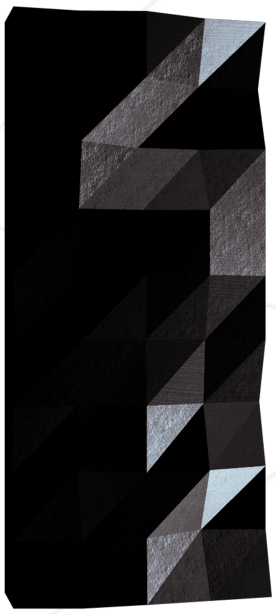

RAG against the machine

Will you answer my questions?

Summary: Retrieval Augmented Generation, that's it.
That's the goal of this project.
Made in collaboration with @ldevelle, @pcamaren, @crfernan

Version: 1.1

Contents

Chapter I

Foreword

The **birthday paradox** is a classic problem in probability theory that demonstrates how counterintuitive probability can be. It shows that even when the probability of an event seems very low, it can occur more frequently than our intuition suggests when there are enough opportunities.

The problem is often presented as follows:

> **In a classroom of 23 students, what is the probability that at least two of them share the same birthday?**

This is a veridical paradox—a statement that appears false but is actually true.
The answer: `50%`! Surprising, isn't it? Here's the formula:

$$1 - (\frac{364}{365})^{n(n-1)/2}$$

Where n represents the number of students. The probability reaches approximately 50% when $n = 23$.

Even more remarkable: with 70 students in a classroom, the probability rises to approximately 99.9% that at least two share the same birthday.

"Interesting trivia, but what does this have to do with the project?" you might ask.
In cryptography, there exists an attack that exploits the birthday paradox to find collisions in hash functions. It's aptly named: **the birthday attack**.

Now that you understand how our intuitions can mislead us and how mathematics can guide us toward brute-force solutions, let's proceed to the project.

Chapter II

AI Instructions

● Context

During your learning journey, AI can assist with many different tasks. Take the time to explore the various capabilities of AI tools and how they can support your work. However, always approach them with caution and critically assess the results. Whether it's code, documentation, ideas, or technical explanations, you can never be completely sure that your question was well-formed or that the generated content is accurate. Your peers are a valuable resource to help you avoid mistakes and blind spots.

● Main message

- ☛ Use AI to reduce repetitive or tedious tasks.
- ☛ Develop prompting skills — both coding and non-coding — that will benefit your future career.
- ☛ Learn how AI systems work to better anticipate and avoid common risks, biases, and ethical issues.
- ☛ Continue building both technical and power skills by working with your peers.
- ☛ Only use AI-generated content that you fully understand and can take responsibility for.

● Learner rules:

- You should take the time to explore AI tools and understand how they work, so you can use them ethically and reduce potential biases.
- You should reflect on your problem before prompting — this helps you write clearer, more detailed, and more relevant prompts using accurate vocabulary.
- You should develop the habit of systematically checking, reviewing, questioning, and testing anything generated by AI.
- You should always seek peer review — don't rely solely on your own validation.

● Phase outcomes:

- Develop both general-purpose and domain-specific prompting skills.
- Boost your productivity with effective use of AI tools.
- Continue strengthening computational thinking, problem-solving, adaptability, and collaboration.

● Comments and examples:

- You'll regularly encounter situations — exams, evaluations, and more — where you must demonstrate real understanding. Be prepared, keep building both your technical and interpersonal skills.
- Explaining your reasoning and debating with peers often reveals gaps in your understanding. Make peer learning a priority.
- AI tools often lack your specific context and tend to provide generic responses. Your peers, who share your environment, can offer more relevant and accurate insights.
- Where AI tends to generate the most likely answer, your peers can provide alternative perspectives and valuable nuance. Rely on them as a quality checkpoint.

✓ Good practice:

I ask AI: "How do I test a sorting function?" It gives me a few ideas. I try them out and review the results with a peer. We refine the approach together.

✗ Bad practice:

I ask AI to write a whole function, copy-paste it into my project. During peer-evaluation, I can't explain what it does or why. I lose credibility — and I fail my project.

