Semantic Kernel — Components (Concise Reference)

A short, visual-friendly document summarizing the Semantic Kernel components (from Microsoft Learn).

---

1. One-line overview

Semantic Kernel (SK) is an orchestration layer that lets you combine LLMs, memory (vector stores), plugins (functions), prompt templates and planners to build agentic applications.

---

2. Core components (quick bullets)

- **AI Service Connectors** — Abstracts Chat Completion / Text Generation / Embeddings / TTS / STT and other model services so SK can call different providers through a common interface.
- **Vector Store (Memory) Connectors** — Abstraction for vector databases used for memory and retrieval (RAG). SK does not use them automatically; expose them as plugins or use via prompt templates.
- **Plugins & Functions** — Named containers of functions. Sources:
  - Native code functions
  - OpenAPI-backed functions
  - Prompt templates (registered as functions)
  - Text-search (ITextSearch) adapters for RAG

  Plugins can be:

  1. Advertised to the chat model (so the model can request them)
  2. Called directly from templates or code
- **Prompt Templates** — Combine context, instructions, user input and function output. Can be used as the starting point for a chat flow or registered as plugin functions. Templates can call other plugins and produce multi-step flows.
- **Filters** — Hooks that run before/after events during the chat flow (before/after prompt rendering, before/after function invocation). Filters can be nested and are registered with the kernel.

---

3. How they fit together (flow diagram, simple)

```
User message
   ↓
Prompt Template (renders) ⟶ executes hardcoded plugin calls (if any)
   ↓
Kernel invokes Chat Completion AI (via AI Service Connector)
   ↓
Chat model may request Plugin functions (advertised) or return text
   ↓
Kernel (using Filters) runs pre/post hooks
   ↓
Optional: Vector Store used by a plugin for retrieval
   ↓
Final answer returned to user
```

---

4. Minimal C# snippet (registering services)

```csharp
// PSEUDO-CODE (concise)
var kernel = Kernel.Builder.Build();

kernel.Config.AddOpenAIChatService("gpt-deploy", endpoint:
"<ENDPOINT>", apiKey: "<KEY>");

kernel.Memory.RegisterVectorStore("my-vs", new
AzureSearchVectorStore(...));

kernel.RegisterNativeFunction("utils", "Summarize", (input) =>
SummarizeText(input));

var result = await kernel.RunAsync("Hello, how can I help?");
```

5. Minimal Python snippet (registering services, plugins, vector store, filters)

```python
import asyncio
import logging

from semantic_kernel import Kernel
from semantic_kernel.utils.logging import setup_logging
from semantic_kernel.functions import kernel_function
from semantic_kernel.connectors.ai.open_ai import AzureChatCompletion
from semantic_kernel.connectors.ai.function_choice_behavior import
FunctionChoiceBehavior
from semantic_kernel.connectors.ai.chat_completion_client_base import
ChatCompletionClientBase
from semantic_kernel.contents.chat_history import ChatHistory
from semantic_kernel.functions.kernel_arguments import KernelArguments
from
```

```
semantic_kernel.connectors.ai.open_ai.prompt_execution_settings.azure_
chat_prompt_execution_settings import (
    AzureChatPromptExecutionSettings,
)

async def main():
    # Initialize the kernel
    kernel = Kernel()

    # Add Azure OpenAI chat completion
    chat_completion = AzureChatCompletion(
        deployment_name="your_models_deployment_name",
        api_key="your_api_key",
        base_url="your_base_url",
    )
    kernel.add_service(chat_completion)

    # Set the logging level for semantic_kernel.kernel to DEBUG
    setup_logging()
    logging.getLogger("kernel").setLevel(logging.DEBUG)

    # Add a plugin (LightsPlugin should be defined below)
    kernel.add_plugin(
        LightsPlugin(),
        plugin_name="Lights",
    )

    # Enable planning
    execution_settings = AzureChatPromptExecutionSettings()
    execution_settings.function_choice_behavior =
FunctionChoiceBehavior.Auto()

    # Create a history of the conversation
    history = ChatHistory()

    # Chat loop
    while True:
        userInput = input("User > ")

        if userInput.lower() == "exit":
            break

        history.add_user_message(userInput)

        result = await chat_completion.get_chat_message_content(
            chat_history=history,
            settings=execution_settings,
            kernel=kernel,
        )

        print("Assistant > " + str(result))
```

```
        history.add_message(result)

# Run the main function
if __name__ == "__main__":
    asyncio.run(main())
```csharp
// PSEUDO-CODE (concise)
var kernel = Kernel.Builder.Build();

// Register AI service (Chat completion / text generation)
kernel.Config.AddOpenAIChatService("gpt-deploy", endpoint:
"<ENDPOINT>", apiKey: "<KEY>");

// Register a vector store connector (memory)
kernel.Memory.RegisterVectorStore("my-vs", new
AzureSearchVectorStore(...));

// Register a native function plugin
kernel.RegisterNativeFunction("utils", "Summarize", (input) =>
SummarizeText(input));

// Run a prompt template
var result = await kernel.RunAsync("Hello, how can I help?");
```

---

6. Practical notes / tips

- **Plugins are powerful**: expose data sources and business logic as plugin functions rather than embedding everything in prompts.
- **Prompt templates as functions** enable language-defined behaviour (describe the function in natural language and register it).
- **Use Filters** for logging, telemetry, auth, or modifying context before/after calls.
- **Vector stores are explicit**: SK doesn't automatically search them; wire a retrieval plugin for RAG.

---

7. Quick use-cases

- Knowledge-base assistant (KB search plugin + SK prompt templates)
- Multi-step enterprise workflows (planner + plugins)
- Personalization and session memory (vector store + memory connectors)
- Tool-enabled agents (calendar, DB, search via OpenAPI plugins)

```