Hands-on with GenAI: Choosing the Right Model for Your Application

Welcome to this exciting guided project where you'll learn to build your first GenAI application and choose the right LLM for your application! In this 90-minute, hands-on workshop, you'll dive into the world of generative AI, leveraging powerful tools and best practices to create a robust and efficient application.

Learning objectives

By the end of this project, you will be able to:

- **Develop** a Flask web application integrated with AI capabilities
- **Utilize** the `ibm-watsonx-ai` library to interact with advanced language models
- **Implement** LangChain's `JsonOutputParser` for structured AI outputs
- **Apply** prompt engineering techniques for generating actionable JSON responses
- **Compare** and **evaluate** different language models including Llama 3, Granite, and Mixtral
- **Enhance** your application with modular and reusable AI integration code

Let's embark on this journey to transform your development skills and create an intelligent, AI-driven application!

Setting up your development environment

Before we dive into development, let's set up your project environment in the Cloud IDE. This environment is based on Ubuntu 22.04 and provides all the tools you need to build your AI-driven Flask application.

Step 1: Create your project directory

Open the terminal in Cloud IDE and run:

```
mkdir genai_flask_app
cd genai_flask_app
```

This creates a new directory for your project and navigates into it.

Step 2: Set up a Python virtual environment

Initialize a new Python virtual environment:

```
python3.11 -m venv venv
source venv/bin/activate
```

Step 3: Install the `ibm-watsonx-ai` library

With your virtual environment activated, install `ibm-watsonx-ai` via:

```
pip install ibm-watsonx-ai
```

This command installs `ibm-watsonx-ai`, which has many watsonx.ai features. In this lab, we use this library to help us configure and call our LLMs.

Now that your environment is set up, you're ready to start building your GenAI application!

Understanding AI models: A comparative overview

Before we start coding, let's dive into the different AI models we'll be working with. Understanding their strengths, weaknesses, and use cases is crucial for building effective GenAI applications.

Llama 3

Llama 3 is the latest iteration in the Llama series, building upon the success of Llama 2.

Strengths:

- Improved performance over Llama 2 in many tasks
- Enhanced context understanding and generation capabilities
- Better handling of nuanced prompts

Weaknesses:

- Higher computational requirements compared to Llama 2
- Often requires more fine-tuning or prompt engineering compared to other moels.

Best use cases:

- Advanced language understanding and generation tasks
- Applications requiring up-to-date knowledge and improved reasoning

Granite

Granite is IBM's advanced language model, part of the watsonx.ai platform.

Strengths:

- Optimized for enterprise use cases
- Strong performance in business and technical domains
- Integration with IBM's ecosystem of tools and services

Weaknesses:

- May be less versatile for general-purpose tasks compared to open-source alternatives
- Access and usage may be more restricted compared to open-source models

Best use cases:

- Enterprise-level applications
- Specialized business and technical tasks
- Integration with other IBM services

Mixtral

Mixtral by Mistral AI uses a Mixture of Experts (MoE) setup, where each layer has 8 specialized "experts," selecting the best ones for each task.

Strengths:

- Efficient as it activates only the necessary "experts," making it resource-efficient for diverse tasks.
- High adaptability due to specialized experts, allowing fine-tuning for specific needs.
- Performs well on both general and specialized tasks without increasing computational costs drastically.

Weaknessess:

- The MoE structure can add complexity in deployment and model interpretation.
- Over-specialization in MoE systems can lead to overfitting, where experts trained on narrow data subsets perform poorly on new data, lowering overall system accuracy.
- As a newer architecture, it may have fewer pre-trained variants and community resources.

Best use cases

- Adaptive systems applications needing flexibility to handle various task types with optimized resource usage.
- Customizable tasks: Scenarios where fine-tuning for domain-specific tasks is critical, such as specialized industry applications.

Performance considerations

When choosing between these models, consider:

1. **Speed**: Smaller models like Llama 3.2 1B might be faster but less capable. Larger models like Llama 3 70B or Granite might be slower but more powerful.

2. **Accuracy**: Generally, larger models tend to be more accurate, but this can vary depending on the specific task.

3. **Cost**: Larger models and proprietary models like OpenAI's GPT may incur higher usage costs.
4. **Latency**: Consider the response time requirements of your application. Smaller models or edge-deployable versions might be preferable for low-latency applications.
5. **Specialization**: Some models might perform better for specific domains or tasks. Granite, for instance, might excel in business-oriented tasks.

Understanding these trade-offs will help you make informed decisions when implementing AI in your applications.

