

# The chronicler of Runeterra

In this exercise we will build a RAG system for a small LLM ([Qwen2.5-3B-Instruct](#) will be used) for the purpose of having a model that can answer league of legends lore related questions without hallucinations

The pipeline for this is composed of these steps:

1. Scraping the lore: the official league of legends universe website will be scraped to get official information about champions and regions (for example: Aatrox lore, The void Lore)
2. Creating the initial corpus: we will be using the output of the first step, to create a unified corpus
3. Chunking the corpus: to make your RAG system efficient and useful, chunking your data into meaningful and semantically coherent parts is very important
4. Building the index: now that we have our chunked data, we will convert it into searchable vector database
5. Lore RAG chabot: we will turn the built lore vector index into a working [question-answering chatbot](#)

## Note:

- The tasks will be about the **3rd** and **4th** step. The output data of the first and second step will be provided, feel free to go through the code and understand what was done. The 5th step is mainly about writing a good prompt, it takes some time to find the perfect one, so again feel free to try different prompts in your own time!

## 1. Scraping the lore

Script: [01\\_scrape\\_league\\_lore.py](#)

This step collects lore text from the [League of Legends Universe](#) website by:

- Using two manually prepared lists of identifiers:
  - one for **champions**
  - one for **regions**
- Visiting each corresponding Universe page and extracting the **short lore description** stored in the page metadata.
- Cleaning the extracted text into a consistent, readable format.
- Storing each entry with basic structured info (type, name, slug, URL, and text).
- Adding a small delay between requests to avoid hammering the site.
- Saving the results into **two separate JSON files**:
  - one for champion lore
  - one for region lore

## 2. Creating the initial corpus

Script: [02\\_full\\_corpus.py](#)

This step turns the two scraped JSON datasets into a **single unified corpus** by:

- Loading both previously generated JSON files (champions + regions).
- Converting every entry into a standardized “document format” with:
  - a unique document ID
  - a consistent schema (type, name, slug, URL, source, and text)
- Merging champions and regions into one combined list.
- Running sanity checks to ensure:
  - no duplicate document IDs exist
  - every document follows the exact same schema
  - texts are not suspiciously short (with warnings if they are)
- Saving the final merged dataset into one clean file:
  - [lol\\_universe\\_corpus.json](#)

## 3. Chunking the corpus

Script: [03\\_corpus\\_chunks\\_blank.py](#)

This step takes the unified lore corpus and turns each document into multiple **meaningful text chunks** by:

- Loading a language model (spacy) to understand sentence boundaries and basic linguistic structure.
- Splitting each document into sentences, then extracting lightweight “semantic signals” per sentence, such as:

- important content words (via lemmas)
- named entities
- the main grammatical subject (when detectable)
- Building chunks by grouping sentences together **as long as they stay topically consistent**, using overlap-based similarity checks (so chunks don't randomly mix unrelated ideas).
- Enforcing minimum and maximum chunk sizes so chunks are:
  - not too tiny to be useful
  - not too large to retrieve efficiently
- Giving every chunk a unique ID and storing it with metadata (document origin, type, name, URL, etc.).
- Saving the final output as a JSON chunk dataset (`lol_universe_chunks.json`) that is ready for retrieval.

**Some functions are not yet implemented in the code, you will find guided tasks in the code, follow them to implement the code**

**Note:** as already stated, the chunking step is very important, in the exercise session we will be implementing the approach described above. In your own time, you can try to do pursue another approach

## 4. Building the index

**Script:** `04_build_index_blank.py`

This step converts the chunked lore dataset into a **searchable vector database** by:

- Loading all chunks and extracting only their text content for embedding.
- Using a pretrained sentence embedding model to turn each chunk into a dense numerical vector that captures semantic meaning.
- Normalizing embeddings so similarity search behaves like cosine similarity.
- Creating a FAISS index to store all embeddings and enable fast nearest-neighbor retrieval.
- Saving both:
  - the FAISS index file (for fast search)
  - the chunk metadata file (so retrieved vectors can be mapped back to actual text + source info)
- Running a quick test query to confirm the index works and returns the most relevant lore chunks.

**The main function has some missing parts that you will implement, you will find some guided tasks and some explanations**

## 5. Lore RAG Chatbot:

**Script:** `05_RAG_inference.py`

This script turns the previously built lore vector index into a working **question-answering chatbot** by:

- Loading the FAISS vector index and the stored chunk metadata (the actual lore text + IDs).
- Loading the same embedding model used during indexing, so user questions can be converted into compatible vectors.
- Loading an instruction-tuned language model to generate answers.

For each user question, it:

- Embeds the question and retrieves the **top-K most relevant lore chunks** from the FAISS index.
- Displays which chunks were retrieved (including their IDs and similarity scores).
- Builds an "in-world historian" prompt that:
  - forces the model to write like a Runeterra chronicler
  - forbids modern/meta language
  - requires every claim to be **grounded in the provided context** only
  - stops immediately if the answer isn't directly supported
- Generates the answer in a **streamed** way (token-by-token), so you won't get bored waiting for the whole answer to be generated.