**Long Chain Prompt**

**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

CopyEdit

# 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.

**3) Output control — structured output & parsers**

When you need machine-readable results (JSON, specific fields), instruct the model to *output JSON* and then use an **Output Parser** to validate and parse the model output reliably.

**Option A — StructuredOutputParser (response schema approach)**

python

CopyEdit

from langchain.output\_parsers import StructuredOutputParser, ResponseSchema

# 1) define the schema(s) you want in the model output

response\_schemas = [

ResponseSchema(name="summary", description="One-sentence summary"),

ResponseSchema(name="sentiment", description="Sentiment label: positive|negative|neutral"),

]

# 2) create the parser and get format instructions (text you put in the prompt)

parser = StructuredOutputParser.from\_response\_schemas(response\_schemas)

format\_instructions = parser.get\_format\_instructions()

# 3) include format\_instructions in your prompt

prompt\_template = (

"You will receive a short text. Do two things:\n"

"1) Provide a one-sentence summary.\n"

"2) Provide sentiment.\n\n"

f"{format\_instructions}\n\nText: { {text} }"

)

# (then format prompt\_template with actual text and send to LLM)

# 4) after you get `raw\_output\_text` from the LLM, parse:

# parsed = parser.parse(raw\_output\_text)

# parsed is a dict like {"summary": "...", "sentiment":"positive"}

**Why this helps**: get\_format\_instructions() tells the model *exactly* how to structure output (e.g., a JSON object with specified fields). parser.parse(...) attempts to extract & validate those fields.

**Option B — PydanticOutputParser (schema & type validation)**

If you want types & auto-validation, use a Pydantic model:

python

CopyEdit

from pydantic import BaseModel

from langchain.output\_parsers import PydanticOutputParser

# 1) define pydantic schema

class MyOutput(BaseModel):

summary: str

sentiment: str

# 2) create PydanticOutputParser wrapper

pyd\_parser = PydanticOutputParser(pydantic\_object=MyOutput)

format\_instructions = pyd\_parser.get\_format\_instructions()

# 3) include format\_instructions in your prompt as before

# 4) when you receive LLM text:

# parsed\_obj = pyd\_parser.parse(raw\_output\_text) # returns MyOutput instance with validated fields

**Benefit:** strong typing and automatic error if the model output can't be coerced to the schema.

**Option C — simple regex / custom fallback parser**

When you only need one value (e.g., a single number), a tiny regex parser can be faster and robust:

python

CopyEdit

import re

def extract\_total(text):

# look for "Total: $123.45" or "Total: 123.45"

m = re.search(r"Total:\s\*\$?([0-9]+(?:\.[0-9]{1,2})?)", text)

return {"total": float(m.group(1))} if m else None

# Use this after the LLM returns a string; if regex fails, you can:

# - ask model to reformat, or

# - run a small repair prompt that says: "Output only JSON: {\"total\": ...}"

**When to use regex:** quick, low-overhead, for predictable short outputs. For complex structured outputs use structured/pydantic parsers.

**Putting it together — example workflow**

1. **Build prompt** with ChatPromptTemplate (system + human), include:
   * short system rules,
   * a few-shot example block (static or selected dynamically),
   * format\_instructions from your parser to constrain output.
2. **Send to LLM** (chat model). Use low temperature for deterministic structured outputs.
3. **Parse result** with parser.parse(...) (Structured or Pydantic). If parse fails:
   * attempt an **output-fix** prompt (ask model to reformat to desired JSON),
   * or apply a regex fallback and log the failing output for later debugging.

**Minimal pseudo-code sketch**

python

CopyEdit

# 1. select examples (maybe dynamic)

examples\_str = build\_examples\_text(selected\_examples)

# 2. get parser instructions

format\_instructions = pyd\_parser.get\_format\_instructions()

# 3. build final chat prompt (system + user including facts, examples, and format instructions)

final\_prompt = f"""

SYSTEM: You are concise and follow instructions.

EXAMPLES:

{examples\_str}

INSTRUCTIONS:

{format\_instructions}

TEXT TO PROCESS:

{text\_to\_process}

"""

# 4. call chat LLM (chat model)

# response\_text = call\_your\_chat\_model(final\_prompt)

# 5. parse

# parsed = pyd\_parser.parse(response\_text)

**Best practices & debugging checklist**

* **Always include format instructions** (via parser.get\_format\_instructions()) rather than asking for ad-hoc JSON.
* **Use low temperature** (0–0.2) for structured outputs to reduce variation.
* **Test corner cases**: empty input, very long input, missing facts.
* **Log both** raw model output and parsed result — you’ll need raw text when parsing fails.
* **Provide few-shot examples** that show the exact format you want (e.g., example JSON).
* **Have a repair strategy**: if parsing fails, ask the model to re-output following the format instructions (or fall back to regex).
* **Avoid ambiguous instructions** — add explicit rules (e.g., “If unknown, return null for that field”).