✓ Good practice:

I use AI to help design a parser. Then I walk through the logic with a peer. We catch two bugs and rewrite it together — better, cleaner, and fully understood.

✗ Bad practice:

I let Copilot generate my code for a key part of my project. It compiles, but I can't explain how it handles pipes. During the evaluation, I fail to justify and I fail my project.

Chapter III

Common Instructions

III.1 General Rules

- Your project must be written in **Python 3.10 or later**.
- Your project must adhere to the **flake8** coding standard.
- Your functions should handle exceptions gracefully to avoid crashes. Use `try-except` blocks to manage potential errors. Prefer context managers for resources like files or connections to ensure automatic cleanup. If your program crashes due to unhandled exceptions during the review, it will be considered non-functional.
- All resources (e.g., file handles, network connections) must be properly managed to prevent leaks. Use context managers where possible for automatic handling.
- Your code must include type hints for function parameters, return types, and variables where applicable (using the `typing` module). Use `mypy` for static type checking. All functions must pass mypy without errors.
- Include docstrings in functions and classes following PEP 257 (e.g., Google or NumPy style) to document purpose, parameters, and returns.

III.2 Makefile

Include a `Makefile` in your project to automate common tasks. It must contain the following rules (mandatory lint implies the specified flags; it is strongly recommended to try `-strict` for enhanced checking):

- **install**: Install project dependencies using `pip`, `uv`, `pipx`, or any other package manager of your choice.
- **run**: Execute the main script of your project (e.g., via Python interpreter).
- **debug**: Run the main script in debug mode using Python's built-in debugger (e.g., pdb).
- **clean**: Remove temporary files or caches (e.g., ___pycache___, .mypy_cache) to keep the project environment clean.

- **lint**: Execute the commands `flake8 .` and `mypy . --warn-return-any --warn-unused-ignores --ignore-missing-imports --disallow-untyped-defs --check-untyped-defs`
- **lint-strict** (optional): Execute the commands `flake8 .` and `mypy . --strict`

III.3 Additional Guidelines

- Create test programs to verify project functionality (not submitted or graded). Use frameworks like `pytest` or `unittest` for unit tests, covering edge cases.
- Include a `.gitignore` file to exclude Python artifacts.
- It is recommended to use virtual environments (e.g., venv or conda) for dependency isolation during development.

If any additional project-specific requirements apply, they will be stated immediately below this section.

III.4 Overview

A new project is often linked to new techniques and new skills: we've seen **function calling** in *call_me_maybe*, and we will continue our exploration into the world of AI in this project. The main topic we're going to approach is **RAG**. But before seeing what **RAG** is about in its substance, let's focus on what it does! To do so, let's take a step back.

When creating an AI model, one of the first steps is to train it. You want the model to develop skills such as **language understanding, reasoning, and structural analysis**, and to achieve this, you feed it a huge amount of data. After training, the model "remembers" what it has learned, but it only "knows" the data it has been given. If you want it to have more recent knowledge, you must retrain it — a process that takes a long time.

Training is a *technique*, and **RAG** is another one. Instead of feeding the model data directly, RAG gives the model access to an external source of information and that source is of *your choice*.
The two techniques can be combined: the model must still be trained on the key concepts we've mentioned before (language understanding, reasoning, and structural analysis) to build its foundation, but for knowledge, it can combine its trained data with the external connection.

III.5 What is Retrieving Augmented Generation (RAG)?

