Integrating AI Services into Semantic Kernel: A Case Study on Enhancing Functionality with Google PaLM and Large Language Models

Alisha Maddy

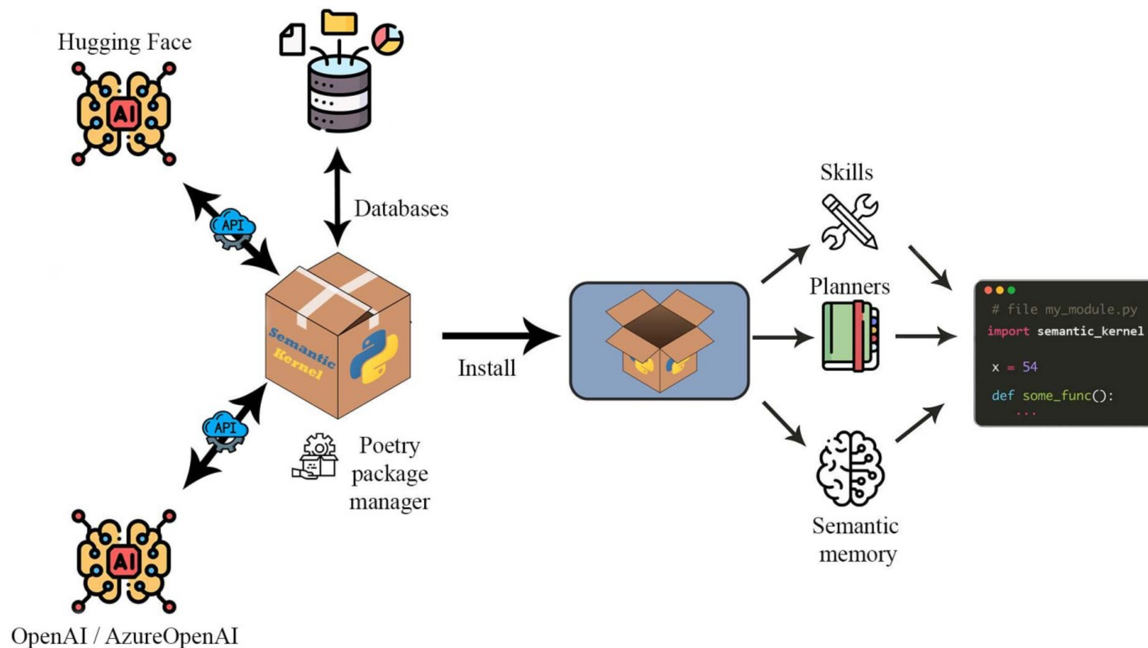

Figure 1: System Overview

Background

Microsoft's Semantic Kernel is a software development kit (SDK) designed for C# and Python developers to integrate AI large language models (LLM) such as Chat-GPT into their existing applications. The Semantic Kernel leverages AI services like OpenAI, Azure OpenAI, and Hugging Face via their respective APIs. This SDK simplifies the utilization of AI technology for developers by enabling the orchestration of AI plugins.

The Semantic Kernel offers a suite of connectors that facilitate the incorporation of memories and models into AI-powered applications. Memories provide the LLM with context and specific knowledge, enhancing its ability to answer questions and engage in meaningful conversations. A model refers to a specific instance of an LLM, such as GPT-3. Additionally, the Semantic Kernel streamlines the addition of skills to applications through AI plugins, which consist of prompts and native functions that respond to triggers and execute actions. Furthermore, planners can be used to allow the AI to autonomously select appropriate skills when fulfilling a user's request.

