ChatGPT

Unified Azure AI Foundry & Semantic Kernel Python SDK Wrapper: Comprehensive Guide

Introduction

Enterprise software engineers working with **Azure AI Foundry** and **Microsoft Semantic Kernel (SK)** need a unified Python SDK to streamline AI tasks across Azure-hosted resources. This guide outlines a Python SDK wrapper that combines **Azure AI Foundry** (preview) capabilities with SK's powerful orchestration, tailored for Azure Machine Learning (Azure ML) pipeline integration. We will cover model deployment and chat completion, embeddings generation, agent creation with planning/orchestration, tool & plugin registration (OpenAPI and SK plugins), retrieval-augmented generation (RAG), multi-agent communication, security (content scanning & auth), testing, CI/CD, and best practices. The focus is on enterprise scenarios: robust Azure integration, compliance, and scalability.

Overview of Azure AI Foundry SDK and Semantic Kernel

Azure AI Foundry is a unified platform for building and operating AI applications, combining models, tools, and governance in an enterprise-ready environment [1] [2]. Azure AI Foundry introduces the concept of **AI Projects** and **AI Agents**. Key client libraries include:

- `azure-ai-projects` (Python SDK, preview) – The entry point for Foundry projects. It lets you manage projects, **deployments, connections, datasets, indexes, evaluations, and agents** [3] [4]. Notably, you can create and run agents via `AIProjectClient.agents`, get Azure OpenAI or other model clients via `project_client.inference`, and enumerate resources like models and indexes [3] [5].
- `azure-ai-agents` – A sub-package (installed with `azure-ai-projects`) containing the **Agents** client and models. As of version 1.0.0b11, agent-specific functionality was refactored into this package [6]. Developers still use `project_client.agents` property to create/run agents, but behind the scenes it leverages `azure-ai-agents`.
- `azure-ai-inference` – A client library to perform model **inference** (chat completions, text embeddings, image embeddings) on models deployed via Azure AI Foundry or Azure OpenAI [7] [8]. This provides unified clients like `ChatCompletionsClient` and `EmbeddingsClient` for various model hosts (OpenAI, Azure OpenAI, open-source models, etc.). It's recommended for comparing models uniformly [9] (though the official OpenAI SDK can also be used for Azure OpenAI models [10]).
- **Semantic Kernel (SK)** – An open-source SDK enabling AI **plugins, planning, and orchestration** in applications. The `semantic-kernel[azure]` package includes Azure-specific connectors (for Azure OpenAI, Azure Cognitive Search, etc.) [11]. SK allows building complex AI workflows by defining **functions (plugins)** – either **semantic functions** (prompt templates) or **native functions** (code) – and using **planners** or agents to compose them. SK's **Agent Framework** (experimental) integrates

with Azure AI Agents, providing classes like `AzureAIAgent` and `AgentGroupChat` for multi-agent systems [12] [13].

By combining these, our unified SDK wrapper will allow developers to: deploy and invoke models, generate embeddings, create intelligent agents that use tools, orchestrate multi-step or multi-agent processes, and integrate all of this into Azure ML pipelines with enterprise-grade security and monitoring.

Model Deployment and Chat Completion

Model Deployment: In Azure AI Foundry, you deploy models (from OpenAI or open-source) via the portal or SDK. Each deployment gets a name and endpoint in your Foundry Project. Ensure you have the project's **endpoint URL** and proper authentication. Typically, set an environment variable `PROJECT_ENDPOINT` to your project's URL of the form `https://<your-resource>.services.ai.azure.com/api/projects/<project-name>` [14]. For Azure OpenAI models, deployments exist under an Azure OpenAI resource with a different endpoint (ending in `.openai.azure.com/...`) [15].

Authentication: Use **DefaultAzureCredential** (from `azure.identity`) for Entra ID (formerly AAD) token auth – this is the recommended method for enterprise scenarios [16] [17]. Ensure your identity has the required role (e.g. *Azure AI User* or *Project Owner* on the Foundry project) [18]. In development, you can authenticate via `az login` for DefaultAzureCredential to pick up your user token [19]. Alternatively, for Azure OpenAI endpoints you may use an API key via `AzureKeyCredential` if not using token auth [20] [21].

Chat Completion: The wrapper should expose a simple interface to get chat completions from a deployed model. Under the hood, it can leverage Azure's inference SDK. For example, using `azure-ai-projects` you can obtain an authenticated chat client in one line:

```
from azure.ai.projects import AIProjectClient
from azure.identity import DefaultAzureCredential

project_client = AIProjectClient(credential=DefaultAzureCredential(),
                                 endpoint=os.environ["PROJECT_ENDPOINT"])
# Get a ChatCompletionsClient for any model in the project
with project_client.inference.get_chat_completions_client() as chat_client:
    response = chat_client.complete(
        model="gpt-4o-mini",  # your deployed model name
        messages=[{"role": "user", "content": "How many feet are in a mile?"}]
    )
    print(response.choices[0].message.content)
```

This uses the Azure AI Inference `ChatCompletionsClient` under the hood [22]. The `complete` method sends a list of messages (in OpenAI chat format) and returns a response with choices. If you specifically need an **Azure OpenAI** client (e.g., to set an API version or use the OpenAI SDK interface), you can get one similarly via `project_client.inference.get_azure_openai_client(api_version=<...>)` [23] [24]. The Azure OpenAI client from the `openai` package can then be used to create chat completions as well [25].

Sample Chat Prompt Completion:

Suppose we deployed a model called `"gpt-4"` in our project. We can query it:

```
with project_client.inference.get_azure_openai_client(api_version="2024-10-21")
as aoai_client:
    completion = aoai_client.chat.completions.create(
        model="<deployment-name>",  # Foundry or connected AOAI deployment name
        messages=[{"role": "user", "content": "Hello, how are you?"}]
    )
print(completion.choices[0].message.content)
```

Both approaches yield the model's response. The wrapper can abstract these by exposing a `chat_complete(prompt: str, model: str)` method that internally constructs the message payload and calls the appropriate client.

Environment Variables: In an Azure ML pipeline, you might store `PROJECT_ENDPOINT` and rely on DefaultAzureCredential (which in Azure ML can use the workspace's managed identity if configured). For Azure OpenAI, if not using Entra ID, you'd provide `AZURE_OPENAI_API_KEY` and resource info. Our SDK wrapper could automatically read common env vars (like `AZURE_AI_INFERENCE_ENDPOINT` and `AZURE_AI_INFERENCE_API_KEY` used by SK's Azure connector [26] [27]) for convenience.

Dependency Versions: Use the latest preview of `azure-ai-projects` (e.g. 1.0.0b11 as of May 2025) and ensure `azure-ai-inference` is installed (e.g. `pip install azure-ai-inference>=1.0.0b9`). The OpenAI SDK (`openai` Python package) is optional – only needed if you want to use OpenAI's native client interface via `get_azure_openai_client`. Our wrapper can default to using the Azure inference SDK for uniformity.

Embeddings Generation

Embeddings are vector representations of text or images, useful for semantic search or similarity. The SDK wrapper should provide a method to generate embeddings from a given input text (or list of texts) using a chosen model.

Using Azure's SDK, one can get a text `EmbeddingsClient` via the Project client or call SK's connector. For instance:

Via Azure AI Inference SDK:

```
from azure.ai.inference import EmbeddingsClient
from azure.core.credentials import AzureKeyCredential

emb_client =
EmbeddingsClient(endpoint=os.environ["AZURE_AI_INFERENCE_ENDPOINT"],
```

```
credential=AzureKeyCredential(os.environ["AZURE_AI_INFERENCE_API_KEY"]))
result = emb_client.embed(model="<embedding-model-deployment>",
                          input=["Hello world", "Contoso Inc"])
vector1 = result.data[0].embedding  # embedding for "Hello world"
```

However, if we have a Project client, we can simplify:

```
with project_client.inference.get_chat_completions_client() as client:

# The ChatCompletionsClient can also do embedding if we had EmbeddingsClient
similarly
    # (Alternatively, azure-ai-projects may expose get_embeddings_client)
    pass
```

Azure's docs suggest using the `EmbeddingsClient` or even a convenience method. The samples indicate that `project_client.inference` can provide an `EmbeddingsClient` as well [28].

