Technical Report: Synapse LLM based system for Learning

Synapse Group

December 4, 2025

Abstract

Synapse is an LLM-based system designed to support university studies through the automatic generation of educational material. This report explores the system architecture, focusing on the integration of agentic design patterns such as **Retrieval-Augmented Generation**, **Reflection**, **Few-Shot**, **Tool Use**, and advanced export functionalities.

Contents

1 Introduction

The evolution of Large Language Models (LLMs) has opened new possibilities in the field of assisted education. However, the "naive" use of these models presents significant limitations: hallucinations, superficiality in responses, and a lack of adherence to specific study materials.

We created *Synapse* to solve these problems through an architecture that combines:

- **RAG:** To ground LLM responses in user-provided documents.

- **Reflection:** To improve the quality and correctness of generated flashcards through a draft-critique-refine cycle.

- **Few-Shot:** To enhance LLM performance by providing examples within prompts.

- **Tool Use:** By searching on the web to supplement user-provided knowledge (if provided) with up-to-date information from the web.

- **Exporting Capabilities:** To produce flashcards compatible also with the popular spaced repetition study tool Anki.

2 System Architecture

The system is built in Python and follows a modular service-oriented architecture.

2.1 Service Overview

The logical core resides in the `services/` directory, which decouples responsibilities:

- `ai_service.py`: Abstraction for interaction with LLMs (supports Google Gemini and local models via Ollama).

- `rag_service.py`: Handles document indexing, chunking, and vector retrieval.

- `reflection_service.py`: Implements the logic for the draft-critique-refine cycle of the flashcard generation.

- `web_search_service.py`: Provides additional context from the web when local documents are insufficient.

- `export_service.py`: Manages data conversion into interoperable formats (CSV, Anki).

2.2 User Interface

The graphical interface is built with **PyQt6**.

The UI structure (`ui/`) separates presentation logic ('main_window.py', 'subject_window.py') from configuration dialogs ('settings_dialog.py').

3 Agentic Design Patterns: Reflection

Instead of passively accepting the LLM's output, the system establishes a self-improvement cycle. The flashcard generation process follows three distinct phases, orchestrated by `reflection_service.py`:

3.1 1. Draft (Draft Generation)

The LLM is instructed to act as an expert in metacognition. The prompt imposes "atomic" rules based on Andy Matuschak's principles, requiring structured JSON output.

```
prompt = f"""
You are an expert in learning and metacognition. Your goal is to create
    ONE 'atomic' flashcard based on Andy Matuschak's principles.

REQUIRED TOPIC (MANDATORY): {topic}

ATTENTION: The flashcard MUST ONLY be about '{topic}'.
- If the context does not contain information relevant to '{topic}',
    indicate that there is insufficient information.
- DO NOT create flashcards on topics other than '{topic}'.

Use ONLY information from the following context that is RELEVANT to '{
    topic}':
{context}

Follow these 5 ABSOLUTE RULES:
1.  **Focused**: The question (front) must relate to ONLY ONE concept or
     fact RELATED TO '{topic}'.
2.  **Precise**: The question must not be ambiguous. It must make it
    clear exactly what is required on '{topic}'.
3.  **Consistent**: The answer (back) must be the only correct answer
    and always the same.
4.  **Ask 'Why'**: If possible, prefer questions about 'why' or
    implications rather than dry definitions.
5.  **Cognitive Effort**: The answer should NOT be obvious from the
    question (avoid trivial clues or binary Yes/No questions).

Return the response in JSON format with this exact structure:
{{
'front': 'Atomic, precise question requiring effort on {topic}',
'back': 'Concise and accurate response based on context'
}}

Do not invent information that is not present in the context. If the
    context does not mention '{topic}', return:
{{
'front': 'Information not available',
'back': 'The context provided does not contain relevant information
    about {topic}'
}}
"""
```

Listing 1: Prompt for draft generation

Even if not highlighted in a separate section, this example shows how the prompt also used **Few-Shot** examples to guide the model towards a well-formatted JSON file.

3.2   2. Critique (Critical Analysis)

In this phase, the system assumes an "adversarial" role. A second prompt asks the LLM to evaluate the newly generated draft. This is not a simple review, but a validation against specific criteria.

```
prompt = f"""
You are an expert reviewer of educational materials who follows Andy
    Matuschak's principles.

Analyse this flashcard based on the context provided.

FLASHCARD:
Question: {front}
Answer: {back}

AVAILABLE CONTEXT:
{context}

Evaluate the flashcard EXCLUSIVELY according to these 5 RULES:
1.   **Focused**: Does it ask about a single concept? Or is it too broad
     (e.g., does it ask for a list)?
2.   **Precise**: Is it ambiguous? Is it clear what it wants?
3.   **Context**: Is the answer correct and based ONLY on the context?
4.   **Cognitive Effort**: Is the answer too obvious when reading the
     question?
5.   **Conceptual**: Is it a dry definition (negative) or does it ask for
     the 'why', a difference, or an implication (positive)?

Provide CONSTRUCTIVE criticism in 2-3 sentences.
- If the flashcard is already excellent and follows the rules, say so (e
    .g., "Excellent, follows all the principles").
- If it does not follow the rules, explain WHAT to improve (e.g. 'It is
    not focused, it asks two things. Break it down.' OR 'The question is
    too vague, make it more precise.' OR 'The answer is obvious, rephrase
    the question to require more effort.').
"""
```

Listing 2: Prompt for critique

3.3   3. Refine (Refinement)

If the critique highlights defects (detected via keyword matching such as "not focused", "vague", etc.), the system invokes the LLM again, passing:

1. The original flashcard.

2. The received critique.

3. The original context.

The model is forced to produce a new version that specifically resolves the reported issues.

