Using SAP Cloud SDK for AI to Evaluate Prompts

Objective

After completing this lesson, you will be able to evaluate prompts for a larger data set using functions in SDK.

Using SAP Cloud SDK for AI to Evaluate Prompts

After learning to develop and refine prompts in the previous lesson, the next critical step is to ensure their quality and reliability. In this lesson, we'll explore how to establish a systematic evaluation framework for your prompts using the **SAP Cloud SDK for AI.** You'll learn to automate the assessment of your LLM's output, setting a crucial baseline for continuous improvement in your generative AI applications.

Evaluate Prompts Using the SAP Cloud SDK for AI

You've successfully developed a prompt that assigns urgency, sentiment, and categories to customer messages in JSON format. However, for the facility solutions company, the accuracy of these outputs is important as they directly impact critical customer-facing applications and operational decisions.

To guarantee this accuracy, automated and consistent evaluation of prompts across various scenarios is essential, ensuring reliability and efficiency. This is where custom evaluation functions, implemented using the **SAP Cloud SDK for AI,** become indispensable.

relevance, coherence, fluency) to quantitatively assess response quality.

- **Tailored Evaluations:** It allows you to create custom evaluators that meet specific needs, enabling more precise and relevant assessments.
- **Scalable Results:** It supports large-scale evaluations, making it easier to test prompts on extensive datasets.

Implementing Evaluation Functions

1. We start the evaluation with the import of packages.

Python

```python
class RateLimitedIterator:
    def __init__(self, iterable, max_iter
        self._iterable = iter(iterable)
        self._max_iterations_per_minute =
        self._min_interval = 1.0 / (max_i
        self._last_yield_time = None

    def __iter__(self):
        return self

    def __next__(self):
        current_time = time.time()

        if self._last_yield_time is not N
            elapsed_time = current_time -
            if elapsed_time < self._min_i
                time.sleep(self._min_inte

        self._last_yield_time = time.time
        return next(self._iterable)
```

The code defines a "RateLimitedIterator" class to control the rate at which you can iterate over an iterable. By specifying a maximum number of iterations per minute, it ensures that the iteration process adheres to a defined speed, preventing hitting rate limits. It uses the "tqdm" library for progress visualization and the "time" module for timing control.

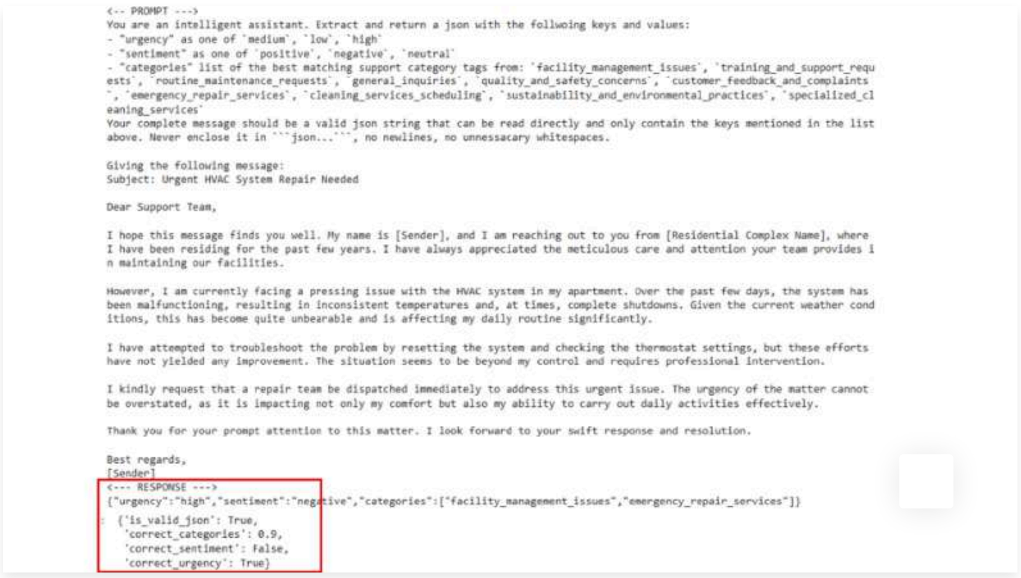
2. Next we define an evaluation function.

Python

```python
def evaluation(mail: Dict[str, str], extr
    response = extract_func(input=mail["m
    result = {
        "is_valid_json": False,
        "correct_categories": False,
    except json.JSONDecodeError:
        result["is_valid_json"] = False
    else:
        result["is_valid_json"] = True
        result["correct_categories"] = 1
        result["correct_sentiment"] = pre
        result["correct_urgency"] = pred[
    return result
evaluation(mail, f_8)

```

This code evaluates the predictions made by a function processing an email message. It uses a provided extraction function to analyze the email's content and compares the results against predefined ground truth data, checking for valid JSON, correct categories, sentiment, and urgency. This ensures that the extraction function performs accurately and consistently.

The last sentence evaluates the combined prompt function that we developed previously.



```
<-- PROMPT --->
You are an intelligent assistant. Extract and return a json with the follwoing keys and values:
- "urgency" as one of `medium`, `low`, `high`
- "sentiment" as one of `positive`, `negative`, `neutral`
- "categories" list of the best matching support category tags from: `facility_management_issues`, `training_and_support_requ
ests`, `routine_maintenance_requests`, `general_inquiries`, `quality_and_safety_concerns`, `customer_feedback_and_complaints
`, `emergency_repair_services`, `cleaning_services_scheduling`, `sustainability_and_environmental_practices`, `specialized_cl
eaning_services`
Your complete message should be a valid json string that can be read directly and only contain the keys mentioned in the list
above. Never enclose it in ```json...```, no newlines, no unnessacary whitespaces.

Giving the following message:
Subject: Urgent HVAC System Repair Needed

Dear Support Team,

I hope this message finds you well. My name is [Sender], and I am reaching out to you from [Residential Complex Name], where
I have been residing for the past few years. I have always appreciated the meticulous care and attention your team provides i
n maintaining our facilities.

However, I am currently facing a pressing issue with the HVAC system in my apartment. Over the past few days, the system has
been malfunctioning, resulting in inconsistent temperatures and, at times, complete shutdowns. Given the current weather cond
itions, this has become quite unbearable and is affecting my daily routine significantly.

I have attempted to troubleshoot the problem by resetting the system and checking the thermostat settings, but these efforts
have not yielded any improvement. The situation seems to be beyond my control and requires professional intervention.

I kindly request that a repair team be dispatched immediately to address this urgent issue. The urgency of the matter cannot
be overstated, as it is impacting not only my comfort but also my ability to carry out daily activities effectively.

Thank you for your prompt attention to this matter. I look forward to your swift response and resolution.

Best regards,
[Sender]
<--- RESPONSE --->
{"urgency":"high","sentiment":"negative","categories":["facility_management_issues","emergency_repair_services"]}
 {'is_valid_json': True,
  'correct_categories': 0.9,
  'correct_sentiment': False,
  'correct_urgency': True}
```

You can see that the evaluation shows that all predictions are correct except sentiment.