Via Semantic Kernel: SK has an Azure Inference connector class `AzureAIInferenceTextEmbedding`. For example:

```
from semantic_kernel.connectors.ai.azure_ai_inference import
AzureAIInferenceTextEmbedding

# Assuming AZURE_AI_INFERENCE_ENDPOINT & API_KEY env vars are set and model
deployed
embedding_generator = AzureAIInferenceTextEmbedding(ai_model_id="<embedding-
model-deployment>")
embeddings = await embedding_generator.generate_embeddings(
    texts=["My favorite color is blue.", "I love to eat pizza."]
)
for emb in embeddings:
    print(len(emb)),  # length of embedding vector
```

This uses the environment variables for endpoint and key if not explicitly provided [26] [29]. The output `embeddings` is a list of float vectors. Our wrapper could wrap this call to return the vectors and handle any batching if needed.

Choosing Models: Ensure you use an embedding model deployment for embeddings generation (e.g., if using OpenAI models, a text embedding model like `text-embedding-ada-002`, or if using Azure's model catalog, something like `cohere-embed-multilingual-v3` as in Foundry's catalog [30]). The wrapper could allow specifying a model name or default to a configured embedding model.

Example – Text Embedding:

```
vectors = sdk_wrapper.generate_text_embeddings(
    ["How to optimize Azure costs?", "Azure AI Foundry architecture"])
# vectors is a list of embedding vectors (one per input string)
```

Under the hood, this might call SK's `AzureAIInferenceTextEmbedding` or the Azure SDK's `EmbeddingsClient.embed` method. For image embeddings (if needed for vision tasks), Azure AI Inference also provides `ImageEmbeddingsClient` similarly [31], which could be integrated for multimodal capabilities.

AI Agent Creation, Planning, and Orchestration

Creating an **AI Agent** is central to Azure AI Foundry. An *agent* pairs an LLM (model deployment) with instructions (system prompt defining its role/behavior) and optionally a set of tools it can use [32]. The Azure AI **Agent Service** manages agent conversations (threads), tool use, and reasoning under the hood [33]. Our SDK wrapper will simplify creating and using agents for developers.

Creating an Agent: With the `azure-ai-projects` client, you can create an agent using `project_client.agents.create_agent()`. At minimum you provide the model (deployment name) and instructions. For example:

```
agent = project_client.agents.create_agent(
    model=os.environ["MODEL_DEPLOYMENT_NAME"],
    name="my-agent",
    instructions="You are a helpful agent that answers travel questions."
)
print(f"Created agent, ID: {agent.id}")
```

This returns an `Agent` object with a unique ID (prefixed likely with `asst_...`). The agent is persisted in the project (until deleted) and can be re-used. By default, this is a **basic agent** with just the prompt; it won't have any tools unless specified (we'll cover tool integration shortly).

Agent Threads & Messages: To have a conversation or run an agent, we need a **Thread** – a conversation session that holds messages between user and agent [34]. Use `project_client.agents.threads.create()` to start a new thread:

```
thread = project_client.agents.threads.create()
print(f"Created thread, ID: {thread.id}")
```

We can then add user messages to this thread. For example, if we want to ask the agent something:

```
msg = project_client.agents.messages.create(
    thread_id=thread.id,
```

```
    role="user",
    content="What's the weather in Seattle today?"
)
```

Now we trigger the agent to process the thread. Azure provides a one-shot call to **create and execute a run**:

```
run = project_client.agents.runs.create_and_process(thread_id=thread.id,
agent_id=agent.id)
print(f"Run finished with status: {run.status}")
```

This call starts the agent, which reads all messages in the thread (in this case, just the user's question) and produces a response. The `create_and_process` API handles the entire loop of the agent reasoning, including any tool calls the agent decides to make, and appends the assistant's answer to the thread messages [35]. By the end, `run.status` should be "succeeded" if everything went well. We can then retrieve messages:

```
for message in project_client.agents.messages.list(thread_id=thread.id):
    print(f"{message.role}: {message.content}")
```

We would see the assistant's reply (with role "assistant") in the list [36].

Agent Reasoning & Planning: One powerful aspect of the Agent Service is that it automates the **planning** and tool use. The agent uses the LLM to decide if a tool is needed (e.g., calling an API) or if it can answer directly, following a thought process. Developers do **not** have to manually parse the model's intent or manage function calls – the Agent Service does that on the server side (this is "automatic tool calling") [37]. Essentially, the agent dynamically **plans steps** to fulfill the user's request: it may retrieve info, call a tool, then formulate an answer. This aligns with the ReAct pattern (reasoning and acting) but offloads orchestration to Azure.

If using Semantic Kernel's approach, SK also offers a **Planner** that can generate a sequential plan of function calls for a given goal, but in our context the Azure agent's internal planner suffices. SK's integration with Azure Agents allows you to wrap the Azure agent in an SK `AzureAIAgent` object, giving you additional programmatic control if needed [12] [38]. For example, you could use SK to maintain a local conversation memory or to combine the agent with SK's other plugins in a unified `Kernel`. This is advanced usage – our wrapper can initially rely on Azure to handle planning.

Running an Agent (Chat Completion vs. Agent Run): It's important to differentiate a basic completion call from an agent run. A raw chat completion (like in the previous section) takes a prompt and returns a completion. An **agent run**, however, involves a thread with state and possibly multiple model invocations (the agent might have to think and call tools multiple times). The Agent Service tracks a **"Run"** object representing this conversation turn (including intermediate steps, which can be inspected via `agents.run_steps.list()` [39] [40]). For simple Q&A, an agent run might just result in one answer; for complex tasks, the agent might perform several actions before concluding. Our SDK wrapper should hide

this complexity for straightforward uses but allow deeper inspection for debugging if needed (for instance, logging `run_steps` to see which tools were invoked and any errors).

Example – Creating and Querying an Agent:

```
# Initialize our SDK wrapper with project endpoint and credentials
sdk = AzureFoundrySKWrapper()  # hypothetical initializer in our wrapper
agent = sdk.create_agent(model="gpt-4o-mini", name="WeatherAgent",
                         instructions="You answer weather questions concisely.")
answer = sdk.run_agent(agent, user_message="What's the weather in Seattle
today?")
print(answer)
```

Under the hood, `create_agent` calls `project_client.agents.create_agent`, and `run_agent` creates a thread, posts the user message, and calls `create_and_process`. The result `answer` is extracted from the last assistant message.

Agent Orchestration: Our wrapper can provide higher-level orchestration such as: - **Continuous Conversations**: If you want a multi-turn conversation, you can reuse the same `thread` and keep calling `agents.runs.create_and_process` for each new user message in that thread. The agent will have access to prior context (the thread will automatically truncate history if it grows too large, according to the service's rules [41]). - **Parallel Agent Runs**: You can run multiple agents concurrently (for different tasks or users). The Azure service scales as needed. Keep an eye on rate limits (e.g., Azure OpenAI throughput limits) – if the underlying model is an Azure OpenAI deployment, you still must respect its TPM/RPM quotas. - **Error Handling**: If a run fails (`run.status == "failed"`), the `run.last_error` field provides an error message [42]. Our wrapper should catch exceptions (like `HttpResponseError`) and surface meaningful errors. For example, if credentials are wrong or a model name is invalid, you'd get a 401/404 error which we can catch and log [43].

In summary, the SDK wrapper makes agent usage as simple as calling `create_agent()` and then `run_agent()` with input, while internally managing threads and runs. Yet, it should also expose hooks for advanced use (e.g., retrieving run steps or continuing an existing thread for multi-turn dialogues).

Tool and Plugin Registration (OpenAPI & Semantic Kernel Plugins)

One of the most powerful features of agents is their ability to use **tools/plugins** – these are essentially functions that the agent can invoke to get information or take actions beyond what the base model knows. Tools can be web APIs, database queries, code execution, or any custom function. Our unified SDK should allow registering tools easily, especially **OpenAPI-described APIs** and SK-defined plugins.

Integrating OpenAPI Tools

Azure AI Agent Service allows you to add tools based on OpenAPI specifications. This means you can enable the agent to call external REST APIs (with authentication) by providing the API's OpenAPI (Swagger) definition. In Azure AI Foundry portal, you can add an OpenAPI tool to an agent by pasting the spec and

configuring auth (API key via a connection or Managed Identity) [44] [45]. In code, we use the `OpenApiTool` model from `azure.ai.agents.models` to achieve the same.

