

动手实践GenAI：为您的应用选择合适的模型

欢迎来到这个激动人心的指导项目，在这里您将学习构建您的第一个GenAI应用，并选择适合您应用的LLM！在这个90分钟的动手工作坊中，您将深入探索生成性AI的世界，利用强大的工具和最佳实践来创建一个健壮高效的应用。

学习目标

到项目结束时，您将能够：

- **开发** 一个集成AI能力的Flask web应用
- **利用** `ibm-watsonx-ai`库与高级语言模型进行交互
- **实现** LangChain的`JsonOutputParser`以获得结构化的AI输出
- **应用** 提示工程技术生成可操作的JSON响应
- **比较** 和 **评估** 不同的语言模型，包括Llama 3、Granite和Mixtral
- **增强** 您的应用，使用模块化和可重用的AI集成代码

让我们开始这段旅程，提升您的开发技能，创建一个智能的、由AI驱动的应用！

设置您的开发环境

在我们开始开发之前，让我们在云 IDE 中设置您的项目环境。该环境基于 Ubuntu 22.04，并提供构建您的 AI 驱动 Flask 应用程序所需的所有工具。

第一步：创建您的项目目录

在云 IDE 中打开终端并运行：

```
mkdir genai_flask_app
cd genai_flask_app
```

这将为您的项目创建一个新目录并进入其中。

步骤 2：设置 Python 虚拟环境

初始化一个新的 Python 虚拟环境：

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

第3步：安装 `ibm-watsonx-ai` 库

在激活虚拟环境后，通过以下命令安装 `ibm-watsonx-ai`：

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

此命令安装了 `ibm-watsonx-ai`，它具有许多 `watsonx.ai` 的功能。在本实验中，我们使用这个库来帮助我们配置和调用我们的 LLM。

现在您的环境已设置好，您可以开始构建您的 GenAI 应用程序！

理解AI模型： 比较概述

在我们开始编码之前，让我们深入了解将要使用的不同AI模型。理解它们的优缺点和应用场景对于构建有效的GenAI应用至关重要。

Llama 3

Llama 3是Llama系列的最新版本，建立在Llama 2的成功基础上。

优点：

- 在许多任务上相较于Llama 2有了性能提升
- 增强了上下文理解和生成能力
- 更好地处理细微的提示

缺点：

- 与Llama 2相比，计算需求更高
- 相较于其他模型，通常需要更多的微调或提示工程

最佳应用场景：

- 高级语言理解和生成任务
- 需要最新知识和改进推理的应用

Granite

Granite是IBM的高级语言模型，是watsonx.ai平台的一部分。

优点：

- 针对企业用例进行了优化
- 在商业和技术领域表现强劲
- 与IBM的工具和服务生态系统集成

缺点：

- 相较于开源替代品，可能在通用任务上灵活性较差
- 访问和使用可能比开源模型更受限制

最佳应用场景：

- 企业级应用
- 专业的商业和技术任务
- 与其他IBM服务的集成

Mixtral

Mistral AI的Mixtral使用专家混合（MoE）设置，每层有8个专门的“专家”，为每个任务选择最佳的专家。

优点：

- 仅激活必要的“专家”，在处理多样任务时资源效率高。
- 由于有专门的专家，高适应性允许针对特定需求进行微调。
- 在一般和专业任务上表现良好，而不会大幅增加计算成本。

缺点：

- MoE结构可能在部署和模型解释上增加复杂性。
- MoE系统的过度专业化可能导致过拟合，训练在狭窄数据子集上的专家在新数据上表现不佳，从而降低整体系统的准确性。
- 作为一种较新的架构，可能有较少的预训练变体和社区资源。

