Robinson Chatbot - Project RAG

Kasjan Śmigielski

January 31, 2025

Contents

List of Tables

List of Figures

1 Introduction

The aim of the project was to implement the RAG method for any selected text. The project was created based on the reading: *Robinson Crusoe* telling about the adventures of the titular Robinson, who after a sea disaster becomes shipwrecked on a desert island. He spends many years there, learning to live independently and survive, using his ingenuity and resourcefulness. Before delving into the technical details of the project, it is worth setting the context of the RAG methodology.

Retrieval-Augmented Generation (RAG) is a technique that enables generative AI models to perform information retrieval. It adjusts interactions with large language models (LLMs) so that they can respond to user queries by referencing a specific set of documents. This approach allows the model to supplement information derived from its extensive, static training data with the retrieved documents[1].

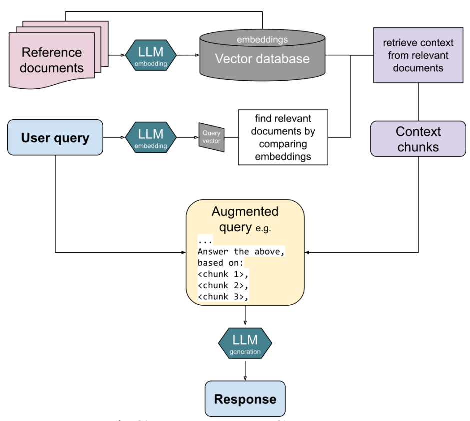

Figure 1: RAG diagram. Source: Wikipedia[1]

To summarize, implementing RAG involves the following steps:

1. **Text preprocessing**:
   - Lemmatization / Stemming.
   - Tokenization.
   - Cleaning the text (removing unnecessary information, e.g. page numbers).
2. **Chunking** - dividing the text into smaller fragments.
3. **Generating embeddings** (both for the resulting chunks and the query) using the LLM model.
4. **Placing embeddings in the vector database**.

[1] Wikipedia, `https://en.wikipedia.org/wiki/Retrieval-augmented_generation`

5. **Retrieving for similar vectors** based on a vectorized query:

6. **Generating context** and passing it as an augmented query to the LLM model:

   - Use of prompt engineering techniques.

7. **Generating response by LLM**:

   - Use of dynamic thresholding.

8. **Verification of response quality** and possible iterative improvement.

The project utilized AWS communication and employing two Amazon LLM models: one for transforming text into embeddings and another for generating responses. Finally, the results were compared using a different LLM model from OpenAI. All the tools and techniques used are discussed in the next chapter.

2 Applied technologies

The following technologies were used for the implementation of RAG:

1. **Boto3 protocol** - for communication with AWS.

2. **Text lemmatization** - by reducing derivative word forms to a single root word, lemmatization assist information retrieval systems and deep learning models in recognizing morphologically related words[2].

3. **Different methods of text fragmentation (chunking)**:

   - Chunking by chapters.
   - Chunking by paragraphs.
   - Chunking by based on a specified number of characters.
   - Chunking using window sliding.
   - Chunking by sentences.

4. **Amazon Titan Text G1 Embeddings** - an LLM model offering functionalities related to natural language processing, enabling the transformation of text (specific chunks) into vector representations in the form of embeddings.

5. **Faiss** - a library developed by Facebook AI Research that is used for efficient similarity search and clustering of dense vectors. It is particularly useful for handling large-scale datasets and is commonly employed in applications like retrieval-augmented generation (RAG) to store and search vector embeddings.

[2] IBM, `https://www.ibm.com/think/topics/stemming-lemmatization`

6. **Cosine Similarity** - in data analysis, cosine similarity is a measure of similarity between two non-zero vectors in an inner product space. It is calculated as the cosine of the angle between the vectors, which is the dot product of the vectors divided by the product of their magnitudes. As a result, cosine similarity depends only on the angle between the vectors, not their size. Cosine similarity values lie within the range $[-1, 1]$[3].

7. **Prompt Engineering** - different methods increasing the precision of queries:
   - Context prompting.
   - Zero-shot prompting.
   - Few-show prompting.
   - Chain-of-thought.

8. **Amazon Titan Text G1 Express** - an LLM model offering generate responses based on extended context (RAG).

9. **OpenAI gpt-4o-mini** - an LLM model used to generate the responses - for comparison with the Amazon model.

All the above tools and methods allowed for the implementation of RAG in the project. The next chapter presents the tests and criteria for selecting specific technologies in the final implementation.

3 RAG implementation

3.1 Communitaction with AWS

To utilize the LLM models available on AWS, the Boto3 library was used for server communication. Auxiliary variables were also introduced to store the names of specific models:

```python
import boto3

client = boto3.client('bedrock-runtime', region_name='eu-
    central-1')

MODEL_ID_RETRIEVAL = 'amazon.titan-embed-text-v1'
MODEL_ID_GENERATION = 'amazon.titan-text-express-v1'
```

[3] Wikipedia, https://en.wikipedia.org/wiki/Cosine_similarity

3.2 Text preprocessing

Then the text was cleaned of unnecessary information (page numbers, information about the author, dedications) and after that the entire text of the reading was lemmatized. For this purpose, the spacy library was used, which supports the Polish language:

```
import spacy

with open("robinson-crusoe.txt", "r", encoding="utf-8") as
    file:
     full_text = file.read() # full_text - before
         lemmatization

nlp = spacy.load('pl_core_news_sm')

doc = nlp(full_text)
lemmas = [token.lemma_ for token in doc]

new_full_text = " ".join(lemmas) # new_full_text - after
    lemmatization
```