Example – Adding an OpenAPI tool: Suppose we have two APIs: a Weather API and a Countries info API, each with an OpenAPI JSON. We can load those specs and create a tool:

```
import jsonref
from azure.ai.agents.models import OpenApiTool, OpenApiAnonymousAuthDetails

# Load OpenAPI specs from files
with open("weather.json") as f:
    openapi_weather = jsonref.loads(f.read())
with open("countries.json") as f:
    openapi_countries = jsonref.loads(f.read())

# Set up auth for the APIs (here, assume both are public or require no auth)
auth = OpenApiAnonymousAuthDetails()

# Create an OpenApiTool definition combining both APIs
openapi_tool = OpenApiTool(
    name="get_weather",
    spec=openapi_weather,
    description="Retrieve weather information for a location",
    auth=auth
)
openapi_tool.add_definition(
    name="get_countries",
    spec=openapi_countries,
    description="Retrieve country info by currency",
    auth=auth
)
```

Here we created an OpenApi tool with two functions: “get_weather” and “get_countries”. We then pass this tool when creating the agent:

```
agent = project_client.agents.create_agent(
    model=os.environ["MODEL_DEPLOYMENT_NAME"],
    name="MyAgentWithAPIs",
    instructions="You are a helpful agent with access to weather and country
info.",
    tools=openapi_tool.definitions  # pass the list of tool definitions
)
print(f"Created agent, ID: {agent.id}")
```

The agent is now configured to use those API functions. At runtime, if the user asks, *"What's the weather in Seattle and the population of the country that uses currency THB?"*, the agent can call `get_weather` and `get_countries` internally to fetch data, then formulate a combined answer [46] [39]. This all happens automatically via the Agent Service's reasoning engine (the model's prompt will include tool descriptions, and it will output a decision to call them). Our wrapper's responsibility is to make it easy to register such tools by perhaps providing a method like `add_openapi_tool(agent, spec_path, auth_type='anonymous' or connection details)`.

Authentication for OpenAPI tools: In enterprise scenarios, your APIs might require API keys or OAuth tokens. The Azure SDK supports: - **Connection-based auth**: If you have stored an API key in Azure Key Vault or as a "custom key" connection in Foundry, you can use `OpenApiConnectionAuthDetails` with a `connection_id` [47]. The agent service will insert the key from that connection when calling the API. - **Managed Identity auth**: Use `OpenApiManagedAuthDetails` and specify the `audience` (resource URI) for the token [48]. The agent service will use the project's managed identity to fetch a token for that resource. This is useful for calling Azure REST APIs securely. - **Anonymous**: as shown above, for public or no-auth endpoints.

Our wrapper could allow passing a `requests.Auth` or similar, but likely better to accept either a path to spec plus an auth type (and credentials info) to internally construct the appropriate `OpenApiTool` object.

SK Plugins vs OpenAPI Tools: Semantic Kernel also provides a way to import OpenAPI plugins into the kernel. For example, you can do:

```
await kernel.add_plugin_from_openapi(
    plugin_name="weather",
    openapi_document_path="https://example.com/weatherSwagger.json",

execution_settings=OpenAPIFunctionExecutionParameters(enable_payload_namespacing=True)
)
```

This will generate SK functions for each operation in the OpenAPI spec, and the SK planner/agent can call these functions [49] [50]. In essence, both Azure Agent Service and SK offer similar plugin functionality. Our unified SDK might choose to primarily rely on Azure's agent tools for production (since the Agent Service handles them robustly with logging, retries [33]), while possibly integrating SK's approach for local or offline scenarios.

We should note that when adding OpenAPI tools, certain specs might need slight modifications to be AI-friendly (clear descriptions, unique parameter names, etc.) [51] [52]. Semantic Kernel documentation provides tips on making OpenAPI specs more LLM-friendly and how SK handles naming conflicts by payload namespacing [53] [54]. Our wrapper docs can reference these best practices.

Registering Semantic Kernel Plugins and Custom Tools

Beyond OpenAPI-described APIs, you may want to add **custom code functions** as tools. There are two approaches: 1. **As Azure Functions** or **Python functions** exposed via an OpenAPI (or Logic App, etc.): You

could wrap your function as a web API and then use the OpenAPI method above. 2. **Directly in Semantic Kernel**: SK allows you to create native functions and include them in an agent's skill set. For example, you might write a Python function to perform a calculation or database lookup and register it as a SK plugin. SK's agent can then call it like any other function.

If using SK's `ChatCompletionAgent` or `AzureAIAgent`, you can add SK plugins to the kernel and the agent will consider them. For instance:

```
from semantic_kernel import Kernel
kernel = Kernel()
# Import a native skill (say a MathSkill with a function `add` we wrote)
math_plugin = kernel.import_skill(MathSkill(), skill_name="math")
# Now math_plugin['add'] is a callable the agent could use
```

If we have an SK `AzureAIAgent` (wrapping an Azure Foundry agent) we could attach the kernel or functions to it. In .NET, they do `agent.Kernel = kernel` to give the agent access to those functions [55]. In Python, this API may differ, but conceptually, linking SK functions with Azure agent could enable more complex scenarios: e.g., the Azure agent can use both Azure-registered tools and local SK functions.

For our wrapper, if deep integration is needed, we can manage an internal SK `Kernel` and register any Python-callable tools as SK plugins. The wrapper's agent logic could first attempt Azure agent tools, and fallback to SK functions if appropriate. However, this introduces complexity in syncing state between Azure's thread and SK's chat history. A simpler approach is: **if using Azure Agent Service, stick to Azure tools**; if using SK's local orchestration (like in an offline mode), use SK plugins. The wrapper can support both modes but should clarify that they operate somewhat distinctly.

Tool Naming Constraints: One pitfall – ensure that tool names and function parameters are unique and descriptive. Azure will expose tool function names to the LLM; if two tools have a function with the same name, one might be skipped [56]. Similarly in SK, if two functions share a name, the planner might be confused. So always give unique plugin names and function names that reflect their purpose.

Example – Registering a Tool via Wrapper:

```
# Add an OpenAPI tool to agent using the wrapper
sdk.register_openapi_tool(agent, "weather.json", auth_type="api_key",
connection_name="WeatherAPIKeyConn")
# Now the agent can call the Weather API internally when needed.
```

This would internally load the spec, create the `OpenApiTool` with an `OpenApiConnectionAuthDetails` (pointing to a Foundry Connection that stores the API key), and update the agent (possibly recreating it or updating its config). Currently, Azure's API is to specify tools at agent creation; if you need to add later, you might recreate the agent with the new tools or have had them preconfigured. Our wrapper could manage a set of tool definitions and create the agent with all of them in one go.

Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation is crucial for enterprise apps: it means grounding the LLM's responses on external knowledge (your documents, knowledge bases) to improve accuracy and relevance. Our SDK wrapper should facilitate RAG by integrating with Azure's search and storage solutions.

Azure Cognitive Search Integration: Azure AI Foundry allows you to create a **Search Index** resource in your project (or connect an existing Azure Cognitive Search index) and use it as a knowledge source. In the SDK, `project_client.indexes` provides operations to manage indexes [57] [58]. For example, you can create an index referencing an existing Azure Cognitive Search index:

```
from azure.ai.projects import AzureAISearchIndex
index = project_client.indexes.create_or_update(
    name="faq-index", version="1",
    body=AzureAISearchIndex(connection_name="<AI_SEARCH_CONNECTION>",
index_name="<cog_search_index_name>")
)
```

This registers a search index in the Foundry project (backed by an Azure Cognitive Search service). Once an index is connected, you can **upload documents** to it via the `datasets` API [59] [60], or directly push content to the Cognitive Search service. Foundry's agent can then use an **Azure AI Search tool** implicitly to retrieve data. Indeed, Azure AI Agent Service offers out-of-the-box a "**Azure AI Search**" tool for indexing and querying documents [61]. If an agent is configured with an index, when a user asks something that might need data from documents, the agent can perform a retrieval step (LLM decides when to use it).

From an implementation perspective, to leverage this we should: - Ensure an index is created and populated with content (e.g., by using `project_client.datasets.upload_file()` to add documents [59] and then perhaps building an index over that dataset). - Possibly attach the index as a tool. It's not entirely clear if the agent automatically knows about the index or if we must add a specific tool. In the Foundry portal, you might "ground" an agent with a data source by enabling an index tool. The blog snippet mentions Bing and Azure Search as built-in tools available [61]. Likely, if an Azure Search connection is present, you can add the **AzureSearchTool** to the agent's tools at creation (similar to how CodeInterpreter was added in the quickstart [62]).