最佳应用场景

- 需要灵活处理各种任务类型并优化资源使用的自适应系统应用。
- 可定制任务：在特定领域任务微调至关重要的场景，例如专业行业应用。

性能考虑

在选择这些模型时，请考虑：

1. **速度**：像Llama 3.2 1B这样的小型模型可能更快，但能力较弱。像Llama 3 70B或Granite这样的大型模型可能更慢，但更强大。

- 2. **准确性**：通常，大型模型往往更准确，但这可能因特定任务而异。
- 3. **成本**：大型模型和像OpenAI的GPT这样的专有模型可能会产生更高的使用成本。
- 4. **延迟**：考虑应用的响应时间要求。对于低延迟应用，小型模型或边缘可部署版本可能更为合适。
- 5. **专业化**：某些模型在特定领域或任务上可能表现更好。例如，Granite可能在商业导向任务中表现优异。

理解这些权衡将帮助您在应用中实施AI时做出明智的决策。

使用 ibm-watsonx-ai Python 库

让我们首次调用 Meta 最新的模型之一，Llama 3.2。对于这个模型，我们将使用 3.2 1b instruct，这是一个非常小且高效的模型。欢迎尝试其他模型！

创建文件 capital.py：

Open capital.py in IDE

让我们先添加导入：

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

这导入了所需的模块，以进行身份验证、与 API 交互、定义模型和设置参数。

```
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
)
```

这将设置凭证对象以便与 IBM Watsonx AI 进行身份验证。API 密钥通常会被添加以确保安全访问。将创建一个 APIClient 的实例，使我们能够与 IBM Watsonx API 进行交互。

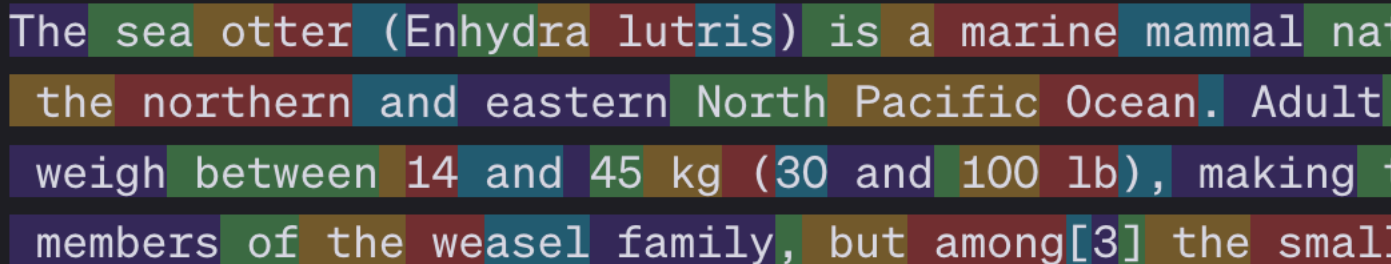
```
params = {
    GenTextParamsMetaNames.DECODING_METHOD: "greedy",
    GenTextParamsMetaNames.MAX_NEW_TOKENS: 100
}
```

params对象定义了LLM生成输出的关键设置。以下是我们正在调整的内容：

- DECODING_METHOD：这控制了LLM如何选择下一个标记。默认是贪婪解码，模型总是选择最可能的下一个标记。或者，将其设置为采样可以影响模型选择的随机性（温度是一个关键因素）。如果您希望获得更确定、可预测的响应，请使用贪婪解码。对于更具创意和多样化的输出，请选择采样。
- MAX_NEW_TOKENS：这设置了LLM在响应中可以生成的最大标记数。由于输入和输出标记通常都会影响使用模型的成本，因此此参数在管理使用时非常重要。在这里，我们将输出限制为100个新标记。

您可能会问自己：“什么是标记？”它是文本的一个小单位，可以是一个字符、一个词的一部分，甚至是一个完整的词，具体取决于语言模型的标记器。当LLM处理文本时，它将输入分解为这些标记，以理解和生成响应。

例如，请查看此图像，直观展示了文本如何被标记化。



```
model = ModelInference(
    model_id='ibm/granite-13b-instruct-v2',
    params=params,
    credentials=credentials,
    project_id="skills-network"
)
```