3.3 Chunking

The next step was to prepare chunking functions according to the methods mentioned in the chapter 2. All chunking implementations are listed below:

1. Chunking by chapters:

```
def split_text_by_chapters(full_text):

    if 'Rozdzial' in full_text:
        chapters = full_text.split('Rozdzial')
    elif 'rozdzial' in full_text:
        chapters = full_text.split('rozdzial')

    if chapters[0].strip() == '':
        chapters = chapters[1:]
        chapters = ['Rozdzial' + chapter for chapter
             in chapters]

    return chapters
```

2. Chunking by chapters:

```
def split_text_by_paragraphs(full_text, group_size
    =5):

    paragraphs = full_text.split('\n\n')
```

```
    filtered_paragraphs = [paragraph.strip() for
        paragraph in paragraphs if paragraph.strip()
         != '' and 'Rozdzial' not in paragraph]

    grouped_paragraphs = []
    for i in range(0, len(filtered_paragraphs),
        group_size):

        group = '\n\n'.join(filtered_paragraphs[i:i
            + group_size])
        grouped_paragraphs.append(group)

    return grouped_paragraphs
```

3. Chunking by based on a specified number of characters:

```
def split_text_by_fixed_length(full_text, max_length
    ):

    return [full_text[i:i + max_length] for i in
        range(0, len(full_text), max_length)]
```

4. Chunking using window sliding.:

```
def split_text_by_fixed_length(full_text, max_length
    ):

    return [full_text[i:i + max_length] for i in
        range(0, len(full_text), max_length)]
```

5. Chunking by sentences:

```
def dynamic_chunking(text, max_sentences=5):

    sentences = text.split()

    dynamic_chunks = []
    current_chunk = []

    for sentence in sentences:
        current_chunk.append(sentence)
        if len(current_chunk) >= max_sentences:
            dynamic_chunks.append(" ".join(
                current_chunk))
            current_chunk = []

    if current_chunk:
        dynamic_chunks.append(" ".join(current_chunk
            ))
```

```
        return dynamic_chunks
```

3.4 Generating embeddings

After the text was divided into appropriate chunks, `Amazon Titan Text G1 Embeddings` was used to convert these chunks into their numeric representations (embeddings):

```
def get_embeddings(chunks):

    if isinstance(chunks, str):
        chunks = [chunks]

    embeddings = []
    for chunk in chunks:
        request_body = {
        'inputText' : chunk
        }

        response = client.invoke_model(
        modelId=MODEL_ID_RETRIEVAL,
        body=json.dumps(request_body)
        )

        model_response = json.loads(response['body'].
            read())

        embedding = model_response['embedding']
        embeddings.append(embedding)

    return embeddings
```

3.5 Placing embeddings in the vector database

After generating the embeddings, they were placed in a suitable database to enable subsequent searching. For this purpose, the `Faiss` library and its vector indexing capabilities were used.

To enable index search, one of the vector comparison methods had to be used. Therefore, the following were compared: Euclidean comparison and Cosine comparison:

1. Euclidean Distance:

```
def embeddings_to_faiss(embeddings):

    embedding_size = len(embeddings[0])
    index = faiss.IndexFlatL2(embedding_size)
```

```
    embedding_array = np.array(embeddings).
        astype('float32')
    index.add(embedding_array)

    return index
```

2. Cosine Similarity:

```
def normalize_embeddings(embeddings):

    norms = np.linalg.norm(embeddings, axis=1,
        keepdims=True)

    return normalized_embeddings

def embeddings_to_faiss_cosine(embeddings):

    embedding_array = np.array(embeddings).
        astype('float32')
    normalized_embeddings = normalize_embeddings
        (embedding_array)

    embedding_size = len(normalized_embeddings
        [0])
    index = faiss.IndexFlatIP(embedding_size)

    index.add(normalized_embeddings)

    return index
```

When using Faiss, the choice of distance metric does not significantly impact the results because even when using Cosine Similarity, distance normalization is applied beforehand[4].

In narrative texts like *Robinson Crusoe*, is important conceptual and thematic similarities between text fragments. Cosine similarity better captures these subtle connections, while Euclidean distance may be more sensitive to differences in embedding length. Therefore, cosine similarity was ultimately chosen as it aligns more closely with the need to identify semantic relationships in the narrative.

3.6 Retrieving for similar vectors

An index was created to search for `k-Nearest Neighbors (k-NN)`:

```
index = embeddings_to_faiss_cosine(embeddings)
```

An example query was then generated, transformed into a normalized embedding, and passed as an argument to a function that searches the vector database for the most similar embeddings:

[4]Medium, `https://medium.com/@stepkurniawan/comparing-similarity-searches-distance-metrics-in-vector-stores-ra`

```python
index = embeddings_to_faiss_cosine(embeddings)

query_text = "Jakie cechy charakteru pozwolily Robinsonowi
    na przetrwanie na wyspie i co czynilo go wyjatkowym w
    porownaniu z innymi rozbitkami z literatury?" # sample
    query

query_embedding = get_embeddings(query_text)

normalized_query_embedding = normalize_embeddings(
    query_embedding)

def retrieve_knn(normalized_query_embedding, k=5):

    distance, indices = index.search(np.array(
        query_embedding).reshape(1, -1).astype('float32'), k)

    retrieved_chunks = [chunks[i] for i in indices[0]]

    retrieved_similarities = distances[0]

    return retrieved_chunks, retrieved_similarities
```