System Overview

To provide a comprehensive understanding of the project's structure and technology stack, a high-level system diagram is presented. Semantic Kernel is available in two versions: C# and Python. This case study focuses on the Python version,

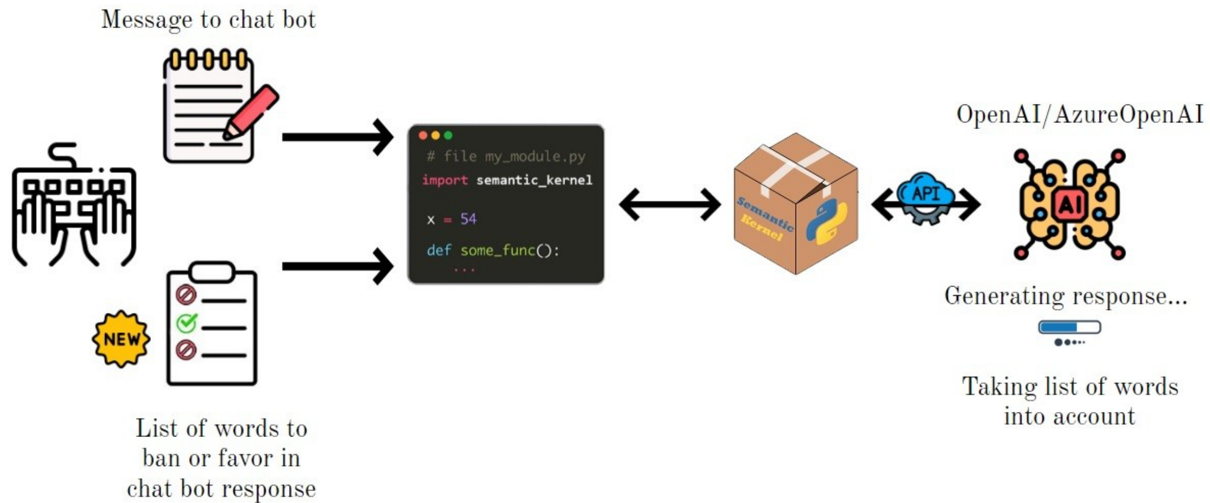

Figure 2: The user sends a message to the chat bot along with a dictionary of words to filter out or favor.

which is managed as a Python package using Poetry.

Within the kernel, direct interactions with the AI service APIs and databases are abstracted from the user. Upon installing Semantic Kernel, various methods and objects can be imported into the code, categorized under skills/plugins, planners, and semantic memory. After instantiating an AI service class, the capabilities of AI can be harnessed through Semantic Kernel orchestration, with data for semantic memory being effortlessly stored in databases. (Figure 1)

A Good First Issue

To become acquainted with the Semantic Kernel project, the team selected a 'good first issue' to address collaboratively. This issue involved modifying the API calls to include a parameter known as logit-bias, which allows developers to influence the probability of certain words appearing in responses from OpenAI models. Logit-bias is a dictionary that maps words to integer values, with -100 completely banning a word and 100 exclusively favoring a word, with varying degrees in between.

Since logit-bias is already implemented by OpenAI, the primary task was to create an example file demonstrating its functionality. Due to unfamiliarity with the Semantic Kernel's functions, the exploration of the codebase was necessary whenever errors were encountered. This exploration provided valuable insights into the project's structure. The team's pull request (PR) was successfully merged, marking a significant milestone in their engagement with the project. (Figure 2)

Opening Our Own Issue

The team's mentor encouraged them to challenge themselves by working on issues individually. He suggested that each member come up with an idea for a new connector to add to the project. This guidance and encouragement proved invaluable, as it prompted taking on a more ambitious task within the project. Initially unfamiliar with what a

connector was, this marked the beginning of a more in-depth exploration.

