

# Final Report

Jérémy Mathet - Clovis Piedallu

January 17, 2025

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Prompt Engineering</b>	<b>2</b>
<b>3</b>	<b>RAG (Retrieval-Augmented Generation)</b>	<b>3</b>
<b>4</b>	<b>AI Agent</b>	<b>5</b>
<b>5</b>	<b>Conclusion</b>	<b>5</b>

## 1 Introduction

This report outlines the plan to analyze the file structure, document each function, explain the prompting strategy, and describe fine-tuning techniques. The technical report includes the implementation details of the Kaggle submission, focusing on the core components and import structure. The implementation relies on key libraries such as the OpenAI API for LLM interactions, pandas for data handling, the RAG system for context retrieval, and custom agents for Python execution.

So we present the different fine-tuning techniques we used to improve the model's performance. We detail the adjustments made to hyperparameters, regularization strategies, and validation methods employed to optimize the results.

Prompt engineering is a crucial step to obtain precise and relevant responses from the model. We carefully designed the prompts to define a clear output format, allow multiple answers with comma separation, limit to a maximum of two distinct answers, include context awareness instructions, and support Python calculations when necessary. However, we will see that our prompt can surely be improved, which could lead to a better score.

Our RAG system uses a custom class to retrieve relevant documentation chunks, combine multiple chunks for broader context, and maintain the original text structure with separators. This integration allows for providing precise contextual responses using similarity search and retrieval-augmented generation techniques. We tried to use a full reliability documentation, which is actually perhaps too big.

The AI agent is responsible for executing Python code when necessary. It uses the OpenAI function calling format to enable numerical calculations, handle execution results in follow-up prompts, and adjust temperature control for more deterministic or exploratory outputs. This agent plays a key role in managing reliability engineering QCM questions.

## 2 Prompt Engineering

Here is our prompt:

```
You will be given a multiple choice question about reliability engineering.
Choose the correct answer.
You will be given some context followed by "Context".
Use it to reason step-by-step.
```

When performing calculations:

1. Think step-by-step and derive an equation.
2. Generate a python script using primarily the 'reliability' library.  
If needed, import specific distributions using:  
`from reliability.Distributions import [distribution]`.
3. Include error handling in your script using try-except blocks.
4. Print results with `print(QoI)`.
5. Use `execute_python_script` to run the code.

If the calculation fails:

- DO NOT return None or invalid results
- Fall back to theoretical reasoning using the reliability documentation context
- Use approximations if necessary
- Always provide a reasoned answer choice

For reliability library usage:

- Reference the provided documentation
- Prefer reliability library over scipy when documentation shows clear advantages
- Double-check distribution parameter requirements

At the end, use format:

`[Answer] [letter choice]`

For multiple answers: `[Answer] [a], [b]`

Maximum response: 400 words.

This prompt uses several classic prompt engineering techniques, including:

- Chain of Thought (CoT):

The prompt encourages step-by-step thinking to solve calculations. This helps structure thought and improve response accuracy (“Think step-by-step and derive an equation”).

- Few-Shot Learning:

Although the prompt does not provide explicit examples, it guides the user through a series of specific steps to solve the problem. This can be considered an implicit form of few-shot learning where detailed instructions serve as models. Example: “Generate a python script using primarily the ‘reliability’ library.”

- Error Handling:

The prompt includes instructions for handling errors, which is crucial for robust and reliable responses. Example: “Include error handling in your script using try-except blocks.”

- Fallback Mechanism:

If the calculation fails, the prompt asks to revert to theoretical reasoning using the reliability library documentation. This ensures that the user always provides a reasoned answer (theoretically). Example: “Fall back to theoretical reasoning using the reliability documentation context.”

- Specific Instructions:

The prompt gives precise instructions on the use of libraries and distributions, which helps guide the user towards correct and optimized solutions. Example: “Prefer reliability library over scipy when documentation shows clear advantages.”

- Output Formatting:

The prompt specifies the response format, which helps standardize responses and make them easier to evaluate, which was already given.