Now that we know where we're at, you might ask yourself (if you haven't looked it up yet!) what is **RAG**? To understand it, we'll break it down into its three key concepts:

- **Indexing**: Before retrieval, the data must be indexed, this step structures and organises the information to make it searchable later on.
- **Retrieving**: Since the model is not trained on your specific data, it needs to search the database to *retrieve* the most useful snippets. First, the model needs to understand your question. Once that's done, it matches the query with the indexed database to choose the best results and finally pulls out the most relevant pieces of information. This involves **query encoding**, **similarity search**, and **ranking**.
- **Augmenting**: Once the AI has retrieved the information, it can combine it with what it already "knows." However, in most practical applications, we try to rely as much as possible on the retrieved data rather than the model's internal knowledge — since mixing both may lead to outdated or hallucinatory answers. Starting from the retrieved results, you can clean and filter them to remove irrelevant snippets (to avoid potential **noise**), insert them into the **context window**.
- **Generating**: Now that you have retrieved the information and augmented it, the AI can finally generate an answer! Whether it's writing text, explaining a concept, or producing code snippets, this is the visible outcome of RAG. To do so, the AI reads the **context window**, understands the task at hand, blends the knowledge, and generates the output. Modern RAG systems often refine while writing, adjusting phrasing on the fly to maintain coherence and match the tone requested in the query.

Now that everything is clear, let's move forward !

Chapter IV

Common Instructions

IV.1 General Rules

- You must use **Python 3.10** for this project.
- All classes must use `pydantic` for validation and type safety.
- Your project must adhere to the **flake8** coding standard. Bonus files are also subject to this standard.
- Your functions must handle exceptions gracefully to avoid crashes. Use try-except blocks to manage potential errors. If your program crashes due to unhandled exceptions during the review, it will be considered non-functional.
- All resources (e.g., file handles, database connections) must be properly managed to prevent leaks.

IV.2 Additional Guidelines

- You may use any libraries you want, we highly recommend `dspy`, `fire`, `tqdm`, `langchain`, `bm25s`, `chromadb`, `chonkie` packages.
- You need to use the following models:
  - **Qwen/Qwen3-0.6B** (default)
  - You can use other models as long as it is working with **Qwen/Qwen3-0.6B**
- You must use `uv` as a project and package manager.
- Your system must provide a Command-Line Interface (CLI) using Python Fire.
- Progress bars should be implemented for long-running operations using `tqdm`.

Chapter V

Mandatory part

V.1 Summary

In this project, you will build a **Retrieval-Augmented Generation (RAG) system** that can answer questions about a codebase. Specifically, you will:

1. **Ingest** the vLLM repository (provided as attachment) and create a searchable knowledge base

2. **Search** this knowledge base to find relevant code snippets and documentation for given questions

3. **Answer** questions using an LLM (Qwen/Qwen3-0.6B) with the retrieved context

4. **Evaluate** your retrieval system's quality using recall@k metrics

Your system will be tested on its ability to correctly retrieve relevant source code locations when asked questions about the vLLM project, and to generate accurate answers based on the retrieved context.

V.2 What You Must Deliver

You must create a Python application that includes:

V.2.1 Knowledge Base Ingestion System

- Read and process all files from the vLLM repository
- Implement intelligent chunking for Python code and Markdown documentation
- Create a searchable index using TF-IDF or BM25
- Store the index for fast retrieval (maximum 5 minutes indexing time)

V.2.2 Retrieval System

- Implement semantic search over the indexed knowledge base
- Return top-k most relevant code snippets for any query
- Each result must include: file_path, first_character_index, last_character_index
- Support batch processing of multiple questions from JSON datasets
- Achieve at least 65% recall@5 on English questions

V.2.3 Answer Generation System

- Use Qwen/Qwen3-0.6B model to generate natural language answers
- Pass retrieved context to the LLM within token limits
- Generate answers based on the retrieved code and documentation
- Output structured JSON following the provided pydantic models
- Answer questions in maximum 2 seconds per question

V.2.4 Evaluation System

- Implement recall@k metric to measure retrieval quality
- Compare retrieved sources against ground truth annotations
- Calculate overlap between retrieved and correct sources (minimum 5% overlap counts as found)
- Provide detailed performance metrics

V.2.5 Command-Line Interface

- Provide a CLI using Python Fire with these commands:
  - `ingest`: Index the repository
  - `search`: Search for a single query
  - `search_dataset`: Process multiple questions and output search results
  - `answer_dataset`: Generate answers from search results
  - `evaluate`: Evaluate search results against ground truth
  - `answer`: Answer a single question with context
- Include progress bars for long-running operations
- Handle errors gracefully with clear messages

```
Start simple!  Begin with basic TF-IDF or BM25 retrieval and measure
your recall@k score.  Once you have a working baseline with good
metrics, you can experiment with more sophisticated approaches.
```

V.3 Core functionalities

Yes, yes, we know you know. Since you've read the summary carefully, you already understand that you'll be coding a **Retrieval-Augmented Generation (RAG)**.