Python

```python
from tqdm.auto import tqdm

def transpose_list_of_dicts(list_of_dicts
    keys = list_of_dicts[0].keys()
    transposed_dict = {key: [] for key ir
    for d in list_of_dicts:
        for key, value in d.items():
            transposed_dict[key].append(v
    return transposed_dict

def evalulation_full_dataset(dataset, fur
    results = [evaluation(mail, func, _pr
    results = transpose_list_of_dicts(res
    n = len(dataset)
    for k, v in results.items():
        results[k] = sum(v) / len(dataset
    return results


def pretty_print_table(data):
    # Get all row names (outer dict keys)
    row_names = list(data.keys())

    # Get all column names (inner dict ke
```

This code now performs evaluation on the entire dataset, evaluates each entry through a function with rate limiting, transposes the results for better aggregation, and then pretty-prints the final evaluation metrics in a tabular format. It uses the "tqdm" library to show a progress bar, making it easier to track processing status. The entire flow ensures streamlined, efficient processing and clear presentation of results.

4. Implement the final function to the final combined function.

```
3
```

You can get the following output:

Code Snippet

```
  0%|          | 0/20 [00:00<?, ?it/s]
                     is_valid_json correct_cat
=============================================
basic--llama3.1-70b        100.0%
```

Using SAP Cloud SDK for AI to Leverage the Power of LLMs

Objective

After completing this lesson, you will be able to build basic prompts using SAP Cloud SDK for AI.

Using SAP Cloud SDK for AI to Leverage the Power of LLMs

In this lesson, we'll explore the practical art of prompt development within the generative AI hub. Building on our understanding of the SAP Cloud SDK for AI and orchestration services, you'll learn an iterative, step-by-step approach to refining prompts that extract precise, structured information from text and are ready for consumption by your SAP applications.

You've installed and configured SAP Cloud SDK for AI (Python) - generative. You've also configured orchestration services. We begin with installing packages and then loading data.

Python

```python
!pip install -U "generative-ai-hub-sdk>=3.1" tq
```

You need to set up an AI Core client for interacting with the AI Core service, and import necessary modules, load credentials from a JSON file, and set environment variables required for authentication and service access. See code here for details.

Refer to the detailed code in the repository here. and the relevant readme file.

Python

```python
from typing import Literal, Type, Union, Dict,
import re, pathlib, json, time
from functools import partial
EXAMPLE_MESSAGE_IDX = 10

```

This code begins by importing necessary modules and types from various Python libraries, including typing, re, pathlib, json, time, and functools. It then defines a constant, EXAMPLE_MESSAGE_IDX set to 10 to select a random 10th message from a dataset.

Next, you need to load and preprocess a dataset of emails stored in a JSON file. You can split dataset into development and testing sets with a smaller test set created for more focused evaluation. You can also process email data, to extract and organize categories, urgency, and sentiment into sets for further analysis.

Helper Functions

Before developing prompts using SAP Cloud SDK for AI (Python) - generative, you need to enable the use of AI models in generative AI hub. This is done through helper functions. You can see detailed functions in the repository here.

One of the function interfaces with the AI Core SDK to manage deployment tasks efficiently. It includes a spinner function to provide real-time updates during long operations, maintaining user engagement. The function checks some of the troubleshooting tips.

The following function sets up and sends requests to an AI orchestration service.

Python

```python
import pathlib
import yaml

from gen_ai_hub.proxy import get_proxy_clie
from ai_api_client_sdk.models.status import

from gen_ai_hub.orchestration.models.config
from gen_ai_hub.orchestration.models.llm im
from gen_ai_hub.orchestration.models.messag
from gen_ai_hub.orchestration.models.templa
from gen_ai_hub.orchestration.service impor

client = get_proxy_client()
deployment = retrieve_or_deploy_orchestrati
orchestration_service = OrchestrationServic

def send_request(prompt, _print=True, _mode
    config = OrchestrationConfig(
        llm=LLM(name=_model),
        # template=Template(messages=[Syste
        template=prompt
    )
    template_values = [TemplateValue(name=k
    answer = orchestration_service.run(conf
```

It imports necessary libraries and modules, sets up a proxy client, and retrieves or deploys an orchestration configuration. It defines a function, "send_request", which configures and sends prompts to an AI model, then formats and prints the response. The function can access all the configured

Develop a Prompt Using SAP Cloud SDK for AI (Python) - generative

We will now find the urgency and sentiment in an incoming mail.

Use the following code to read a random mail with the ID assigned earlier.

Python

```python
mail = dev_set[EXAMPLE_MESSAGE_IDX]
```

As we saw earlier, developing a prompt to solve a business problem is an iterative process. This process is explained in the following steps:

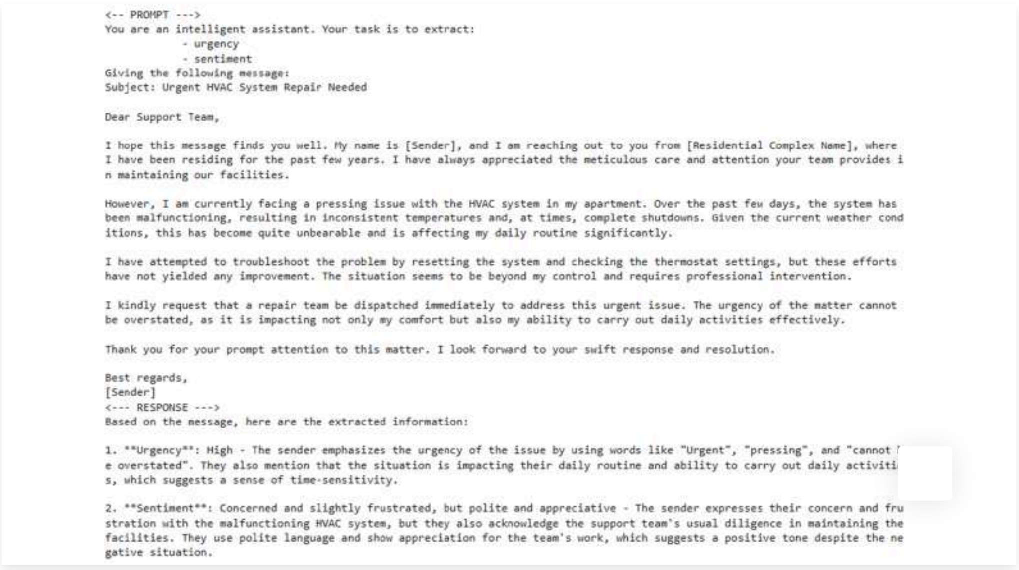
1. **Develop a basic prompt**: We'll start with a basic prompt and then develop the prompt, finding an output that can be used by applications within the company. The first prompt is:

Python

```python
from gen_ai_hub.orchestration.models.mess
from gen_ai_hub.orchestration.models.temp

prompt_1 = Template(
    messages=[
        SystemMessage(
            """You are an intelligent ass
            - urgency
            - sentiment
Giving the following message:"""
        ),
        UserMessage("{{?input}}")
    ]
)

f_1 = partial(send_request, prompt=prompt

response = f_1(input=mail["message"])
```

`gen_ai_hub.orchestration.models` package to create a structured prompt for an intelligent assistant. It defines the system and user messages. The assistant is tasked with extracting urgency and sentiment from a given message. The `partial` function sets up the request with the prompt, and the code sends the request using the input message from `mail["message"]` .

