Semantic Kernel: A bridge between large language models and your code
At first glance, building a large language model (LLM) like GPT-4 into your code might seem simple. The API is a single REST call, taking in text and returning a response based on the input. But in practice things get much more complicated than that. The API is perhaps better thought of as a domain boundary, where you're delivering prompts that define the format the model uses to deliver its output. But that's a critical point: LLMs can be as simple or as complex as you want them to be.

When we integrate an AI model into our code, we're crossing a boundary between two different ways of computing, like the way in which programming a quantum computer is much like designing hardware. Here we're writing descriptions of how a model is intended to behave, expecting its output to be in the format defined by our prompts. As in quantum computing, where constructs like the Q# language provide a bridge between conventional computing and quantum computers, we need tools to wrap LLM APIs and provide ways to manage inputs and outputs, ensuring that the models remain focused on the prompts we've defined and that outputs remain relevant.

This is an important point to stress: How we interact with a LLM is very different from traditional programming. What's needed is a Q# equivalent, a higher-level abstraction that translates between the different domains, helping us take data and use it to craft prompts, while providing a way to manage the essential context between calls that avoid exhausting the available tokens in a conversation while still keeping outputs grounded in source data.

[Also on InfoWorld: How to choose a cloud machine learning platform]

Introducing Semantic Kernel

A few weeks ago, I looked at Microsoft's first LLM wrapper, the open-source Prompt Engine. Since then, Microsoft has released a larger and more powerful C# tool for working with Azure OpenAI (and with OpenAI's own APIs), Semantic Kernel. It too is open source and available on GitHub, along with a selection of sample applications to help you get started. A Python version of Semantic Kernel is under development as well.

The choice of name is intriguing, as it shows a better understanding of what a LLM is used for. Whereas Prompt Engine was about managing inputs to APIs, Semantic Kernel has a wider remit, focusing on natural language inputs and outputs. Microsoft describes its approach as "goal oriented," using the initial request from a user (the "ask") to direct the model, orchestrating passes through the resources associated with the model to fulfill the request, and returning a response to the request (the "get").

So calling Semantic Kernel a kernel makes sense. It's like an operating system for the LLM API, taking inputs, processing them by working with the model, and returning the output. The kernel's role as an orchestrator is key here, as it's able to work not just with the current prompt and its associated tokens but also with memories (key-value pairs, local storage, and vector or "semantic" search), with connectors to other information services, and with predefined skills that mix prompts and conventional code (think LLM functions). The Semantic Kernel tooling provides more effective ways of building and using the types of constructs you needed to wrap around Prompt Engine, simplifying what could become quite complex programming tasks, especially when it comes to handling context and supporting actions that include multiple calls of a LLM API.

Vectors and semantic memory

A key element of processing user asks is the concept of memories. This is how Semantic Kernel manages context, working with familiar files and key-value storage. However, there's a third option, semantic memory. This approach is close to the way that a LLM processes data, treating content as vectors, or embeddings, which are arrays of numbers the LLM uses to represent the meanings of a text. Similar texts will have similar vectors in the overall space associated with your model and its content, much like the way a search engine generates ranked results.

A LLM like GPT uses these embedded vectors to extract the context of a prompt, helping the underlying model maintain relevance and coherence. The better the embedding, the less likely a model is to generate purely random output. By breaking up large prompts into blocks of text that can be summarized by a LLM, we can generate an embedding vector for each summary, and then use these to create complex prompts without exhausting the available tokens for a request (GPT-4 has a limit of 8,192 tokens per input, for example).

This data can be stored in a vector database for fast retrieval. Specific vector databases can be created for specialized knowledge, using summarized content to help keep the LLM on track. So, for example, an application that uses GPT-4 for medical case note summaries, could use a vector database of embeddings from medical papers, suitable anonymized notes, and other relevant texts, to ensure that its output is coherent and in context. This approach goes some way toward explaining why Microsoft's first big GPT-based application is its Bing search engine, as it already has the appropriate vector database ready for use.

Connecting to external data sources

Connectors are an interesting feature of the Semantic Kernel, as they’re a way of integrating existing APIs with LLMs. For example, you can use a Microsoft Graph connector to automatically send the output of a request in an email, or to build a description of relationships in your organization chart. There’s no set point in a Semantic Kernel application to make the call; it can be part of an input, or an output, or even part of a sequence of calls to and from the LLM. You can build prompts from API calls that themselves build further API calls, perhaps by using a Codex code-based model to inject the resulting output into a runtime.

One of the more interesting features of a connector is that it applies some form of role-based access control to an LLM. If you’re using, say, Microsoft Graph queries to construct a prompt, these will be in the context of the user running the application, using their credentials to provide data. Passing credentials to a connector ensures that outputs will be tailored to the user, based on their own data.

Building skills to mix prompt templates and code

The third main component of Semantic Kernel is skills, which are containers of functions that mix LLM prompts and conventional code. These functions are similar in concept and operation to Azure Functions and can be used to chain together specialized prompts. An application could have one set of functions that generates text using GPT, then use that text as a prompt for Codex and DALL-E to go from a description to a prototype web application (similar to how the natural language programming tools work in Microsoft’s low-code and no-code Power Platform).

Once you have your skills, memories, and connectors in place, you can start to build an LLM-powered application, using skills to turn a request into prompts that are delivered to the underlying models. This approach lets you build flexible skills that your code can select and use as required. Semantic Kernel distinguishes between semantic functions, templated prompts, and native functions, i.e. the native computer code that processes data for use in the LLM’s semantic functions. Outputs of one function can be chained to another function, allowing you to build a pipeline of functions that mix native processing and LLM operations.

From just this brief look at Semantic Kernel, it’s clear that this is a powerful tool, but one that needs careful thought and planning. You can use Semantic Kernel to build and manage complex prompts and the pipelines that work with the chains of inputs and outputs to deliver interesting and useful results. Naturally, you’ll get the best results when you use each element appropriately, with native code to handle calculations and models to focus on directed goals (or

as the documentation calls them, in very Microsoft fashion, “the asks”).

Using a tool like Semantic Kernel to marshal and orchestrate inputs and functions will certainly make working with LLMs a lot more effective than simply passing a prompt to an input. It will allow you to sanitize inputs, guiding the LLM to produce useful outputs. To help you get started, Microsoft provides a list of best practice guidelines (the Shillace Laws of Semantic AI) learned from building LLM applications across its business. They’re a useful primer into how to build code around LLMs like GPT, and should help you get as much value as possible from these new tools and techniques, while steering clear of unrealistic expectations.