Orchestration Service

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}}: {{?text}}"
        ),
    ],
    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.

Referencing Templates in the Prompt Registry

You can also use a prompt template reference. This approach 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

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."),
        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."
        },
            "lastName": {
            "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"
# }
```

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

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

llm = LLM(name="gpt-4o", version="latest", parameters={"max_tokens": 256
```

This configuration initializes the language model to use the `gpt-4o` model with the `latest` version. The model will generate responses up to 256 tokens in length and produce more predictable and focused output due to the low temperature setting.

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,  # or use a referenced prompt template from Step
    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="text", value="The Orchestration Service is worki
])
```

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

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: {{?
        ]
    ),
    llm=LLM(
        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)
```

```
The CV summarized is for an experienced financial manager who specialize
```

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:

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

service = OrchestrationService(
    api_url=YOUR_API_URL
```

```
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:
    print(chunk.orchestration_result)
    print("*" * 20)
```

Modules that influence or process streaming results, such as `OutputFiltering`, might need specific stream options. The `overlap` option allows you to include extra context during the filtering process:

```
config = OrchestrationConfig(
    template=Template(
        messages=[
            SystemMessage("You are a helpful AI assistant."),
            UserMessage("{{?text}}"),
        ]
    ),
    llm=LLM(
```

```
        parameters={
            "max_tokens": 256,
            "temperature": 0.0
        }
    ),
    output_filtering=OutputFiltering(
        filters=[AzureContentFilter(
            hate=AzureThreshold.ALLOW_ALL,
            violence=AzureThreshold.ALLOW_ALL,
            self_harm=AzureThreshold.ALLOW_ALL,
            sexual=AzureThreshold.ALLOW_ALL
            )
        ],
        stream_options={ 'overlap': 100 }
    )
)

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

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

Advanced Examples

```
service = OrchestrationService(api_url=YOUR_API_URL)
```

Translation Service

This example extends the initial walkthrough of a basic orchestration pipeline by abstracting the translation task into its own reusable `TranslationService` class. Once the configuration is established, it can be easily adapted and reused for different translation scenarios.

```
from gen_ai_hub.orchestration.models.config import OrchestrationConfig
from gen_ai_hub.orchestration.models.llm import LLM
from gen_ai_hub.orchestration.models.message import SystemMessage, UserM
```

```
from gen_ai_hub.orchestration.service import OrchestrationService


class TranslationService:
    def __init__(self, orchestration_service: OrchestrationService):
        self.service = orchestration_service
        self.config = OrchestrationConfig(
            template=Template(
                messages=[
                    SystemMessage("You are a helpful translation assista
                    UserMessage(
                        "Translate the following text to {{?to_lang}}: {
                    ),
                ],
                defaults=[
                    TemplateValue(name="to_lang", value="English"),
                ],
            ),
            llm=LLM(name="gpt-4o"),
        )

    def translate(self, text, to_lang):
        response = self.service.run(
            config=self.config,
            template_values=[
                TemplateValue(name="to_lang", value=to_lang),
                TemplateValue(name="text", value=text),
            ],
        )

        return response.orchestration_result.choices[0].message.content
```

```
translator = TranslationService(orchestration_service=service)
```

```
result = translator.translate(text="Hello, world!", to_lang="French")
print(result)
```

```
result = translator.translate(text="Hello, world!", to_lang="Spanish")
print(result)
```

```
result = translator.translate(text="Hello, world!", to_lang="German")
print(result)
```

Chatbot with Memory

This example demonstrates how to integrate the `OrchestrationService` with a chatbot to handle conversational flow.

When making requests to the orchestration service, you can specify a list of messages as `history` that will be prepended to the templated content and processed by the templating module. These messages are plain, non-templated messages, as they typically represent past conversation outputs — such as in this chatbot scenario.

It's important to note that managing conversation history / state is handled locally in the `ChatBot` class, not by the orchestration service itself.

```
from typing import List

from gen_ai_hub.orchestration.models.config import OrchestrationConfig
from gen_ai_hub.orchestration.models.llm import LLM
from gen_ai_hub.orchestration.models.message import Message, SystemMessa
from gen_ai_hub.orchestration.models.template import Template, TemplateV
from gen_ai_hub.orchestration.service import OrchestrationService


class ChatBot:
    def __init__(self, orchestration_service: OrchestrationService):
        self.service = orchestration_service
        self.config = OrchestrationConfig(
            template=Template(
                messages=[
                    SystemMessage("You are a helpful chatbot assistant."
                    UserMessage("{{?user_query}}"),
                ],
            ),
            llm=LLM(name="gpt-4"),
        )
        self.history: List[Message] = []

    def chat(self, user_input):
        response = self.service.run(
            config=self.config,
            template_values=[
                TemplateValue(name="user_query", value=user_input),
            ],
            history=self.history,
        )
```

```
        self.history = response.module_results.templating
        self.history.append(message)

        return message.content

    def reset(self):
        self.history = []
```

```
bot = ChatBot(orchestration_service=service)
```

```
print(bot.chat("Hello, how are you?"))
```

```
print(bot.chat("What's the weather like today?"))
```

```
print(bot.chat("Can you remember what I first asked you?"))
```

```
bot.reset()
```

```
print(bot.chat("Can you remember what I first asked you?"))
```