If needed, adding Azure Search tool in code might be done via a provided class (e.g., perhaps `AzureSearchToolDefinition`). For instance, .NET might have it. If not directly exposed, one approach is to use OpenAPI route: Azure Cognitive Search has a REST API that could be wrapped as an OpenAPI plugin. But since it's mentioned as out-of-box, the better approach is to use the built-in tool.

Our wrapper could simplify RAG by providing methods like: - `create_index(name)`: which sets up an index in the project. - `upload_documents(index_name, files)`: to add documents. - `attach_index_tool(agent, index_name)`: to ensure the agent can use that index.

Direct RAG via SK or LangChain: In cases you want more manual control (outside the agent service), you could do retrieval in the application code: - Use SK's memory connectors (e.g.,

`AzureCognitiveSearchMemory` in `semantic_kernel.connectors.memory` if available) to perform a vector similarity search on your index. - Or use the Azure Cognitive Search SDK to query by keyword or vector (if using semantic or vector search). - Then feed the top results into the prompt (e.g., as part of system/message content) before asking the LLM to answer.

This approach might bypass the agent's internal tool calling but gives you deterministic control. However, it requires prompt engineering to ensure the model uses the context. The Agent Service approach is more automatic: the model is prompted with tool descriptions like "you have a search tool, use it for relevant questions," and it will decide when to call it.

RAG Use Case Example: Imagine an agent that answers questions about internal company policies by searching a corpus of documents: 1. **Index Creation**: You create an index for the policy documents and upload all PDFs. 2. **Agent Setup**: Create the agent with instructions like "You answer questions about company policy. Use the `PolicySearch` tool to find relevant content in the knowledge base when needed." Add the Azure Search tool pointing to that index. 3. **User Query**: "What is our vacation policy for contractors?" – The agent will likely invoke the PolicySearch tool (which queries the index for "vacation policy contractors"), get relevant snippet, and then formulate an answer citing that info. 4. **Answer**: The agent's response is thus grounded in real data, reducing hallucination.

Our wrapper should help configure steps 1 and 2 easily.

Ensuring Relevant Results: Typically, you'd use **embeddings** for semantic RAG. You might embed documents and user query and do vector similarity. Azure Cognitive Search now supports vectors, which you can use via the AzureAISearchIndex (Foundry might abstract some of that). If manual, SK or LangChain could be used to handle the embedding and similarity scoring. But since Foundry provides evaluation and indexing tooling, leveraging that is preferred for enterprise (plus it integrates with the agent content filters and monitoring).

Citation or Logging: In RAG, it's often useful to trace what was retrieved. The Azure agent's `run_steps` will show tool calls, including the query and possibly some result info [40] [63]. We can log these for audit (ensuring no sensitive data is logged unless needed). For compliance, any use of RAG might require verifying that the model doesn't leak info beyond what's in documents or appropriate (hence content filtering at output still applies).

Multi-Agent Communication and Configuration

Multi-agent systems involve multiple agents collaborating or communicating to solve tasks. This can be powerful in complex workflows – e.g., one agent may specialize in data analysis and another in explanation, and together they handle a request. Azure AI Foundry Agent Service supports **agent-to-agent messaging** as a first-class concept [64], and Semantic Kernel provides constructs to orchestrate group conversations (the **AgentGroupChat**).

Use Cases: - Agents with distinct roles (e.g., a "StockExpertAgent" that knows finance data and an "InvestmentAdvisorAgent" that uses the stock agent's info to give advice [13] [65]). - Agents that represent different knowledge domains collaborating on a single query. - An agent calling another as a tool (since an agent's tool could be an API that internally triggers another agent – though this can get complex).

Azure Agent Service Approach: The service allows multi-agent coordination by sharing messages among agents. Practically, each agent would have its own ID, but they can be part of a shared thread or use the agent messaging API to send messages to each other. Azure's docs emphasize structured **threads** that include agent↔agent messages, not just user↔agent [66]. This means you could potentially create a thread and post a message from Agent A to it, then have Agent B respond, etc., using the `agents.messages.create` with appropriate roles (maybe a role like "agent" with agent's ID). However, managing this at the REST API level is intricate.

A simpler method is to leverage **Semantic Kernel's AgentGroupChat** which abstractly manages a set of agents in a conversation. SK's `AgentGroupChat` allows you to add multiple `Agent` instances and will handle turn-taking via configurable strategies [67] [68].

Example – Multi-Agent with SK: (Based on SK experimental support)

```
from semantic_kernel.agents import AzureAIAgent, AgentGroupChat
from semantic_kernel.agents.strategies import KernelFunctionSelectionStrategy,
KernelFunctionTerminationStrategy

# Assume we already created two agents in Azure Foundry and have their IDs
agent_def1 = await client.agents.get(agent_id1)  # pseudo-call to get agent
definition
agent_def2 = await client.agents.get(agent_id2)
# Wrap them as AzureAIAgent for SK
agent1 = AzureAIAgent(client=client, definition=agent_def1)
agent2 = AzureAIAgent(client=client, definition=agent_def2)

# Create selection and termination strategies (could use an SK prompt or simple
round-robin)
selection_function = ...  # (some SK function or lambda that picks which agent
replies next)
termination_function = ...  # (function that decides when to stop, e.g., if one
agent says "DONE")

chat = AgentGroupChat(
    agents=[agent1, agent2],
    selection_strategy=KernelFunctionSelectionStrategy(initial_agent=agent1,
function=selection_function, kernel=kernel, ...),
    termination_strategy=KernelFunctionTerminationStrategy(agents=[agent1],
function=termination_function, kernel=kernel, ...)
)
await chat.add_chat_message("User asks a question that requires both agents.")
async for response in chat.invoke():
    print(f"{response.name}: {response.content}")
```

In this pseudocode, we configure a group chat with two agents and provide strategies: - **Selection strategy**: decides which agent speaks next. Could be as simple as alternating, or based on a "role" function

output. In an example from Microsoft, they used a special function that reads the last message and suggests the next speaker (Reviewer vs Writer agent scenario) [67] [68]. - **Termination strategy**: decides when the conversation is done (e.g., if a certain keyword or a maximum turns reached) [68] [69].

Once set, adding a user message and invoking the chat triggers the agents to start exchanging messages, with our code capturing the streaming conversation [70] [71]. The output would label each agent's response. This is essentially automated multi-agent orchestration.

Simpler Approach: If designing our wrapper for enterprise devs, we might not expose all of the strategy configuration (which can be complex). Instead, we can provide some presets: - A **turn-based** strategy: agents respond in a fixed order, until a condition. - A **role-based** strategy: one agent designated as lead, another as subordinate, etc. - Or simply allow one agent to call another as needed (though implementing that may mean one agent's action triggers a message to the other).

The **AutoGen** framework (by Microsoft Research) is another approach for multi-agent, where agents can dynamically create new agents or tasks. The blog references integrating with AutoGen and SK for multi-agent workflows [72]. In our context, SK's AgentGroupChat is sufficient to demonstrate multi-agent collab.

Example Use Case: Agent A is an expert in a database (it has a tool to query database), Agent B is a report generator. A user asks: "Find the total sales for last quarter and summarize in a report." Agent A can query the DB, answer with the data, Agent B takes that and produces a summary. Using group chat, Agent A and B can be set to alternate: user question -> Agent A (with data) -> Agent B (with summary) -> done. The selection strategy can enforce this sequence.

Configuration Considerations: When creating agents for a multi-agent scenario, each agent's **instructions** should clarify its role (so they don't overlap or conflict). You may also want to **name** the agents distinctly and possibly use those names in the conversation for clarity. Our wrapper could enforce that by requiring a `role` or `purpose` attribute when creating multiple agents.

Resource Management: Running two agents means two model invocations potentially in parallel for a single user query. Be mindful of costs and rate limits. Multi-agent is powerful but doubles the usage if both agents use large models.

