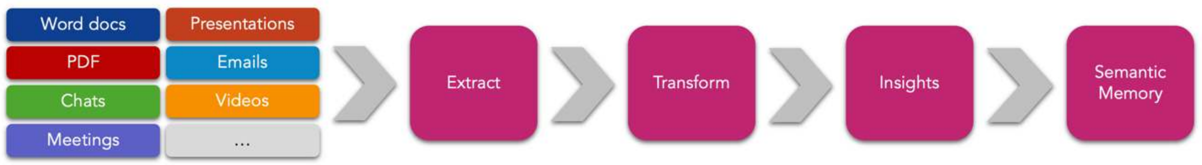
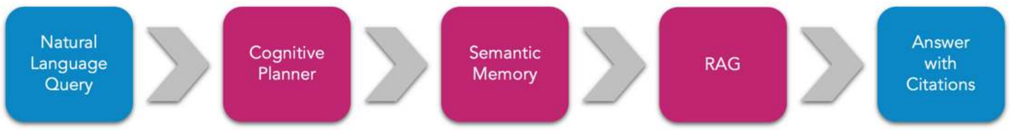
Kernel Memory

Kernel Memory (KM) is a **multi-modal [AI Service]** specialized in the efficient indexing of datasets through custom continuous data hybrid pipelines, with support for **Retrieval Augmented Generation** (RAG), synthetic memory, prompt engineering, and custom semantic memory processing.

KM includes a GPT **Plugin**, **web clients**, a .NET library for embedded applications, and as a Docker container.



Utilizing advanced embeddings and LLMs, the system enables Natural Language querying for obtaining answers from the indexed data, complete with citations and links to the original sources.



Designed for seamless integration as a Plugin with Semantic Kernel, Microsoft Copilot and ChatGPT, Kernel Memory enhances data-driven features in applications built for most popular AI platforms.

Repository Guidance

This repository presents best practices and a reference architecture for memory in specific AI and LLMs application scenarios. Please note that **the provided code serves as a demonstration** and is **not an officially supported** Microsoft offering.

Kernel Memory (KM) and Semantic Memory (SM)

Semantic Memory (SM) is a library for C#, Python, and Java that wraps direct calls to databases and supports vector search. It was developed as part of the Semantic Kernel (SK) project and serves as the first public iteration of long-term memory. The core library is maintained in three languages, while the list of supported storage engines (known as "connectors") varies across languages.

Kernel Memory (KM) is a service built on the feedback received and lessons learned from developing Semantic Kernel (SK) and Semantic Memory (SM). It provides several features that would otherwise have to be developed manually, such as storing files, extracting text from files, providing a framework to secure users' data, etc. The KM codebase is entirely in .NET, which eliminates the need to write and maintain features in multiple languages. As a service, **KM can be used from any language, tool, or platform, e.g. browser extensions and ChatGPT assistants.**

Here's a few notable differences:

Feature	Semantic Memory	Kernel Memory
Data formats	Text only	Web pages, PDF, Images, Word, PowerPoint, Excel, Markdown, Text, JSON, more being added
Search	Cosine similarity	Cosine similarity, Hybrid search with filters, AND/OR conditions
Language support	C#, Python, Java	Any language, command line tools, browser extensions, low-code/no-code apps, chatbots, assistants, etc.
Storage engines	Azure AI Search, Chroma, DuckDB, Kusto, Milvus, MongoDB, Pinecone, Postgres, Qdrant, Redis, SQLite, Weaviate	Azure AI Search, Elasticsearch, Postgres, Qdrant, Redis, SQL Server, In memory KNN, On disk KNN. In progress: Chroma

and **features available only in Kernel Memory**:

- RAG (Retrieval Augmented Generation)
- RAG sources lookup
- Summarization
- Security Filters (filter memory by users and groups)
- Long running ingestion, large documents, with retry logic and durable queues
- Custom tokenization
- Document storage
- OCR via Azure Document Intelligence
- LLMs (Large Language Models) with dedicated tokenization
- Cloud deployment
- OpenAPI
- Custom storage schema (partially implemented/work in progress)
- Short Term Memory (partially implemented/work in progress)
- Concurrent write to multiple vector DBs

Supported Data formats and Backends

- MS Office: Word, Excel, PowerPoint
- PDF documents
- Web pages

- JPG/PNG/TIFF Images with text via OCR
- MarkDown and Raw plain text
- JSON files
- 🧠 AI
  - Azure OpenAI
  - OpenAI
  - LLama - thanks to llama.cpp and LLamaSharp
  - Azure Document Intelligence
- ↗ Vector storage
  - Azure AI Search
  - Postgres+pgvector
  - Qdrant
  - MSSQL Server (third party)
  - Elasticsearch (third party)
  - Redis
  - Chroma (work in progress)
  - In memory KNN vectors (volatile)
  - On disk KNN vectors
- 📀 Content storage
  - Azure Blobs
  - Local file system
  - In memory, volatile content
- ⏳ Orchestration
  - Azure Queues
  - RabbitMQ
  - Local file based queue
  - In memory queues (volatile)

Kernel Memory in serverless mode

Kernel Memory works and scales at best when running as a service, allowing to ingest thousands of documents and information without blocking your app.