这将使用定义的参数和凭据初始化模型 `granite-13b-instruct-v2`。

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

这设置了一个文本提示，并使用模型的生成方法获取响应，然后打印生成的文本。

```
python capital.py
```

运行这段代码，你应该得到预期的答案：

```
_渥太华_。
```

干得好 – 你已经调用了你的第一个 LLM！

尝试其他 LLM 模型

IBM 和其他提供商提供了众多 LLM，每个模型都有其自身的优势和使用场景。新的模型不断涌现，因此保持对该领域最新进展的了解非常重要。

如何选择合适的 LLM

首先，选择 LLM 模型看似复杂（这本身就是一个完整的话题）。虽然专注于规格——如令牌限制、训练数据或参数数量——是很有诱惑力的，但这些细节只能提供有限的帮助。真正的考验在于评估模型在特定用例中的表现。

在选择模型时，有几个重要因素需要考虑：

- **能力**：模型是否满足你的需求？例如，有些模型是多模态的，意味着它们可以处理图像和文本，而其他模型仅限于文本任务。
- **成本**：使用模型的成本是多少，包括输入和输出令牌？平衡成本与性能是确保长期价值的关键。
- **速度**：模型生成响应的速度有多快？在某些用例中，速度与准确性同样重要，尤其是在实时应用中。
- **质量**：模型输出的准确性和相关性如何？你需要进行测试以评估响应是否符合你的质量标准。

- **其他考虑：**考虑一下你可能需要合作的特定供应商、许可限制或与现有系统的集成。

最终，你需要进行实验并进行真实世界的测试，以找到适合你需求的模型。规格可以为你提供指导，但针对你自己用例的实际测试才是判断模型是否适合你独特场景的唯一方法。

现在让我们试着用更新的 LLM 模型 `llama-3-2-1b-instruct` 更新我们的代码。

确保你仍然打开着 `capital.py`：

[Open `capital.py` in IDE](#)

现在只需将模型从 `ibm/granite-13b-instruct-v2` 更新为 `meta-llama/llama-3-2-1b-instruct`。新代码应如下所示：

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

现在在终端中再次运行代码：

```
python capital.py
```

运行代码时，我们得到（注意：您可能会得到稍微不同的输出）

“””

- A) 多伦多
- B) 渥太华
- C) 温哥华
- D) 蒙特利尔

正确答案是 B) 渥太华。

解释：渥太华是加拿大的首都，位于安大略省。这里是国家议会和许多国家机构的所在地。多伦多、温哥华和蒙特利尔都是加拿大的重要城市，但它们不是首都。

这个问题要求通过排除错误选项来识别正确答案。学生需要知道渥太华是
“””

嗯……这不是我们预期的答案。为什么会这样呢？（别担心，我们将在下一部分解释。）

尝试使用不同的模型，看看您能得到什么！

以下是一些在 WatsonX 中可用的最新模型列表（截至 2024 年 10 月 21 日）。只需将代码中的 `model_id` 替换为下面的其中一个，然后再次运行程序！

提供者	模型 ID	用例	上下文长度	每百万个令牌的价格（美元）
IBM	ibm/granite-3-8b-instruct	支持问答（Q&A）、摘要、分类、生成、提取、RAG 和编码任务。	4096	0.2
IBM	ibm/granite-3-2b-instruct	支持问答（Q&A）、摘要、分类、生成、提取、RAG 和编码任务。	4096	0.1
IBM	ibm/granite-20b-multilingual	支持法语、德语、葡萄牙语、西班牙语和英语的问答、摘要、分类、生成、提取、翻译和 RAG 任务。	8192	0.6
IBM	ibm/granite-13b-instruct-v2	支持问答、摘要、分类、生成、提取和 RAG 任务。	8192	0.6
IBM	ibm/granite-34b-code-instruct	针对代码的特定任务模型，通过自然语言提示生成、解释和翻译代码。	8192	0.6