Sentiment Analysis with Few Shot Learning

This example demonstrates the different message `roles` in the templating module through a few-shot learning use case with the `FewShotLearner` class.

- **Message Types:** Different message types (`SystemMessage` , `UserMessage` , `AssistantMessage`) structure the interaction and guide the model's behavior.
- **Templating:** The template includes these examples, ending with a `placeholder` ({{?user_input}}) for dynamic user input.
- **Few-Shot Examples:** Pairs of UserMessage and AssistantMessage show how the model should respond to similar queries.

The FewShotLearner class manages the dynamic creation of the template and ensures the correct message roles are used for each user input.

```
from typing import List, Tuple

from gen_ai_hub.orchestration.models.config import OrchestrationConfig
from gen_ai_hub.orchestration.models.llm import LLM
from gen_ai_hub.orchestration.models.message import (
    SystemMessage,
    UserMessage,
    AssistantMessage,
)
from gen_ai_hub.orchestration.models.template import Template, TemplateV
from gen_ai_hub.orchestration.service import OrchestrationService


class FewShotLearner:
    def __init__(
            self,
            orchestration_service: OrchestrationService,
            system_message: SystemMessage,
            examples: List[Tuple[UserMessage, AssistantMessage]],
    ):
        self.service = orchestration_service
        self.config = OrchestrationConfig(
            template=self._create_few_shot_template(system_message, exam
            llm=LLM(name="gpt-4o-mini"),
        )

    @staticmethod
    def _create_few_shot_template(
            system_message: SystemMessage,
            examples: List[Tuple[UserMessage, AssistantMessage]],
    ) -> Template:
        messages = [system_message]

        for example in examples:
            messages.append(example[0])
            messages.append(example[1])
        messages.append(UserMessage("{{?user_input}}"))

        return Template(messages=messages)

    def predict(self, user_input: str) -> str:
        response = self.service.run(
            config=self.config,
            template_values=[TemplateValue(name="user_input", value=user
        )
```

```
sentiment_examples = [
    (UserMessage("I love this product!"), AssistantMessage("Positive")),
    (UserMessage("This is terrible service."), AssistantMessage("Negativ
    (UserMessage("The weather is okay today."), AssistantMessage("Neutra
]
```

```
sentiment_analyzer = FewShotLearner(
    orchestration_service=service,
    system_message=SystemMessage(
        "You are a sentiment analysis assistant. Classify the sentiment
    ),
    examples=sentiment_examples,
)
```

```
print(sentiment_analyzer.predict("The movie was a complete waste of time
```

```
print(
    sentiment_analyzer.predict("The traffic was fortunately unusually li
)
```

```
print(
    sentiment_analyzer.predict("I'm not sure how I feel about the recent
)
```

Async Support

The `OrchestrationService` also supports asynchronous calls. Use:

- `arun` from the async version of `run`
- `astream` from the async version of `stream`

```
import asyncio
```

```
from gen_ai_hub.orchestration.models.template import Template, TemplateV
from gen_ai_hub.orchestration.models.llm import LLM
from gen_ai_hub.orchestration.models.config import OrchestrationConfig

from IPython.display import display, Markdown # just for pretty print in


config = OrchestrationConfig(
            llm=LLM(name="gemini-1.5-pro"),
            template=Template(
                messages=[
                    SystemMessage("This is a system message."),
                    UserMessage("Write a markdown cheatsheet!"),
                ],
            ),
        )

# Instantiate the orchestration service.
from gen_ai_hub.orchestration.service import OrchestrationService
orchestration_service = OrchestrationService(config=config)
```

```
async def test_async():
    async_result = await orchestration_service.arun()
    display(Markdown(async_result.orchestration_result.choices[0].messag

await test_async()
```

```
async def test_streaming_async():
    streamed_content = ""
    async for chunk in await orchestration_service.astream():
        if chunk.orchestration_result.choices:
            streamed_content += chunk.orchestration_result.choices[0].de
            display(Markdown(streamed_content), clear=True)

await test_streaming_async()
```