3.7 Generating response by LLM

The next step was to focus on generating an extended query context. This involved first generating an example query, transforming it into an embedding, and using the k-NN search function to locate the closest vectors:

```python
retrieved_chunks, retrieved_similarities = retrieve_knn(
    normalize_query_embedding, k=5)

augmented_context = query_text + " " + " ".join(
    retrieved_chunks)
```

When creating queries, the prompt engineering methods described in the chapter were used 2:

1. **Context prompting** - this technique involves providing background information or context in the prompt to help the model understand the specific scenario or task better. It sets the stage by supplying relevant details that guide the model's response. Example:

    - Normal query: *Describe how Robinson Crusoe organized his daily life on the island to survive?*
    - Query with context prompting: *After becoming the sole survivor of a shipwreck, Robinson Crusoe had to adapt to life on an unfamiliar island. How did he organize his daily life to survive in such conditions?*

2. **Zero-shot prompting** - in zero-shot prompting, the model is asked to perform a task without being given any specific examples. It relies on its pre-trained knowledge to understand and execute the task directly from the prompt. Example:

   - Query with zero-shot prompting (normal query): *Describe how Robinson Crusoe organized his daily life on the island to survive?*

3. **Few-shot prompting** - involves giving the model a small number of examples within the prompt to demonstrate the desired task or output. It helps the model recognize patterns and apply them in generating responses. Example:

   - Normal query: *Describe how Robinson Crusoe organized his daily life on the island to survive?*
   - Query with few-shot prompting: *Similar to 'Cast Away' where the main character organizes a new life for himself on a deserted island, describe how Robinson Crusoe organized his daily life on the island in order to survive?*

4. **Chain-of-thought** - this method encourages the model to generate intermediate reasoning steps or a sequence of thoughts that lead to the final answer. It helps in tasks requiring logical reasoning and complex problem-solving by breaking down the process into smaller, understandable parts. Example:

   - Normal query: *Describe how Robinson Crusoe organized his daily life on the island to survive?*
   - Query with few-shot prompting: *Robinson Crusoe had to make many decisions to survive on the island. Step by step, how did he organize his daily life, from finding shelter, to getting food, to adapting to new conditions?*

`Amazon Titan Text G1 Express` was used to generate the final response and the following functions were implemented:

```
def generate_text(context):

    model_id = MODEL_ID_GENERATION #

    request_body = {
    "inputText": context,
    "textGenerationConfig": {
        "maxTokenCount": 150, #
        "temperature": 0,
        "topP": 0.01,
        "stopSequences": []
        }
    }
```

```
    response = client.invoke_model(
        modelId=model_id,
        body=json.dumps(request_body)
    )

    model_response = json.loads(response['body'].read())

    return model_response['results'][0]['outputText']
```

Finally, the response was generated by LLM - using dynamic thresholding (the minimum acceptable threshold of the cosine similarity metric was established):

```
# dynamic thresholding (min -> 0.6)
threshold = update_threshold([sim for sim in
    similarity_metrics if sim >= threshold])

if max(retrieved_similarities) < threshold:
    print("Nie znaleziono wystarczajacego podobienstwa, by
        wygenerowac odpowiedz.")
else:
   similarity_metrics.append(max(retrieved_similarities))

    context = query_text + " " + " ".join(retrieved_chunks)

    generated_response = generate_text(context)

    print(f'PYTANIE: {query_text}')
    print()
    print(f"WYGENEROWANA ODPOWIEDZ: {generated_response}")
```

4 Tested solutions

As part of the project, different types of chunking were tested (both before and after lemmatization), and the differences between various types of prompts were examined (based on prompt engineering methods).

4.1 Chunking tests

During chunk verification the following criteria were taken into account:

- number of k-NN (KNN).
- number of groupings - if present, e.g. when chunking by paragraphs $N = 5$, it concerns grouping of 5 paragraphs (N).
- window sliding settings: window size (WS) and step (S) - applies only to chunking using this method.
- number of chunks (NC).

- average number of tokens per 1 chunk (using the first chunk as an example) (FCT).
- cosine Similarity for the first vector found (before lemmatization) (CS).
- cosine Similarity for the first vector found after lemmatization) (CSL).
- embedings generation time (in seconds) (TR).
- response generation time (in seconds) (TG).
- subjective opinion on the correct answer (SO):
  - The opinion was determined using a 1-5 scale:
    * **1**: Answer is imprecise and misleading.
    * **2**: Contains a lot of information, but misses the main point.
    * **3**: Average answer – correct, but lacks details or syntax of the answer is incorrect.
    * **4**: A good answer includes most of the necessary information, but does not take into account the full context of the question.
    * **5**: Very precise and fully contextual answer.

The chunking methods presented in subsection 3.3 were tested on one selected prompt (created using the zero-shot prompting method):

```
query_text = "Jakie cechy charakteru pozwolily Robinsonowi
    na przetrwanie na wyspie i co czynilo go wyjatkowym w
    porownaniu z innymi rozbitkami z literatury?"
```

. In this case the `Amazon Titan Text G1 Express` was used, adopting its target settings:

- maxTokenCount = 200 - which means the maximum number of tokens that will be included in the generated response.
- `temperature = 0.7` - this parameter controls the randomness of the generated text. A lower temperature results in more deterministic outputs, while a higher temperature produces more diverse and creative responses (ranges from 0 to 1).
- `topP = 0.9` - affects the variety of model responses, lower values will cause the model to consider only the most likely tokens, while higher values will allow for more diverse results (ranges from 0 to 1).