提供者	模型 ID	用例	上下文长度	每百万个令牌的价格（美元）
IBM	ibm/granite-20b-code-instruct	针对代码的特定任务模型，通过自然语言提示生成、解释和翻译代码。	8192	0.6
Meta	meta-llama/llama-3-2-90b-vision-instruct	支持法语、德语、葡萄牙语、西班牙语和英语的问答、摘要、分类、生成、提取、翻译和 RAG 任务。	128k	2.00
Meta	meta-llama/llama-3-2-11b-vision-instruct	支持图像描述、图像到文本的转录（OCR），包括手写，数据提取和处理，上下文问答，物体识别	128k	0.35
Meta	meta-llama/llama-3-2-1b-instruct	支持英语、德语、法语、意大利语、葡萄牙语、印地语、西班牙语和泰语的问答、摘要、生成、编码、分类、提取、翻译和 RAG 任务	128k	0.1
Mistral	mistralai/mistral-large	支持法语、德语、意大利语、西班牙语和英语的问答、摘要、生成、编码、分类、提取、翻译和 RAG 任务。	128k	10.00
Google	google/flan-t5-xl	支持问答、摘要、分类、生成、提取和 RAG 任务。可用于提示调优	4096	0.6

标记化和提示格式化

我们错过了一个非常重要的步骤。Llama 使用特殊的标记来提高其功能性、控制力和适应性，以应对多样化的任务。没有特殊标记，Llama 3 的响应可能会变得不可预测，因为它缺乏必要的提示来解释输入的结构、上下文或意图。这些标记充当指导，告诉模型如何响应。

Llama 3

标记名称	描述
< begin_of_text >	指定提示的开始。
< end_of_text >	指定提示的结束。
< start_header_id >	这些标记包围特定消息的角色，始终与 < end_header_id > 配对。可能的角色有：[system, user, assistant, 和 ipython]。
< end_header_id >	与 < start_header_id > 配对，以定义特定消息的角色。
< eot_id >	回合结束。表示模型已经确定它完成了与用户消息的交互，该消息启动了其响应。这个标记向执行者发出信号，表明模型已完成生成响应。

角色

除了提示格式化，我们还需要理解角色的概念（需要被包围在 <|start_header_id|> 和 <|end_header_id|> 标签中）。在 Llama 中，有 4 个角色。

- System（系统）：** 指定助手的行为、上下文或个性。它设定了指导方针或指令，塑造助手如何与用户互动、响应和提供帮助。这可以包括语气、正式程度以及更好地协助所需的任何背景知识。
- User（用户）：** 代表与助手互动的人。这个角色包含用户提出的查询、请求或命令。例如，如果用户问：“法国的首都是什么？”助手将根据这个输入生成相关的响应。
- Assistant（助手）：** 在这里提供 AI 生成的响应。根据用户的输入和系统的指令，助手在此处撰写满足用户需求的回复。
- iPython：** 在 Llama 3.1 中引入的新角色。这个角色用于标记从执行者发送回模型的工具调用的输出消息。我们在这里不会使用这个角色。

Mixtral

标记名称	描述
<S>	标记句子或序列的开始。
<\S>	标记句子或序列的结束。
[INST]	表示指令消息或命令的开始。通常用于指令。
[/INST]	标记指令消息的结束。

Granite

标记名称	描述
< system >	确定指令，通常称为基础模型的系统提示。
< user >	需要回答的查询文本。
< assistant >	提示末尾的一个提示，表示期待生成的答案。

再试一次

所以让我们更新我们的代码以使用上述特殊标记。

```
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|>
"""
```

我们现在看到我们的输出是：

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

那么为什么会发生这种情况？

请记住，虽然大型语言模型（LLM）非常灵活，但它们尚未完全具备真正的逻辑推理能力。它们将内容转换为标记，然后预测下一个标记。这意味着当被问到“为什么鸡要过马路？”时，LLM可能会回答“这是一个常见的谜语笑话”，就像回答“为了到达另一边”一样，因为它是基于概率而不是理解来选择响应。通过使用特殊标记来更好地定义LLM的角色，我们可以对其响应进行更严格的控制，使输出更贴近我们的预期结果。

1. 现在尝试用其他模型来做这件事。

► [点击这里查看答案](#)

什么是 LangChain?

