Development of a scalable and secure RAG-as-a-Service infrastructure

Bachelor Thesis - Spring Term 2025

Department:
Computer Science

Field of Study:
Software Engineering,
Artificial Intelligence,
Networks, Security & Cloud Infrastructure

Authors:

Lukas Ammann
(lukas.ammann@ost.ch)

Sara Ott
(sara.ott@ost.ch)

Advisor: Prof. Dr. Marco Lehmann (marco.lehmann@ost.ch)

External Co-Examiner: Andreas Landerer (andreas.landerer@gmail.com)

Internal Co-Examiner: Prof. Dr. Mitra Purandare (mitra.purandare@ost.ch)

Version: 1.0
Date: 2025-06-05

Abstract

Large Language Models (LLMs) have become very popular with the introduction of chatbots such as ChatGPT or Gemini. LLMs are very good at Natural Language Processing (NLP), which means they have the ability to interpret and communicate in human language. However, they are limited to the knowledge used during training, so it is difficult and resource-intensive to keep them up-to-date and/or to integrate domain-specific knowledge. In addition, LLMs tend to hallucinate and give inaccurate answers when the specific data is not available in the language model.

To overcome these limitations, Retrieval-Augmented Generation (RAG) has been introduced. This novel approach facilitates the incorporation of up-to-date and domain-specific knowledge, while reducing the hallucination of LLMs by providing missing information in a targeted manner. These substantial benefits have led to the popularity of RAG.

One of the most pressing concerns in many RAG implementations is the security and privacy of the data involved, especially when handling sensitive or classified information. Ensuring that data remains within authorized boundaries, maintaining full traceability, and preventing unauthorized data exposure are critical requirements.

To address these challenges, we propose an architectural blueprint and core functionality for a secure and scalable RAG-as-a-Service infrastructure. This design emphasizes local data processing and containment within system boundaries, enabling predictable data flows and robust privacy protection. The system incorporates the security risks and mitigation strategies identified in our prior research, ensuring adaptability and resilience through a modular and customizable core framework. Furthermore, the architecture is designed for seamless scalability and to host multiple systems on a single infrastructure. This makes it suitable for a wide range of use cases and deployment scenarios.

The system's core components were developed using a microservice-based design and deployed via Kubernetes to ensure scalability and adaptability. Security was a central concern throughout the implementation process. In addition to encrypting all external traffic, we integrated a modern authentication solution based on the OAuth 2.0 and OpenID Connect standards to safeguard our RAG system. The resulting platform is fully operational and will be used during our hands-on workshop at the IEEE Swiss Conference on Data Science (SDS2025) on June 26, 2025, at the Circle Convention Center, Zurich Airport. Additional steps included comprehensive system testing and thorough preparation for the upcoming workshop.

Keywords: Retrieval-Augmented Generation (RAG), RAG Security, RAG-as-a-Service, Data Security, Privacy, Scalable Architecture, Secure AI Systems, Local RAG Pipeline, Large Language Models (LLMs), Natural Language Processing (NLP), Microservice Architecture, Docker, Kubernetes, Workshop, SDS2025, IEEE

Management Summary

Introduction

Retrieval-Augmented Generation (RAG) systems are valuable tools for enhancing AI language models by incorporating real-time, specialized, or domain-specific knowledge. This integration improves accuracy and relevance, reduces hallucination, and unlocks numerous business applications such as improving customer service, streamlining operations, and supporting decision making. By increasing productivity, RAG systems can deliver significant financial benefits. While implementation can be straightforward thanks to frameworks, complexity rapidly increases with custom requirements.

Problems

The deployment of RAG systems introduces significant security and privacy challenges. These systems often process sensitive data, raising concerns around unauthorized access, misuse, and operational vulnerabilities. Without proper safeguards, organizations face the risk of regulatory penalties, reputational damage, and financial loss.

RAG systems require specialized components, such as Large Language Models (LLMs) and vectorization services. Due to the significant computational power and complex infrastructure needed to support these systems, there is a strong temptation to rely on third-party hosted solutions. However, doing so can lead to reduced traceability of the data flow and further introduce operational risks.

Building and maintaining a secure and fully self-hosted RAG system is a non-trivial endeavor. It requires deep expertise across various technical domains and a significant investment of time and resources to ensure secure and reliable operation. This challenge is particularly pronounced for small and medium-sized enterprises, which may lack the necessary infrastructure, personnel, or budget to implement such systems.

Objectives

The objective of this thesis is to design and develop a secure and scalable RAG-as-a-Service infrastructure. This includes designing and describing the system's overall architecture, implementing its core functionalities, and analyzing a related topic of interest. Finally, the resulting infrastructure is intended to be ready for use during our workshop at the IEEE Swiss Conference on Data Science (SDS2025) on June 26, 2025, at the Circle Convention Center, Zurich Airport.

Methodology

To support effective planning and informed decision-making, we conducted a comprehensive evaluation of various architectural approaches, frameworks, and tools. Based on this analysis, we developed an architectural blueprint centered on a microservice architecture.

The microservice approach enables us to meet the scalability requirement by allowing individual components to scale independently and efficiently using Kubernetes, a proven container orchestration platform.