Below are the generated answer examples depending on the chunking method used:

1. **Chunking by chapters**:

    “ Robinson był wytrwałym i upartym człowiekiem, który miał silną wolę przetrwania. Był też bardzo kreatywny i nieustępliwy w poszukiwaniu jedzenia i środków do życia. Jego wyjątkowe cechy charakteru obejmowały także umiejętność dostosowywania się do nowych warunków, znajomość roślin i zwierząt na wyspie oraz zdolność do współpracy z innymi rozbitkami, gdy się pojawiają. ”

2. **Chunking by paragraphs**:

    “ Robinson był wyjątkowy w porównaniu z innymi rozbitkami z literatury, ponieważ był bardzo silny i wytrzymały psychicznie, a także miał dużą wiedzę i umiejętności, które pomogły mu przetrwać na wyspie. ”

3. **Chunking by specified number of characters**:

    “ Robinson, oprócz swojej niezwykłej siły fizycznej, charakteryzował się też wielką wytrzymałością psychologiczną i odpornością na stres. Udało mu się poradzić z wieloma przeciwnościami, które napotkał na wyspie, a także utrzymać przy życiu swoją żonę i dzieci. Był też niezwykle inteligentny i praktyczny, co pozwoliło mu na konstruowanie różnych narzędzi i urządzeń, które ułatwiały mu życie na wyspie. Robinson był też człowiekiem bardzo religijnym, który często modlił się i rozważał swoje życie. Jego wytrwałość i upór doprowadziły go do sukcesu na wyspie, a jego wyjątkowa osobowość i cechy charakteru uczyniły go legendą. ”

4. **Chunking using window sliding**:

    “ Robinson był niezwykły, ponieważ był spokojny i opanowany w najtrudniejszych sytuacjach. Hiszpański żeglarz nie miał problemów z zaaklimatyzowaniem się na bezludnej wyspie. Był bardzo silny fizycznie, co pozwoliło mu uporać się z trudnymi warunkami. Robinson był również bardzo sprytny, a jego zdolność do konstruowania narzędzi i budowy schronienia pomogła mu przetrwać. Robinson był też bardzo inteligentny, co pozwoliło mu na naukę języka i obserwację przyrody. Jego cechy charakteru pozwoliły mu na przetrwanie na wyspie, a jego wyjątkowość polegała na spokojnym i opanowanym charakterze, siły fizycznej i sprytu oraz inteligencji. ”

5. **Chunking by sentences**:

    " Robinson był inteligentny, charyzmatyczny, wytrwały, odporny na przeciwności, miał wiedzę na temat survivalu, był inteligentny, charyzmatyczny, wytrwały, odporny na przeciwności, miał wiedzę na temat survivalu, był inteligentny, charyzmatyczny, wytrwały, odporny na przeciwności, miał wiedzę na temat survivalu. "

The test results are summarized in the table below:

Chunking type	KNN	N	WS	S	NC	FCT	CS	CSL	TR	TG	SO
Chapters	1	-	-	-	14	5463	0.59	0.54	5	5	4
Paragraphs	10	5	-	-	212	337	0.72	0.75	22	4	5
Characters	25	500	-	-	509	168	0.72	0.69	55	20	4
Window sliding	10	-	300	100	388	610	0.73	0.70	80	20	4
Sentences	5	20	-	-	127	475	0.73	0.69	15	15	3

Table 1: Results of testing different types of chunking.

After analyzing the data in the table, the following conclusions can be drawn:

- RAG works correctly (chunking, embeddins generation, k-NN search).
- Cosine Similarity is on average from 0.6 to 0.75, so these are satisfactory results
- `Amazon Titan Text G1 - Express` - answers correctly but in most cases does not take into account the full context of the question.
- used LLM model sometimes has a problem with the syntax of the response (e.g. the response is duplicated several times).
- lemmatization helped only with chunking by paragraphs.
- if RAG works well and the answers are not always completely precise, a more accurate LLM model should be used to generate the answers.
- lemmatization helped only with chunking by paragraphs and this method of chunking also achieved the highest similarities among k-NN, which is why **chunking by paragraphs method was considered the most accurate and was used in further tests prompts engineering and other LLM model**.

4.2 Prompts tests

After selecting the optimal type of chunking, the prompt engineering methods listed in the subsection 3.7 were also tested:

1. **Zero-shot prompting**:

```
query_text = "Jak Robinson Crusoe zorganizowal swoje
    codzienne zycie na wyspie, aby przetrwac?"
```

2. **Few-shot prompting**:

```
query_text = "Podobnie jak w 'Cast Away', gdzie
    glowny bohater organizuje sobie nowe zycie na
    bezludnej wyspie, opisz, jak Robinson Crusoe
    zorganizowal swoje codzienne zycie na wyspie,
    aby przetrwac?"
```

3. **Context prompting**:

```
query_text = "Po tym jak zostal jedynym ocalalym z
    katastrofy statku, Robinson Crusoe musial
    przystosowac sie do zycia na nieznanej wyspie.
    Jak zorganizowal swoje codzienne zycie, aby
    przetrwac w takich warunkach?"
```

