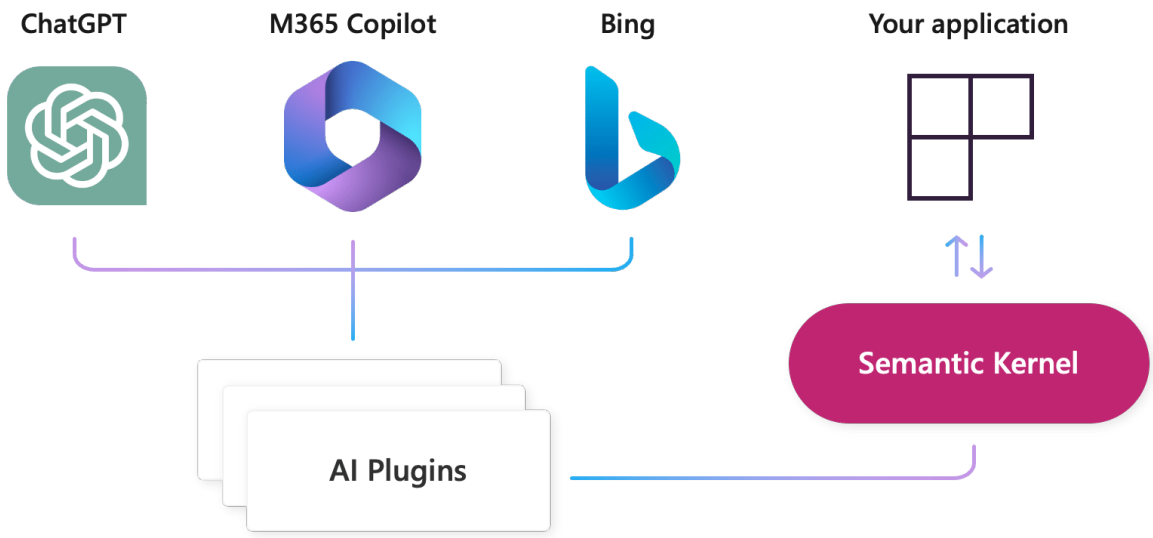
# The skills of LLM - Plugins

We have learned the foundation of Semantic Kernel. One of the major features of Semantic Kernel is its powerful plugins, which solve intelligent business problems by combining custom/predefined plugins. Let traditional code and smart plugins work together to flexibly connect to application scenarios to simplify the process of transforming traditional applications into intelligent ones. In this chapter, we mainly introduce how to use plugins.

## What's Plugins




We know that the original data of LLMs is time-limited, and if you want to add real-time content or enterprise knowledge, there are considerable shortcomings. OpenAI connects ChatGPT to third-party applications via plugins. These plugins enable ChatGPT to interact with developer-defined APIs, thereby enhancing ChatGPT's functionality and allowing a wider range of operations, such as:

1. Retrieve real-time information, such as sports scores, stock prices, latest news, etc.
2. Retrieve knowledge base information, such as company documents, personal notes, etc.
3. Assist users to perform related operations, such as booking flights, ordering meals, etc.

Semantic Kernel follows the plugin specifications of OpenAI plug-ins and can easily access and export plug-ins (such as plug-ins based on Bing, Microsoft 365, OpenAI), which allows developers to easily call different plugin services. In addition to plug-ins compatible with OpenAI, Semantic Kernel also has its own plugin definition method. Not only can Plugins be defined in a specified template format, but Plugins can also be defined within a function.

### Plugins Style

In early preview versions of Semantic Kernel, Plugins were defined as Skills. As mentioned above, on par with OpenAI. But the specific goals have not changed. You can understand that the plug-in of Semantic Kernel is to provide developers with various functions to complete intelligent business requirements. You can think of Semantic Kernel as the base of Lego. To complete the construction of a Great Wall, you need various functional modules. And these functional modules are what we call plug-ins.

We may have a business-based plug-in set, such as HRPlugins, which contains different functions, such as:

| Plugins | Intro |
| --- | --- |
| Holidays | holiday content |
| Contracts | About employee contracts |
| Departments | About organizational structure |

These sub-functions can be effectively combined according to different requirements to complete different planned tasks. For example, the following structure:

```
|-plugins
    |-HRPlugins
        |-Holidays
        |-Contract
        |-Departments
    |-EmailsPlugins
        |-HRMail
        |-CustomMail
    |-PeoplesPlugins
        |-Managers
        |-Workers
```

We issue an instruction to the LLMs, "Please send an email to the manager whose contract has expired." In fact, we find a combination from different components, and finally complete the relevant work based on Contract + Managers + HRMail.

# Semantic Kernel's plugins

## Define plugins through templates

We know we can have conversations with LLMs through prompt engineering. For an enterprise or start-up company, when we deal with business, it may not be a prompt project, but may need a collection of prompt engineerings. We can put these prompt engineerings sets for business capabilities into the Semantic Kernel plugin collection. For plugins that combine prompt projects, Semantic Kernel has a fixed template. Prompt engineerings are placed in the skprompt.txt file, and related parameter settings are placed in the config.json file. The final file structure is like this

```
|-plugins
    |-HRPlugins
        |-Holidays
            |-skprompt.txt
            |-config.json
        |-Contract
```

```
            |-skprompt.txt
            |-config.json
        |-Departments
            |-skprompt.txt
            |-config.json
    |-EmailsPlugins
        |-HRMail
            |-skprompt.txt
            |-config.json
        |-CustomMail
            |-skprompt.txt
            |-config.json
    |-PeoplesPlugins
        |-Managers
            |-skprompt.txt
            |-config.json
        |-Workers
            |-skprompt.txt
            |-config.json
```

Let's first take a look at the definition of **skprompt.txt**. This is generally where business-related prompts are placed and can support multiple parameters. Each parameter is placed in {{$parameter's name}}, as in the following format:

```
Translate {{$input}} into {{$language}}
```

Our job here is to translate the input content into a specific language. Input and language are two parameters, which means you can give any value to these two parameters.