LangChain 提供了一个抽象层，覆盖多个语言模型，使开发者能够使用一致的 API 和工具集，根据需求在不同模型之间切换或组合。它包括内置的实用工具，用于管理提示、链接响应、解析输出和构建对话，使其成为构建复杂 AI 应用程序的强大工具包。

为什么使用 LangChain?

- **一致且模块化的集成**，可重用组件，简化将 AI 模型集成到您的应用程序中的过程，例如能够在不进行重大代码更改的情况下更换模型。
- **使用 JSON 解析器的结构化输出**，确保语言模型的响应一致且易于解析。
- **支持多步骤工作流程**，允许您创建复杂的多步骤工作流程，涉及多个提示与多个不同模型之间的沟通。

在 GenAI 应用程序中使用 LangChain，使开发者能够通过简化模型交互的管理，构建强大、高效且可维护的 AI 解决方案，并确保输出结构化且可靠。因此，LangChain 使开发者能够专注于更高层次的功能，提升 AI 驱动应用程序的整体性能和可用性。

创建你的 Flask 应用程序

现在我们已经了解了我们的 AI 模型，让我们开始创建你的 Flask 应用程序的骨架。我们将设置一个基本结构，随后在接下来的步骤中增强其 AI 功能。

在开始编码之前，让我们安装 Flask 和 LangChain 库：

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

此命令安装：

- 用于 web 开发的 Flask
- 用于高级 AI 能力的 LangChain 库

第一步：创建主应用程序文件

创建一个名为 `app.py` 的新文件：

Open **app.py** in IDE

添加以下代码以设置基本的 Flask 应用：

```
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)
```

让我们分解一下这段代码：

- 我们导入 Flask 所需的模块。
- 我们创建一个 Flask 应用实例。
- 我们定义一个处理 POST 请求的路由 `/generate`。这就是我们 AI 逻辑所在的地方。
- 目前，它返回一个简单的 JSON 响应。
- `if __name__ == '__main__':` 块确保当我们直接执行此文件时，Flask 开发服务器会运行。

您已经搭建好了 GenAI 应用的基础。在接下来的部分中，我们将集成 AI 功能并增强其功能。

将AI模型与LangChain集成

现在，让我们使用 `langchain` 库和各种语言模型将AI功能集成到您的Flask应用程序中。我们将重点创建一个模块化结构，以便于维护和扩展。

第一步：创建模型配置文件

首先，让我们创建一个配置文件来存储我们的模型参数和凭据。创建一个名为 `config.py` 的新文件：

Open **config.py** in IDE

添加以下代码：

```
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"
MISTRAL_MODEL_ID = "mistralai/mistral-large"
```


此配置文件集中管理我们的模型设置，使其更容易进行管理和更新。

第2步：创建模型集成文件

现在，让我们创建一个文件来处理我们的AI模型集成。创建一个名为 `model.py` 的新文件：

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
```

让我们来分析一下导入内容

1. `ChatWatsonx` 将是我们与 IBM Watsonx AI 模型交互的接口。
2. `PromptTemplate` 允许我们创建带有占位符的动态提示，以供 AI 输入。
3. `PARAMETERS`, `LLAMA3_MODEL_ID`, 等等 是我们之前定义的配置值，用于设置不同的 AI 模型。

```
# 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)
```

我们将再次初始化我们的模型，这次我们将利用 LangChain 的 `ChatWatsonx`，这是 WatsonX API 客户端的一个包装器。

```
# 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"]
)
```

为了使我们的提示更加可重用和适应不同的聊天场景，我们可以使用 `PromptTemplate` 类。这使我们能够定义带有占位符的模板，这些占位符可以在运行时动态填充特定的输入。

通过定义像 `system_prompt` 和 `user_prompt` 这样的占位符，这些模板可以与不同的内容一起重用，使其在与 AI 模型的各种互动中更加灵活。

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

函数`get_ai_response`允许我们将提示模板和AI模型链接在一起。我们可以使用管道操作符`|`直接将模板的输出作为模型的输入。