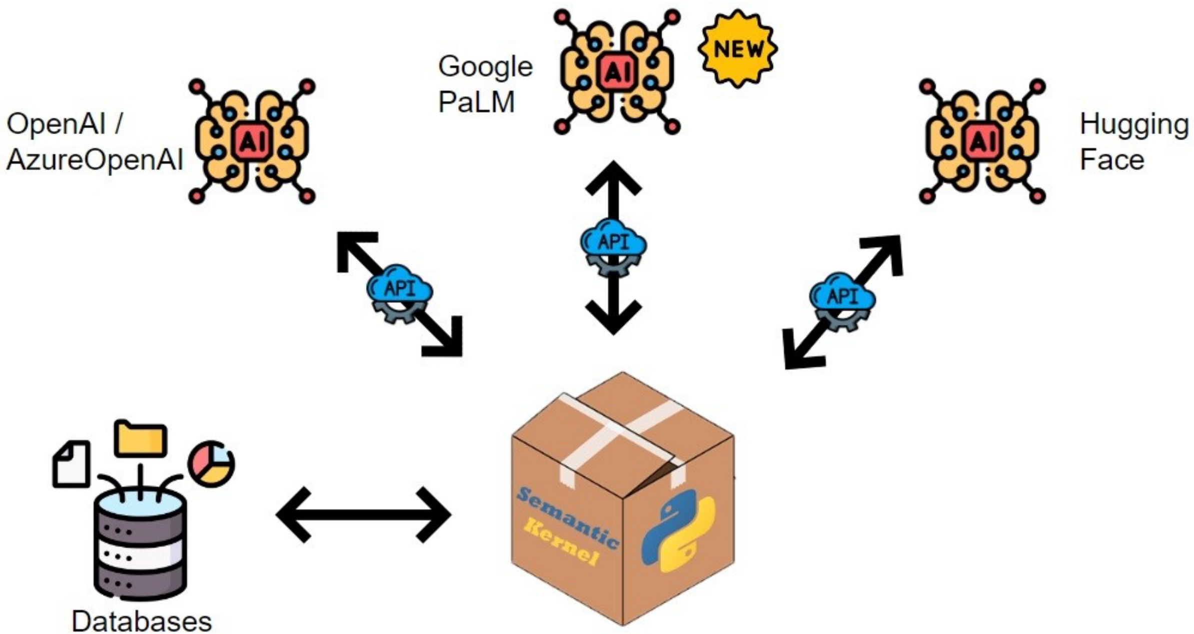

Figure 3: The Semantic Kernel and its connections to LLMs and databases, including the new PaLM connector.

Finding a Starting Point

The exploration of the codebase commenced to locate where the connectors were implemented. By examining the existing implementations, it was determined that a connector is a class designed to manage interactions with APIs and databases. With this foundational understanding, the search for gaps in the project began. It was observed that the project had a limited number of connectors for large language models, specifically only three: one for OpenAI, one for Hugging Face, and another for Azure OpenAI, which itself does not have proprietary LLMs and merely serves as an API service for OpenAI.

Subsequently, an investigation was undertaken to identify additional AI services that could be integrated. This task was challenging due to the prevalent waitlists for API keys. After joining the waitlist for a Google PaLM API key, fortune smiled, and an API key was received the following day. The next step involved thoroughly studying the PaLM documentation to ensure compatibility with the project. Confident in its feasibility, an issue was opened to add a Google PaLM connector to the Semantic Kernel project. (Figure 3)

Integration of PaLM's Capabilities

To integrate Google PaLM's text completion, chat completion, and text embedding functionalities into the Semantic Kernel, a structured approach was adopted. The text completion feature leverages the text-bison-001 model, which generates a completion based on a provided text prompt. This model can handle various language tasks, including translation, text summarization, and text generation. The initial focus was on integrating text completion.

Implementing a Solution

A new class, GooglePalmTextCompletion, was developed to handle the text completion functionality. This class manages user requests, communicates with the API, and returns responses to users. The similarity between Google's generative AI library functions for text completion and OpenAI's functions facilitated a consistent integration process. Significant effort was dedicated to understanding the codebase comprehensively, examining all relevant directories and files in detail.

To demonstrate the usage of the new class, an example file was created, serving as a tool for manual debugging. Following successful manual testing, six new integration tests were implemented to ensure compatibility with core kernel functions. Additionally, three unit tests were developed to verify the successful initialization of the class and the accuracy of API calls.

Testing and Simulating API Calls

While integration and unit testing did not reveal any bugs, testing the API calls posed a challenge due to the need to avoid consuming actual resources. Therefore, asynchronous API calls were simulated using Python's `MagicMock` patching. This approach involved several steps:

1. **Creating an Asynchronous Future Object**: An `asyncio.Future` object was created, with its result set to the string "Example" to simulate a successful API response.
2. **Creating a MagicMock Response**: A `MagicMock` object was generated to represent the API call response.
3. **Assigning the Future Result**: The `MagicMock` response's `result` attribute was set to the `asyncio.Future` object, simulating an asynchronous response.
4. **Mocking the GooglePalmTextCompletion Class**: Another `MagicMock` object was created to represent the `GooglePalmTextCompletion` class, with its return value set to the mock response.
5. **Patching the Class**: The actual class was temporarily replaced with the `MagicMock` object using patching.
6. **Conducting the Test**: Functions in the patched class were called, returning the mock response object with the future result "Example".