**config.json** contains configuration-related content. In addition to setting parameters related to LLMs, you can also set input parameters and related descriptions.

```
{
    "schema": 1,
    "type": "completion",
    "description": "Translate sentenses into a language of your choice",
    "completion": [
      {
        "max_tokens": 2000,
        "temperature": 0.7,
        "top_p": 0.0,
        "presence_penalty": 0.0,
        "frequency_penalty": 0.0,
        "stop_sequences": [
```

```
            "[done]"
          ]
        }
      ],
      "input": {
        "parameters": [
          {
            "name": "input",
            "description": "sentense to translate",
            "defaultValue": ""
          },
           {
            "name": "language",
            "description": "Language to translate to",
            "defaultValue": ""
          }
        ]
      }
  }
```

### Define plugins through functions

We can also define different plugins through functions. This is a bit like Function Calling released by gpt-3.5-turbo on June 13, which enhances the capabilities of the OpenAI model by adding external functions and calling them. As mentioned at the beginning of this chapter.

#### Function Calling

By describing functions, LLMs call JSON objects of the parameters of these functions. This is a way to connect GPT functionality with external tools and APIs. Supports Azure OpenAI Service's or OpenAI's models:

- gpt-4
- gpt-4-1106-preview
- gpt-4-0613
- gpt-3.5-turbo
- gpt-3.5-turbo-1106
- gpt-3.5-turbo-0613

Semantic Kernel also supports Function Calling, but it is rarely used unless you have a Function Calling method customized for your business.

#### Semantic Kernel definition function plugin

Semantic Kernel defines function plugins, which is simpler than using Function Calling, and there are related definition methods before the birth of Function Calling. It is not limited by the model. You can use this method to perform knowledge and analysis on LLMs in early models. Data augmentation. It is recommended that all custom functions be placed in the same folder of plugins for easy management.

**.NET**

To define function extension, macro definition needs to be added

```
[KernelFunction,Description("search weather")]
public string WeatherSearch(string text)
{
    return "Guangzhou, 2 degree,rainy";
}
```

***Note:*** It is recommended to use business class encapsulation to define different extension functions to make it easier to call, such as

```
using Microsoft.SemanticKernel;
using System.ComponentModel;
using System.Globalization;


public class CompanySearchPlugin
{
    [KernelFunction,Description("search employee infomation")]
    public string EmployeeSearch(string input)
    {
        return "talk about hr information";
    }

    [KernelFunction,Description("search weather")]
    public string WeatherSearch(string text)
    {
        return text + ", 2 degree,rainy";
    }
}
```

The calling method :

```
var companySearchPluginObj = new CompanySearchPlugin();

var companySearchPlugin =
kernel.ImportPluginFromObject(companySearchPluginObj,
"CompanySearchPlugin");

var weatherContent = await kernel.InvokeAsync(
companySearchPlugin["WeatherSearch"],new(){["text"] = "guangzhou"});
```

```
weatherContent.GetValue<string>()
```

**Python**

To define function extension, macro definition needs to be added

```
@sk_function_context_parameter(name="city", description="city string")
def ask_weather_function(self, context: SKContext) -> str:
    return "Guangzhou's weather is 30 celsius degree , and very hot."
```

Place all custom functions in **native_function.py** and define them through classes, such as

```
from semantic_kernel.skill_definition import sk_function,
sk_function_context_parameter
from semantic_kernel import SKContext


class API:
    @sk_function(
        description = "Get news from the web",
        name = "NewsPlugin"
    )
    @sk_function_context_parameter(name="location", description="location
name")
    def get_news_api(self, context: SKContext) -> str:
        return """Get news from the """ + context["location"] + """."""

    @sk_function(
            description="Search Weather in a city",
            name="WeatherFunction"
        )
    @sk_function_context_parameter(name="city", description="city string")
    def ask_weather_function(self, context: SKContext) -> str:
        return "Guangzhou's weather is 30 celsius degree , and very hot."

    @sk_function(
            description="Search Docs",
            name="DocsFunction"
        )
    @sk_function_context_parameter(name="docs", description="docs string")
    def ask_docs_function(self, context: SKContext) -> str:
```

```
        return "ask docs :" + context["docs"]
```

Calling functions：

```
api_plugin = kernel.import_native_skill_from_directory(base_plugin ,
"APIPlugin")

context_variables = sk.ContextVariables(variables={
    "location": "China"
})

news_result = await api_plugin["NewsPlugin"].invoke_async(
variables=context_variables)

print(news_result)
```

### Predefined plugin

Semantic Kernel has a lot of predefined plug-ins as related capabilities for solving general business. Of course, as Semantic Kernel matures, more built-in plug-ins will be incorporated.

## Samples

***.NET Samples*** Click

***Python Samples*** Click

## Summary

Through plugins, you can complete more work based on business scenarios. Through studying this chapter, you have initially mastered the definition of plugins and how to use plugins to work. I hope you can create a plugin library for your enterprise based on business scenarios in real business work.