**Quick troubleshooting recipes**

* **Parser.parse raises error** → print raw model output, then craft a small “format-fix” prompt:  
  "The output you gave is not valid JSON. Please reformat EXACTLY to the schema: {format\_instructions} ..."
* **Model ignores examples** → raise their prominence: move examples closer to the user question, reduce temperature, include fewer but very relevant examples.
* **Model hallucinates fields** → add explicit rule in system that *only* information present in the context may be used; otherwise return "unknown".

**Want a runnable example combining templates + semantic example selector + Pydantic parser?**

I can produce a **single runnable Python file** that:

* loads a tiny example dataset,
* selects semantic few-shot examples,
* builds a ChatPromptTemplate with PydanticOutputParser instructions,
* calls OpenAI (or a mock if you prefer),
* parses and prints the typed Pydantic output.

Say “yes” and I’ll generate that exact runnable script (with step-by-step comments and instructions for running locally).

Sources

**You said:**

yes

**ChatGPT said:**

Alright — let’s go **deep** into the **second building block of LangChain: Prompts**, starting with the **first type: Templates**.

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.

If you want, in our **next conversation** 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.

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

**You said:**

yes please

**ChatGPT said:**

Alright — let’s go step-by-step into **LangChain’s PromptTemplate** since it’s the **first and most fundamental** part of the "Prompts" building block.

**1️⃣ What is a PromptTemplate?**

A **PromptTemplate** in LangChain is like a reusable **prompt mold**.  
You define placeholders (variables) in a template, and later fill them with actual values to generate a final prompt for the LLM.

Think of it like:

*"Dear {name}, your order {order\_id} is ready."*  
Where {name} and {order\_id} are **dynamic fields**.

**2️⃣ Why use PromptTemplate instead of just strings?**

* **Reusability** – Define once, reuse many times.
* **Consistency** – Prevents typos and ensures prompt structure remains intact.
* **Parameter Injection** – Fill in values at runtime from user input or code.
* **Validation** – Ensures all required variables are passed before sending to LLM.

**3️⃣ Basic Syntax**

python

CopyEdit

from langchain.prompts import PromptTemplate

# 1. Define the template string with placeholders

template\_str = "Write a short {tone} poem about {topic}."

# 2. Create PromptTemplate

prompt = PromptTemplate(

input\_variables=["tone", "topic"], # must match placeholders

template=template\_str

)

# 3. Fill in variables

final\_prompt = prompt.format(tone="funny", topic="cats")

print(final\_prompt)

**Output:**

cpp

CopyEdit

Write a short funny poem about cats.

**4️⃣ Validating Variables**

If you try:

python

CopyEdit

prompt.format(topic="cats")

It will raise:

vbnet

CopyEdit

ValueError: Missing value for 'tone'

✅ This ensures your prompt won’t be incomplete.

**5️⃣ PromptTemplate + LLM**

python

CopyEdit

from langchain.llms import OpenAI

llm = OpenAI(model="gpt-3.5-turbo", temperature=0.7)

response = llm(prompt.format(tone="romantic", topic="moonlight"))

print(response)

Now you’re **feeding a dynamically created prompt** to the LLM.

**6️⃣ Jinja2-style Templates (Advanced)**

LangChain also supports **Jinja2** for more complex logic inside prompts.  
Example:

python

CopyEdit

template\_str = """

Generate {{ style }} review for {{ product }}.

{% if rating %}

Give it a rating of {{ rating }} stars.

{% endif %}

"""

prompt = PromptTemplate.from\_template(template\_str)

print(prompt.format(style="positive", product="coffee", rating=5))

This is useful when prompts need **conditional text**.

**7️⃣ When to use PromptTemplate**

* Multi-turn chat apps (templates for each turn).
* API automation (injecting dynamic parameters).
* Few-shot learning (consistent formatting).
* Structured data extraction (fixed query style).

✅ Next, we could move to **ChatPromptTemplate** (which is similar but supports **system/human/AI roles** for chat models).  
If you want, I can break that down **step-by-step** like this too before we move to **few-shot examples**.

Do you want me to continue with **ChatPromptTemplate** next?

**You said:**

yes please

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.

Which of those would you like next?

Sources

**You said:**

go with - Build a small FastAPI endpoint around this flow so the frontend can request select+prompt+parse from a server.

**ChatGPT said:**

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