However, you can use Kernel Memory also serverless, embedding the MemoryServerless class in your app.

Importing documents into your Kernel Memory can be as simple as this:

```
var memory = new KernelMemoryBuilder()

  .WithOpenAIDefaults(Env.Var("OPENAI_API_KEY"))

  .Build<MemoryServerless>();
```

```
// Import a file

await memory.ImportDocumentAsync("meeting-transcript.docx", tags: new() { { "user", "Blake" } });



// Import multiple files and apply multiple tags

await memory.ImportDocumentAsync(new Document("file001")

    .AddFile("business-plan.docx")

    .AddFile("project-timeline.pdf")

    .AddTag("user", "Blake")

    .AddTag("collection", "business")

    .AddTag("collection", "plans")

    .AddTag("fiscalYear", "2023"));
```

Asking questions:

```
var answer1 = await memory.AskAsync("How many people attended the meeting?");



var answer2 = await memory.AskAsync("what's the project timeline?", filter: new MemoryFilter().ByTag("user", "Blake"));
```

The code leverages the default documents ingestion pipeline:

1. Extract text: recognize the file format and extract the information
2. Partition the text in small chunks, to optimize search
3. Extract embedding using an LLM embedding generator
4. Save embedding into a vector index such as Azure AI Search, Qdrant or other DBs.

Documents are organized by users, safeguarding their private information. Furthermore, memories can be categorized and structured using **tags**, enabling efficient search and retrieval through faceted navigation.

Data lineage, citations

All memories and answers are fully correlated to the data provided. When producing an answer, Kernel Memory includes all the information needed to verify its accuracy.

Using Kernel Memory Service

Depending on your scenarios, you might want to run all the code **locally inside your process, or remotely through an asynchronous service.**

If you're importing small files, and need only C# and can block the process during the import, local-in-process execution can be fine, using the **MemoryServerless** seen above.

However, if you are in one of these scenarios:

- I'd just like a web service to import data and send queries to answer
- My app is written in **TypeScript, Java, Rust, or some other language**
- I want to define **custom pipelines mixing multiple languages** like Python, TypeScript, etc
- I'm importing **big documents that can require minutes to process**, and I don't want to block the user interface
- I need memory import to **run independently, supporting failures and retry logic**

then you can deploy Kernel Memory as a service, plugging in the default handlers or your custom Python/TypeScript/Java/etc. handlers, and leveraging the asynchronous non-blocking memory encoding process, sending documents and asking questions using the **MemoryWebClient**.

Here you can find a complete set of instruction about how to run the Kernel Memory service.

Quick test using the Docker image

If you want to give the service a quick test, use the following command to **start the Kernel Memory Service** using OpenAI:

```
docker run -e OPENAI_API_KEY="..." -it --rm -p 9001:9001 kernelmemory/service
```

If you prefer using custom settings and services such as Azure OpenAI, Azure Document Intelligence, etc., you should create an appsettings.Development.json file overriding the default values set in appsettings.json, or using the configuration wizard included:

```
cd service/Service

dotnet run setup
```

Then run this command to start the Docker image with the configuration just created:

```
docker run --volume ./appsettings.Development.json:/app/data/appsettings.Production.json \

  -it --rm -p 9001:9001 kernelmemory/service
```

To import files using Kernel Memory web service, use **MemoryWebClient**:

```
#reference clients/WebClient/WebClient.csproj
```

```
var memory = new MemoryWebClient("http://127.0.0.1:9001"); // <== URL where the web service is running


// Import a file (default user)

await memory.ImportDocumentAsync("meeting-transcript.docx");


// Import a file specifying a Document ID, User and Tags

await memory.ImportDocumentAsync("business-plan.docx",

    new DocumentDetails("user@some.email", "file001")

        .AddTag("collection", "business")

        .AddTag("collection", "plans")

        .AddTag("fiscalYear", "2023"));
```

Getting answers via the web service

You can find a full example here.