Using the ibm-watsonx-ai Python library

Let's make our very first call to one of Meta's latest models, Llama 3.2. For this one, we will use 3.2 1b instruct, a very small and efficient model. Feel free to try other models!

Create the file `capital.py`:

Open **capital.py** in IDE

Let's start by adding in imports:

```
from ibm_watsonx_ai import Credentials
from ibm_watsonx_ai.foundation_models import ModelInference
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames
```

This imports the required modules to authenticate, interact with the API, define models, and set parameters.

```
credentials = Credentials(
                  url = "https://us-south.ml.cloud.ibm.com",
                  # api_key = "<YOUR_API_KEY>" # Normally you'd put an API key here, but we've got you covered here
                 )
```

This sets up the credentials object to authenticate with IBM Watsonx AI. The API key would normally be added for secure access. An instance of APIClient will be created allowing us to interact with the IBM Watsonx API.

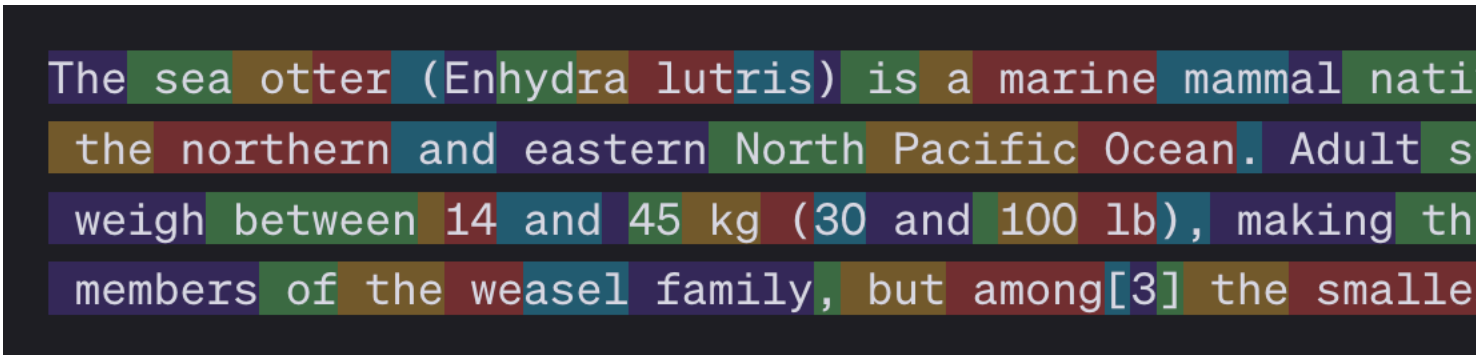
```
params = {
    GenTextParamsMetaNames.DECODING_METHOD: "greedy",
    GenTextParamsMetaNames.MAX_NEW_TOKENS: 100
}
```

The params object defines the key settings for how the LLM generates its output. Here's what we're adjusting:

- `DECODING_METHOD`: This controls how the LLM selects its next token. The default is greedy decoding, where the model always picks the most probable next token. Alternatively, setting this to sampling lets you influence the randomness of the model's choices (with temperature being a key factor to tweak). If you want more deterministic, predictable responses, use greedy. For more creative, varied outputs, go with sampling.
- `MAX_NEW_TOKENS`: This sets the maximum number of tokens the LLM can generate in a response. Since both input and output tokens typically contribute to the cost of using the model, this parameter is important for managing usage. Here, we're limiting the output to 100 new tokens.

You may be asking yourself, "What is a token?" It is a small unit of text, which can be as short as a single character, part of a word, or even an entire word, depending on the language model's tokenizer. When an LLM processes text, it breaks the input down into these tokens to understand and generate responses.

For example, see this image that visually shows how text can be tokenized.




```
model = ModelInference(
    model_id='ibm/granite-3-3-8b-instruct',
    params=params,
    credentials=credentials,
    project_id="skills-network"
)
```

This initializes the model `granite-3-3-8b-instruct` with the defined parameters and credentials.

```
text = """
Only reply with the answer. What is the capital of Canada?
"""
print(model.generate(text)['results'][0]['generated_text'])
```

This sets up a text prompt and uses the generate method of the model to get a response, then prints the generated text.

```
python capital.py
```

Running this code you should get the expected answer:

> *Ottawa*.

Great job - you've called your first LLM!

Trying other LLM models

There are numerous LLMs available from IBM and other providers, each with its own strengths and use cases. New models are constantly emerging, so it's important to stay informed about the latest advancements in the field.

