.run(test\_description)

print("Movie Description:", test\_description)

print("Predicted Genre:", result.strip())

**How It Works**

1. **examples** – our labeled training examples for genres.
2. **SemanticSimilarityExampleSelector** –
   * Stores examples in a **FAISS vector database**.
   * Embeds new query (test\_description) and retrieves the **most similar** examples.
3. **FewShotPromptTemplate** –
   * Dynamically injects the top 3 relevant examples into the prompt.
   * Adds the user’s input (test\_description) at the end.
4. **LLMChain** – connects the prompt with the LLM for execution.
5. **Run** – outputs the predicted genre.

✅ This approach is **smart** because it:

* Avoids **hardcoding** all few-shot examples into every request.
* Always **adapts** examples to be most relevant to the query.
* Improves accuracy vs zero-shot.

If you want, I can **extend this further** by printing the **exact prompt** that the LLM sees — so you can debug and understand *why* it chose that answer.  
That will make this deep dive even richer.

Do you want me to add that debug output?

**You said:**

yes please -That will make this deep dive even richer.

**ChatGPT said:**

Alright, Rajeev — let’s make this a **full deep dive** on **Few-shot Prompts** with **ExampleSelector** and run it end-to-end in a single runnable Python script using LangChain.

We’ll cover:

1. **What is a Few-shot Prompt?**
2. **What is an ExampleSelector?**
3. **How they work together**
4. **Full runnable code** (Few-shot Prompt + ExampleSelector + LLM Chain + One execution run)
5. **Output walkthrough**

**1. What is a Few-shot Prompt?**

A **few-shot prompt** is a prompt to the LLM that **includes multiple examples of input-output pairs** before asking it to complete a new query.  
It helps the LLM **learn the pattern** or **style** without fine-tuning.

Example:

vbnet

CopyEdit

Q: What's 2 + 2?

A: 4

Q: What's 3 + 5?

A: 8

Q: What's 10 + 20?

A:

The last answer is expected to follow the same pattern.

**2. What is an ExampleSelector?**

An **ExampleSelector** dynamically **selects the most relevant examples** from a large pool, based on the new query, instead of hardcoding them.  
Benefits:

* More **context-relevant** examples
* Keeps prompt size **within token limits**
* Adapts to different domains

**3. How they work together**

* The ExampleSelector chooses **best-fit examples** for the query.
* The Few-shot Prompt Template injects them before the user’s question.
* The LLM sees relevant patterns and gives more accurate answers.

**4. Full Runnable Code**

python

CopyEdit

from langchain.prompts import FewShotPromptTemplate

from langchain.prompts.example\_selector import SemanticSimilarityExampleSelector

from langchain.prompts.prompt import PromptTemplate

from langchain.vectorstores import FAISS

from langchain.embeddings import OpenAIEmbeddings

from langchain.llms import OpenAI

from langchain.chains import LLMChain

# -------------------

# 1. Example Dataset

# -------------------

examples = [

{"question": "What's the capital of France?", "answer": "Paris"},

{"question": "What's the capital of Germany?", "answer": "Berlin"},

{"question": "What's the capital of Spain?", "answer": "Madrid"},

{"question": "What's the capital of Italy?", "answer": "Rome"},

{"question": "What's the capital of Canada?", "answer": "Ottawa"},

{"question": "What's the capital of Australia?", "answer": "Canberra"},

{"question": "What's the capital of Japan?", "answer": "Tokyo"},

]

# -------------------

# 2. Create ExampleSelector

# -------------------

example\_prompt = PromptTemplate(

input\_variables=["question", "answer"],

template="Q: {question}\nA: {answer}"

)

example\_selector = SemanticSimilarityExampleSelector.from\_examples(

examples,

OpenAIEmbeddings(), # Embedding model

FAISS, # Vector store

k=3 # Select top 3 similar examples

)

# -------------------

# 3. FewShot Prompt Template

# -------------------

few\_shot\_prompt = FewShotPromptTemplate(

example\_selector=example\_selector,

example\_prompt=example\_prompt,

prefix="Answer the following questions based on previous examples:",

suffix="Q: {input}\nA:",

input\_variables=["input"]

)

# -------------------

# 4. LLM Chain

# -------------------

llm = OpenAI(temperature=0) # Deterministic output

chain = LLMChain(llm=llm, prompt=few\_shot\_prompt)

# -------------------

# 5. Run

# -------------------

user\_query = "What's the capital of India?"

response = chain.run(user\_query)

print("\n--- Generated Prompt ---")

print(few\_shot\_prompt.format(input=user\_query))

print("\n--- LLM Response ---")

print(response)

**5. How It Works (Execution Flow)**

1. **ExampleSelector**
   * Takes "What's the capital of India?"
   * Finds **3 most semantically similar** Q&A from our dataset (probably capitals of other countries).
2. **FewShotPromptTemplate**
   * Injects these examples before your actual question.
3. **LLMChain**
   * Passes the constructed prompt to the LLM.
4. **Output**
   * Since the LLM has seen a consistent Q/A pattern about capitals, it confidently answers:

New Delhi