

# Technical Report: Synapse - Neurosymbolic AI and RAG for Learning

Synapse Group

November 24, 2025

## Abstract

This document provides an in-depth technical analysis of *Synapse*, an intelligent system designed to support university studies through the automatic generation of educational material. The report explores the system architecture, focusing on the integration of **Retrieval-Augmented Generation (RAG)** technologies, **Neurosymbolic** methodologies (Reflection Pattern), integration with web search services, and advanced export functionalities.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>System Architecture</b>	<b>2</b>
2.1	Service Overview . . . . .	2
2.2	User Interface . . . . .	2
<b>3</b>	<b>Neurosymbolic AI: The Reflection Pattern</b>	<b>2</b>
3.1	The Draft-Critique-Refine Cycle . . . . .	3
3.1.1	1. Draft (Draft Generation) . . . . .	3
3.1.2	2. Critique (Critical Analysis) . . . . .	3
3.1.3	3. Refine (Refinement) . . . . .	3
<b>4</b>	<b>Retrieval-Augmented Generation (RAG)</b>	<b>3</b>
4.1	How it works: A Simple Example . . . . .	4
4.2	Recursive Chunking . . . . .	4
4.3	Vector Store and Embeddings . . . . .	4
<b>5</b>	<b>Web Search Integration</b>	<b>5</b>
5.1	Service Operation . . . . .	5
5.2	Fallback Strategy . . . . .	5
<b>6</b>	<b>Export and Interoperability</b>	<b>5</b>
6.1	Anki Package Generation (.apkg) . . . . .	5

# 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.

*Synapse* was created to solve these problems through a hybrid architecture that combines:

- **Neural Component:** The generative capacity and natural language understanding of LLMs (Gemini, Ollama).
- **Symbolic Component:** Logical rules, structural constraints, and deterministic control flows that guide the AI.
- **Knowledge Retrieval:** A RAG system to anchor responses to verified documents.

## 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 neurosymbolic logic for flashcard generation and validation.
- `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**, offering a native and responsive desktop experience. The UI structure (`ui/`) separates presentation logic (`'main_window.py'`, `'subject_window.py'`) from configuration dialogs (`'settings_dialog.py'`).

## 3 Neurosymbolic AI: The Reflection Pattern

One of *Synapse*'s main innovations is the implementation of the **Reflection** pattern. Instead of passively accepting the LLM's output, the system establishes a self-improvement cycle.

## 3.1 The Draft-Critique-Refine Cycle

The flashcard generation process follows three distinct phases, orchestrated by `reflection_service.py`:

### 3.1.1 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.

```
1 prompt = f"""You are an expert in learning and metacognition...
2 Follow these 5 ABSOLUTE RULES:
3 1. Focused: The question must address ONLY ONE concept.
4 2. Precise: It must not be ambiguous.
5 3. Consistent: The answer must be the only correct one.
6 4. Ask "Why": Prefer questions about implications.
7 5. Cognitive Effort: The answer must NOT be intuitive from the question.
8
9 Return the response in JSON format..."""
```

Listing 1: Prompt for draft generation

### 3.1.2 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.

```
1 prompt = f"""You are an expert critic...
2 Evaluate the flashcard EXCLUSIVELY according to these 5 RULES:
3 1. Focused: Does it ask for a single concept? Or is it too broad?
4 2. Precise: Is it ambiguous?
5 3. Context: Is the answer correct and based ONLY on the context?
6 ...
7 Provide CONSTRUCTIVE criticism in 2-3 sentences."""
```

Listing 2: Prompt for critique

### 3.1.3 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 Retrieval-Augmented Generation (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 converts the question into a numerical vector (embedding) 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?"*

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:

```

1 cursor.execute('''
2     CREATE TABLE notes (
3         id INTEGER PRIMARY KEY,
4         guid TEXT NOT NULL,
5         mid INTEGER NOT NULL,
6         ...
7         flds TEXT NOT NULL, -- Contains Front and Back separated
8         ...
9     )
10 ''')
```

Listing 3: 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.