Multi-Agent in Azure ML Pipelines: A scenario could be a pipeline step that uses multi-agents to analyze data (e.g., one agent reads a dataset, another validates the findings). The wrapper should support launching such multi-agent sessions within a pipeline job. This means handling async event loop inside pipeline code (since SK is async). We can hide async behind sync methods in our wrapper (using `asyncio.run()` internally when needed) or encourage pipeline components to be written async-friendly.

In summary, our SDK will allow configuration of multiple agents and orchestrate their communication. This can be as simple as providing a `run_agents_together([agent1, agent2], user_prompt)` method. Under the hood, it could use SK's AgentGroupChat or sequential logic. The result would be a combined answer or a transcript of the agent dialogue.

Security, Content Scanning, and Authentication

Enterprise environments demand strict security and compliance, especially when dealing with AI outputs. Our unified SDK wrapper should integrate Azure's security features and provide hooks for additional safeguards.

Authentication & Access Control: We've largely covered auth (DefaultAzureCredential, API keys, etc.). The key is to use **managed identities or service principals** in production rather than user accounts. In Azure ML, you can assign a managed identity to the compute that has access to the Azure AI Foundry project (via the appropriate RBAC role) [18]. The DefaultAzureCredential will then seamlessly authenticate. No secret keys need to be stored in code. If API keys are used (for OpenAI or other services), store them in Azure Key Vault and link via Foundry connections or pipeline secrets, not in plaintext.

Network Security: Azure AI Foundry can be configured with private endpoints, meaning all operations (model inference, agent calls) stay within a VNet if needed. Our wrapper should allow specifying custom endpoints (e.g., if using a private DNS name) and perhaps work with Azure ML's VNet integration. For instance, if Azure ML workspace and Foundry are in the same VNet or peered, the pipeline can reach the Foundry endpoint securely. We should document that any necessary NSG or firewall rules need to allow the traffic, but those are infra concerns.

Content Safety & Scanning: Azure AI Agent Service includes **content filtering** out-of-the-box. All outputs are governed by policies and content filters to mitigate harmful content or data leaks [64]. For example, if a user prompt or an agent's response violates certain policies (hate, violence, etc.), the service can block or modify it. Additionally, when the agent uses tools, the results from tools are also subject to moderation (ensuring, say, that if an API returns disallowed content it's handled). As developers, you will get an error or a flagged content indication if a message was filtered. Our wrapper should propagate such information (perhaps via exceptions or special return values) so the application can handle it (e.g., inform the user that content can't be shown).

For further safety, Azure provides the **Azure AI Content Safety** service (separate resource) which can detect and score content for inappropriate material. One could integrate it for scanning user inputs before sending to the model, or scanning final outputs, for additional compliance (some orgs require a second layer of filtering). This is not built into Foundry as of writing, but our wrapper could optionally call the Content Safety API on outputs. However, given the Agent Service already does some filtering, this might be redundant.

Encryption and Data Handling: Data sent to Azure AI (prompts, chat histories) might be sensitive. Azure ensures data in transit is encrypted (HTTPS) and at rest in the service. The Foundry platform also allows BYO Storage (your own Azure Storage account for data like uploaded files) and BYO keys (customer-managed keys) for encryption [73] [74]. While our code may not directly handle encryption keys, be aware that using a managed storage vs. custom storage could be a config at project creation. If the enterprise requires that all data remain in certain regions or owned storage, configure the Foundry project accordingly (e.g., link your own Blob Storage and Key Vault). Our wrapper can then just use the project as usual, with no difference in code.

Secrets in Code: The wrapper itself should avoid printing or logging secrets (like API keys or access tokens). If we implement logging, ensure sensitive info is redacted. Azure SDKs by default redact secrets in logs and

require an explicit flag to log raw content [75] [76]. We will follow the same practice. For instance, enabling debug logging on our wrapper might reveal request URLs and status codes, but not the actual content or auth headers, unless explicitly allowed (and even then, caution is needed) [77] [78].

Auditing and Observability: Azure Agent Service offers structured logging and the ability to hook into Application Insights for monitoring agent conversations [79]. In the Foundry portal or via logging, you can get a full trace of a conversation thread (every message and tool call) [66]. For CI/CD and debugging, this is extremely useful. Our wrapper should allow toggling verbose logging or telemetry integration. Azure's `enable_telemetry()` function can send traces to stdout or an OTLP endpoint [80] [81]. Setting `AZURE_TRACING_ENABLED` and related environment variables can capture prompts and completions for debugging, though content recording is off by default for privacy [82]. In an enterprise pipeline, you might route telemetry to App Insights to monitor usage and detect failures (e.g., to alert if agent calls are failing or content filters trigger frequently).

Testing Security: As part of CI, one should include tests for security-related scenarios: - Does the wrapper correctly enforce using DefaultAzureCredential (e.g., fails if no creds found, so we don't accidentally use a fallback unsafe method)? - If an unauthorized identity is used, do we handle the 401 error gracefully (e.g., catching `HttpResponseError` with status code 401 and printing a clear message) [43]? - If content is flagged, do we return a specific exception or result code?

Rate Limits: Azure OpenAI and some other model endpoints have rate limits. For example, an `gpt-4` deployment might allow N requests/minute. If our wrapper will be used heavily (like in parallel in a pipeline), consider implementing a **retry with backoff** for rate-limit errors (HTTP 429). Azure SDKs don't always auto-retry on 429, so we could integrate a policy or simply catch 429 and sleep a bit. The user should be informed of these limits in documentation. Best practice is to handle `TooManyRequests` errors by backing off exponentially. We can integrate this at the HTTP pipeline level or with simple try/except logic around our calls.

Content Length and Prompt Limits: Ensure the wrapper checks or at least warns if prompts are too long. Each model has a context length (e.g., GPT-4 might be 8K or 32K tokens). The agent service likely handles truncation via the thread manager (dropping oldest messages) [83] [84], but if the user sends a huge one-shot prompt (outside of thread usage), we should possibly chunk it or warn.

In summary, our unified SDK will stand on Azure's robust security foundation – **Entra ID auth, RBAC, network isolation, content filtering by design** – and supplement it with convenient testing and logging features. By following Azure's guidelines and using the SDKs, we inherit a lot of security features automatically [74]. Our job is not to break them (e.g., don't bypass the content filter) and to make it easy for developers to comply (like one-liner auth, and no secrets needed explicitly).

Testing Strategies and CI/CD Integration

Developing a reliable SDK wrapper requires thorough testing and a solid CI/CD setup. Below we outline testing strategies including unit tests, integration tests (with mocking or record/playback), and how to integrate these into a CI/CD pipeline (particularly in Azure DevOps or GitHub Actions, and aligned with Azure ML pipelines).

Testing Strategies

1. Unit Testing with Mocks: For most SDK functionality, especially logic that doesn't need a live Azure call, use mocks. Python's `unittest.mock` or `pytest` fixtures can simulate the Azure clients. For example, when testing `sdk_wrapper.chat_complete()`, instead of calling the real Azure service, inject a fake `ChatCompletionsClient` that returns a preset response. This isolates our logic and avoids external dependencies. We can mock `project_client.agents.create_agent` to return a dummy agent ID, etc. This is critical for testing error handling: we can simulate `HttpResponseError` exceptions from the Azure SDK to ensure our wrapper catches and handles them properly (e.g., we might simulate a 401 and check that our wrapper raises a custom `AuthenticationError` with a helpful message).

2. Integration Testing (Recorded): To ensure the wrapper works end-to-end, write a few integration tests that actually call the Azure services. However, calling live services has downsides: requires Azure credentials, consumes quota, and tests might fail due to transient cloud issues. The solution is **recorded tests**. Azure's SDK test framework uses a recording proxy that captures HTTP interactions and saves them as YAML files, which can be replayed for offline testing. We can achieve similar with `pytest-recording` or `vcr.py`. For example, run a test that calls `sdk_wrapper.run_agent()` with a known prompt, record the HTTP calls to Azure (requests and responses). On subsequent runs, the responses are served from the recording, making tests fast and deterministic.

When recording, ensure to mask sensitive data (the recording framework usually filters out Authorization headers and any predefined secrets). This way, we can commit the recordings without leaking keys or tokens. Azure SDK's guidelines show that by default, logging and recording redact secrets like keys and tokens [85] [86].