```
# 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)
```

模型特定的函数各自调用这个通用函数，传入相应的模型和模板，确保在生成响应时为每个AI模型使用适当的格式。

让我们分解一下这段代码：

1. 我们导入必要的模块和配置。
2. 我们定义一个函数 `initialize_model` 来创建模型实例，促进代码重用。
3. 我们使用这个函数初始化我们的模型。
4. 我们为每个模型创建提示模板，因为它们可能有不同的首选格式。
5. `get_ai_response` 函数处理格式化提示、获取响应的过程。
6. 我们定义特定于模型的响应函数，这些函数使用通用的 `get_ai_response` 函数。

这种模块化的方法允许轻松添加新模型或修改现有模型。

合理性检查

这段代码挺多的，在我们继续之前，先运行一下代码，看看结果如何。我们通过将所有模型作为一个函数一起调用，来给它们进行测试。

创建文件 `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 c
```

并运行以下内容：

```
python llm_test.py
```

如果一切顺利，您应该会得到类似以下的输出：

“”

Llama3 响应：

加拿大的首都渥太华。

关于渥太华的一个有趣事实是，它是联合国教科文组织世界遗产名录中的里多运河的所在地，这是北美最古老的持续运营的运河。在冬季，这条运河会结冰，成为世界上最大的天然冰滑冰场，横跨城市中心，长达7.8公里（4.8英里）。

Granite 响应：

加拿大的首都渥太华。它位于渥太华河的南岸，以其历史建筑、博物馆和充满活力的文化景观而闻名。关于渥太华的一个有趣事实是，它拥有世界上最大的室内滑冰场——里多运河滑冰道，亦是联合国教科文组织世界遗产名录中的一部分。

Mixtral 响应：

加拿大的首都渥太华。关于渥太华的一个有趣事实是，它是世界上最寒冷的首都之一。在冬季，气温可以降到-40°C（-40°F），使其成为冬季运动和活动的热门目的地，例如在里多运河上滑冰，这里成为世界上最大的天然冰滑冰场。

“”

设置 JSON 输出

我们需要解决一个重要步骤：确保 AI 的输出遵循明确的格式。这对于将输出无缝集成到其他系统中，例如网站，是至关重要的。

我们可以使用 Pydantic 来定义 AI 响应的清晰模式，以确保结构的一致性和验证。这强制执行正确的格式，使数据集成更加顺畅和可靠。

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

我们将使用 `BaseModel` 和 `Field` 来定义我们的 JSON 输出结构。为了简化我们的工作，我们还将使用 `JsonOutputParser` 自动解析和验证 AI 的输出，以符合我们定义的结构化格式。

Pydantic 模型

```
# 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")
```

为了将此结构无缝集成到我们的代码中，我们使用 `JsonOutputParser`。该解析器确保 AI 返回的输出会自动验证并解析为 `AIResponse` 格式。

JSON 输出解析器

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

在这里，我们使用 `AIResponse` Pydantic 模型定义预期输出，指定字段如 `summary`、`sentiment`、`action` 和 `response`。`JsonOutputParser` 将确保 AI 输出符合此结构，为我们应用中的进一步使用提供格式良好、经过验证的数据。

更新链条

```
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_instructio
```

您可以看到我们将 `json_parser` 添加到我们的链中，并调用 `json_parser.get_format_instructions()`，这最终会根据 `AIResponse` 类更新我们的提示，以便以良好结构的 JSON 格式进行响应。

整合所有内容

那么让我们把这个添加到链中！为此，我们需要将 `AIResponse` 和 `json_parser` 添加到 `model.py` 的顶部，并在 `get_ai_response` 中为我们的链对象添加另一个链接。您的代码应该如下所示：