You can see the following output:



```
<-- PROMPT --->
You are an intelligent assistant. Your task is to extract:
            - urgency
            - sentiment
Giving the following message:
Subject: Urgent HVAC System Repair Needed

Dear Support Team,

I hope this message finds you well. My name is [Sender], and I am reaching out to you from [Residential Complex Name], where
I have been residing for the past few years. I have always appreciated the meticulous care and attention your team provides i
n maintaining our facilities.

However, I am currently facing a pressing issue with the HVAC system in my apartment. Over the past few days, the system has
been malfunctioning, resulting in inconsistent temperatures and, at times, complete shutdowns. Given the current weather cond
itions, this has become quite unbearable and is affecting my daily routine significantly.

I have attempted to troubleshoot the problem by resetting the system and checking the thermostat settings, but these efforts
have not yielded any improvement. The situation seems to be beyond my control and requires professional intervention.

I kindly request that a repair team be dispatched immediately to address this urgent issue. The urgency of the matter cannot
be overstated, as it is impacting not only my comfort but also my ability to carry out daily activities effectively.

Thank you for your prompt attention to this matter. I look forward to your swift response and resolution.

Best regards,
[Sender]
<--- RESPONSE --->
Based on the message, here are the extracted information:

1. **Urgency**: High - The sender emphasizes the urgency of the issue by using words like "Urgent", "pressing", and "cannot
e overstated". They also mention that the situation is impacting their daily routine and ability to carry out daily activiti
s, which suggests a sense of time-sensitivity.

2. **Sentiment**: Concerned and slightly frustrated, but polite and appreciative - The sender expresses their concern and fru
stration with the malfunctioning HVAC system, but they also acknowledge the support team's usual diligence in maintaining the
facilities. They use polite language and show appreciation for the team's work, which suggests a positive tone despite the ne
gative situation.
```

You can utilize different models using the same function. This is a benefit of the orchestration feature of generative AI hub.

The current lengthy response is not useful for our objective of assigning urgency and sentiment to emails for software input, so we should continue to develop it.

this learning journey.

When you execute the same prompt in your machine, an LLM produces varying outputs due to its probabilistic nature, temperature setting, and non-deterministic architecture, leading to different responses even with slight setting changes or internal state shifts.

2. **Assign values to urgency and sentiment**: We'll now assign some basic urgency and sentiment values and execute the prompt.

Python

```python
prompt_2 = Template(
    messages=[
        SystemMessage(
            """You are an intelligent assist
- "urgency" as one of {{?urgency}}
- "sentiment" as one of {{?sentiment}}
Giving the following message:"""
        ),
        UserMessage("{{?input}}")
    ]
)

f_2 = partial(send_request, prompt=prompt_2,

response = f_2(input=mail["message"])

```

The code defines a prompt for extracting "urgency" and "sentiment" from a given message. It uses the "partial" function to configure a request with predefined options. Finally, it processes the email message by passing it through the configured request to get the needed details. This ensures consistent and accurate data extraction from messages.

it's a lightweight, language-independent, and easy-to-parse format that enables efficient data exchange and simplifies development.
We use the following code:

Python

```
prompt_3 = Template(
    messages=[
        SystemMessage(
            """You are an intelligent assist
- "urgency" as one of {{?urgency}}
- "sentiment" as one of {{?sentiment}}
Your complete message should be a valid json
Giving the following message:"""
        ),
        UserMessage("{{?input}}")
    ]
)
f_3 = partial(send_request, prompt=prompt_3,

response = f_3(input=mail["message"])

```

This code prepares and sends a request to extract specific information from a message. It uses a predefined prompt template to instruct the system to extract "urgency" and "sentiment" and return them in JSON format. The "partial" function sets up this request with fixed options, and then it sends the request using the message content.

You will see a JSON output.

4. **Ensure that the JSON formatting is correct**: This step is needed when the JSON output format is not clear in the previous prompt. You can make it clearer using the following code:

Python

```
- "urgency" as one of {{?urgency}}
- "sentiment" as one of {{?sentiment}}
Your complete message should be a valid json

Giving the following message:"""
        ),
        UserMessage("{{?input}}")
    ]
)

f_4 = partial(send_request, prompt=prompt_4,

response = f_4(input=mail["message"])

```

This code defines a prompt template called "prompt_4" that specifies instructions for extracting "urgency" and "sentiment" from an input message and returning it as a JSON string. This prompt gives clear instruction to provide a clean format without any quotes or whitespaces.

You will see that the JSON output is clear now. This is ready for software consumption.

5. **Simple categories based on business functions**: We have associated urgency and sentiment to a mail. However, we also need more tags for each message to categorize them for business needs. We can start with a simple code.

Python

```
prompt_5 = Template(
    messages=[
        SystemMessage(
            """You are an intelligent ass

Giving the following message:"""
        ),
response = f_5(input=mail["message"])

```

This code sets up a template for assigning support categories based on user messages. It prepares a system message stating the assistant's role, incorporates user input, and defines a partial function to send the request. Finally, it uses this function to process the message and generate a response, helping categorize support queries efficiently.

You will see that categories are assigned to the message. The response assigns the support categories to the original message, streamlining customer support processes.

6. **Assigning values to categories from a list**: The categories generated above have overlapping values, such as urgency, and should be aligned with company-defined values to address business needs effectively.

Python

```
prompt_6 = Template(
    messages=[
        SystemMessage(
            """You are an intelligent ass
{{?categories}}

Giving the following message:"""
        ),
        UserMessage("{{?input}}")
    ]
)

# Prepare the options
option_lists = {
    'categories': ', '.join(f"`{entry}`"
}

```

This code sets up a template for classifying messages by support categories. It defines the system and user messages, prepares a list of category options, and creates a function to send the classification request using these options. Then, it calls the function with the input message to get the response, effectively assigning suitable tags based on the content of the message.

You will see that categories are assigned to values from the list. These are streamlined for business processing.

7. **Generate JSON output for categories values**: Similar to step 3 earlier, we need JSON output for processing these values in a software. We also ensure that the format of JSON output is without any unnecessary symbols or space.
   We use the following code:

   Python

```
- "categories" list of the best matching sup
Your complete message should be a valid json

Giving the following message:"""
            ),
            UserMessage("{{?input}}")
        ]
)
f_7 = partial(send_request, prompt=prompt_7,

response = f_7(input=mail["message"])

```

This code creates a precise prompt template to extract specific category tags from an input message and return them in a valid JSON format. It then uses a predefined function, "send_request", with the constructed prompt and some options to generate the necessary response from the given input message. This ensures consistency and accuracy in data extraction.

You will see categories in JSON output that can be processed in a software.

8. **Combining all the steps**: We have used SAP Cloud SDK for AI (Python) - generative to arrive at proper values of urgency, sentiment, and categories in JSON format step-by-step. Now, let's combine all these steps into a single prompt.