3. Live Testing with a Test Instance: In some cases (especially before a release), you might run a live test against a real Azure project. For CI, this could be done by provisioning a test Azure AI Foundry project (via Terraform or script) and using a service principal's credentials. This is more complex but can be done in Azure DevOps pipelines with service connections. Given the preview nature of Foundry, recorded tests might suffice until GA.

4. Testing Async Behavior: Many methods (especially with SK) are `async`. Use `pytest.mark.asyncio` to test async functions. For group chat, for instance, you'd run an async test that awaits `sdk_wrapper.multi_agent_chat()` and assert it returns expected dialogues. If using `asyncio.run` internally in wrapper for sync interface, ensure it doesn't block event loops in environments like Jupyter (maybe allow an async interface as well for advanced use).

5. Edge Cases: Write tests for: - Creating an agent with invalid model (should raise error from Azure). - Running an agent with no messages (maybe it should just not produce anything or return gracefully). - Tool using: if an agent tool call fails (simulate an API returning 500), does the agent respond with an error message? The run might mark failed. We should handle that either by raising or returning info to user. - Ensure thread reuse works: send two sequential messages and see that second answer references first (if applicable). - Multi-agent termination: ensure our termination strategy stops the loop.

6. Performance Testing: Not typical in unit tests, but you might want a test that ensures the wrapper can handle, say, 10 parallel requests (maybe using threads or asyncio tasks) without race conditions (especially

if our wrapper is stateful). If the wrapper holds any global state (like an SK kernel), ensure it's thread-safe or document that it's not.

CI/CD Integration

Continuous Integration (CI): On each push or PR, run the test suite. Use a matrix to test across Python versions (e.g., 3.9, 3.10, 3.11 as our SDK requires 3.9+). If using GitHub Actions, set up `python-version` matrix. Install dependencies including azure SDKs and semantic-kernel. If using recorded tests, you don't need Azure credentials for CI (since tests will play back). For safety, have a mode to re-record easily (maybe via an env var like `RECORD=true` that will run tests live and update cassettes, but in CI we keep it false).

Integrate linting (flake8, black, isort) and type checking (mypy) in CI to maintain code quality.

Continuous Deployment (CD): If this wrapper is internal, you might publish it to an artifact feed or PyPI. Set up a pipeline to build the package (ensure version bumping strategy, e.g., use Git tags for version). Possibly use Azure DevOps or GitHub Actions to deploy on a merge to main or a release creation.

Azure ML Pipeline Alignment: If the main use is within Azure ML pipelines, consider creating an **Azure ML component** (which is basically a packaged piece of code) that uses the wrapper. This could be a custom component, e.g., `agent_ask_question` that takes parameters like `question` and outputs an `answer`. In CI, you can validate this component by running it in a test pipeline (maybe using Azure ML CLI with a dummy workspace). This ensures that when integrated, everything is set (the environment has the required packages, etc.).

DevOps Consideration: The wrapper should be installable via pip in the Azure ML environment. You might define it in the `conda.yml` of your pipeline step or use `pip install git+https://...` if not in PyPI. For CI, perhaps build a Docker image for the Azure ML component that includes the wrapper – then the pipeline just uses that image. This can be done in a CI step using Azure ML CLI or the Python SDK.

Testing in CI with Azure ML (optional advanced): Write a test that submits a real Azure ML pipeline (with perhaps a small model or a dummy endpoint) to ensure end-to-end flow. This might be too heavy for each CI run, but maybe as a nightly build.

Recorded Data in CI: Ensure to include the recorded test files in the repo so CI can use them. Verify that no secrets are in them before commit.

Error Handling Verification: In CI, deliberately cause errors to ensure our error messages are helpful. For example, run `sdk_wrapper.run_agent()` without logging in (DefaultAzureCredential not found) – our wrapper should catch and say "Authentication failed, please login or set credentials" instead of an ugly stack trace. We can simulate that with environment isolation.

CI Secrets: If running any live tests or deployment steps, use CI secret stores (e.g., GitHub Secrets or Azure DevOps Library) for things like AZURE_CLIENT_ID, AZURE_TENANT_ID, AZURE_CLIENT_SECRET (for a service principal), or subscription IDs. Never hardcode these. Use them in pipeline YAML as environment variables. Our tests can pick up those env vars if needed.

Example CI Setup (GitHub Actions):

```
name: CI
on: [push, pull_request]
jobs:
  build-test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python: [3.9, 3.10, 3.11]
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v4
        with:
          python-version: ${{ matrix.python }}
      - name: Install deps
        run: pip install -r requirements.txt
      - name: Run lint
        run: flake8 .
      - name: Run tests
        env:
          AZURE_AI_INFERENCE_ENDPOINT: ${{ secrets.TEST_ENDPOINT }}   # if
needed
          AZURE_AI_INFERENCE_API_KEY: ${{ secrets.TEST_API_KEY }}     # if
needed for recording
        run: pytest
```

This runs tests on each Python version. We include env secrets only if certain tests need them (or for re-recording; otherwise the secrets aren't required for playback mode).

Continuous Delivery: Once tests pass and code is merged, you could have another job to package and deploy. If using PyPI, ensure the `setup.py` or equivalent is set up, and use API tokens stored as secrets to upload.

Azure ML Integration in CI: Optionally, maintain a **sample pipeline** in the repo that uses the wrapper. For example, a simple pipeline that calls the agent on a trivial task. In CI, after packaging the code, run that pipeline (with a real Azure ML workspace and compute). This serves as an integration test with the actual environment.

Test Matrix for Azure: The Azure SDK is in preview; test with the latest preview and possibly pinned versions to catch any breaking changes. If MS releases `azure-ai-projects==1.0.0b12`, we'd want to test that too. The CI could parameterize the azure SDK version (though that may be overkill; at least be aware to update and test when new versions come out).

Finally, always include a **human-in-the-loop** before production deployment: perhaps a QA engineer or the developer should review outputs of critical agent flows manually, especially around content filtering (since automated tests might not cover subtle policy edge cases).

Weekend Implementation Plan (Accelerated Schedule)

If we were to implement this unified SDK wrapper over a weekend, here's a breakdown of tasks by day:

Day 1: Saturday

- **Morning:** Project setup. Initialize a Python project (e.g., using Poetry or pipenv) for the SDK wrapper. Set up the repository structure: create modules for core functionalities (e.g., `foundry_client.py` for connecting to Azure Foundry, `agent_wrapper.py` for agent operations, etc.). Install required packages (`azure-ai-projects`, `azure-ai-inference`, `semantic-kernel[azure]`, etc.). Verify you can authenticate and connect to a test Azure AI Foundry project (perhaps using Azure CLI credential locally).

- **Midday:** Implement **Model Inference APIs**. Write functions or classes for chat completion and embeddings:

- `ModelClient` class that wraps ChatCompletions and Embeddings. Implement `complete_chat(prompt, model=None)` that uses `project_client.inference.get_chat_completions_client()` internally [22]. Similarly, `get_embeddings(texts, model=None)`.
- Ensure to handle both sync/async. Possibly use sync for simplicity initially, with the option to add async later. Test these against a known deployment (if available, e.g., a small GPT-3 model or open-source model deployed).

- Also implement an optional direct call to SK's connectors for comparison.

- **Afternoon:** Implement **Agent Creation and Basic Run**:

- Create an `AgentClient` class or similar. Implement `create_agent(model, name, instructions, tools=None)` using `project_client.agents.create_agent` [87]. If tools are provided, ensure they match the expected format (we might accept our own `Tool` dataclass and convert to SDK models like OpenApiTool or CodeInterpreter).
- Implement `start_thread(agent)` that returns a new thread ID (via `threads.create()`).
- Implement `send_message(thread, message, role="user")` using `messages.create()` to add user or agent messages.
- Implement `run_agent(agent, thread)` to call `runs.create_and_process()` [88] and wait for result. This should return the assistant's reply content.

- Test the flow end-to-end manually: call `create_agent`, then `start_thread`, `send_message` with a prompt, then `run_agent` and see if you get a sensible answer. Use a simple model (maybe GPT-3.5) for quick responses. Debug any issues (like mis-set endpoint or missing roles).