In that context, the minimum functionalities to implement are:

- Build an indexed knowledge base from the project attached files.
- Retrieve and rank the most relevant pieces of information.
- Pass them to the LLM within context limitations.
- Generate structured JSON output as described in the output section.
- Implement intelligent chunking strategies for the different file types.
- Provide a comprehensive CLI interface for all operations.
- Include evaluation metrics and performance analysis.

```
Dont́ panic !  Start by measuring your error using the simplest
approach.  Advance to complex methods once your error measurement
is improving.
```

V.4 Chunking Strategy

Your program must implement different chunking strategies for the different types of files.

- Python code chunking
- Text chunking

```
The maximum chunk size is 2000 characters and it has to be
configurable through a variable.
```

V.5 Retrieving Method

One of the two following retrieving methods must be implemented, the choice is up to you:

- TF-IDF
- BM25

You can also explore other retrieving methods, as long as one of the two above is implemented.

V.6 Command-Line Interface

Using `Python Fire`, the following commands are required:

- **index**: Ingest and index the repository

  Index command

  ```
  uv run python -m src index --max_chunk_size 2000 \textbackslash
     Ingestion complete! Indices saved under data/processed/
  ```

- **search**: Search the indexed repository

  Search command

  ```
  uv run python -m src search "OpenAI compatible server" -{}-k 10
  ```

  Optional parameters: `k` - number of results to retrieve

- **search_dataset**: Search dataset and output `StudentSearchResults`

  Search dataset command

  ```
  uv run python -m src search_dataset \textbackslash
     -{}-dataset\_path \textbackslash\\
     data/datasets/UnansweredQuestions/dataset\_docs\_public.json \textbackslash\\
     -{}-k 10 -{}-save\_directory data/output/search\_results \textbackslash\\
     \textbackslash\\
  Saved student\_search\_results to \textbackslash\\
  ```

  Optional parameters: `k` (number of results), `save_directory`
  Output: JSON file of type `StudentSearchResults` with filename matching input dataset

- **evaluate**: Evaluate search results

  Evaluate search results

  ```
  uv run python -m moulinette evaluate\_student\_search\_results \textbackslash\\
     -{}-student\_answer\_path \textbackslash\\
     data/output/search\_results/dataset\_docs\_public.json \textbackslash\\
     -{}-dataset\_path \textbackslash\\
     data/datasets/AnsweredQuestions/dataset\_docs\_public.json \textbackslash\\
  \textbackslash\\
  Student data is valid: True \textbackslash\\
  \textbackslash\\
  Total number of questions: 100 \textbackslash\\
  Total number of questions with sources: 100 \textbackslash\\
  Total number of questions with student sources: 100 \textbackslash\\
  \textbackslash\\
  Evaluation Results \textbackslash\\
  ======================================== \textbackslash\\
  Questions evaluated: 100 \textbackslash\\
  Recall@1: 0.450 (45.0\%) \textbackslash\\
  Recall@3: 0.590 (59.0\%) \textbackslash\\
  Recall@5: 0.650 (65.0\%) \textbackslash\\
  Recall@10: 0.720 (72.0\%) \textbackslash\\
  ```

- **answer_dataset**: Generate answers from search results

  Answer dataset command

```
uv run python -m src answer_dataset \textbackslash
    -{}-student\_search\_results\_path data/output/search\_results/dataset\_docs\_public.json \
        textbackslash\\
    -{}-save\_directory data/output/search\_results\_and\_answer \textbackslash\\
\textbackslash\\
Loaded 100 questions from data/output/search\_results/dataset\_docs\_public.json \textbackslash\\
\textbackslash\\
100/100 [02:30<00:00, 1.50s/it]
```

