Prompts:

Prompts are the input instructions or queries given to a model to guide its output

- types: text based
- multimodal prompts - text with -> image, sound , video.
- types of text prompts:
- static : The prompt is written directly by the user, and sent to the LLM as-is.
- Problems:
    - Inconsistent outputs.
    - Users may make mistakes (spelling, unclear instructions).
    - Not a good user experience for production apps.
- to overcome this issue we have to create a **prompt template!**
- This approach is known as **dynamic prompting**.

A Prompt Template allows you to create flexible prompts by inserting variables (placeholders) into a predefined structure. Instead of rewriting the same prompt every time, you define it once and fill in the blanks dynamically at runtime. This makes your code more reusable, maintainable, and ideal for scenarios involving user input or automated tasks.

In LangChain, PromptTemplate provides a more structured and robust way to manage prompts compared to using plain f-strings.

Why use PromptTemplate instead of f-strings?

1. Built-in validation – Ensures all required variables are provided.(validate_template = True)
2. Reusability – Prompts can be saved, shared, or reused easily.
3. Seamless integration – Works well across the LangChain ecosystem, including chains, agents, and retrievers.

(Generate prompt template for easy use)

CHAINS: (overview, in depth after Messages topic)

What is a Chain in LangChain?

A **chain** is just a **sequence of steps** that connect together to complete a task using a language model.

Think of it like a **pipeline**:
Input → Prompt → Model → Output

Instead of writing all the logic yourself each time, a chain helps you organize and reuse that flow.

Why use Chains?

Without chains, you would need to manually handle each part:

- Format the prompt
- Send it to the model
- Parse the output
- Add logic for tools, memory, etc.

With **chains**, LangChain handles all that for you in an organized way.

Example:

1. User asks a question
2. You add the question to a prompt template
3. The LLM generates an answer
4. You show the answer to the user

Chain in use:

In the original code, we are calling invoke two times.

First, we use it to fill the template with the user's name.
Second, we use it to send the filled prompt to the model and get the response.

```
Langchain_code > 4.prompts > Prompt_template.py > ...
19
20  # Define the prompt template
21  template2 = PromptTemplate(
22      template='Greet this person in 5 languages. The name of th
23      input_variables=['name']
24  )
25
26  # Fill the template with user input
27  prompt = template2.invoke({'name': name})
28
29  # Get the result from the model
30  result = model.invoke(prompt)
31
32  # Print the output
33  print(result.content)
34
```

Using a chain, we can combine both steps into one by linking the prompt template and the model together. This makes the code cleaner and easier to maintain

Chatbot:

- ```
  Langchain_code > 4.prompts > Prompt_template.py > ...
  52  # Ask user for name
  53  name = input("Enter your name: ")
  54
  55  # Define the prompt template
  56  template = PromptTemplate(
  57      template="Greet this person in 5 languages. The name o
  58      input_variables=["name"]
  59  )
  60
  61  # Create a chain: prompt → model
  62  chain = template | model
  63
  64  # Run the chain with user input
  65  result = chain.invoke({"name": name})
  66
  67  # Print the output
  68  print(result.content)
  ```

  ```
  You: hi
  AI: Hello! How can I help you today?
  You: I am shree
  AI: Hello Shree! Nice to meet you. How are you doing today? Is there anything I can help you with?
  You: what is my name?
  AI: I don't know your name. I don't have access to personal information about you unless you choose to share it with me. If you'd like to tell me your name, I'd be happy to use it in
  our conversation!
  You:
  ```

the most imp part of a chatbot is its memory. if it remembers the chat history its easier to carry the conversation forward without having to give it context repeatedly
  ```

- chatbot response after creating a chat_history:

```
You: hi i am shree.
AI: Hi Shree! Nice to meet you. How are you doing today? Is there anything I can help you with?
You: what's total number of letters in my name?
AI: Hi Shree! Nice to meet you too!