4. **Chain-of-thought**:

```
query_text = "Robinson Crusoe musial podejmowac
    wiele decyzji, by przetrwac na wyspie. Krok po
    kroku, jak zorganizowal swoje codzienne zycie,
    zaczynajac od zdobycia schronienia, przez
    zdobywanie zywnosci, az po adaptacje do nowych
    warunkow?"
```

During testing prompts, a more detailed model was used: `OpenAI gpt-4o-mini` using previously selected chunking by paragraphs. The following settings were used:

- maxTokenCount = 500.
- `temperature = 0.7`.
- `topP = 0.9`.
- `k = 10` - number of nearest vectors sought.
- `chunks = 212` - number of separated text fragments.
- `tokensPerChunk = 300-500` - average range of tokens per chunk.
- `lemmatization = True` - applying the lemmatization procedure before chunking the text.

When testing prompts the only evaluation criteria were:

- cosine similarity (CS) - the range of this metric for k-NN (from the farthest to the nearest neighbor).
- subjective opinion (SO) - verification of the answer based on readability, consistency and context.

Below are the generated **fragments** of sample answers depending on the prompting method used:

1. **Zero-shot prompting**:

   “ Robinson Crusoe, bohater książki Daniela Defoe, znalazł się na bezludnej wyspie po katastrofie morskiej i musiał nauczyć się organizować swoje codzienne życie, aby przetrwać. Oto jak sobie radził:

   1. Zabezpieczenie podstawowych potrzeb: Po rozbiciu się na wyspie, Robinson musiał przede wszystkim zapewnić sobie schronienie, jedzenie i wodę. Znalazł jaskinię, którą przekształcił w swój dom, a także nauczył się polować na zwierzęta i zbierać owoce. Z czasem zaczął uprawiać rośliny, takie jak jęczmień i ryż, co pozwoliło mu na bardziej zróżnicowaną dietę.

   2. Tworzenie narzędzi i broni: Robinson był zmuszony do samodzielnego tworzenia narzędzi i broni z materiałów, które znalazł na wyspie. Udało mu się skonstruować m.in. łuk i strzały, co ułatwiło mu polowanie. [...] ”

2. **Few-shot prompting**:

   “ Robinson Crusoe, podobnie jak bohater "Cast Away", musiał zorganizować swoje życie na bezludnej wyspie, aby przetrwać. Po katastrofie morskiej, która uczyniła go jedynym ocalałym, Robinson musiał szybko dostosować się do nowej rzeczywistości. Oto, jak zorganizował swoje codzienne życie:

   1. Budowa schronienia: Robinson zbudował sobie schronienie, które nazwał swoim „zameczkiem”. Było to miejsce, w którym mógł się schronić przed dzikimi zwierzętami i niekorzystnymi warunkami atmosferycznymi. Wykorzystał naturalne ukształtowanie terenu oraz materiały znalezione na wyspie, takie jak drewno i liście palmowe.

   2. Zabezpieczenie żywności: Aby przeżyć, Robinson musiał znaleźć źródła pożywienia. Polował na dzikie zwierzęta, łowił ryby i zbierał owoce. Z czasem nauczył się także uprawiać zboża, co pozwoliło mu na pieczenie własnego chleba. [...] ”

3. **Context prompting**:

   “ Robinson Crusoe, jako jedyny ocalały z katastrofy statku, musiał przystosować się do życia na nieznanej wyspie. W pierwszym rozdziale jego opowieści dowiadujemy się o jego pochodzeniu i skłonności do podróży, które zaprowadziły go do obecnej sytuacji. Po rozbiciu statku i cudownym

   W kolejnych rozdziałach dowiadujemy się, jak Robinson przystosowuje się do życia na wyspie. Z czasem zdobywa umiejętności hodowli bydła i uprawy roli, a także staje się biegły w rzemiośle, budując różne potrzebne przedmioty. Pomimo niebezpieczeństw związanych z obecnością ludożerców na wyspie, Crusoe rozwija swoje gospodarstwo i planuje środki obrony.ocaleniu, Crusoe zaczyna organizować swoje codzienne życie, aby przetrwać w trudnych warunkach. [...] "

4. **Chain-of-thought**:

   > " Robinson Crusoe, znany podróżnik i poszukiwacz przygód, musiał podjąć wiele kluczowych decyzji, aby przetrwać na bezludnej wyspie. Jego historia, pełna wyzwań i zmagania z samotnością, zaczyna się od dramatycznego wydarzenia - rozbicia statku, które zmusiło go do znalezienia się w zupełnie nowej rzeczywistości.
   >
   > Po katastrofie, Robinson był zmuszony do szybkiego działania, aby zapewnić sobie podstawowe potrzeby. Pierwszym krokiem było zorganizowanie schronienia. Znalazł odpowiednie miejsce, które mógł przekształcić w swoją tymczasową siedzibę. Używając dostępnych materiałów, stworzył prostą chatę, która miała go chronić przed niekorzystnymi warunkami atmosferycznymi. [...] "

The test results are summarized in the table below:

Prompting method	CS	SO
Zero-shot	0.60–0.72	Specific and correct response, presented in a clear form
Few-shot	0.68–0.79	Even more precise response presented in a clear form
Context	0.63–0.78	Correct, but vague, verbose response
Chain-of-thought	0.65–0.79	Correct response, but but less readable

Table 2: Results of testing different method of prompting.