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)
```

练习：增强 JSON 结构

现在，让我们练习增强我们的 JSON 结构。您的任务是向 `AIResponse` 类添加一个新字段，推荐支持代表可能采取的下一步措施来解决此问题。

1. 更新 `model.py` 中的 `AIResponse` 类。
2. 修改 `app.py` 中的系统提示以包含这个新字段。
3. 使用各种用户消息测试您的更改。

▶ [点击这里查看答案](#)

增强您的 Flask 应用程序的 AI 功能

现在我们已经设置好了 AI 模型，让我们将它们集成到我们的 Flask 应用程序中。

第一步：更新您的 Flask 应用程序

让我们更新 app.py 以使用这些 AI 功能：

Open **app.py** in IDE

用以下内容更新 app.py：

```
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)
```

让我们来分解一下这些更改：

1. 我们导入特定于模型的响应函数。
2. 在 /generate 路由中，我们现在期望包含 "message" 和 "model" 字段的 JSON 输入。
3. 我们添加了对缺失输入的错误处理。
4. 我们使用 try-except 块来处理 AI 处理中的潜在错误。
5. 我们测量并将处理时间包含在响应中。

这个设置使我们能够处理不同模型的请求，并提供强大的错误处理。

第 2 步：创建简单的 HTML 文件

创建文件 templates/index.html：

Open **index.html** in IDE

更新 templates/index.html 的内容为：

```
<!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>
</body>
</html>
```

```
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.tc
    }
  })
  .catch((error) => {
    console.error('Error:', error);
    document.getElementById('response').innerText = 'Error: ' + error;
  });
});
</script>
</body>
</html>
```

这是一些简单的 HTML，将为我们提供一个表单，允许我们调用 /generate 端点，传递消息和模型选择。

第 3 步：测试您的 AI 驱动应用程序

首先让我们运行我们的 Flask 应用程序，执行：

```
python app.py
```

您应该看到输出，指示 Flask 开发服务器正在端口 5000 上运行。

Flask 应用程序现在在 Cloud IDE 上本地运行。要访问它，请点击以下按钮：

测试您的应用程序

尝试使用不同的消息和模型，看看响应如何变化。

恭喜您创建了支持 LLM 的 Flask 应用程序！

结论与后续步骤

恭喜您完成这个指导项目！您成功构建了一个使用 Flask 的 GenAI 应用程序后端，集成了多个 AI 模型，并实现了结构化的 JSON 输出以增强功能。

主要收获

- 您学习了如何设置具有 AI 功能的 Flask 应用程序。
- 您集成并比较了多个语言模型（Llama 3、Granite 和 Mixtral）。
- 您实现了 LangChain 的 JsonOutputParser 以获得结构化的 AI 输出。
- 您对提示工程和模型性能分析有了深入了解。
- 您创建了一个模块化且易于维护的 AI 集成代码库。

后续步骤

为了进一步提升您的技能和应用：

1. **实现缓存：**添加缓存机制，以提高重复查询的性能。
2. **探索高级 LangChain 功能：**了解如内存等功能，以保持对话上下文。
3. **添加更多模型：**尝试集成通过 watsonx.ai 提供的其他模型。
4. **实施 A/B 测试：**创建一个系统，以比较不同模型对同一查询的响应。
5. **增强错误处理：**实现更强大的错误处理和日志记录。
6. **探索 IBM Cloud 服务：**考虑集成其他 IBM Cloud 服务，以扩展您应用程序的功能。

进一步学习

- 探索 [IBM watsonx.ai 文档](#) 以获取更多高级功能。
- 深入了解 [LangChain](#)，以获取更复杂的 AI 应用架构。
- 学习 [提示工程技术](#)，以改善 AI 模型输出。

请记住，GenAI 领域正在快速发展。继续实验、学习和构建，以保持在这一激动人心的技术前沿！

感谢您参与此次研讨会。我们希望您发现它有价值，并受到启发，继续在 AI 驱动的应用开发之旅中前行！



Skills Network