Step 1: Define templates for system and human messages.

These templates can include placeholders for dynamic content or be static text based on the task requirements.

```
system_message_template = """You are an expert Python coder. Your task is to write a very short {language} function that will {task}.
Please also provide a brief description of what the code does. Return the output as a JSON object with 'code' and 'description' fields."""
human_message_template = "{task}"
```

Step 2: Convert these templates into structured prompt templates.

Use the appropriate methods (e.g., SystemMessagePromptTemplate, HumanMessagePromptTemplate) to structure the templates.

```
from langchain_core.prompts import SystemMessagePromptTemplate, HumanMessagePromptTemplate

system_message_prompt_template = SystemMessagePromptTemplate.from_template(system_message_template)
human_message_prompt_template = HumanMessagePromptTemplate.from_template(human_message_template)
```

Step 3: Combine different prompt templates into a ChatPromptTemplate.

Integrate both system and human message templates into a unified prompt structure that the LLM will process.

```
from langchain_core.prompts import ChatPromptTemplate
```

```
chat_prompt_template = ChatPromptTemplate.from_messages(
    [system_message_prompt_template, human_message_prompt_template]
)
```

Step 4: Format the prompt by replacing placeholders with specific values using .format_prompt().

This step involves dynamically inserting actual data (e.g., task details, language) into the template placeholders.

```
formatted_prompt = chat_prompt_template.format_prompt(language="python", task="return a list of numbers")
```

Step 5: Convert the formatted prompt into a list of messages using .to_messages().

This step prepares the structured prompt for submission to the LLM.

```
final_prompt_messages = formatted_prompt.to_messages()
```

Step 6: Use the .invoke() method to send the formatted and structured prompt to the LLM.

The LLM processes the prompt and generates a response based on the input provided.

```
from langchain_openai import ChatOpenAI


llm = ChatOpenAI(openai_api_key="your_api_key")
llm_response = llm.invoke(final_prompt_messages)
```

Step 7: Extract the content of the LLM's response for further use in your application.

The response content can be used directly or processed further depending on the application's requirements.

```
response_content = llm_response.content
print("Generated Code:", response_content)
```

Step 8: If maintaining an ongoing conversation, save the prompt and response to memory to

retain context.

This step ensures continuity in conversation by storing past interactions, enabling the LLM to remember previous exchanges.

```
from langchain.memory import ConversationSummaryMemory


memory = ConversationSummaryMemory(
    memory_key="messages",
    return_messages=True,
    llm=llm
)


# Save the context to memory
memory.save_context({'input': final_prompt_messages}, {'output': llm_response.content})


# Retrieve conversation history
history_messages = memory.load_memory_variables({'input': 'messages'})['messages']
```