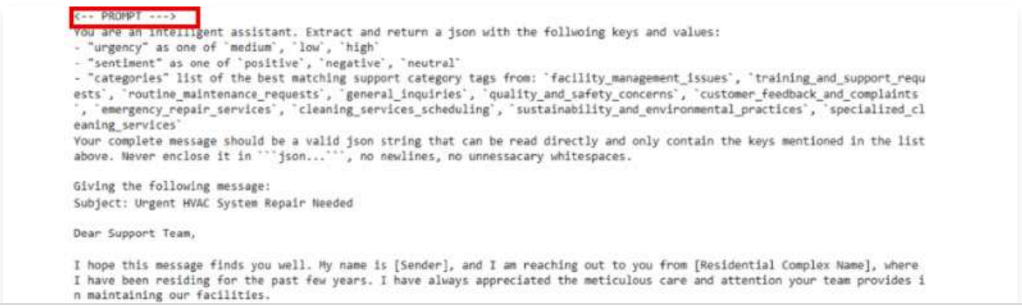
Python

```
prompt_8 = Template(
    messages=[
        SystemMessage(
            """You are an intelligent ass
- "urgency" as one of {{?urgency}}
- "sentiment" as one of {{?sentiment}}
- "categories" list of the best matching
```

```
    ]
)

option_lists = {
    'urgency': ', '.join(f"`{entry}`" for
    'sentiment': ', '.join(f"`{entry}`" f
    'categories': ', '.join(f"`{entry}`"
}
f_8 = partial(send_request, prompt=prompt

response = f_8(input=mail["message"])

```

The code combines all the previous steps in one prompt. It helps processing an email message to identify and return key information. By using a predefined template, it extracts details like urgency, sentiment, and categories, then formats them as a JSON string. It ensures that the output is clean and directly usable by invoking a function with the given parameters.

The following screenshot shows the output of this code.



```
<-- PROMPT --->
You are an intelligent assistant. Extract and return a json with the follwoing keys and values:
- "urgency" as one of `medium`, `low`, `high`
- "sentiment" as one of `positive`, `negative`, `neutral`
- "categories" list of the best matching support category tags from: `facility_management_issues`, `training_and_support_requ
ests`, `routine_maintenance_requests`, `general_inquiries`, `quality_and_safety_concerns`, `customer_feedback_and_complaints
`, `emergency_repair_services`, `cleaning_services_scheduling`, `sustainability_and_environmental_practices`, `specialized_cl
eaning_services`
Your complete message should be a valid json string that can be read directly and only contain the keys mentioned in the list
above. Never enclose it in ```json...```, no newlines, no unnessacary whitespaces.

Giving the following message:
Subject: Urgent HVAC System Repair Needed

Dear Support Team,

I hope this message finds you well. My name is [Sender], and I am reaching out to you from [Residential Complex Name], where
I have been residing for the past few years. I have always appreciated the meticulous care and attention your team provides i
n maintaining our facilities.
```

/ Examples / Native Client Integrations

Our SDK offers a developer-friendly way to consume foundational models available in the SAP generative AI hub. We strive to facilitate seamless interactions with these models by providing integrations that act as drop-in replacements for the native client SDKs and LangChain. This allows developers to use familiar interfaces and workflows. Usage is as follows.

Native Client Integrations

Completions

OpenAI

`Completions` equivalent to `openai.Completions`. Below is an example usage of Completions in generative AI hub sdk. All models that support the legacy completion endpoint can be used.

```
from gen_ai_hub.proxy.native.openai import completions

response = completions.create(
    model_name="gpt-4o-mini",
    prompt="The Answer to the Ultimate Question of Life, the Universe, a
    max_tokens=20,
    temperature=0
)
print(response)
```

`ChatCompletions` equivalent to `openai.ChatCompletions` Below is an example usage of ChatCompletions in generative AI hub sdk.

```
from gen_ai_hub.proxy.native.openai import chat

messages = [{"role": "system", "content": "You are a helpful assistant."
            {"role": "user", "content": "Does Azure OpenAI support custo
            {"role": "user", "content": "Do other Azure Cognitive Servic

kwargs = dict(model_name='gpt-4o-mini', messages=messages)
response = chat.completions.create(**kwargs)

print(response)
```

```
#example where deployment_id is passed instead of model_name parameter

from gen_ai_hub.proxy.native.openai import chat

messages = [{"role": "system", "content": "You are a helpful assistant."
            {"role": "user", "content": "Does Azure OpenAI support custo
            {"role": "assistant", "content": "Yes, customer managed keys
            {"role": "user", "content": "Do other Azure Cognitive Servic

response = chat.completions.create(deployment_id="dcef02e219ae4916", mes
print(response)
```

Structured model outputs

LLM output as json objects is a powerful feature that allows you to define the structure of the output you expect from the model.

see https://platform.openai.com/docs/guides/structured-outputs/examples

```
from pydantic import BaseModel
from gen_ai_hub.proxy.native.openai import chat

class Person(BaseModel):
    name: str
    age: int

response = chat.completions.parse(
    model="gpt-4o-mini",
    messages=[{"role": "user", "content": "Tell me about John Doe, aged
    response_format=Person
)
person = response.choices[0].message.parsed  # Fully typed Person
print(person)
```

Google GenAI

Generate Content

```
from gen_ai_hub.proxy.native.google_genai.clients import Client
from gen_ai_hub.proxy.core.proxy_clients import get_proxy_client

proxy_client = get_proxy_client('gen-ai-hub')
client = Client(proxy_client=proxy_client)

response = client.models.generate_content(model="gemini-2.5-flash",
    contents="How many paws are there for a dog?"
)

print(response)
# Using another model
response = client.models.generate_content(model="gemini-2.0-flash",
                                          contents="Explain the theory o
print(response)
```

Generate Content streaming

```
from gen_ai_hub.proxy.native.google_genai.clients import Client
from gen_ai_hub.proxy.core.proxy_clients import get_proxy_client

proxy_client = get_proxy_client('gen-ai-hub')

client = Client(
    proxy_client=proxy_client,
)


response_stream = client.models.generate_content_stream(model="gemini-2.
contents="Explain singularity in short terms.")

for chunk in response_stream:
    print("Chunk: ", chunk.text)
```

Functional Calling of Google Genai

```
from google genai import types
from gen_ai_hub.proxy.native.google_genai.clients import Client
from gen_ai_hub.proxy.core.proxy_clients import get_proxy_client

def get_current_weather(location: str) -> str:
  """Returns the current weather.

  Args:
    location: The city and state, e.g. San Francisco, CA
  """
  return 'sunny'


proxy_client = get_proxy_client('gen-ai-hub')

client = Client(
    proxy_client=proxy_client,
)
response = client.models.generate_content(
  model='gemini-2.5-flash',
  contents='What is the weather like in Boston?',
  config=types.GenerateContentConfig(tools=[get_current_weather]),
)
response
```

Amazon

Invoke Model

```
import json
from gen_ai_hub.proxy.native.amazon.clients import Session

bedrock = Session().client(model_name="amazon--nova-premier")
body = json.dumps(
    {
        "inputText": "Explain black holes in astrophysics to 8th graders
        "textGenerationConfig": {
            "maxTokenCount": 3072,
            "stopSequences": [],
            "temperature": 0.7,
            "topP": 0.9,
        },
    }
)
response = bedrock.invoke_model(body=body)
response_body = json.loads(response.get("body").read())
print(response_body)
```