- **Evening: Tool Integration (OpenAPI & Code Interpreter)**:

- Implement a way to register a tool. Perhaps create classes: `OpenAPIToolSpec(name, spec_path, auth_type, auth_info)` and `CodeInterpreterToolSpec`.
- For OpenAPI, use `OpenApiTool` model to load spec (as shown earlier) [89]. This might involve reading and using `jsonref` to resolve $refs in the spec. Aim to support at least anonymous and API key auth on Day 1. (Managed Identity auth can be added if time permits).
- For Code Interpreter (which executes Python code and can handle files), see if `azure.ai.agents` has a class or if it's only in .NET. The quickstart shows using `CodeInterpreterToolDefinition()` in .NET [62]. In Python, this might not be exposed. Perhaps we skip implementing CodeInterpreter on Day 1 or attempt a workaround (maybe there's an OpenAPI behind it, or it's not available in Python SDK yet). Mark as a to-do if not straightforward.
- Integrate tools into agent creation: modify `create_agent` to handle a list of tool specs. For OpenAPI tools, if multiple, combine definitions or allow list. For now, handle single OpenAPI spec. Ensure `tools=...` parameter is passed correctly [90].

- Quick test: Try adding a known public OpenAPI (maybe a weather API) and see if agent can call it (this requires the actual API to be accessible and returning data). If not possible to fully test, at least ensure no exceptions in creating agent with tool.

- **Night:** Write **basic documentation** (in README or a docs folder) for what was implemented today – usage examples for chat and agent simple Q&A. This will help tomorrow when expanding features and for eventual users. Also, commit code regularly.

Day 2: Sunday

- **Morning:** Focus on **Advanced Agent Features**:
- Implement multi-agent orchestration. Possibly create a `MultiAgentManager` class that can take multiple agent IDs or definitions and coordinate a conversation. Simplest approach: round-robin agent responses. Advanced: integrate SK's AgentGroupChat.
    - Given time constraints, perhaps implement a basic loop: alternate agents responding to each other until a condition or max turns. Use our own `send_message` and `run_agent` in alternation: send user message to Agent1's thread, run Agent1 to get answer, take answer text and send as *user* message to Agent2's thread, run Agent2, and so on. However, the Azure service might not treat an agent's answer as a user message for another agent out-of-the-box (lack of a role for agent message). To simplify, you might just string their outputs logically.
    - Alternatively, use SK's AgentGroupChat if possible. Check if `semantic_kernel.agents.AzureAIAgent` and `AgentGroupChat` can be used easily. If SK's integration is not yet well-documented in Python, use pseudo-orchestration as above.

- If time permits, incorporate SK Planner: e.g., an option like `plan_and_execute(goal)` that uses SK to break a goal into steps and calls either our agent or functions. This might be too ambitious for one day, so likely skip or leave stub.

- **Midday: Testing and Hardening**:

- Write tests (as discussed) for the core parts. Create dummy objects for ProjectClient if needed. Possibly use Azure's recorded tests technique: record a simple agent Q&A and a simple embedding call. Ensure tests cover success and failure scenarios.
- Fix any bugs uncovered by tests. For example, ensure context managers (`with ... as client:`) are handled or replaced if causing issues. Possibly refactor long methods for clarity.

- Check thread-safety: If our wrapper holds a global `AIProjectClient`, is it safe to reuse across threads? Azure clients are generally thread-safe for read operations. If not certain, one could create clients per call or use a pooling. Given a pipeline typically runs single-threaded per step, not critical, but note for future.

- **Afternoon: CI/CD Setup and Documentation**:

- Create a GitHub Actions workflow (or Azure DevOps pipeline) as described, to automatically run tests. This might just involve writing the YAML, since actual cloud resources may not be hit thanks to recordings.
- Write comprehensive **documentation** in Markdown: how to install (pip install requirements, plus any extras like `semantic-kernel[azure]`), how to configure environment variables, code examples for each major feature. Much of this content is what we have in this report (we can repurpose it). Ensure to highlight enterprise use cases (like "to use this in Azure ML, do X").
- Provide **dependency versions** in docs: e.g., "Requires azure-ai-projects >= 1.0.0b11, semantic-kernel >= 0.3.**".

- Ensure the README covers at least one example end-to-end (maybe a small snippet where an agent with an OpenAPI tool is created and used).

- **Evening: Final Tasks and Best Practices**:

- Implement any remaining important feature that was missing: e.g., error translation (wrap Azure exceptions in our custom exceptions for clarity), or content safety hook (e.g., if we want to integrate an explicit moderation call).
- Review best practices and pitfalls and ensure our code either handles them or clearly documents them. For instance, if the Azure SDK call is async but we used sync, did we ensure to use the sync client (`azure.ai.projects` vs `azure.ai.projects.aio`)? If using async anywhere, did we properly manage the event loop? Possibly decide to keep everything synchronous for now (so that Azure ML pipeline integration is straightforward), which is fine since the Azure SDK offers sync methods. SK usage might force async; if so, we can provide a sync wrapper around it (like `asyncio.run(SKAgentGroupChat.invoke())` inside).
- Test one more time critical paths (manually if possible) – maybe run a quick script that calls multiple features.
- **Wrap up:** commit all changes, push, ensure CI passes. If doing a release, tag the repo, build distribution (wheel) and upload to internal feed or PyPI.

This aggressive plan assumes familiarity with Azure SDK and SK – but given the preparation (this report), a developer should be able to implement core functionality within the timeframe. The key is to prioritize: basic agent and model calls first (those are essential), tools and multi-agent next (value-add features), and make sure testing and documentation aren't forgotten.

Pitfalls and Best Practices

Finally, let's summarize some common pitfalls and best practices when developing and using this SDK in enterprise scenarios:

- **Async vs Sync:** Many Azure and SK operations are async. If mixing them, be careful to avoid blocking the event loop or creating nested event loops. Use the sync clients for Azure operations in synchronous code (e.g., `AIProjectClient` vs `AIProjectClient.aio`). If you need to use SK's async functions in a sync context (like in an Azure ML pipeline which is typically sync), wrap them with `asyncio.run()` or provide sync alternatives. Best practice: **keep high-level pipeline code synchronous** for simplicity, and confine async usage inside the wrapper or use separate async methods that advanced users can call.

- **Thread and Conversation Management:** Each agent thread holds conversation state. If you mistakenly use a single thread for multiple unrelated queries (especially with different users or tasks), the agent might confuse contexts. Best practice is to **start a fresh thread for each independent conversation** (unless you intentionally want it to have memory). Our wrapper should make it easy to get a new thread and perhaps automatically handle thread creation in `run_agent` if none provided. Conversely, if doing multi-turn with the same agent, reuse the thread to leverage context (and call `runs.create_and_process` for each turn). Also remember to **reset or create a new agent** if you change its instructions or tools significantly; you can't change an agent's instructions on the fly without creating a new one (the agent config is static after creation).

- **Resource Cleanup:** Agents created via `create_agent` persist in the Azure project. If you create many during tests or dynamic scenarios, you might clutter the project or hit limits. It's a good practice to delete agents that are no longer needed using `project_client.agents.delete_agent(agent.id)` [91]. Our wrapper could offer a cleanup method or context manager to create an agent temporarily. Similarly, threads and messages are ephemeral (they don't necessarily count against quotas heavily), but if there's a concept of thread cleanup, do that as needed (not critical, as threads likely just expire or can be reused).

- **Naming Conventions:** Name your agents and tools clearly and uniquely. For tools, avoid names that could overlap with model's own abilities or other tools. For example, naming an OpenAPI tool function `search` could conflict with a Bing Search tool. A more specific name like `CustomSearchAPI` is better. Azure might require tool names to be lowercase and no spaces (check if any restrictions). Similarly, use descriptive agent names – while not used in conversation, it helps manage them in the portal.

- **Tool Conflicts and "Bleed":** As seen in some SK discussions, if two agents or two tools share some state or names, they might interfere. SK's AgentGroupChat issue "tool-bleed" suggests one agent might inadvertently access another's function if not isolated [92]. Ensure that when orchestrating multi-agents, each agent has its own set of tools and SK functions. The Azure Agent Service keeps tools per agent ID, so that should be isolated. But if using SK's Kernel with multiple agents, don't register all plugins globally without scoping – perhaps instantiate separate Kernel or use skill naming to isolate.

- **Rate Limits and Retries:** We mentioned this in security but as a best practice: implement retry logic for transient errors (429, 503, etc.). Azure SDK often has built-in retries for idempotent calls, but an agent run might not retry if the error occurs mid-run. If `create_and_process` fails due to a transient issue (network glitch or function timeout), you might consider a second attempt. However, caution: if the agent had side effects (e.g., partially executed a tool), retrying might do it twice. In most Q&A scenarios it's fine. Maybe expose a parameter for max retries.