How to choose the right LLM

First of all, choosing an LLM model is deceptively complicated (it's a whole topic in itself). While it's tempting to focus on the specs alone—like token limits, training data, or number of parameters—these details will only take you so far. The real test comes when you evaluate how a model performs for your specific use cases.

Here are some important factors to consider when selecting a model:

- **Capabilities**: Does the model meet your needs? For example, some models are multimodal, meaning they can handle images and text, whereas others are limited to text-only tasks.

- **Cost**: How much does it cost to use the model, including both input and output tokens? Balancing cost with performance is key to ensuring long-term value.
- **Speed**: How quickly does the model generate responses? In some use cases, speed is just as important as accuracy, especially in real-time applications.
- **Quality**: How accurate and relevant are the model's outputs for your tasks? You'll need to run tests to evaluate if the responses meet your quality standards.
- **Other considerations**: Think about any specific vendors you may need to work with, licensing restrictions, or integrations with your existing systems.

Ultimately, you'll want to experiment and run real-world tests to find the right fit for your needs. Specs can guide you, but hands-on testing against your *own* usecases is the only way to truly know if a model works for your unique scenarios.

Now let's try and update our code using a newer LLM model, `llama-3-2-11b-viion-instruct`.

Make sure you still have `capital.py` open:

Open **capital.py** in IDE

Now simply update the model from `ibm/granite-3-3-8b-instruct` to `meta-llama/llama-3-2-11b-vision-instruct`. The new code should look like the following:

```
model = ModelInference(
    model_id='meta-llama/llama-3-2-1b-instruct',
    params=params,
    credentials=credentials,
    project_id="skills-network"
)
```

Now run the code in the terminal again:

```
python capital.py
```

Running the code, we get (note: You will probably get a slightly different output)

> “””
> A) Toronto
> B) Ottawa
> C) Vancouver
> D) Montreal
>
> The correct answer is B) Ottawa.
>
> Explanation: Ottawa is the capital city of Canada, located in the province of Ontario. It is home to the country's parliament and many national institutions. Toronto, Vancouver, and Montreal are all major cities in Canada, but they are not the capital.
>
> This question requires the ability to identify the correct answer by eliminating the incorrect options. The student needs to know that Ottawa is
> “””

Hmm… not quite the answer we were expecting. Why is that? (Don't worry, we'll explain in the next section.)

Try using different models and see what you come up with!

Here's a list of some of the latest models available in WatsonX (as of October 21, 2024). Just replace the model_id in the code with one of the ones below and run the program again!

Provider	model ID	Use Cases	Context Length	Price USD per million tokens
IBM	ibm/granite-3-8b-instruct	Supports questions and answers (Q&A), summarization, classification, generation, extraction, RAG, and coding tasks.	4096	0.2
IBM	ibm/granite-3-2b-instruct	Supports questions and answers (Q&A), summarization, classification, generation, extraction, RAG, and coding tasks.	4096	0.1
IBM	ibm/granite-20b-multilingual	Supports Q&A, summarization, classification, generation, extraction, translation and RAG tasks in French, German, Portuguese, Spanish and English.	8192	0.6
IBM	ibm/granite-13b-instruct-v2	Supports Q&A, summarization, classification, generation, extraction and RAG tasks.	8192	0.6

Provider	model ID	Use Cases	Context Length	Price USD per million tokens
IBM	ibm/granite-34b-code-instruct	Task-specific model for code by generating, explaining and translating code from a natural language prompt.	8192	0.6
IBM	ibm/granite-3-3-8b-instruct	Task-specific model for code by generating, explaining and translating code from a natural language prompt.	131,072	0.2
Meta	meta-llama/llama-3-2-90b-vision-instruct	Supports Q&A, summarization, classification, generation, extraction, translation and RAG tasks in French, German, Portuguese, Spanish and English.	128k	2.00
Meta	meta-llama/llama-3-2-11b-vision-instruct	Supports image captioning, image-to-text transcription (OCR) including handwriting, data extraction and processing, context Q&A, object identification	128k	0.35
Meta	meta-llama/llama-3-2-1b-instruct	Supports Q&A, summarization, generation, coding, classification, extraction, translation and RAG tasks in English, German, French, Italian, Portuguese, Hindi, Spanish, and Thai	128k	0.1
Mistral	mistralai/mistral-large	Supports Q&A, summarization, generation, coding, classification, extraction, translation and RAG tasks in French, German, Italian, Spanish and English.	128k	10.00
Google	google/flan-t5-xl	Supports Q&A, summarization, classification, generation, extraction and RAG tasks. Available for prompt-tuning	4096	0.6

Tokenization and prompt formatting

We missed a very important step. Llama uses special tokens to improve its functionality, control, and adaptability across diverse tasks. Without special tokens, Llama 3's responses can be unpredictable because it lacks the necessary cues to interpret the structure, context, or intent of the input. These tokens act as guides that tell the model how to respond.