  Input: Output file from `search_dataset` (type `StudentSearchResults`) *Optional parameters*: `save_directory`
  Output: JSON file of type `StudentSearchResultsAndAnswer` with filename matching input dataset

- **answer**: Answer single query with context

  Answer command

```
uv run python -m src answer "How to configure OpenAI server?" -{}-k 10
```

  Optional parameters: `k` - number of results to retrieve

Think of these commands as a base template: you can add extra commands or customize them with flags according to your project's needs.

V.7 Data Models

The following `pydantic` models for type-safe data handling must be implemented. These models ensure data integrity and provide automatic validation throughout the pipeline. The **MinimalSource** model represents a minimal source of information:
MinimalSource Model

```
class MinimalSource(BaseModel):
    file_path: str
    first_character_index: int
    last_character_index: int
```

The **UnansweredQuestion** and **AnsweredQuestion** models represent an unanswered question and an answered question:
UnansweredQuestion and AnsweredQuestion Models

```
class UnansweredQuestion(BaseModel):
    question_id: str = Field(default_factory=lambda:
str(uuid.uuid4()))
    question: str

class AnsweredQuestion(UnansweredQuestion):
    sources: List[MinimalSource]
    answer: str
```

The **RagDataset** model represents a dataset of RAG questions:
RagDataset Model

```
class RagDataset(BaseModel):
    rag_questions: List[AnsweredQuestion | UnansweredQuestion]
```

The **MinimalSearchResults** and **MinimalAnswer** models represent the search results and an answer:
MinimalSearchResults and MinimalAnswer Models

```
class MinimalSearchResults(BaseModel):
    question_id: str
    question: str
    retrieved_sources: List[MinimalSource]

class MinimalAnswer(MinimalSearchResults):
    answer: str
```

The **StudentSearchResults** and **StudentSearchResultsAndAnswer** models represent search results and search results with answers:
StudentSearchResults and StudentSearchResultsAndAnswer Models

```
class StudentSearchResults(BaseModel):
    search_results: List[MinimalSearchResults]
    k: int

class StudentSearchResultsAndAnswer(StudentSearchResults):
    search_results: List[MinimalAnswer]
```

The provided models are a foundation. You can expand them by adding new models or extra fields (for example in the search results model) if your implementation requires it.

V.8 Input

Ingestion Options:

- **Repository**: Index all needed files in the repository
- **Selective Ingestion**: Only process files mentioned in questions.tsv (recommended for testing)

For each query, your system must retrieve relevant chunks of the repository and generate an evidence-based response in the same form as the output.

`Linked to the different chunking strategies, you can create different indexes for the different types of files.`

V.9 Output

The output must conform to the provided Pydantic models and must be a comprehensive JSON file containing detailed results and metadata as follows:

- **For search operations**: Use `StudentSearchResults` model with:
  - `search_results`: List of `MinimalSearchResults` containing question_id and retrieved_sources
  - `k`: Number of results requested
- **For answer generation**: Use `StudentSearchResultsAndAnswer` model with:
  - `search_results`: List of `MinimalAnswer` containing question_id, retrieved_sources, and answer
  - `k`: Number of results requested
- **Source information**: Each `MinimalSource` contains:
  - `file_path`: Full path to the source file
  - `first_character_index`: Starting character position
  - `last_character_index`: Ending character position

Output Format

The output must respect the minimal basis of the provided models but can be enhanced as follows:
Example: StudentSearchResults Output

```
"search_results": [
    {
        "question_id": "q1",
        "retrieved_sources": [
            {
                "file_path": "docs/serving/openai_compatible_server.md",
                "first_character_index": 9867,
                "last_character_index": 10100
            },
            {
                "file_path": "vllm/entrypoints/openai/api_server.py",
                "first_character_index": 267,
                "last_character_index": 400
            }
        ]
    }
],
"k": 10
```