Converse

```
from gen_ai_hub.proxy.native.amazon.clients import Session

bedrock = Session().client(model_name="anthropic--claude-4-sonnet")
conversation = [
    {
        "role": "user",
        "content": [
            {
                "text": "Describe the purpose of a 'hello world' program
            }
        ],
    }
]
response = bedrock.converse(
    messages=conversation,
    inferenceConfig={"maxTokens": 512, "temperature": 0.5, "topP": 0.9},
)
print(response)
```

Embeddings

OpenAI

`Embeddings` are equivalent to `openai.Embeddings`. See below examples of how to use `Embeddings` in generative AI hub sdk.

```
from gen_ai_hub.proxy.native.openai import embeddings

response = embeddings.create(
    input="Every decoding is another encoding.",
    model_name="text-embedding-ada-002"
)
print(response.data)
```

```
from gen_ai_hub.proxy.native.openai import embeddings
# example with encoding format passed as parameter
response = embeddings.create(
    model_name="text-embedding-ada-002",
    encoding_format='base64'
)
print(response.data)
```

Amazon

```
import json
from gen_ai_hub.proxy.native.amazon.clients import Session
bedrock = Session().client(model_name="amazon--nova-premier")
body = json.dumps(
    {
        "inputText": "Please recommend books with a theme similar to the
    }
)
response = bedrock.invoke_model(
    body=body,
)
response_body = json.loads(response.get("body").read())
print(response_body)
```

```
from gen_ai_hub.proxy.native.openai import embeddings
# example with encoding format passed as parameter
response = embeddings.create(
    input="Every decoding is another encoding.",
    model_name="text-embedding-ada-002",
    encoding_format='base64'
)
print(response.data)
```

Langchain Integration

LangChain provides an interface that abstracts provider-specific details into a common interface. Classes like Chat and Embeddings are interchangeable.

The list of the available models can be found here: Supported Models

Harmonized Model Initialization

The `init_llm` and `init_embedding_model` functions allow easy initialization of langchain model interfaces in a harmonized way in generative AI hub sdk

```
from langchain.prompts import PromptTemplate
from langchain_core.output_parsers import StrOutputParser
from gen_ai_hub.proxy.langchain.init_models import init_llm

template = """Question: {question}
    Answer: Let's think step by step."""
prompt = PromptTemplate(template=template, input_variables=['question'])
question = 'What is a supernova?'

llm = init_llm('gpt-5-nano', max_tokens=300)
chain = prompt | llm | StrOutputParser()
response = chain.invoke({'question': question})
print(response)
```

```
from gen_ai_hub.proxy.langchain.init_models import init_embedding_model

text = 'Every decoding is another encoding.'

embeddings = init_embedding_model('text-embedding-ada-002')
response = embeddings.embed_query(text)
print(response)
```

LLM

```
from langchain import PromptTemplate

from gen_ai_hub.proxy.langchain.openai import OpenAI  # langchain class
from gen_ai_hub.proxy.core.proxy_clients import get_proxy_client

proxy_client = get_proxy_client('gen-ai-hub')
# non-chat model
model_name = "mistralai--mistral-small-instruct"

llm = OpenAI(proxy_model_name=model_name, proxy_client=proxy_client)  #

template = """Question: {question}

Answer: Let's think step by step."""
```

```
prompt = PromptTemplate(template=template, input_variables=["question"])
llm_chain = prompt | llm

question = "What NFL team won the Super Bowl in the year Justin Bieber w

print(llm_chain.invoke({'question': question}))
```

Chat model

```
from langchain.prompts.chat import (
    AIMessagePromptTemplate,
    ChatPromptTemplate,
    HumanMessagePromptTemplate,
    SystemMessagePromptTemplate,
)

from gen_ai_hub.proxy.langchain.openai import ChatOpenAI
from [illegible] import get_proxy_client

proxy_client = get_proxy_client('gen-ai-hub')

chat_llm = ChatOpenAI(proxy_model_name='gpt-4o-mini', proxy_client=proxy
template = 'You are a helpful assistant that translates english to pirat

system_message_prompt = SystemMessagePromptTemplate.from_template(templa

example_human = HumanMessagePromptTemplate.from_template('Hi')
example_ai = AIMessagePromptTemplate.from_template('Ahoy!')
human_template = '{text}'

human_message_prompt = HumanMessagePromptTemplate.from_template(human_te
chat_prompt = ChatPromptTemplate.from_messages(
    [system_message_prompt, example_human, example_ai, human_message_pro

chain = chat_prompt | chat_llm

response = chain.invoke({'text': 'I love planking.'})
print(response.content)
```

Structured model outputs

```
from gen_ai_hub.proxy.langchain.openai import ChatOpenAI
from gen_ai_hub.proxy.core.proxy_clients import get_proxy_client
from langchain.schema import HumanMessage
from pydantic import BaseModel

class Person(BaseModel):
    name: str
    age: int
chat_model = ChatOpenAI(proxy_model_name="gpt-4o-mini", proxy_client=get
chat_model = chat_model.with_structured_output(method="json_schema", sch

message = HumanMessage(content="Tell me about a person named John who is
print(chat_model.invoke([message]))
```

Embeddings

```
from gen_ai_hub.proxy.langchain.openai import OpenAIEmbeddings
from gen_ai_hub.proxy.core.proxy_clients import get_proxy_client

proxy_client = get_proxy_client('gen-ai-hub')

embedding_model = OpenAIEmbeddings(proxy_model_name='text-embedding-ada-

response = embedding_model.embed_query('Every decoding is another encodi

#call without passing proxy_client

embedding_model = OpenAIEmbeddings(proxy_model_name='text-embedding-ada-

response = embedding_model.embed_query('Every decoding is another encodi
print(response)
```

Using New Models Before Official SDK Support

You can use models via Gen AI Hub even before they are officially listed, provided their provider family (e.g., `Google`, `Amazon Bedrock`) is supported.

1. **Native SDK Clients:**

If using the provider's native SDK (like `boto3`, `google-genai`) through the Gen AI Hub proxy, you can often use the new model name/ID directly with existing client methods.

2. **Langchain Integration (`init_llm`):**

   The `init_llm` helper simplifies creating Langchain LLM objects configured for the proxy.