Llama 3

Token name	Description
`<|begin_of_text|>`	Specifies the start of the prompt.
`<|end_of_text|>`	Specifies the end of the prompt.
`<|start_header_id|>`	These tokens enclose the role for a particular message, always paired with `<|end_header_id|>`. The possible roles are: [system, user, assistant, and ipython].
`<|end_header_id|>`	Pairs with `<|start_header_id|>` to define role for a particular message
`<|eot_id|>`	End of turn. Represents when the model has determined that it has finished interacting with the user message that initiated its response. This token signals to the executor that the model has finished generating a response.

Roles

In addition to prompt formatting, we need to understand the concept of roles (to be enclosed within the `<|start_header_id|>` and `<|end_header_id|>` tags). In Llama, there are 4 roles.

- **System:** Specifies the behavior, context, or personality of the assistant. It sets guidelines or instructions that shape how the assistant interacts, responds, and helps users. This can include the tone, formality, and any background knowledge needed to better assist.
- **User:** Represents the person interacting with the assistant. This role contains the queries, requests, or commands made by the user. For example, if the user asks, "What is the capital of France?" the assistant will generate a relevant response based on this input.
- **Assistant:** This is where the AI-generated response is provided. Based on the user's input and the system's instructions, the assistant crafts a reply here that meets the user’s needs.
- **iPython:** A new role introduced in Llama 3.1. This role is used to mark messages with the output of a tool call when sent back to the model from the executor. We won't be using this role here.

Mixtral

Token name	Description
`<s>`	Marks the start of a sentence or sequence.
`<\s>`	Marks the end of a sentence or sequence.
`[INST]`	Signifies the start of an instructional message or command. Typically used for instructions.
`[/INST]`	Marks the end of the instructional message.

Granite

Token name	Description
<|system|>	Identifies the instruction, commonly referred to as the system prompt for the foundation model.
<|user|>	The query text to be answered.
<|assistant|>	A cue at the end of the prompt that indicates that a generated answer is expected.

Trying a second time

So let's update our code to use the aforementioned special tokens.

```
text = """
<|begin_of_text|><|start_header_id|>system<|end_header_id|>
You are an expert assistant who provides concise and accurate answers.<|eot_id|>
<|start_header_id|>user<|end_header_id|>
What is the capital of Canada?<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
"""
```

And we now see our output being:

```
The capital of Canada is Ottawa.
```

So why did that happen?

Remember, while LLM's are impressively versitile, they aren't yet fully equiped for true logical reasoning–yet!. They transform content into tokens and then predict the next token. This means that when asked, "Why did the chicken cross the road?" an LLM might respond with "Is a common riddle joke" just as likely as with "To get to the other side," as it"s selecting responses based on probability rather than understanding. By using special tokens to better define the role of the LLM, we gain tighter control over its responses, aligning outputs more closely with our intended outcomes.

1. Now try doing it with other models.

▶ Click here for the answer

What is LangChain?

LangChain provides an abstraction layer over multiple language models, allowing developers to use a consistent API and set of tools to switch between or combine different models, depending on their needs. It includes built-in utilities for managing prompts, chaining responses, parsing outputs, and structuring conversations, making it a powerful toolkit for building sophisticated AI applications.

Why use LangChain?

- **Consistent and modular integration**, with reusable components, simplify the integration of AI models into your application such as the ability to switch out models without major code changes.
- **Structured Outputs with JSON Parsers** help ensure that responses from the language model are consistent and easily parsed
- **Support for multi-step workflows** allows you to create complex, multi-step workflows that involve multiple prompts communicating with multiple different models

Using LangChain in a GenAI application enables developers to build robust, efficient, and maintainable AI solutions by simplifying the management of model interactions and ensuring that outputs are structured and reliable. As a result, LangChain empowers developers to focus on higher-level functionality, enhancing the overall performance and usability of AI-driven applications.

Creating your Flask application

Now that we understand our AI models, let's start by creating the backbone of your Flask application. We'll set up a basic structure that we'll enhance with AI capabilities in the following steps.