Tests showed that the most accurate and detailed answers were generated using the **few-shot prompting** method, for which the vector search metric also achieved the highest results.

As a result, this type of prompting, combined with the paragraph fragmentation selected from earlier tests, served as the basis for the next step: creating an application, based on `Streamlit`, that will enable the use of a chatbot for the user.

5 Creating a Streamlit application

This chapter explores the process of developing a simple application using the `Streamlit` library. The focus is on applying RAG technology to a selected text to create a functional chatbot interface.

5.1 Application idea

The goal of the project was to implement RAG technology on a selected text. For this purpose, the text of *Robinson Crusoe* was used, and all implementations were developed in the form of an application resembling a chatbot that answers questions related to the book.

The `Streamlit` library was used - for creating simple applications, enabling intuitive use of available functionalities even for non-technical users. The `Robinson Chatbot` application is designed to enable asking questions related to reading and generating answers through LLM models - based on the provided extended context.

5.2 Main functionalities of the application

The application enables several basic functionalities:

1. **Asking questions in chat** - the user can ask a question related to the content of the reading used.
2. **Using ready-made sample prompts** - the user selects options from a list of specially prepared prompts.
3. **Generating responses by chatbot** - based on the extended context of the question - the AI model generates an answer related to the content of the reading.
4. **Ability to select an AI model** - the user selects the model from the drop-down list that will be responsible for generating the response.
5. **Tracking the costs of using AI models** - the user can check how much individual queries and answers generated by AI cost.
6. **Conversation editability** - the user has the ability to create new conversations, switch between them and change their names.
7. **Giving awareness to the chatbot** - the user can experiment and change the chatbot's personality and observe the impact of individual settings on the generated responses.

All these functionalities are combined into one interactive application that anyone can use - without the need for specialist knowledge.

5.3 Description of individual Streamlit components

Below is a list of the most important `Streamlit` components that enable interaction with the user and the use of the provided functionalities. Questions are marked with a red symbol resembling a *human*, while generated answers are marked with yellow *robot* symbols.

1. **Main chat view**:
   Here the user can send a query and read the response obtained:

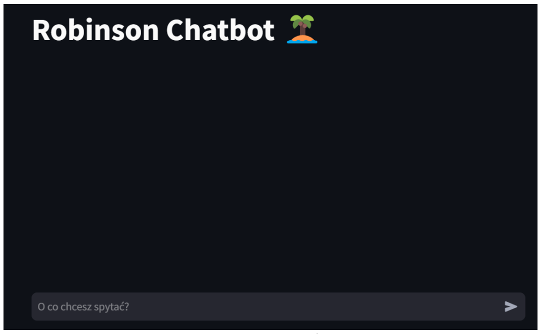

Figure 2: Empty chat view

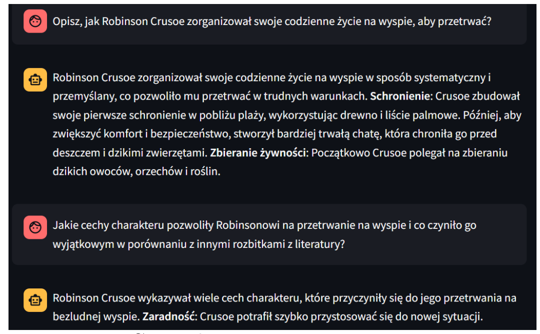

Figure 3: Sample conversation

2. **Sidebar view**:
   Another functionalities are placed on the sidebar, including ready-made prompts and information about the AI model in use. This includes displaying the current model, allowing the user to change the model from a drop-down list, and showing the cost of the conversation in two currencies.

   In addition to this, there are conversation settings such as naming the conversation, a button to start a new conversation, and options to switch between previous conversations. The final element is a window displaying the default chat personality, which can be changed at any time. Below are the visualizations of the listed components:

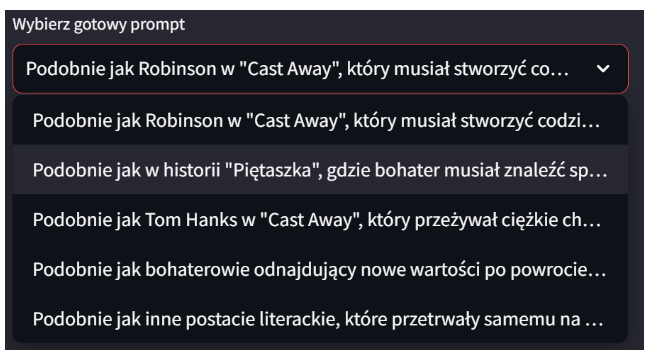

Figure 4: Ready-made prompts

Figure 5: Current model name, usages

Figure 6: Models list

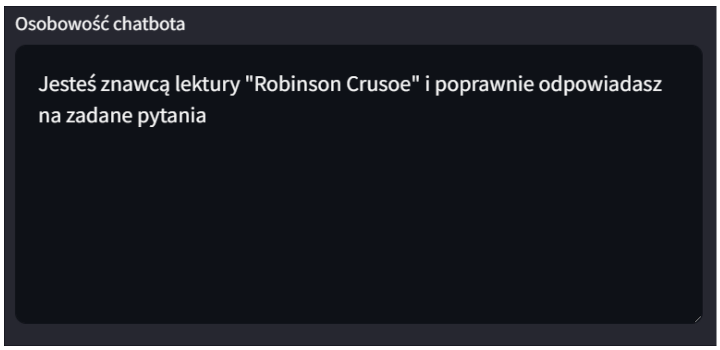

Figure 7: LLM awareness

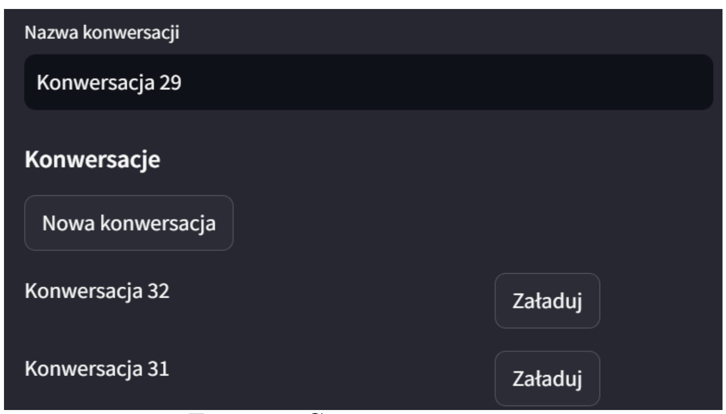

Figure 8: Conversations menu

5.4 Idea for expanding the application

The idea for extending this application is to implement a RAG technology test environment in which the user will be able to:

- Testing the quality of individual text fragments.

- Testing prompt engineering methods and their impact on response accuracy.
- Testing other LLM models.
- Tracking vector comparison metrics (Euclidean Distance, Cosine Similarity).
- The ability to upload any reading text and use RAG technology on it.
- **The possibility of using AI agents to increase the efficiency of generating responses**.

5.5 Summary

To sum up, it managed to implement the RAG technology on the selected text *Robinson Crusoe* and based on it, create an interactive application `Streamlit` - allowing to query AI models about the content of the book.

Ultimately, the application allows you to ask questions and generate contextual answers (based on the principles of similarities used) - which is the foundation of RAG technology. `Streamlit` itself is a perfect combination of the project's backend and frontend, allowing even non-technical people to work with the provided functionalities.

6 AI agents

In order to theoretically imagine the possibility of implementing AI agents into this project, it is necessary to understand how AI agents work.

6.1 How does an AI agent work?

AI agents are designed to make complex tasks simpler and more automated. These autonomous systems generally adhere to a structured process when carrying out assignments:

1. **Defining Objectives**: The agent receives specific goals or instructions from the user and uses these to devise a plan of action. The overall objective is broken down into manageable tasks, which the agent executes following specific criteria or sequences to ensure the final result is meaningful for the user.
2. **Gathering Information**: To effectively carry out the planned tasks, AI agents require relevant information. For instance, to analyze customer feedback, an agent might need to extract dialogue logs. This often involves searching the internet or interacting with other intelligent systems to gather or exchange the necessary data.

3. **Executing Tasks**: Armed with the required information, the agent systematically completes each task. Upon finishing one task, it removes it from the list and moves on to the next. Throughout this process, the agent continually assesses whether the overarching goal has been met by reviewing feedback and its own activity logs. If needed, the agent may generate additional tasks to fully achieve the desired outcome. [5].

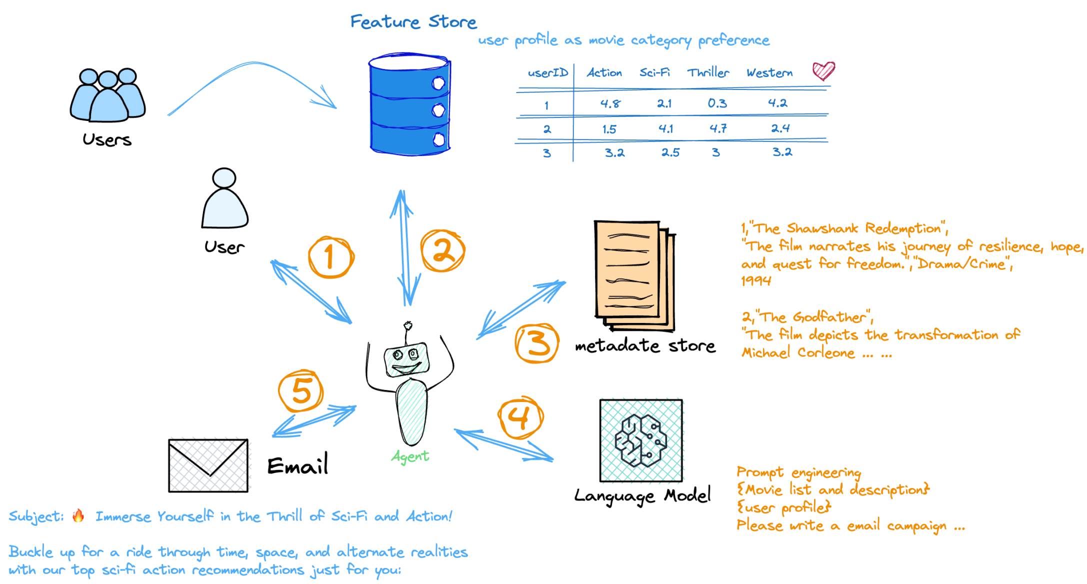

Figure 9: AI agents concept. Source: AWS[5]

6.2 The idea of implementing AI agents into the RAG project

To improve the efficiency and interpretative capabilities of the Robinson Chatbot, AI agents (e.g. available on the AWS platform can be utilized). Constructing an architecture that incorporates two dedicated AI agents with distinct roles contributes to a more comprehensive analysis and a better understanding of the source text.