   - **Alternative:** You can always bypass `init_llm` and instantiate the Langchain classes (e.g., `ChatGoogleGenerativeAI`, `ChatBedrock`, `ChatBedrockConverse`) directly.
   - **Bedrock Specifics**:
     - Requires `model_id` in addition to `model_name`. Find IDs here. `init_llm` automatically selects the appropriate Bedrock API (older Invoke via `ChatBedrock` or newer Converse via `ChatBedrockConverse`) based on known models.
     - **Crucially:** For *new* Bedrock models or to force a specific API (Invoke/Converse), you must pass the corresponding initialization function (`init_chat_model` or `init_chat_converse_model`) to the `init_func` argument of `init_llm`.

```
from gen_ai_hub.proxy.langchain.init_models import init_llm
# Import specific init functions for overriding Bedrock behavior
from gen_ai_hub.proxy.langchain.amazon import (
    init_chat_model as amazon_init_invoke_model,
    init_chat_converse_model as amazon_init_converse_model
)
from gen_ai_hub.proxy.langchain.google_genai import init_chat_model as g

# --- Google Example ---
llm_google = init_llm(model_name='gemini-newer-version', init_func=googl

# --- Bedrock Example (New Model requiring Converse API) ---
model_name_amazon = 'anthropic--claude-newer-version'
model_id_amazon = 'anthropic.claude-newer-version-v1:0' # Use actual ID

llm_amazon = init_llm(
    model_name_amazon,
    model_id=model_id_amazon,
    init_func=amazon_init_converse_model # Explicitly select Converse AP
)

# --- Bedrock Example (Explicitly using older Invoke API) ---
#     'some-model-name',
#     model_id='some-model-id',
#     init_func=amazon_init_invoke_model # Explicitly select Invoke API
# )
```

Streaming

Streaming in AI models enables real-time data generation. With native SDKs, invocation and response formats vary by provider and model. Langchain simplifies this by offering a unified stream method.

Native SDKs

OpenAI - ChatGPT

```
from gen_ai_hub.proxy.native.openai import chat

def stream_openai(prompt, model_name='gpt-4o-mini'):
    messages = [
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": prompt}
    ]

    kwargs = dict(model_name=model_name, messages=messages, max_tokens=5
    stream = chat.completions.create(**kwargs)

    for chunk in stream:
        if chunk.choices:
            content = chunk.choices[0].delta.content
            if content:
                print(content, end='')
```

```
stream_openai("Why is the sky blue?")
```

Structured model outputs

```
from gen_ai_hub.proxy.native.openai import chat, OpenAI
from pydantic import BaseModel

class Person(BaseModel):
    name: str
    age: int

messages = [{"role": "user", "content": "Tell me about John Doe, aged 30

def stream_openai_structured_outputs(messages, response_object, model_na
     # For more information, see:
     # https://www.github.com/openai/openai-python#with_streaming_respon
     # https://platform.openai.com/docs/guides/structured-outputs#stream
    with chat.completions.with_streaming_response.parse(
        model=model_name,
        messages=messages,
        response_format=Person
    ) as stream:
        response = stream.parse() # takes care of the stream chunks and
    return response.choices[0].message.parsed

print(stream_openai_structured_outputs(messages, Person, "gpt-4o-mini"))
```

```
def stream_beta_openai_structured_outputs(messages, response_object, mod
    chat = OpenAI(proxy_client=get_proxy_client())
    with chat.beta.chat.completions.stream(
        model=model_name,
        messages=messages,
        response_format=Person
    ) as stream:
        response = stream.get_final_completion() # This will wait for th
    return response.choices[0].message.parsed

print(stream_beta_openai_structured_outputs(messages, Person, "gpt-4o-mi
```

Google - GenAI

```
from gen_ai_hub.proxy.core.proxy_clients import get_proxy_client
from gen_ai_hub.proxy.native.google_genai.clients import Client
from google.genai.types import GenerateContentConfig

def stream_genai(prompt, model_name='gemini-2.0-flash'):
    proxy_client = get_proxy_client('gen-ai-hub')
    response = client.models.generate_content_stream(
        model=model_name,
        contents=prompt,
        config=GenerateContentConfig(max_output_tokens=500),
    )
    for chunk in response:
        print(chunk.text, end='')
```

```
stream_genai("Why is the sky blue?")
```

Anthropic - Claude

```
import json
from gen_ai_hub.proxy.native.amazon.clients import Session

def stream_claude(prompt, model_name='anthropic--claude-3-haiku'):
    bedrock = Session().client(model_name=model_name)
    body = json.dumps({
      "max_tokens": 500,
      "messages": [{"role": "user", "content": prompt}],
      "anthropic_version": "bedrock-2023-05-31"
    })

    response = bedrock.invoke_model_with_response_stream(body=body)
    stream = response.get("body")

    for event in stream:
        chunk = json.loads(event["chunk"]["bytes"])
        if chunk["type"] == "content_block_delta":
          print(chunk["delta"].get("text", ""), end="")
```

```
stream_claude("Why is the sky blue?")
```

Amazon - Bedrock

```
import json
from gen_ai_hub.proxy.native.amazon.clients import Session

def stream_bedrock(prompt, model_name='amazon--nova-pro'):
    bedrock = Session().client(model_name=model_name)
    body = json.dumps({
        "messages": [{"role": "user", "content": [{"text": prompt}]}],
        "system": [{"text": "Act as a creative writing assistant."}],
        "inferenceConfig": {"maxTokens": 500, "topP": 0.9, "topK": 20, "
    })

    response = bedrock.invoke_model_with_response_stream(body=body)
    stream = response.get("body")
    chunk_count = 0
    answer = ""
    if stream:
        for event in stream:
            chunk = event.get("chunk")
            if chunk:
                chunk_json = json.loads(chunk.get("bytes").decode())
                content_block_delta = chunk_json.get("contentBlockDelta"
                if content_block_delta:
                    chunk_count += 1
                    answer += content_block_delta.get("delta").get("text
        print(f"Total chunks: {chunk_count}")
        print("Final answer:", answer)
    else:
        print("No response stream received.")
```

```
stream_bedrock("Why is the sky blue?")
```

Langchain

```
from gen_ai_hub.proxy.langchain import init_llm

def stream_langchain(prompt, model_name):
    llm = init_llm(model_name=model_name, max_tokens=500)

    for chunk in llm.stream(prompt):
        print(chunk.content, end='')
```

```
stream_langchain("How do airplanes stay in the air?", model_name='gpt-4o
stream_langchain("How do airplanes stay in the air?", model_name='anthro
```

```
stream_langchain("How do airplanes stay in the air?", model_name='amazon
```

Prompt Registry

The Prompt Registry API allows you to create, manage, and retrieve prompt templates for use in SAP AICore when working with the models of the Generative AI Hub.

In this notebook, we'll walk through the design-time process of creating prompt templates using the SDK. (This is also known as imperative Prompt Template creation.) Here, one can create and modify the Prompt Template. It will be versioned automatically and the versions can be retrieved by an id.

See SAP Help for the difference between **imperative** and **declarative** prompt templates.

Prerequisite

Provide the credentials to authenticate the client and establish a connection with the Prompt Registry API.

See options for providing credentials here.