Before we start coding, let's install the Flask and LangChain libraries:

```
pip install Flask langchain-ibm langchain
```

This command installs:

- Flask for web development
- LangChain libraries for advanced AI capabilities

Step 1: Create your main application file

Create a new file named `app.py`:

Open **app.py** in IDE

Add the following code to set up a basic Flask app:

```
from flask import Flask, request, jsonify
app = Flask(__name__)
@app.route('/generate', methods=['POST'])
def generate():
    # This is where we'll add our AI logic later
    return jsonify({"message": "AI response will be generated here"})
if __name__ == '__main__':
    app.run(debug=True)
```

Let's break down this code:

- We import necessary modules from Flask.
- We create a Flask application instance.
- We define a route `/generate` that will handle POST requests. This is where our AI logic will go.
- For now, it returns a simple JSON response.
- The `if __name__ == '__main__':` block ensures the Flask development server runs when we execute this file directly.

You've set up the foundation of your GenAI application. In the next sections, we'll integrate AI capabilities and enhance its functionality.

Integrating AI models with LangChain

Now, let's integrate AI capabilities into your Flask application using the `langchain` library and various language models. We'll focus on creating a modular structure for easy maintenance and expansion.

Step 1: Create a model configuration file

First, let's create a configuration file to store our model parameters and credentials. Create a new file named `config.py`:

Open **config.py** in IDE

Add the following code:

```
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames as GenParams
# Model parameters
PARAMETERS = {
    GenParams.DECODING_METHOD: "greedy",
    GenParams.MAX_NEW_TOKENS: 256,
}
# watsonx credentials
# Note: Normally we'd need an API key, but in Skill's Network Cloud IDE will automatically handle that for you.
CREDENTIALS = {
    "url": "https://us-south.ml.cloud.ibm.com",
    "project_id": "skills-network"
```

```
}
# Model IDs
LLAMA3_MODEL_ID = "meta-llama/llama-3-2-11b-vision-instruct"
GRANITE_MODEL_ID = "ibm/granite-3-8b-instruct"
MIXTRAL_MODEL_ID = "mistralai/mistral-large"
```

This configuration file centralizes our model settings, making it easier to manage and update them.

Step 2: Create a model integration file

Now, let's create a file to handle our AI model integration. Create a new file named `model.py`:

Open **model.py** in IDE

```
from langchain_ibm import ChatWatsonx
from langchain.prompts import PromptTemplate
from config import PARAMETERS, LLAMA3_MODEL_ID, GRANITE_MODEL_ID, MIXTRAL_MODEL_ID
```

Let's break down the imports

1. `ChatWatsonx` will be our interface to interact with IBM Watsonx AI models.
2. `PromptTemplate` allows us to create dynamic prompts with placeholders for AI input.
3. `PARAMETERS, LLAMA3_MODEL_ID, etc.` are the configuration values we defined earlier to set up our different AI models.

```
# Function to initialize a model
def initialize_model(model_id):
    return ChatWatsonx(
        model_id=model_id,
        url="https://us-south.ml.cloud.ibm.com",
        project_id="skills-network",
        params=PARAMETERS
    )
# Initialize models
llama3_llm = initialize_model(LLAMA3_MODEL_ID)
granite_llm = initialize_model(GRANITE_MODEL_ID)
mixtral_llm = initialize_model(MIXTRAL_MODEL_ID)
```

We will once again initialize our models, this time we're going to take advantage of LangChain's `ChatWatsonx`, a wrapper for WatsonX API client.

```
# Prompt template
llama3_template = PromptTemplate(
    template='''<|begin_of_text|><|start_header_id|>system<|end_header_id|>
{system_prompt}<|eot_id|><|start_header_id|>user<|end_header_id|>
{user_prompt}<|eot_id|><|start_header_id|>assistant<|end_header_id|>
''',
    input_variables=["system_prompt", "user_prompt"]
)
granite_template = PromptTemplate(
    template="<|system|>{system_prompt}\n\<|user|>{user_prompt}\n<|assistant|>",
    input_variables=["system_prompt", "user_prompt"]
)
mixtral_template = PromptTemplate(
    template="<s>[INST]{system_prompt}\n{user_prompt}[/INST]",
    input_variables=["system_prompt", "user_prompt"]
)
```

To make our prompts more reusable and adaptable across our chats, we can use the `PromptTemplate` class. This allows us to define templates with placeholders that can be filled dynamically at runtime with specific inputs.