Following the successful completion of these steps, a pull request was submitted. Subsequently, attention shifted to integrating chat completion and text embedding capabilities, with a new issue being opened to address these features.

Chat With Google's Large Language Model

The chat completion functionality utilizes the chat-bison-001 model, which is more akin to the Chat-GPT models that OpenAI is renowned for. This model engages users in a conversational manner, maintaining the context of the conversation history. Integrating Google PaLM's chat completion capabilities into the Semantic Kernel posed greater challenges compared to text completion. Significant differences exist between the structures of chat functions in Google's generative AI package and the OpenAI package, necessitating subjective decisions to ensure a consistent user experience across different AI services.

Challenges and Solutions

OpenAI employs "system messages" to provide the chatbot with conversational context and to prime it with specific behaviors or knowledge. These messages are part of a list containing the entire chat history, which is passed as a parameter in each API call for chatting. Users can add system messages to give the bot skills or context. Conversely, PaLM uses a parameter called context, which is a string separate from the message parameter. To maintain a similar user experience, the conversion of system messages to the context parameter was abstracted away from the user.

Additionally, OpenAI allows constructing a conversation history that never occurred and passing it to the API call, enabling Chat-GPT to use it for context. PaLM, however, does not support passing chat history to its functions; only the current message can be given, with a non-mutable chat history stored in the response object. To accommodate this, the GooglePalmChatCompletion class was designed to concatenate the entire chat history into a string and pass it as the context parameter.

A comprehensive understanding of Semantic Kernel's offerings was necessary to deliver this feature, ensuring seamless integration with the project's workflow. Three example files were created to demonstrate the usage of the new classes and facilitate manual debugging: one for normal chatting, another for chatting with skills and system messages, and a third for chatting with memory, leading to the text embedding feature.

Text Embedding

Text embedding is a capability provided by PaLM and other AI services, utilized by Semantic Kernel for semantic memory. It processes words and phrases into a list of integers representing their semantic meaning, which can be used to measure text relatedness. Users can embed information they want to provide to the chatbot, store the embeddings in a database, and query the database to build prompts. For instance, a question or any text string can be embedded to find a related string in the database, enabling the chatbot to provide context-specific responses based on stored embeddings.

A class was created to handle sending and receiving data from PaLM's API for text embedding. An example was developed to build a prompt filled with embeddings stored in memory, which is then sent to the bot for context. This allows the bot to access personal details about a hypothetical user that it otherwise would not know. Additional integration and unit tests were implemented, similar to those for text completion. The final pull request for the Google PaLM connector was submitted after thorough testing of chat completion and text embedding.

Chat with CSV Files and Pandas Dataframes

In addition to completing a major feature, collaboration with teammate Sneha led to the development of a feature allowing users to query structured data sources with natural language. This task, unprecedented before the advent of large language models, is challenging due to the data size limitations. For example, Chat-GPT has a text length limit of around 500 words per message, making it impractical to query a dataset with thousands of entries directly. Implementing this feature involved innovative techniques to enable efficient querying of large datasets.

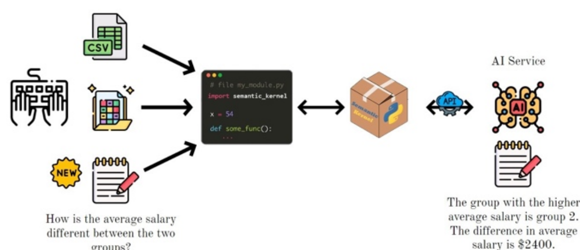

Figure 4: An example of a user asking the LLM about their data using a CSV file and Pandas dataframe source.