Step 0: Initialize Client

Initialize the client to interact with the Prompt Registry.

```
from gen_ai_hub.proxy import get_proxy_client
from gen_ai_hub.prompt_registry.client import PromptTemplateClient
proxy_client = get_proxy_client(proxy_version="gen-ai-hub")
prompt_registry_client = PromptTemplateClient(proxy_client=proxy_client)
```

Step 1: Create Prompt Templates

Define the Prompt Template configuration and post to Prompt Registry.

```
from gen_ai_hub.prompt_registry.models.prompt_template import PromptTemp

prompt_template_spec = PromptTemplateSpec(template=[PromptTemplate(role=

template_id = prompt_registry_client.create_prompt_template(
    scenario='MyScenario', name='prompt_template_name', version='1.0.0',

print(f"Created Prompt Template with ID: {template_id}")
```

Step 2: Retrieve Prompt Templates

Retrieve the Prompt Template by ID.

```
response = prompt_registry_client.get_prompt_template_by_id(template_id)
print(response.spec.template)
```

Step 3: Modify the Prompt Template

We will add an input variable to the existing Prompt Template.

```
prompt_template_spec = PromptTemplateSpec(template=[PromptTemplate(role=
                                                                   conte
                                                                   )
                                                              ]
                                                     )
response = prompt_registry_client.create_prompt_template(scenario='MySce
                                                          prompt_template

input_template_id = response.id
print(response.message)
```

Step 4: Prompt Template History

Retrieve the history of Prompt Templates by id.

```
response = prompt_registry_client.get_prompt_template_history(scenario='
print(response.json())
```

Step 5: Fill Prompt Template

Fill the variables in the Prompt Template.

```
response = prompt_registry_client.fill_prompt_template_by_id(template_id
print(response.parsed_prompt)
```

Orchestration Service [Deprecated]

> ***Important:*** *Note that version 1 of the orchestration service API is deprecated and will be decommissioned October 2026. Please refer to the SAP Note 3634540. Use Orchestration Service V2 API*

This notebook demonstrates how to use the SDK to interact with the Orchestration Service, enabling the creation of AI-driven workflows by seamlessly integrating various modules, such as templating, large language models (LLMs), data masking and content filtering. By leveraging these modules, you can build complex, automated workflows that enhance the capabilities of your AI solutions. For more details on configuring and using these modules, please refer to the Orchestration Service Documentation.

Prerequisite

> ***Important:*** *Before you begin using the SDK, make sure to set up a virtual deployment of the Orchestration Service.*

For detailed guidance on setting up the Orchestration Service, please refer to the setup guide here.

Authentication

By default, the `OrchestrationService` initializes a `GenAIHubProxyClient` , which automatically configures credentials using configuration files or environment variables, as outlined in the *Introduction* section.

If you prefer to set credentials manually, you can provide a custom instance using the `proxy_client` parameter.

Basic Orchestration Pipeline

Let's walk through a basic orchestration pipeline for a translation task.

Step 1: Define the Template and Default Input Values

The `Template` class is used to define structured message templates for generating dynamic interactions with language models. In this example, the template is designed for a translation assistant, allowing users to specify a language and text for translation.

```
from gen_ai_hub.orchestration.models.message import SystemMessage, UserM
from gen_ai_hub.orchestration.models.template import Template, TemplateV

template = Template(
    messages=[
        SystemMessage("You are a helpful translation assistant."),
        UserMessage(
            "Translate the following text to {{?to_lang}}: {{?user_query
        ),
    ],
    defaults=[
        TemplateValue(name="to_lang", value="German"),
    ],
)
```

This template can be used to create translation requests where the language and text to be translated are specified dynamically. The placeholders in the `UserMessage` will be replaced with the actual values provided at runtime, and the default value for the language is set to German.

Step 2: Define the LLM

The `LLM` class is used to configure and initialize a language model for generating text based on specific parameters. In this example, we'll use the `gpt-4o` model to perform the translation task.

Note: The Orchestration Service automatically manages the virtual deployment of the language model, so no additional setup is needed on your end.

```
from gen_ai_hub.orchestration.models.llm import LLM

llm = LLM(name="gpt-5-nano", parameters={"max_completion_tokens": 512})
```

Initializes the language model to use the `gpt-5-nano` model. It will generate responses up to 512 tokens in length.

Step 3: Create the Orchestration Configuration

The `OrchestrationConfig` class defines a configuration for integrating various modules, such as templates and language models, into a cohesive orchestration setup. It specifies how these components interact and are configured to achieve the desired operational scenario.

```
from gen_ai_hub.orchestration.models.config import OrchestrationConfig

config = OrchestrationConfig(
    template=template,
    llm=llm,
)
```

Step 4: Run the Orchestration Request

The `OrchestrationService` class is used to interact with a orchestration service instance by providing configuration details to initiate and manage its operations.

```
from gen_ai_hub.orchestration.service import OrchestrationService

orchestration_service = OrchestrationService(config=config)
```

Call the `run` method with the required `template values`. The service will process the input according to the configuration and return the result.

```
result = orchestration_service.run(template_values=[
    TemplateValue(name="user_query", value="The Orchestration Service is
])
print(result.orchestration_result.choices[0].message.content)
```

```
Der Orchestrierungsdienst funktioniert!
```

Referencing Templates in the Prompt Registry

In Step 3 you can also use a prompt template reference, which allows you to reuse existing templates stored in the Prompt Registry.

```
from gen_ai_hub.orchestration.models.template_ref import TemplateRef

template_by_id = TemplateRef.from_id(prompt_template_id="648871d9-b207-4
template_by_names = TemplateRef.from_tuple(scenario="translation", name=
```

Overview of response_format Parameter Options

The `response_format` parameter allows the model output to be formatted in several predefined ways, as follows:

1. **text**: This is the simplest form where the model's output is generated as plain text. It is suitable for applications that require raw text processing.
2. **json_object**: Under this setting, the model's output is structured as a JSON object. This is useful for applications that handle data in JSON format, enabling easy integration with web applications and APIs.
3. **json_schema**: This setting allows the model's output to adhere to a defined JSON schema. This is particularly useful for applications that require strict data validation, ensuring the output matches a predefined schema.

```
from gen_ai_hub.orchestration.models.message import SystemMessage, UserM
from gen_ai_hub.orchestration.models.template import Template, TemplateV

template = Template(
    messages=[
        SystemMessage("You are a helpful translation assistant."),
        UserMessage("{{?user_query}}")
    response_format="text",
    defaults=[
        TemplateValue(name="user_query", value="Who was the first person
    ]
)

# Response:
# The first man on the moon was Neil Armstrong.
```

```
from gen_ai_hub.orchestration.models.message import SystemMessage, UserM
from gen_ai_hub.orchestration.models.template import Template, TemplateV

template = Template(
    messages=[
        SystemMessage("You are a helpful translation assistant. Format t
        UserMessage("{{?user_query}}")
    ],
    response_format="json_object",
    defaults=[
        TemplateValue(name="user_query", value="Who was the first person
    ]
)

# Response:
# {
#     "First_man_on_the_moon": "Neil Armstrong"
# }
```

Important: When using `response_format` as json_object, ensure that messages contain the word 'json' in some form.

```
from gen_ai_hub.orchestration.models.message import SystemMessage, UserM
from gen_ai_hub.orchestration.models.template import Template, TemplateV
from gen_ai_hub.orchestration.models.response_format import ResponseForm

json_schema = {
    "title": "Person",
    "type": "object",
    "properties": {
            "firstName": {
            "type": "string",
            "description": "The person's first name."
        },
                "type": "string",
                "description": "The person's last name."
            }
        }
}
template = Template(
    messages=[
        SystemMessage("You are a helpful translation assistant."),
        UserMessage("{{?user_query}}")
    ],
    response_format = ResponseFormatJsonSchema(name="person", descriptio
    defaults=[
        TemplateValue(name="user_query", value="Who was the first person
    ]
)

# Response:
# {
#     "firstName": "Neil",
#     "lastName": "Armstrong"
# }
```

Understanding Deployment Resolution

The `OrchestrationService` class provides multiple ways to specify and target orchestration deployments when sending requests. Below are the available options:

Default Behavior

If no parameters are provided, the `OrchestrationService` automatically searches for a `RUNNING` deployment. If multiple running deployments exist, the service selects the most recently created one.