To count the letters in your name "Shree":
S-h-r-e-e

Your name has **5 letters** total.
You:
```

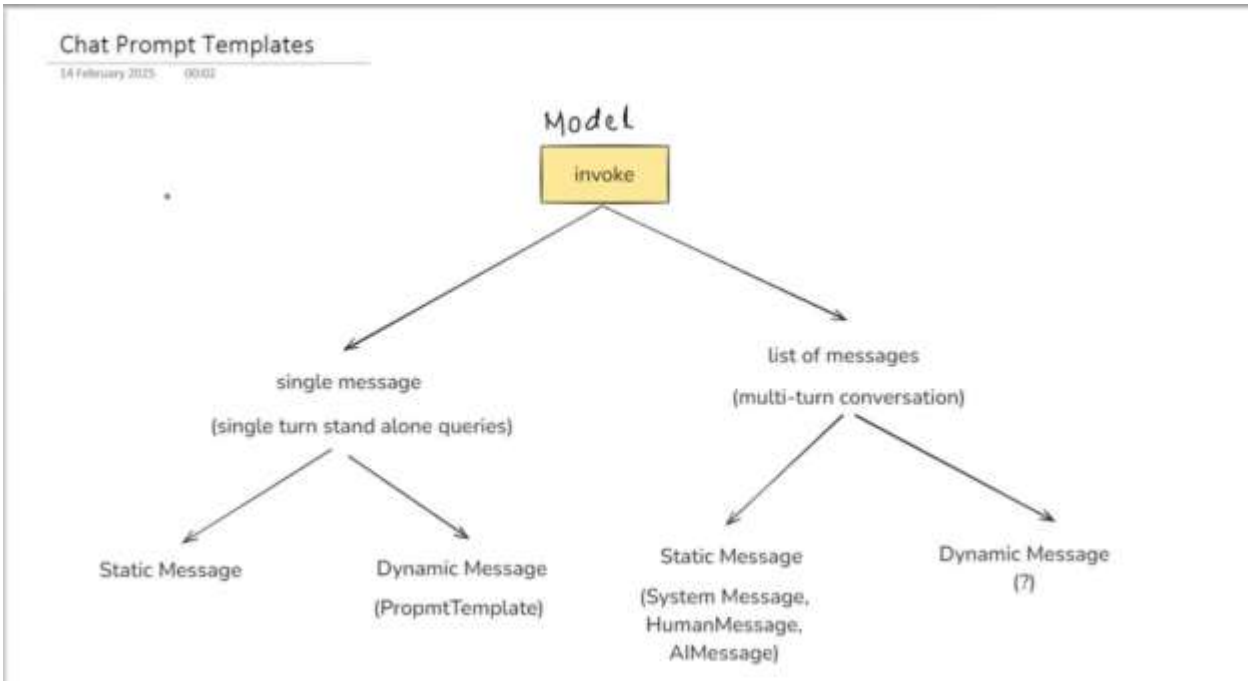
since we are storing our chat history in this format, the model wont be able to identify who sent what response if its the llm or the user as the convo gets longer.

which create problems in the future

```
You: exit
['hi i am shree.', 'Hi Shree! Nice to meet you. How are you doing today? Is there anything I can help you with?', "what's total number of letters in my name?", 'Hi Shree! Nice to meet
 you too!\n\nTo count the letters in your name "Shree":\n\n-h-r-e-e\n\nYour name has **5 letters** total.', 'exit']
```

solution : we'll have to use a dictionary

- since lnagchain has already identified this issue we can simply use the built -in classes that performs this task.



- Static MESSAGES:
- human message: the message user sends the llm (prompt)
- System message: the instructions we give ai before we start the conversation (assigning role: you are a helpful assistant do research before giving me a response)
- ai message:the response ai gives for the system &human message sent
- **FunctionMessage (for function calling mode)**

Used when the model calls or receives results from tools/functions.

- This is used **after** a model **calls a function** and you want to **send the result back**.

Example Scenario:

You ask the AI:

“What’s the weather in Mumbai?”

The LLM says:

“Call the get_weather function with location = 'Mumbai'.”

Your backend calls that function and gets:

{"temperature": "30°C", "condition": "Sunny"}

example :

from langchain_core.messages import FunctionMessage

FunctionMessage(name="get_weather", content='{"temperature": "28°C"}')

ToolMessage (used when LLMs interact with tools in agents)

This is very similar, but it's used with **LangChain Agents** that use tools like:

- calculator
- search
- API call wrappers

Example Scenario:

Agent says:

“I’ll use the Calculator tool to compute 456 * 82.”

Then you run the tool and get the result:

37392

You return this using a ToolMessage:

It’s how the agent **gets the result of a tool call**.

- from langchain_core.messages import ToolMessage
  ToolMessage(tool_call_id="abc123", content="37392")