For answers, the output should follow the `StudentSearchResultsAndAnswer` model:
Example: StudentSearchResultsAndAnswer Output

```
"search_results": [
    {
        "question_id": "q1",
        "retrieved_sources": [
            {
                "file_path": "docs/serving/openai_compatible_server.md",
                "first_character_index": 9867,
                "last_character_index": 10100
            },
            {
                "file_path": "vllm/entrypoints/openai/api_server.py",
                "first_character_index": 267,
                "last_character_index": 400
            }
        ],
        "answer": "To configure the OpenAI compatible server in vLLM..."
    }
],
"k": 10
```

Chapter VI

Evaluation

VI.1 Evaluation metrics

The evaluation of the program is performed using a **recall@k** metric that measures the effectiveness of the retrieval component.

VI.1.1 Recall@k Calculation

The recall@k for a given question is calculated by checking how much the retrieved sources overlap with the correct sources. A source is considered "found" if there is at least 5the retrieved source and any correct source. If there are multiple sources in the question, their retrieval score for that question is number_found

VI.1.2 Performances

Your system must respect some minimal performances that are listed as follow:

- **Indexing time**: 5 minutes maximum
- **Cold start latency**: 60 seconds maximum (first retrieval after system startup, including model loading)
- **Warm retrieval throughput**: 90 seconds maximum for 1000 questions (after cold start)
- **Question answering time**: 2 seconds maximum per question
- **Recall@5**: 75% on English questions and 50% on code

Chapter VII

Readme Requirements

A `README.md` file must be provided at the root of your Git repository. Its purpose is to allow anyone unfamiliar with the project (peers, staff, recruiters, etc.) to quickly understand what the project is about, how to run it, and where to find more information on the topic.
The `README.md` must include at least:

- The very first line must be italicized and read: *This project has been created as part of the 42 curriculum by <login1>[, <login2>[, <login3>[...]]].*
- A “**Description**” section that clearly presents the project, including its goal and a brief overview.
- An “**Instructions**” section containing any relevant information about compilation, installation, and/or execution.
- A “**Resources**” section listing classic references related to the topic (documentation, articles, tutorials, etc.), as well as a description of how AI was used — specifying for which tasks and which parts of the project.

➡ **Additional sections may be required depending on the project** (e.g., usage examples, feature list, technical choices, etc.).

Any required additions will be explicitly listed below.

For this project, the `README.md` must also include:

- **System architecture**: Describe your RAG pipeline components and how they interact
- **Chunking strategy**: Explain your approach to document segmentation
- **Retrieval method**: Detail the retrieval algorithm and ranking mechanism
- **Performance analysis**: Discuss recall@k scores and system performance
- **Design decisions**: Explain key implementation choices

- **Challenges faced**: Document difficulties encountered and solutions
- **Example usage**: Provide clear examples of running your system

`Your README must be written in English.`

Chapter VIII

Submission and peer-evaluation

Submit your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double-check the names of your files to ensure they are correct.

Your repository must contain:

- `src/` directory with your implementation
- `pyproject.toml` and `uv.lock` for dependency management
- `README.md` with comprehensive documentation
- Any additional configuration files needed to run your solution

`Do not include large data files, model weights, or generated outputs in your repository. The evaluator will generate these during the evaluation process.`

VIII.1 Recode instructions

During the evaluation, a brief **modification of the project** may occasionally be requested. This could involve a minor behaviour change, a few lines of code to write or rewrite, or an easy-to-add feature.

While this step may **not be applicable to every project**, you must be prepared for it if it is mentioned in the evaluation guidelines.

This step is meant to verify your actual understanding of a specific part of the project. The modification can be performed in any development environment you choose (e.g., your usual setup), and it should be feasible within a few minutes — unless a specific time frame is defined as part of the evaluation.
You can, for example, be asked to make a small update to a function or script, modify a

display, or adjust a data structure to store new information, etc.

The details (scope, target, etc.) will be specified in the **evaluation guidelines** and may vary from one evaluation to another for the same project.