Direct Deployment Specification

You can explicitly define the target deployment using the following options:

1. **API URL** (`api_url`):

   - Specify the exact URL assigned to the deployment during its creation.

   - Refer to the Prerequisites section for more details on obtaining the deployment URL.

2. **Deployment ID** (`deployment_id`):

   - Use the unique identifier assigned to the deployment instead of the URL.

Config-Based Specification

If you want to target deployments based on their configuration source, use one of the following options:

1. **Configuration ID** (`config_id`):

   - The `OrchestrationService` searches for a `RUNNING` deployment created using the provided configuration ID.

2. **Configuration Name** (`config_name`):

   - The service looks for a `RUNNING` deployment that matches the specified configuration name.

If multiple deployments match the given configuration criteria, the most recently created one will be selected automatically.

Optional Modules

Data Masking

The `Data Masking` module `anonymizes` or `pseudonymizes` personally identifiable information (PII) before it is processed by the LLM module. Currently, `SAPDataPrivacyIntegration` is the only available masking provider.

Masking Types

- **Anonymization**: All identifying information is replaced with placeholders (e.g., MASKED_ENTITY), and the original data cannot be recovered, ensuring that no trace of the original information is retained.

- **Pseudonymization**: Data is substituted with unique placeholders (e.g., MASKED_ENTITY_ID), allowing the original information to be restored if needed.

In both cases, the masking module identifies sensitive data and replaces it with appropriate placeholders before further processing.

Configuration Options

- **entities**: Specify which types of entities to mask (e.g., EMAIL, PHONE, PERSON).
- **allowlist**: Provide specific terms or patterns that should be excluded from masking, even if they match entity types.
- **mask_grounding_input**: When enabled, ensures that masking is also applied to the context provided to the grounding module.

```
from gen_ai_hub.orchestration.utils import load_text_file
from gen_ai_hub.orchestration.models.data_masking import DataMasking
from gen_ai_hub.orchestration.models.sap_data_privacy_integration import
    ProfileEntity

orchestration_service = OrchestrationService()

data_masking = DataMasking(
    providers=[
        SAPDataPrivacyIntegration(
            method=MaskingMethod.ANONYMIZATION,  # or MaskingMethod.PSEU
            entities=[
                ProfileEntity.EMAIL,
                ProfileEntity.PHONE,
                ProfileEntity.PERSON,
                ProfileEntity.ORG,
                ProfileEntity.LOCATION
            ],
            allowlist=["M&K Group"],  # Terms to exclude from masking
        )
    ]
)

config = OrchestrationConfig(
    template=Template(
        messages=[
            SystemMessage("You are a helpful AI assistant."),
            UserMessage("Summarize the following CV in 10 sentences: {{?
        ]
    ),
        name="gpt-4o",
    ),
    data_masking=data_masking
)

cv_as_string = load_text_file("data/cv.txt")

result = orchestration_service.run(
    config=config,
    template_values=[
        TemplateValue(name="orgCV", value=cv_as_string)
    ]
)
```

```
print(result.orchestration_result.choices[0].message.content)
```

Content Filtering

The `Content Filtering` module can be configured to filter both the `input` to the LLM module (input filter) and the `output` generated by the LLM (output filter). The module uses predefined classification services to detect inappropriate or unwanted content. Azure Content Filter sensitivity is controlled by customizable `thresholds`, assuring the content aligns with the desired standards before processing or generating as output. Llama Guard 3 Filter, equipped with 14 categories, runs on a binary mechanism, accepting only true or false. Setting a category to true enables filtering for it. It's possible to execute both filters in a single request, optimizing efficiency.

```
from gen_ai_hub.orchestration.models.content_filtering import ContentFil
from gen_ai_hub.orchestration.models.azure_content_filter import AzureCo
from gen_ai_hub.orchestration.models.llama_guard_3_filter import LlamaGu

input_filter= AzureContentFilter(hate=AzureThreshold.ALLOW_SAFE,
                                  violence=AzureThreshold.ALLOW_SAFE,
                                  self_harm=AzureThreshold.ALLOW_SAFE,
                                  sexual=AzureThreshold.ALLOW_SAFE)
input_filter_llama = LlamaGuard38bFilter(hate=True)
output_filter = AzureContentFilter(hate=AzureThreshold.ALLOW_SAFE,
                                   violence=AzureThreshold.ALLOW_SAFE_LO
                                   self_harm=AzureThreshold.ALLOW_SAFE_L
                                   sexual=AzureThreshold.ALLOW_ALL)
output_filter_llama = LlamaGuard38bFilter(hate=True)
```

```
config = OrchestrationConfig(
    template=Template(
        messages=[
            SystemMessage("You are a helpful AI assistant."),
            UserMessage("{{?text}}"),
        ]
    ),
    llm=LLM(
        name="gpt-4o",
    ),
    filtering=ContentFiltering(
        input_filtering=InputFiltering(filters=[input_filter, input_filt
        output_filtering=OutputFiltering(filters=[output_filter, output_
    )
)
```

```
from gen_ai_hub.orchestration.exceptions import OrchestrationError

try:
    result = orchestration_service.run(config=config, template_values=[
        TemplateValue(name="text", value="I hate you")
    ])
except OrchestrationError as error:
    print(error.message)
```

Streaming

When you initiate an orchestration request, the full response is typically processed and delivered in one go. For longer responses, this can lead to delays in receiving the complete output. To mitigate this, you have the option to stream the results as they are being generated. This helps in rapidly processing or displaying initial portions of the results without waiting for the entire computation to finish.

To activate streaming, use the `stream` method of the `OrchestrationService`. This method returns an object that streams chunks of the response as they become available. You can then extract relevant information from the `delta` field.

Here's how you can set up a simple configuration to stream orchestration results:

```
    template=Template(
        messages=[
            SystemMessage("You are a helpful AI assistant."),
            UserMessage("{{?text}}"),
        ]
    ),
    llm=LLM(
        name="gpt-4o-mini",
        parameters={
            "max_completion_tokens": 256,
            "temperature": 0.0
        }
    ),
)

service = OrchestrationService()

response = service.stream(
    config=config,
    template_values=[
        TemplateValue(name="text", value="Which color is the sky? Answer
    ]
)

for chunk in response:
    print(chunk.orchestration_result)
    print("*" * 20)
```

Note: As shown above, streaming responses contain a delta field instead of a message field.

You can customize the global stream behavior by setting options like `chunk_size` which controls the amount of data processed in each chunk:

```
response = service.stream(
    config=config,
    template_values=[
        TemplateValue(name="text", value="Which color is the sky? Answer
    ],
    stream_options={
        'chunk_size': 25
    }
)

for chunk in response:
```