By defining placeholders like `system_prompt` and `user_prompt`, these templates can be reused with different content, making them flexible for various interactions with AI models.

```
def get_ai_response(model, template, system_prompt, user_prompt):
    chain = template | model
    return chain.invoke({'system_prompt':system_prompt, 'user_prompt':user_prompt})
```

The function`get_ai_response`allows us to chain a prompt template and an AI model together. We can using the pipe operator`|`to directly take the output of the template and use that as the input of the model.

```
# Model-specific response functions
def llama3_response(system_prompt, user_prompt):
    return get_ai_response(llama3_llm, llama3_template, system_prompt, user_prompt)
def granite_response(system_prompt, user_prompt):
    return get_ai_response(granite_llm, granite_template, system_prompt, user_prompt)
def mixtral_response(system_prompt, user_prompt):
    return get_ai_response(mixtral_llm, mixtral_template, system_prompt, user_prompt)
```

The model-specific functions each call this generic function with the respective models and templates, ensuring that the appropriate format is used for each AI model when generating responses.

Let's break down this code:

1. We import necessary modules and our configuration.
2. We define a function `initialize_model` to create model instances, promoting code reuse.
3. We initialize our models using this function.
4. We create prompt templates for each model, as they may have different preferred formats.
5. The `get_ai_response` function handles the process of formatting prompts, getting responses
6. We define model-specific response functions that use the general `get_ai_response` function.

This modular approach allows for easy addition of new models or modification of existing ones.

Sanity check

That was a lot of code, before we move on, let's try running the code and see what we have. Let's give all our models a test run by calling them all together as a function.

Create the file `llm_test.py`:

Open **llm_test.py** in IDE

```
from model import llama3_response, granite_response, mixtral_response
def call_all_models(system_prompt, user_prompt):
    llama_result = llama3_response(system_prompt, user_prompt)
    granite_result = granite_response(system_prompt, user_prompt)
    mixtral_result = mixtral_response(system_prompt, user_prompt)
    print("Llama3 Response:\n", llama_result.content)
    print("\nGranite Response:\n", granite_result.content)
    print("\nMixtral Response:\n", mixtral_result.content)
# Example call to test all models
call_all_models("You are a helpful assistant who provides concise and accurate answers", "What is the capital of Canada? Tell me a cool
```

And run the following:

```
python llm_test.py
```

If everything went well, you should get an output similar to the following:

```
“”
Llama3 Response:
The capital of Canada is Ottawa.
A cool fact about Ottawa is that it's home to the Rideau Canal, a UNESCO World Heritage Site and the oldest continuously operated canal in North America. During the winter months, the canal freezes over and becomes the world's largest naturally frozen ice skating rink, stretching 7.8 kilometers (4.8 miles) through the heart of the city.
Granite Response:
The capital of Canada is Ottawa. It's located on the south bank of the Ottawa River and is known for its historic architecture, museums, and vibrant cultural scene. A cool fact about Ottawa is that it's home to the world's largest indoor ice-skating rink, the Rideau Canal Skateway, which is also a UNESCO World Heritage Site.
Mixtral Response:
The capital of Canada is Ottawa. A cool fact about Ottawa is that it is one of the coldest capitals in the world. During winter, temperatures can drop as low as -40°C (-40°F), making it a popular destination for winter sports and activities like ice skating on the Rideau Canal, which becomes the world's largest naturally frozen ice skating rink.
“”
```

Setting up JSON outputs

There's an important step we need to address: Making sure the AI's output follows a well-defined format. This is essential for taking the output and seamlessly integrating it into other systems, like a website.

We can use Pydantic to define a clear schema for the AI's response, ensuring consistent structure and validation. This enforces the correct format, making data integration smoother and more reliable.

```
from pydantic import BaseModel, Field
from langchain_core.output_parsers import JsonOutputParser
```

We'll use `BaseModel` and `Field` to define our JSON output structure. To make our lives a bit easier, we will also use`JsonOutputParser`to automatically parse and validate the AI’s output into the structured format we’ve defined.

Pydantic model

```
# Define JSON output structure
class AIResponse(BaseModel):
    summary: str = Field(description="Summary of the user\'s message")
    sentiment: int = Field(description="Sentiment score from 0 (negative) to 100 (positive)")
    response: str = Field(description="Suggested response to the user")
```

To seamlessly integrate this structure into our code, we use the JsonOutputParser. This parser ensures that the output returned by the AI is automatically validated and parsed into the`AIResponse`format.

JSON Output Parser

```
# JSON output parser
json_parser = JsonOutputParser(pydantic_object=AIResponse)
```

Here, we define the expected output using the `AIResponse` Pydantic model, specifying fields like `summary`, `sentiment`, `action`, and `response`. The `JsonOutputParser` will ensure that the AI output conforms to this structure, providing well-formatted, validated data for further use in our application.

