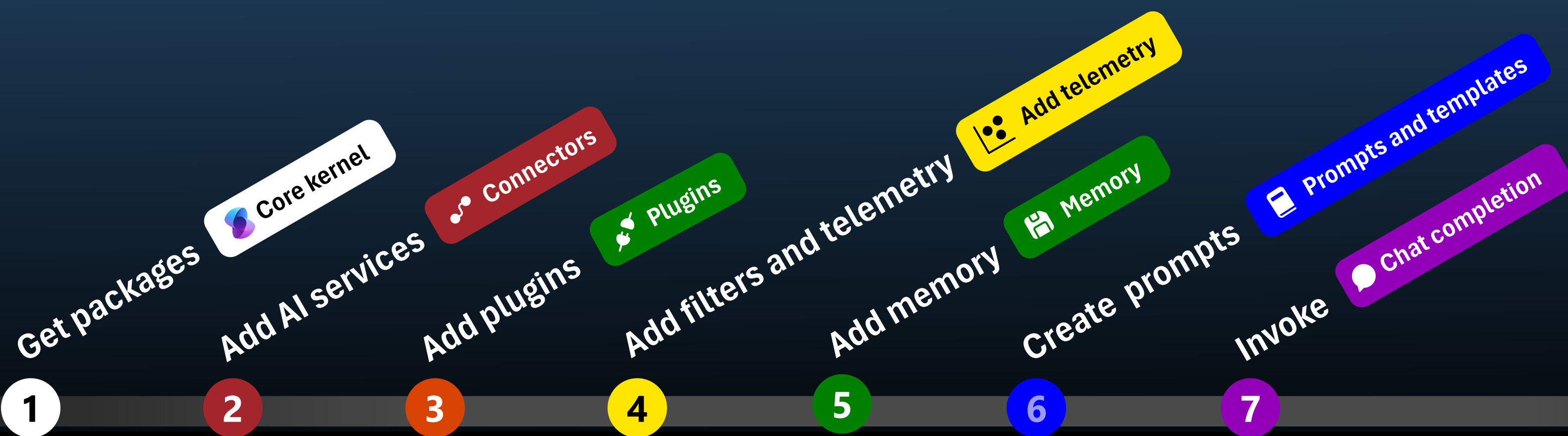




Semantic Kernel for Python

2024 Python version 1.0 map



1 Get packages

<https://aka.ms/sk/kernel>

```
# Get the latest Semantic Kernel Python Package
pip install semantic-kernel

# Get the kernel to build
from semantic_kernel import Kernel
kernel = Kernel()
```

2 Add AI services

<https://aka.ms/sk/aiservices>

```
# Semantic Kernel allows you to add and swap out different AI services depending on your needs. In addition to Azure OpenAI and OpenAI, Semantic Kernel supports Google Gemini, MistralAI, Ollama, local models, and more.

from semantic_kernel.connectors.ai.open_ai import AzureChatCompletion, OpenAIChatCompletion, AzureTextToImage

# Azure OpenAI example
chat_service = AzureChatCompletion(deployment_name=AOAI_DEP_NAME, endpoint=AOAI_ENDPOINT, api_key=AOAI_KEY, )
kernel.add_service(chat_service)

# OpenAI example
chat_service = OpenAIChatCompletion(api_key=AOAI_KEY, ai_model_id="gpt-4o", )
kernel.add_service(chat_service)

# other modalities
text_to_image_service = AzureTextToImage(deployment_name=AOAI_DEP_NAME, endpoint=AOAI_ENDPOINT, api_key=AOAI_KEY, )

# other services
from semantic_kernel.connectors.ai.mistral_ai import MistralAIChatCompletion

kernel.add_service(MistralAIChatCompletion(ai_model_id=MAI_MODEL_ID, api_key=MAI_KEY))
```

3 Add plugins

<https://aka.ms/sk/plugins>

```
# Plugins are a way to extend the functionality of the kernel. Plugins are added differently depending on where they are stored.

# Import for OpenAPI (most common)
plugin = kernel.add_plugin_from_openapi(plugin_name="WeatherForecast", openapi_document_path="http://127.0.0.1:8000/openapi.json")

#Import Example Plugins from the Semantic Kernel package
from semantic_kernel.core_plugins import TextPlugin

text_plugin = kernel.add_plugin(TextPlugin(), "TextPlugin")

#Define a custom plugin
from semantic_kernel.functions.kernel_function_decorator import kernel_function

@kernel_function(description="Get the current date.")
def date(self) -> str:
    """Get the current date."""
    now = datetime.datetime.now()
    return now.strftime("%A, %d %B, %Y" )

kernel.add_plugin(plugin_name="time" function=date)

# Load from a directory
plugins_directory = "path/to/plugins"
funFunctions = kernel.add_plugin(parent_directory=plugins_directory, plugin_name="FunPlugin")
```