[5]AWS, `https://aws.amazon.com/what-is/ai-agents/`

6.3 Definition of AI Agent Roles

Based on the content of the reading used, two roles could be introduced:

- **Book Researcher Agent**: Engages in an in-depth search and analysis of the content of *Robinson Crusoe*. Responsibilities include understanding the context, identifying key themes, and clarifying relationships between different sections of the text.
- **Book Critic Agent**: Concentrates on evaluating, interpreting, and offering subjective commentary about the content from a literary critic's perspective. The responses are based on the analyses provided by the `Book Researcher Agent`.

6.4 Task Division

For the most effective use of AI agents, a division of appropriate tasks has been prepared:

- **In-Depth Analyses**:
  - `The Book Researcher Agent` explores the text using RAG technology to gather significant data.
  - `The Book Critic Agent` uses the collected data to formulate interpretations and critical remarks.
- **Broad Thematic Scans**:
  - `The Book Researcher Agent` creates a broad database of themes and motifs.
  - `The Book Critic Agent` maps these themes against historical and literary interpretations.

6.5 Scenario of Dialogue Between Agents

To enable effective dialogue, it is necessary for agents to exchange information in real time. Below is a sample scenario:

- **Book Researcher Agent**:

  > "The predominant motifs of isolation and self-discovery in chapters 1-3 have been identified."

- **Book Critic Agent**:

  > "An analysis of these motifs reveals similarities to Enlightenment literature trends, emphasizing individualism and personal growth."

- **Book Researcher Agent**:

  > "Data also indicate significant symbolism regarding nature as a protagonist's support."

- **Book Critic Agent**:

  > "Interestingly, this aspect can be critically interpreted as an allegory for human dependence on nature."

6.6 Generating the Critic Agent's Responses

The responses of the Book Critic Agent can be delivered directly to the chatbot user as an enhanced literary analysis and critical insights. This approach ensures that users receive not only answers to their questions but also additional context that enriches their experience with the text.

6.7 Summary

The introduction of AI agents with predefined roles has the potential to enhance the quality and depth of the interpretation of *Robinson Crusoe*. By efficiently distributing duties between research-analytical and critical-interpretative agents, a richer and more valuable literary experience can be provided to users.

7 Final Summary and Conlusions

The Robinson Chatbot Project aimed to create an intelligent system capable of providing contextually rich interactions about the novel *Robinson Crusoe*. Utilizing advanced technologies such as RAG (Retrieval-Augmented Generation) and embedding models like `Amazon Titan`, the project successfully demonstrated how AI models can enhance user engagement and understanding of literary works.

7.1 Achievements

During the implementation of the project, the following goals were achieved:

1. **Integration of Cutting-Edge Technologies**: Implementing `Amazon Titan Embeddings` and leveraging both `Amazon Titan Text` and `OpenAI GPT` for response generation highlighted the project's ability to utilize state-of-the-art AI to improve user interaction quality.

2. **Development of a Streamlit Application**: The creation of a user-friendly GUI through the `Streamlit` framework allowed users to easily interact with the chatbot, making the application accessible to a broader audience.

3. **Expanding Contextual Understanding**: By employing RAG, the chatbot was able to provide users with contextually accurate answers, thereby enhancing users comprehension and experience of the novel's themes and narrative to utilize state-of-the-art AI to improve user interaction quality.

4. **Innovative Use of AI Agents**: Although theoretical, the exploration of using AI agents with specified roles like `Book Researcher` and `Book Critic` offers a promising avenue for future development, suggesting potential improvements in analytical depth and interactivity.

7.2 Challenges and Learning Points

While working on the project and implementing individual solutions, many challenges were encountered, which were the most valuable lessons during this implementation:

- **Data Accuracy and Model Limitations**: Ensuring the accuracy and relevance of the information provided while dealing with model limitations proved to be an ongoing challenge, highlighting the need for continuous model evaluation and refinement.

- **User Interface Design**: Crafting an intuitive and efficient user interface required attention to both technical detail and user experience principles, underscoring the importance of usability in technology projects.

- **Interdisciplinary Collaboration**: The project emphasized the value of interdisciplinary approaches, combining technical expertise in AI with literary analysis to produce a well-rounded application.

7.3 Future Prospects

Despite the finalization of the project, there are still many opportunities to expand the functionality and ensure greater efficiency of RAG technology:

1. **Agent Integration**: Future development could involve practical implementation of AI agents with defined roles, enhancing both the depth of literary analysis and the quality of user interaction.

2. **Expanded Literary Database**: Extending the system to include more literary works could broaden its appeal and utility, offering diversified learning opportunities.

3. **Continuous Model Improvement**: Ongoing updates and training of the AI models will be critical to maintain accuracy and relevance as new data and technological advancements become available.

7.4 Conlusion

The Robinson Chatbot Project illustrates the transformative potential of artificial intelligence in enriching literary exploration. By effectively integrating AI technologies, the project not only provided an interactive platform for discussing *Robinson Crusoe* but also paved the way for future innovations in the intersection of technology and literature. The successes and learnings from this project set a strong foundation for ongoing advancement and adaptation in similar ventures.