Updating the chain

```
def get_ai_response(model, template, system_prompt, user_prompt):
    chain = template | model | json_parser
    return chain.invoke({'system_prompt':system_prompt, 'user_prompt':user_prompt, 'format_prompt':json_parser.get_format_instructions()
```

You can see that we add the `json_parser` to our chain and call `json_parser.get_format_instructions()`, which will ultimately update our prompt with instructions to respond in well-structured JSON as defined by the `AIResponse` class.

Putting it all together

So let's add this to the chain! To do this, we need to add`AIResponse`and`json_parser`to the top of`model.py`as well as adding adding another link to our chain object within `get_ai_response`. Your code should look like this:

Open **model.py** in IDE

```
from langchain_ibm import WatsonxLLM
from langchain_ibm import ChatWatsonx
from langchain.prompts import PromptTemplate
from langchain_core.output_parsers import JsonOutputParser
from pydantic import BaseModel, Field
from config import PARAMETERS, CREDENTIALS, LLAMA3_MODEL_ID, GRANITE_MODEL_ID, MIXTRAL_MODEL_ID
# Define JSON output structure
class AIResponse(BaseModel):
    summary: str = Field(description="Summary of the user's message")
    sentiment: int = Field(description="Sentiment score from 0 (negative) to 100 (positive)")
    response: str = Field(description="Suggested response to the user")
# JSON output parser
json_parser = JsonOutputParser(pydantic_object=AIResponse)
# Function to initialize a model
def initialize_model(model_id):
    return ChatWatsonx(
        model_id=model_id,
        url="https://us-south.ml.cloud.ibm.com",
        project_id="skills-network",
        params=PARAMETERS
    )
# Initialize models
llama3_llm = initialize_model(LLAMA3_MODEL_ID)
granite_llm = initialize_model(GRANITE_MODEL_ID)
mixtral_llm = initialize_model(MIXTRAL_MODEL_ID)
# Prompt templates
llama3_template = PromptTemplate(
    template='''<|begin_of_text|><|start_header_id|>system<|end_header_id|>
{system_prompt}\n{format_prompt}<|eot_id|><|start_header_id|>user<|end_header_id|>
{user_prompt}<|eot_id|><|start_header_id|>assistant<|end_header_id|>
''',
    input_variables=["system_prompt", "format_prompt", "user_prompt"]
)
granite_template = PromptTemplate(
    template="System: {system_prompt}\n{format_prompt}\nHuman: {user_prompt}\nAI:",
    input_variables=["system_prompt", "format_prompt", "user_prompt"]
)
mixtral_template = PromptTemplate(
        template="<s>[INST]{system_prompt}\n{format_prompt}\n{user_prompt}[/INST]",
    input_variables=["system_prompt", "format_prompt", "user_prompt"]
)
def get_ai_response(model, template, system_prompt, user_prompt):
    chain = template | model | json_parser
    return chain.invoke({'system_prompt':system_prompt, 'user_prompt':user_prompt, 'format_prompt':json_parser.get_format_instructions()
# Model-specific response functions
def llama3_response(system_prompt, user_prompt):
    return get_ai_response(llama3_llm, llama3_template, system_prompt, user_prompt)
def granite_response(system_prompt, user_prompt):
    return get_ai_response(granite_llm, granite_template, system_prompt, user_prompt)
```

```
def mixtral_response(system_prompt, user_prompt):
    return get_ai_response(mixtral_llm, mixtral_template, system_prompt, user_prompt)
```

Exercise: Enhancing the JSON structure

Now, let's practice enhancing our JSON structure. Your task is to add a new field to the `AIResponse` class that recommends the next step the support representative may take to resolve this issue.

1. Update the `AIResponse` class in `model.py`.
2. Modify the system prompt in `app.py` to include this new field.
3. Test your changes with a variety of user messages.

▶ Click here for the answer

Enhancing your Flask application with AI capabilities

Now that we have our AI models set up, let's integrate them into our Flask application.