- **Parallelism:** If using the wrapper in a multi-threaded app or scaling out in Azure ML (like using `ParallelRunStep` or deployment), ensure that each thread or process uses its own `AIProjectClient` or at least its own agent threads. The `AIProjectClient` is threadsafe for reads but not sure about writes – to be safe, instantiate one per thread or use a lock around agent creation calls if multi-threading. In Azure ML, typically each run is isolated so not an issue.

- **Logging and Data Privacy:** By default, Azure SDK logs are sanitized [86]. If you turn on full logging (for debugging in non-prod), remember to turn it off in production to avoid logging sensitive content. Also, consider logging conversation metadata (timestamps, user ID, question, whether answered, etc.) without logging the full content in production, depending on privacy requirements. For instance, you might log that an agent was used and a tool was called, but not store the exact question if it contains PII.

- **Using the Latest SDKs:** Azure AI Foundry is evolving rapidly (as seen with b11 release changes). Keep an eye on the changelogs [93]. Best practice: pin versions for stability in your requirements (`azure-ai-projects==1.0.0b11`) for your production, and test new beta updates in a staging environment. There might be breaking changes in preview versions.

- **Fallback Strategies:** If the agent fails or tool fails, have a graceful fallback. E.g., if the OpenAPI tool fails to fetch weather, the agent might respond "Sorry, I cannot get the weather now." as part of its learned behavior. You can also intercept failures (like `run.status == failed`) [42] and choose to either retry or return a friendly error. Communicate to users appropriately rather than dumping raw errors.

- **Semantic Kernel Best Practices:** When using SK, ensure to **initialize the SK Kernel only once** if possible (it's somewhat heavy to load). Also, load plugins outside of tight loops. If using SK's planner, note it uses an LLM call itself to create plans, which adds latency. Only use planning if the task truly requires dynamic sequencing of multiple steps; for simpler tool selection, the Azure agent's built-in approach is usually enough. SK's advantage would be in orchestrating across multiple independent skills or when you need a different style of coordination than the agent's.

- **Memory and State:** Be cautious about the size of conversation history. The Foundry thread will truncate automatically older messages to stay within context window [70] [94], but if you keep a copy in your app, that copy might grow. If we store conversation transcripts, consider size limits. Also, vector memory (embeddings) can grow if you embed many docs – use a proper store (like cognitive search or a database) rather than storing huge lists in memory.

- **Tool Limits:** The agent may have limits on number of tools or the complexity of OpenAPI specs it can handle (prompt length considerations). If you cram too many tools, the prompt given to the model becomes large (each tool is described in the system prompt). If an agent has, say, 10 tools, the

model might get confused or run out of tokens for instructions. It might be better to have specialized agents each with a smaller set of relevant tools, and then use a multi-agent approach where one agent delegates to another (rather than one monolithic agent). We should mention this as guidance: *design agents with focused responsibilities*. Our wrapper can help by making it easy to spin up multiple agents if needed, rather than trying to give one agent everything.

- **Testing in Production:** Even after all this, AI systems can be unpredictable. It's best practice to monitor outputs especially after deploying to production. Use the observability features (trace logs, App Insights events) to sample conversations and ensure quality. Incorporate user feedback loops if possible (let users flag incorrect answers, etc., and retrain or adjust prompts accordingly). The Foundry evaluation feature (which can automatically evaluate responses on criteria like relevance [95] [96]) might be something to integrate later for continuous improvement.

By being mindful of these points, developers can avoid common pitfalls and build reliable, scalable applications on top of our unified Azure AI Foundry & Semantic Kernel SDK wrapper.

Conclusion

This comprehensive report outlined how a Python SDK wrapper can unify Azure AI Foundry's powerful agent and model services with Semantic Kernel's orchestration capabilities. By following the structured approach above – leveraging official Azure SDKs for deployment, inference, and agent tooling [97] [98], and integrating SK for advanced multi-agent scenarios – enterprise developers can rapidly build AI solutions that are **grounded, extensible, and enterprise-ready**. The weekend implementation plan provides a roadmap to get a functional version quickly, and the testing/CI guidance ensures the solution remains robust in the long run. By adhering to best practices around security, async usage, and tool design, the SDK will serve as a reliable foundation for AI-powered applications integrated into Azure ML pipelines and beyond.

With Azure managing the heavy lifting of scale, compliance, and runtime orchestration [33] [74] and Semantic Kernel adding a flexible developer experience for plugins and multi-agent coordination, this unified SDK will empower developers to focus on solving business problems with AI, rather than wrangling disparate tools. Let's build responsibly and innovatively, bridging Azure's enterprise AI with the creativity of Semantic Kernel to usher in the next generation of intelligent apps.

Sources:

- Azure AI Projects & Agents SDK Documentation [3] [35]
- Azure AI Foundry & Agent Service Overviews [32] [64]
- Semantic Kernel Documentation (Azure Integration, Plugins, Multi-agents) [26] [67]
- Microsoft and Community Tutorials/Blogs [37] [90]

---

[1] [2] What is Azure AI Foundry? - Azure AI Foundry | Microsoft Learn
https://learn.microsoft.com/en-us/azure/ai-foundry/what-is-azure-ai-foundry

3 4 5 6 14 16 17 19 22 23 24 25 28 43 57 58 59 60 85 86 93 95 96 97 98 Azure AI Projects client library for Python | Microsoft Learn
https://learn.microsoft.com/en-us/python/api/overview/azure/ai-projects-readme?view=azure-python-preview

7 8 9 10 15 20 21 31 Azure AI Inference client library for Python | Microsoft Learn
https://learn.microsoft.com/en-us/python/api/overview/azure/ai-inference-readme?view=azure-python-preview

11 26 27 29 30 Develop applications with Semantic Kernel and Azure AI Foundry - Azure AI Foundry | Microsoft Learn
https://learn.microsoft.com/en-us/azure/ai-foundry/how-to/develop/semantic-kernel

12 38 Exploring the Semantic Kernel Azure AI Agent Agent | Microsoft Learn
https://learn.microsoft.com/en-us/semantic-kernel/frameworks/agent/agent-types/azure-ai-agent

13 55 65 Build a Multi-Agent System Using Microsoft Azure AI Agent Service and Semantic Kernel in 3 Simple Steps! | by Akshay Kokane | Data Science Collective | Medium
https://medium.com/data-science-collective/create-multi-agent-system-with-microsofts-azure-ai-agent-service-and-semantic-kernel-framework-in-a6c68b123e54

18 34 35 36 41 62 87 88 91 Quickstart - Create a new Azure AI Foundry Agent Service project - Azure AI Foundry | Microsoft Learn
https://learn.microsoft.com/en-us/azure/ai-services/agents/quickstart

32 33 64 66 74 What is Azure AI Foundry Agent Service? - Azure AI Foundry | Microsoft Learn
https://learn.microsoft.com/en-us/azure/ai-services/agents/overview

37 61 72 73 79 What is Azure AI Agent Service?. Azure AI Agent Service is a stateful... | by Luis Valencia | GoPenAI
https://blog.gopenai.com/what-is-azure-ai-agent-service-ae8ec52fe901?gi=3657c1c0c6f5

39 40 42 44 45 46 47 48 63 89 90 OpenAPI spec code samples - Azure AI Foundry | Microsoft Learn
https://learn.microsoft.com/en-us/azure/ai-services/agents/how-to/tools/openapi-spec-samples

49 50 51 52 53 54 56 Give agents access to OpenAPI APIs | Microsoft Learn
https://learn.microsoft.com/en-us/semantic-kernel/concepts/plugins/adding-openapi-plugins

67 68 69 70 71 83 84 94 How-To Coordinate Agent Collaboration using Agent Group Chat (Experimental) | Microsoft Learn
https://learn.microsoft.com/en-us/semantic-kernel/support/archive/agent-chat-example

75 76 77 78 80 81 82 azure.ai.projects package | Microsoft Learn
https://learn.microsoft.com/en-us/python/api/azure-ai-projects/azure.ai.projects?view=azure-python-preview

92 [python] AgentGroupChat tool-bleed between agents #10297 - GitHub
https://github.com/microsoft/semantic-kernel/discussions/10297