Custom memory ingestion pipelines

On the other hand, if you need a custom data pipeline, you can also customize the steps, which will be handled by your custom business logic:

```
// Memory setup, e.g. how to calculate and where to store embeddings

var memoryBuilder = new KernelMemoryBuilder().WithOpenAIDefaults(Env.Var("OPENAI_API_KEY"));

memoryBuilder.Build();

var orchestrator = memoryBuilder.GetOrchestrator();


// Define custom .NET handlers

var step1 = new MyHandler1("step1", orchestrator);

var step2 = new MyHandler2("step2", orchestrator);

var step3 = new MyHandler3("step3", orchestrator);
```

```
await orchestrator.AddHandlerAsync(step1);

await orchestrator.AddHandlerAsync(step2);

await orchestrator.AddHandlerAsync(step3);


// Instantiate a custom pipeline

var pipeline = orchestrator

    .PrepareNewFileUploadPipeline("user-id-1", "mytest", new[] { "memory-collection" })

    .AddUploadFile("file1", "file1.docx", "file1.docx")

    .AddUploadFile("file2", "file2.pdf", "file2.pdf")

    .Then("step1")

    .Then("step2")

    .Then("step3")

    .Build();


// Execute in process, process all files with all the handlers

await orchestrator.RunPipelineAsync(pipeline);
```

Web API specs

The API schema is available at http://127.0.0.1:9001/swagger/index.html when running the service locally with OpenAPI enabled.

Examples and Tools

Examples

1. Collection of Jupyter notebooks with various scenarios
2. Using Kernel Memory web service to upload documents and answer questions
3. Using KM Plugin for Semantic Kernel
4. Importing files and asking question without running the service (serverless mode)
5. Processing files with custom steps
6. Upload files and ask questions from command line using curl

7. Customizing RAG and summarization prompts
8. Custom partitioning/text chunking options
9. Using a custom embedding/vector generator
10. Using custom LLMs
11. Using LLama
12. Summarizing documents
13. Natural language to SQL examples
14. Writing and using a custom ingestion handler
15. Running a single asynchronous pipeline handler as a standalone service
16. Test project linked to KM package from nuget.org
17. Integrating Memory with ASP.NET applications and controllers
18. Sample code showing how to extract text from files

Tools

1. Curl script to upload files
2. Curl script to ask questions
3. Curl script to search documents
4. Script to start Qdrant for development tasks
5. Script to start RabbitMQ for development tasks
6. .NET appsettings.json generator

.NET packages

- **Microsoft.KernelMemory.WebClient:** The web client library, can be used to call a running instance of the Memory web service. .NET Standard 2.0 compatible.
- **Microsoft.KernelMemory.SemanticKernelPlugin:** a Memory plugin for Semantic Kernel, replacing the original Semantic Memory available in SK. .NET Standard 2.0 compatible.
- **Microsoft.KernelMemory.Abstractions:** The internal interfaces and models shared by all packages, used to extend KM to support third party services. .NET Standard 2.0 compatible.
- **Microsoft.KernelMemory.MemoryDb.AzureAISearch:** Memory storage using **Azure AI Search**.
- **Microsoft.KernelMemory.MemoryDb.Postgres:** Memory storage using **PostgreSQL**.
- **Microsoft.KernelMemory.MemoryDb.Qdrant:** Memory storage using **Qdrant**.
- **Microsoft.KernelMemory.AI.AzureOpenAI:** Integration with **Azure OpenAI** LLMs.
- **Microsoft.KernelMemory.AI.LlamaSharp:** Integration with **LLama** LLMs.
- **Microsoft.KernelMemory.AI.OpenAI:** Integration with **OpenAI** LLMs.
- **Microsoft.KernelMemory.DataFormats.AzureAIDocIntel:** Integration with Azure AI Document Intelligence.
- **Microsoft.KernelMemory.Orchestration.AzureQueues:** Ingestion and synthetic memory pipelines via Azure Queue Storage.
- **Microsoft.KernelMemory.Orchestration.RabbitMQ:** Ingestion and synthetic memory pipelines via RabbitMQ.
- **Microsoft.KernelMemory.ContentStorage.AzureBlobs:** Used to store content on Azure Storage Blobs.

- **Microsoft.KernelMemory.Core:** The core library, can be used to build custom pipelines and handlers, and contains a serverless client to use memory in a synchronous way, without the web service. .NET 6+.

Packages for Python, Java and other languages

Kernel Memory service offers a **Web API** out of the box, including the **OpenAPI swagger** documentation that you can leverage to test the API and create custom web clients. For instance, after starting the service locally, see http://127.0.0.1:9001/swagger/index.html.

A python package with a Web Client and Semantic Kernel plugin will soon be available. We also welcome PR contributions to support more languages.