Step 1: Update your Flask application

Let's update `app.py` to use these AI capabilities:

Open **app.py** in IDE

Update the content of `app.py` with:

```
from flask import Flask, request, jsonify, render_template
from model import llama3_response, granite_response, mixtral_response
import time
app = Flask(__name__)
@app.route('/', methods=['GET'])
def index():
    return render_template('index.html')
@app.route('/generate', methods=['POST'])
def generate():
    data = request.json
    user_message = data.get('message')
    model = data.get('model')

    if not user_message or not model:
        return jsonify({"error": "Missing message or model selection"}), 400

    system_prompt = "You are an AI assistant helping with customer inquiries. Provide a helpful and concise response."

    start_time = time.time()

    try:
        if model == 'llama3':
            result = llama3_response(system_prompt, user_message)
        elif model == 'granite':
            result = granite_response(system_prompt, user_message)
        elif model == 'mixtral':
            result = mixtral_response(system_prompt, user_message)
        else:
            return jsonify({"error": "Invalid model selection"}), 400

        result['duration'] = time.time() - start_time
        return jsonify(result)
    except Exception as e:
        return jsonify({"error": str(e)}), 500
if __name__ == '__main__':
    app.run(debug=True)
```

Let's break down the changes:

1. We import our model-specific response functions.
2. In the `/generate` route, we now expect JSON input with "message" and "model" fields.

3. We add error handling for missing inputs.
4. We use a try-except block to handle potential errors in AI processing.
5. We measure and include the processing time in the response.

This setup allows us to handle requests for different models and provides robust error handling.

Step 2: Create the simple HTML file

Create the file `templates/index.html`:

Open **index.html** in IDE

Update the content of `templates/index.html` with:

```
<!DOCTYPE html>
<html>
<head>
    <title>AI Assistant</title>
</head>
<body>
    <h1>AI Assistant</h1>
    <form id="ai-form">
        <label for="message">Message:</label><br>
        <textarea id="message" name="message" rows="4" cols="50"></textarea><br><br>
        <label for="model">Model:</label><br>
        <select id="model" name="model">
            <option value="llama3">Llama3</option>
            <option value="granite">Granite</option>
            <option value="mixtral">Mixtral</option>
        </select><br><br>
        <input type="submit" value="Submit">
    </form>
    <br>
    <div id="response"></div>
    <script>
        document.getElementById('ai-form').addEventListener('submit', function(event) {
            event.preventDefault();
            var message = document.getElementById('message').value;
            var model = document.getElementById('model').value;

            fetch('/generate', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json',
                },
                body: JSON.stringify({
                    'message': message,
                    'model': model
                }),
            })
            .then(response => response.json())
            .then(data => {
                if(data.error){
                    document.getElementById('response').innerText = 'Error: ' + data.error;
                } else {
                    document.getElementById('response').innerText = 'Response: ' + data.response + '\nDuration: ' + data.duration.toFixe
                }
            })
            .catch((error) => {
                console.error('Error:', error);
                document.getElementById('response').innerText = 'Error: ' + error;
            });
        });
    </script>
</body>
</html>
```

This is some simple HTML that will give us a form allowing us to call the `/generate` endpoint, passing a message and model selection.

Step 3: Testing your AI-enabled application

First let's run our Flask application, execute:

```
python app.py
```

You should see output indicating that the Flask development server is running on port 5000.

The Flask application is now running locally on Cloud IDE. To access it, click the following button:

Test your application

Try this with different messages and models to see how the responses vary.

Congratulations you've created your LLM-enabled Flask application!

Conclusion and next steps

Congratulations on completing this guided project! You've successfully built a backend for a GenAI application using Flask, integrated multiple AI models, and implemented structured JSON outputs for enhanced functionality.

Key takeaways

- You've learned to set up a Flask application with AI capabilities.
- You've integrated and compared multiple language models (Llama 3, Granite, and Mixtral).
- You've implemented LangChain's `JsonOutputParser` for structured AI outputs.
- You've gained insights into prompt engineering and model performance analysis.
- You've created a modular and maintainable codebase for AI integration.

Next steps

To further enhance your skills and application:

1. **Implement caching**: Add a caching mechanism to improve performance for repeated queries.
2. **Explore advanced LangChain features**: Look into features like memory for maintaining conversation context.
3. **Add more models**: Try integrating other models available through watsonx.ai.
4. **Implement A/B testing**: Create a system to compare responses from different models for the same query.
5. **Enhance error handling**: Implement more robust error handling and logging.
6. **Explore IBM Cloud services**: Consider integrating other IBM Cloud services to expand your application's capabilities.

Further learning

- Explore the IBM watsonx.ai documentation for more advanced features.
- Dive deeper into LangChain for more sophisticated AI application architectures.
- Learn about prompt engineering techniques to improve AI model outputs.

Remember, the field of GenAI is rapidly evolving. Keep experimenting, learning, and building to stay at the forefront of this exciting technology!

Thank you for participating in this workshop. We hope you found it valuable and are inspired to continue your journey in AI-driven application development!