Our first submission only with the prompt engineering was very poor (quite average, such as the benchmark), so now we tried to use the different methods seen in class.

### 3 RAG (Retrieval-Augmented Generation)

The RAG integration uses a custom RAG class to retrieve relevant documentation chunks, combines multiple chunks for broader context, and maintains the original text structure with separators.

Thus, our RAG class core functions include cheatsheet processing where the document is loaded from markdown, split into semantic chunks, embeddings generated and cached, and similarity

search on query. The search implementation uses cosine similarity between embeddings, top-k retrieval (default k=1), context preservation with headers, and token count optimization. This implementation focuses on maintaining document structure while enabling efficient semantic search through the reliability engineering documentation.

Let's delve deeper into our functions. We used an open-source reliability library, which we extracted into a 20,000-line markdown file. For the code structure, we were inspired by the `simple_rag_from_scratch` file seen in class.

Our RAG (Retrieval Augmented Generation) class is a response generation system augmented by information retrieval, designed to answer questions using a markdown cheatsheet. Here is a detailed explanation of its operation, including embeddings and various parts of the code:

- RAG Class

The RAG class is initialized with several parameters:

`client`: an instance of the OpenAI client to interact with OpenAI APIs. `cheatsheet_path`: the path to the markdown file containing the cheatsheet. `embedding_model`: the embedding model to use for generating text embeddings. `max_chunk_size`: the maximum size of each text chunk in characters. `embeddings_path`: the path to save or load the embeddings.

- Initialization

During initialization, the class loads and processes the cheatsheet by splitting it into chunks, then computes or loads the embeddings for these chunks.

- Splitting the Cheatsheet The `chunk_markdown` method splits the markdown content into chunks of the specified maximum size. It detects section headers to logically structure the chunks.
- Computing Embeddings

Embeddings are vector representations of texts. The `_compute_embeddings` method computes embeddings for a list of texts using the specified model.

```
def _compute_embeddings(self, texts: list, show_progress: bool = True) -> list:
    if show_progress:
        return [self._get_embedding(chunk) for chunk in tqdm(texts,
            desc="Computing embeddings")]
    return [self._get_embedding(chunk) for chunk in texts]
```

The `_get_embedding` method uses the OpenAI client to obtain the embedding of a text.

```
def _get_embedding(self, text: str) -> list:
    response = self.client.embeddings.create(model=self.embedding_model, input=text)
    return response.data[0].embedding
```

- Loading or computing embeddings

The `_load_or_compute_embeddings` method loads embeddings from a file if they exist, otherwise it computes and saves them.

```
def _load_or_compute_embeddings(self) -> list:
    if os.path.exists(self.embeddings_path):
        with open(self.embeddings_path, "r") as file:
            return json.load(file)
    else:
        embeddings = self._compute_embeddings(self._chunks)
```

```

    self._save_embeddings(embeddings)
    return embeddings

```

- Saving embeddings

The `_save_embeddings` method saves the embeddings in a JSON file.

```

def _save_embeddings(self, embeddings: list):
    with open(self.embeddings_path, "w") as file:
        json.dump(embeddings, file)

```

- Computing similarity

The `_compute_similarity` method calculates the cosine similarity between two embeddings.

```

def _compute_similarity(self, query_embedding: list, chunk_embedding: list) -> float:
    return cosine_similarity(query_embedding, chunk_embedding)

```

The `cosine_similarity` function calculates the cosine similarity between two vectors.

```

def cosine_similarity(v1, v2):
    return np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2))

```

- Retrieving relevant chunks

The `get_relevant_chunks` method retrieves the most relevant chunks for a given query by calculating the similarity between the query embedding and the chunk embeddings.

```

def get_relevant_chunks(self, query: str, top_k: int = 1) -> list:
    query_embedding = self._get_embedding(query)
    similarities = [self._compute_similarity(query_embedding, emb) for emb in self._chunk_embeddings]
    top_indices = np.argsort(similarities)[-top_k:][::-1]
    return [self._chunks[i] for i in top_indices]

```

We got some issues with

## 4 AI Agent

## 5 Conclusion