```python
# Create chat model
model = ChatHuggingFace(llm=llm)

#create a list of messages.
messages=[
    SystemMessage(content='You are a helpful assistant'),
    HumanMessage(content='Tell me about LangChain')
]

result = model.invoke(messages) #send the prompt(list of messages) to receive response from ai

#the response received from ai now gets appended to the list of messages we created for complete context
messages.append(AIMessage(content=result.content))

print(messages)
```

```
[SystemMessage(content='You are a helpful assistant', additional_kwargs={}, response_
metadata={}), HumanMessage(content='Tell me about LangChain', additional_kwargs={}, r
esponse_metadata={}), AIMessage(content="LangChain is a popular open-source framework
 designed to help developers build applications powered by large language models (LLM
s) like GPT, Claude, and others. Here's an overview:\n\n## What is LangChain?\n\nLang
Chain provides tools and abstractions that make it easier to create AI applications b
y connecting LLMs with external data sources, other AI models, and various software c
omponents.\n\n## Key Features\n\n**1. Chains**\n- Pre-built workflows that combine LL
Ms with other components\n- Simplify common patterns like question-answering, summari
ions and make decisions\n- Can interact with tools and APIs to complete complex tasks
\n\n**5. Prompts**\n- Template system for creating and managing prompts\n- Helps stru
cture inputs to LLMs effectively\n\n## Use Cases\n\n- **Chatbots and virtual assistan
ts**\n- **Question-answering systems**\n- **Document analysis and summarization**\n-
**Code generation and explanation**\n- **Research and information retrieval**\n\n## B
enefits\n\n- Reduces boilerplate code\n- Provides best practices for LLM integration\
n- Offers modular, reusable components\n- Supports multiple LLM providers\n- Active c
ommunity and extensive documentation\n\nLangChain has become one of the most popular
frameworks for LLM application development, with both Python and JavaScript versions
available.", additional_kwargs={}, response_metadata={})]
```

dynamic messages: → aka chat-prompt template.

- are **chat messages that include variables** or logic that can change at runtime depending on the user input, memory, or function/tool outputs.

They are often used when you're constructing **chat prompts** dynamically during the conversation.

Langchain_code > 4.prompts > dynamicmessages.py > ...

```python
#ChatPrompt-template.
from langchain_core.prompts import ChatPromptTemplate

# Create the chat prompt template w placeholders.
chat_template = ChatPromptTemplate([
    ('system', 'You are a helpful {domain} expert'),
    ('human', 'Explain in simple terms, what is {topic}')
])