4 Agentic Design Patterns: RAG

The RAG module (`rag_service.py`) is the heart of Synapse's "memory". This technology overcomes the static knowledge limit of LLMs by providing dynamic access to user-uploaded documents (PDFs, slides).

4.1 How it works: A Simple Example

To understand the operation, let's consider a practical example. Imagine the user asks: *"What is the Reflection pattern?"*.

1. **Retrieval:** The system creates the embedding of that question (converting it into a numerical vector) and searches the database (Qdrant) for text fragments (chunks) from the PDFs that are semantically closest to the question.

2. **Augmentation:** The found fragments are inserted into the system prompt. The prompt becomes similar to:

```
Use ONLY the following information to answer:
[Text extracted from PDF: 'Reflection is a pattern...'].
Question: What is the Reflection pattern?"
```

Listing 3: Example of RAG augmented prompt

3. **Generation:** The LLM generates the answer based exclusively on the provided context, ensuring that the explanation is faithful to the study material and not generic.

4.2 Recursive Chunking

Retrieval quality depends drastically on how documents are divided (chunking). A naive approach, cutting text every fixed N characters, risks splitting sentences in half or separating related concepts, making the context incomprehensible to the LLM.
Synapse adopts a **Recursive Character Text Splitting** strategy, which aims to preserve text semantics by respecting the natural structure of language. The algorithm does not cut arbitrarily but tries to divide the text using a hierarchy of separators, in order of priority:

1. **Paragraphs (\n\n):** First attempts to divide text into complete logical blocks (paragraphs).

2. **Lines (\n):** If a paragraph is still too long to handle, it splits by lines.

3. **Sentences (.   ):** If necessary, it goes down to the single sentence level.

4. **Characters:** Only as a last resort does it cut a sentence in half.

   This approach ensures that each "chunk" sent to the model contains, as much as possible, a complete and coherent thought, significantly improving the quality of generated responses.

4.3 Vector Store and Embeddings

The system uses **Qdrant** in embedded mode (local disk storage) to store vectors. This avoids the need for complex external servers to configure.

For embeddings, the system is hybrid:

- **Local (Ollama):** Uses models like `nomic-embed-text`, ensuring total privacy and offline operation.

- **Cloud (Gemini):** Uses Google APIs for high-performance embeddings (`gemini-embedding-001`) ideal for those without powerful hardware.

The search is performed via **Cosine Similarity**, with a configurable relevance threshold (default 0.25) to discard noisy results.

5 Web Search Integration

To ensure responses are always up-to-date and complete, Synapse integrates a web search module (`web_search_service.py`) that intervenes when local documents do not contain the necessary information.

5.1 Service Operation

The system does not limit itself to a simple Google search but uses a specialized tool for AI agents (in this case, **Tavily**) optimized for content extraction. The goal is not to provide links to the user, but to retrieve high-quality *raw context* to inject into the LLM prompt.

The process takes place in three steps:

- **Query Optimization:** The LLM reformulates the user's question into an optimized search query (e.g., from "who is the current president?" to "current president of Italy 2024").

- **Content Extraction:** The search engine visits the most relevant pages in parallel, removing ads, menus, and boilerplate, and extracting only informative text.

- **Context Aggregation:** Text fragments extracted from different sources are aggregated into a single context block passed to the LLM for final response generation.

5.2 Fallback Strategy

The service is designed to be resilient:

1. **AI Engine (Tavily):** Used as the primary source for its ability to aggregate and clean data from multiple web sources in real-time.

2. **Wikipedia API:** If the primary service is unreachable or not configured, the system automatically falls back to public Wikipedia APIs to obtain general definitions and concepts.

6 Export and Interoperability

To be truly useful, the generated material must be usable in daily study tools. The `export_service.py` service handles this need.

6.1 Anki Package Generation (.apkg)

The most advanced feature is the direct creation of packages for **Anki**, the most widespread spaced repetition software. The code interacts directly with Anki's internal SQLite database:

```
cursor.execute('''
    CREATE TABLE notes (
        id INTEGER PRIMARY KEY,
        guid TEXT NOT NULL,
        mid INTEGER NOT NULL,
        ...
        flds TEXT NOT NULL,  -- Contains Front and Back separated
        ...
    )
''')
```

Listing 4: Anki database structure creation

The system generates a valid `.apkg` file that includes:

- The database structure (`collection.anki2`).
- Card models (custom CSS for clean visualization).
- Decks organized by subject.

This allows the user to import hundreds of flashcards into Anki with a double click, maintaining formatting and tags.

6.2 Export to Standard Formats

In addition to the Anki format, the system supports export to standard formats such as CSV (Comma-Separated Values) and TSV (Tab-Separated Values). These formats allow maximum interoperability with spreadsheets, databases, and other flashcard management software, offering users flexibility in using the generated material according to their needs.