4 Filters and telemetry

<https://aka.ms/sk/filters>

```
# Filters enable pre and post event handling for use cases like Responsible AI
from semantic_kernel.filters.filter_types import FilterTypes
from semantic_kernel.filters.prompts.prompt_render_context import PromptRenderContext

# A filter is a piece of custom code that runs at certain points in the process
# Name the function with arbitrary names, but the signature needs to be: 'context, next'

@kernel.filter(FilterTypes.PROMPT_RENDERING)
async def prompt_rendering_filter(context: PromptRenderContext, next):
    await next(context)
    context.rendered_prompt = f"Reply only in French {context.rendered_prompt or ''}"

# enable logging with OpenTelemetry
from opentelemetry.sdk._logs import LoggerProvider, LoggingHandler
from opentelemetry.sdk._logs.export import BatchLogRecordProcessor, ConsoleLogExporter
from opentelemetry.sdk.resources import Resource

# Set up LoggerProvider with resource metadata for context
logger_provider = LoggerProvider(resource=Resource.create({"service.name": "my-service"}))

# Use ConsoleLogExporter for testing; replace with OTLP exporter in production
logger_provider.add_log_record_processor(BatchLogRecordProcessor(ConsoleLogExporter()))

# Create a logging handler with the OpenTelemetry logger provider
handler = LoggingHandler(logger_provider=logger_provider)

# Attach the handler to the root logger and set log level
logger = logging.getLogger()
logger.addHandler(handler)
```

5 Add memory

<https://aka.ms/sk/memory>

```
# Semantic Kernel offers several memory store connectors to vector databases that you can use to store and retrieve information. Including Azure AI Search, Azure SQL Database, Azure CosmosDB, Chroma, DuckDB, Milvus, MongoDB Atlas, Pinecone, Postgres, Qdrant, Redis, Sqlite, Weaviate and more.

@vectorstoremodel
# Define a data model
class Glossary(BaseModel):
    id: Annotated[str, VectorStoreRecordKeyField]
    term: Annotated[str | None, VectorStoreRecordDataField()] = None
    definition: Annotated[str, VectorStoreRecordDataField(
        has_embedding=True, embedding_property_name="definition_vector")]
    ]
    definition_vector: Annotated[list[float] | None,
        VectorStoreRecordVectorField(dimensions=1536, local_embedding=True,
            embedding_settings={"embedding":
                OpenAIEmbeddingPromptExecutionSettings(dimensions=1536)}))] = None

# Add the embedding service
embeddings = AzureTextEmbedding(service_id="embedding", api_key=AOAI_KEY, deployment_name=AOAI_EMBEDDING_DEP_NAME, endpoint=AOAI_ENDPOINT)
kernel.add_service(embeddings)

# Create a collection
collection = InMemoryVectorCollection[Glossary](
    data_model_type=Glossary,
    collection_name="collection_name",
)
await collection.create_collection_if_not_exists()
record1 = Glossary(id = "1", term = "Azure", definition = "A cloud computing service")
record2 = Glossary(id = "2", term = "OpenAI", definition = "An AI research lab")
record3 = Glossary(id = "3", term = "Semantic Kernel", definition = "A powerful AI service that allows you to build and deploy AI models in a few lines of code")

# Add the records to the collection
records = await VectorStoreRecordUtils(kernel).add_vector_to_records(
    [record1, record2, record3], data_model_type=Glossary)
keys = await collection.upsert_batch(records)

# Search the collection
query = "What can deploy AI models?"
query_vector = (await embeddings.generate_raw_embeddings([query]))[0]

# Use vectorized search to search using the vector.
results = await collection.vectorized_search(
    vector=query_vector,
    options=VectorSearchOptions(vector_field_name="definition_vector"),
)

# Print the results
async for result in results.results:
    print(f"{result.record.id}, {result.record.term}: {result.record.definition} (score: {result.score})")
```

6 Create prompts

<https://aka.ms/sk/prompts>

```
# Prompts are a way to interact with the kernel using natural language. Prompts can be used to ask questions, get information, or execute functions.

# Invoke a simple prompt
result = await kernel.invoke_prompt("Tell me about GenAI")

# Invoke a prompt with plugin
result = await kernel.invoke_prompt("The current time is {{time.date}}.")

# Invoke a prompt with arguments
result = await kernel.invoke_prompt("Tell me about {{$topic}}", topic="Dogs")

# Define settings for the prompt, this setting will allow the prompt to automatically execute functions
execution_settings = PromptExecutionSettings(
    function_choice_behavior=FunctionChoiceBehavior.Auto(auto_invoke=True),
)
result = await kernel.invoke_prompt("How many days until Christmas? Explain you thinking.", arguments=KernelArguments(settings=execution_settings))

print(result)
```

7 Chat completion

<https://aka.ms/sk/chat>

```
# ChatCompletionService is a common way to interact with the models
chat_function = kernel.add_function(
    plugin_name="ChatBot",
    function_name="Chat",
    prompt="{{$chat_history}}{{$user_input}}",
    template_format="semantic-kernel",
)

# Create a chat history
chat_history = ChatHistory(system_message="You are a librarian, expert about books")
user_input = "Hi, I'm looking for book suggestions"

# Invoke the chat function
answer = await kernel.invoke(chat_function, KernelArguments(user_input=user_input, chat_history=chat_history))
chat_history.add_user_message(user_input)
chat_history.add_assistant_message(str(answer))
print(answer)
```