# Fill in the template with user-provided values
prompt = chat_template.invoke({'domain':'cricket','topic':'Dusra'})

# Show the final prompt messages
print(prompt)
```

Why is it useful?

- You don't need to manually write a new message for each user or case.
- You can **reuse the same message**, and it **fills in the blanks** with current info (like name, question, location, etc.).
- Makes your chatbot or AI assistant **personalized** and **context-aware**.

Message-place holder:

A **MessagePlaceholder** is a special tool used inside a prompt **to inject dynamic lists of messages** — like conversation history, tool results, or memory.

example:

You:

"I want to go to Paris."

AI:

"Great! When are you planning to travel?"

You:

"Next month."

next day u ask.

You:

"What should I pack?"

when u ask what should i pack after a day the ai must remember that u are going to paris,next month

this is where message placeholder comes in

eg: —after 24 hrs—

System: You are a helpful travel assistant.

[Message Placeholder: insert past messages here]

Human: What should I pack?

now the ai gets the full context.

System: You are a helpful travel assistant.
Human: I want to go to Paris.
AI: Great! When are you planning to travel?
Human: Next month.
Human: What should I pack?

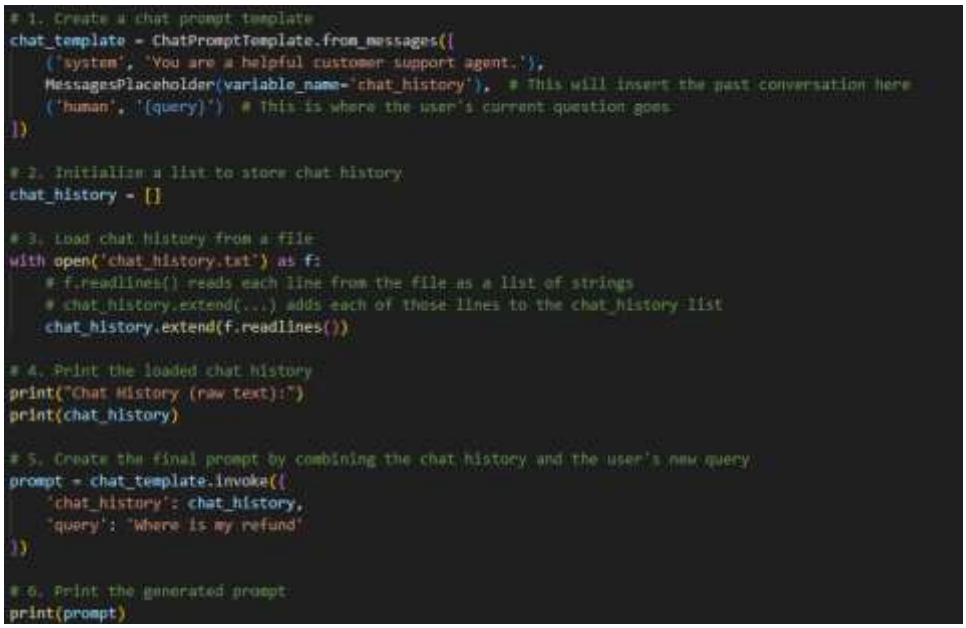
now the ai gives a mush smarter. personalized answer

AI:

"Since you're visiting Paris in the spring, pack light sweaters, a raincoat, and comfortable walking shoes."

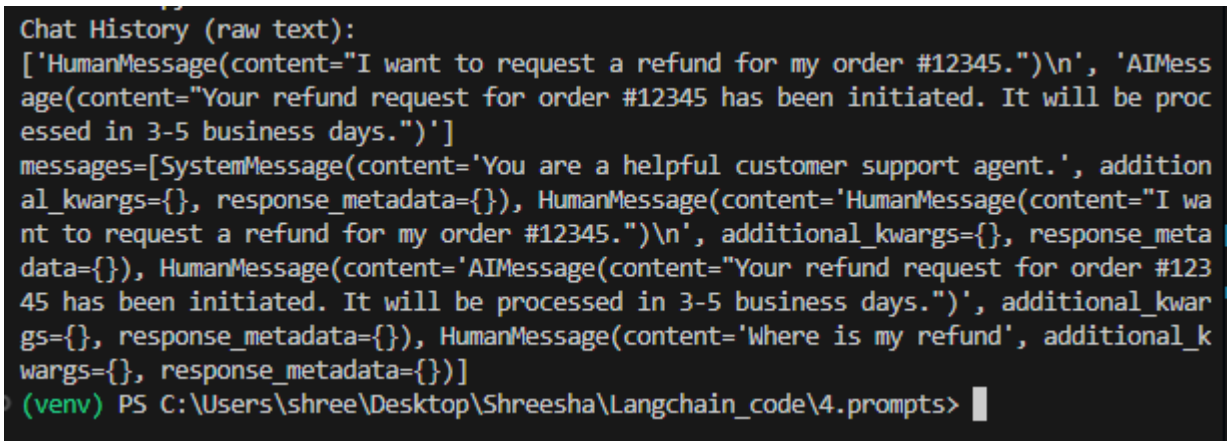
Message Placeholder = recent chat memory
It keeps the AI aware of the full conversation, **without you repeating it manually** every time.

code:



```
# 1. Create a chat prompt template
chat_template = ChatPromptTemplate.from_messages([
    ('system', 'You are a helpful customer support agent.'),
    MessagesPlaceholder(variable_name='chat_history'),  # This will insert the past conversation here
    ('human', '{query}')  # This is where the user's current question goes
])

# 2. Initialize a list to store chat history
chat_history = []

# 3. Load chat history from a file
with open('chat_history.txt') as f:
    # f.readlines() reads each line from the file as a list of strings
    # chat_history.extend(...) adds each of those lines to the chat_history list
    chat_history.extend(f.readlines())

# 4. Print the loaded chat history
print("Chat History (raw text):")
print(chat_history)

# 5. Create the final prompt by combining the chat history and the user's new query
prompt = chat_template.invoke({
    'chat_history': chat_history,
    'query': 'Where is my refund'
})

# 6. Print the generated prompt
print(prompt)
```

output:



w chat_history passed to ai before new user prompt.