Addressing the Issue of Model Training with Structured Data

Upon discovering an issue requesting the ability to 'train' models using a structured data source, an in-depth investigation into fine-tuning OpenAI models was initiated. However, after reviewing OpenAI's documentation and engaging in discussions on the Semantic Kernel (SK) Discord, it became evident that fine-tuning was not recommended for most use cases. Clarifying the issue's requirements necessitated considerable brainstorming. References to Langchain's documentation clarified the desired end result, though the process remained ambiguous. Despite the complexity of Langchain's codebase, a critical insight was gained from its source code: the existence of a tool for executing LLM-generated Python code. Although the structural differences between Langchain and Semantic Kernel prevented direct implementation parallels, this conceptual understanding was invaluable.

Implementing a Solution

The solution began with the development of two new skills: `data_skill` and `code_skill`. The `code_skill` class generates Python code by providing instructions to the LLM, parses the code from the response, and executes it. Although `code_skill` can operate independently, the `data_skill` class relies on it.

The primary focus was on the `data_skill` class, which manages data retrieval and user requests, returning the processed answer. This class converts all data into pandas dataframes and constructs a prompt containing the first three rows of each dataframe, along with instructions for generating Python code and the user's request. The prompt is then sent to the `code_skill`, where the LLM-generated code is executed on the user's data. Finally, the result is formatted in a natural language response by the LLM and returned to the user.

Efficiency Considerations

Executing code locally on the user's system, rather than sending all data to the LLM, significantly enhances efficiency. Each API request consumes tokens, which incur costs and are subject to limits on the number of tokens that can be used per request. This local execution approach mitigates these limitations, optimizing both cost and performance.

Consider the following datasets:

```
data1 = {
        "Name": ["Alice", "Bob", "Charlie",
"David", "Eve"],

        "Age": [25, 32, 28, 22, 29],

        "City": ["New York", "Los Angeles",
"Chicago", "Houston", "Miami"],

        "Salary": [60000, 75000, 52000,
48000, 67000],
    }

    data2 = {
        "Name": ["Amanda", "Brian",
"Catherine", "Daniel", "Emily", "Francis"],

        "Age": [27, 35, 31, 24, 30, 33],

        "City": ["San Francisco", "Seattle",
"Boston", "Austin", "Denver", "Savannah"],

        "Salary": [62000, 80000, 55000,
50000, 67000, 70000],
    }
    df1 = pd.DataFrame(data1)
    df2 = pd.DataFrame(data2)
```

To convert them to dataframes and then send them to the data_skill, a user could ask questions such as:

- User: How old is Bob and what city does Francis live in?
- Output: Bob is 32 years old and Francis lives in Savannah.

- User: Which group has a higher average salary and what is the difference?
- Output: The group with the higher average salary is df2. The difference in average salary between the two groups is $3600.

- User: Explain the correlation between age and income with two decimal places.
- Output: The correlation between age and income is .83, which indicates a strong positive relationship. This means that as age increases, income tends to increase as well.

The possibilities are endless. I prototyped functions for being able to transform data and generate plots as well, which we will continue to work on. I submitted a draft PR to get feedback on what we've done so far.

The Endless Possibilities

The integration of these new capabilities opens up numerous possibilities. Prototyping functions for data transformation and plot generation has been initiated, paving the way for further enhancements. The draft pull request (PR) was submitted to solicit feedback on the progress made thus far.

Prototyping Advanced Functions

In addition to basic data handling and query execution, functions for data transformation and visualization were prototyped. These advanced features enable users to manipulate data and generate various plots, significantly expanding the range of analytical tasks that can be performed. The ongoing development of these functions aims to provide robust and versatile tools for data analysis within the Semantic Kernel framework.

Seeking Feedback

The submission of a draft PR marks an important step in the development process, inviting feedback from the project maintainers and the broader community. This feedback will be instrumental in refining the implementation and ensuring it aligns with the project's goals and standards. The collaborative nature of this process underscores the commitment to delivering a high-quality, user-friendly solution.

Future Work

The work on transforming data and generating plots is set to continue, with the potential to introduce even more sophisticated data analysis capabilities. The integration of these features will not only enhance the functionality of the Semantic Kernel but also empower users to perform complex data-driven tasks efficiently.