To ensure security, we leverage insights from our previous research, including the thesis "Analysis of Risks and Mitigation Strategies in RAG" and the paper "Securing RAG: A Risk Assessment and Mitigation Framework". Building on this foundation, we apply industry-standard security measures to provide robust protection for our RAG system.

Results

The system and its components have been implemented in alignment with the architectural blueprint. This encompasses the creation of a scalable, modular infrastructure built on containerization. Each component is either stateless or handles state management appropriately, with well-defined interfaces facilitating clear communication. To establish a strong security foundation, a robust authentication system was implemented using industry-standard protocols, including OAuth 2.0 and OpenID Connect. Furthermore, all external communications are encrypted to ensure the protection of data in transit.

For our workshop, a three-system approach has been set up, showcasing the system's capability to operate in an as-a-Service model. Additionally, tasks addressing security-related topics and mitigation strategies were planned and prepared for participants. The RAG infrastructure was also evaluated for potential risks and corresponding mitigation strategies — further reinforcing the project's strong emphasis on security.

By consolidating components, the resulting system enables the efficient use of hardware resources for computationally intensive tasks across multiple RAG pipelines, while ensuring strict data separation to maintain privacy. This makes it particularly valuable for small and medium-sized enterprises, offering a secure and private RAG environment without the complexity of in-house development and maintenance. The project also lays the groundwork for broader adoption of scalable, privacy-centric RAG architectures and informs ongoing research in this field. Moreover, the implemented infrastructure supports our upcoming workshop at the IEEE Swiss Conference on Data Science (SDS2025), titled "Hacking RAG: Exploring Risks and Implementing Mitigations," scheduled for June 26, 2025, at the Circle Convention Center, Zurich Airport.

Outlook

This thesis introduces a scalable and secure RAG-as-a-Service infrastructure. While the core components have been implemented, the system is designed with flexibility in mind, allowing for future extensions and adaptations based on specific use cases.

The upcoming workshop will serve as a valuable opportunity to gather insights and direct feedback, helping to evaluate the infrastructure's real-world readiness and inform further development.

Further developments are on the horizon — stay tuned.

Acknowledgements

The successful completion of this thesis was made possible by the invaluable support and contributions of many individuals. We would like to express our sincere gratitude to all those who assisted with this work, with special recognition to the individuals highlighted below.

First and foremost, we would like to express our deep appreciation to our advisor, Prof. Dr. Marco Lehmann, for his invaluable support, guidance, and mentorship throughout this journey. His insightful suggestions and constructive feedback greatly contributed to our progress. We are especially grateful for his initiative in organizing a workshop at the SDS2025, as well as for his ongoing support and collaboration.

We would also like to thank Prof. Dr. Mitra Purandare, our co-reader, for her support throughout the thesis process and for her valuable feedback at our mid-term presentation.

Additionally, we would like to thank Jan Untersander from the Eastern Switzerland University of Applied Sciences in Rapperswil for his constructive and straightforward collaboration, particularly regarding the provision and support of the Kubernetes system used during this project.

Last but not least, we would like to thank you. Thank you, the reader of this thesis, for taking the time to read what we have created over the past semester.

Table of Contents

Glossary

***gRPC* – Google Remote Procedure Call**: High-performance, open-source remote procedure call (RPC) framework that uses HTTP/2 for transport and protocol buffers for interface definition.

***HTTP* – Hypertext Transfer Protocol**: Protocol used to transfer data in networks.

***Ingress* – Kubernetes Ingress**: Kubernetes resource that defines rules for routing external HTTP and HTTPS traffic to services within the cluster. Managed by an Ingress Controller.

***JWT* – JSON Web Token**: Compact, URL-safe token format used to securely transmit information between parties as a JSON object, commonly used for authentication and authorization.

***K8s* – Kubernetes**: Open-source container orchestration platform for automating deployment, scaling, and management of containerized applications.

***MVP* – Minimum Viable Product**: Product with the minimum set of features necessary to satisfy early adopters and gather feedback for future development.

***NIST* – National Institute of Standards and Technology**: U.S. agency that develops standards and guidelines to promote innovation, including widely used frameworks in science, technology, and cybersecurity.

***OAuth 2.0* – Open Authorization 2.0**: Authorization framework that enables third-party applications to obtain limited access to user accounts on an HTTP service.

***OIDC* – OpenID Connect**: Authentication layer built on top of OAuth 2.0 that allows clients to verify the identity of end-users based on the authentication performed by an authorization server.

***OWASP* – Open Worldwide Application Security Project**: Non-profit organization with the aim of improving the security of applications, services and software in general.

***PII* – Personally Identifiable Information**: Information that can be used to identify an individual, such as a name, email address, or phone number.

***SSH* – Secure Shell**: Cryptographic protocol that provides secure remote login and other network services over an unsecured network.

***SSO* – Single Sign-On**: Authentication process that allows users to access multiple applications or systems with one set of login credentials.

***UBAC* – User-Based Access Control**: Access control model that grants or restricts access to resources based on the identity of individual users.

***UML* – Unified Modeling Language**: Standardized modeling language used to visualize, specify, construct, and document the artifacts of a software system.

Part I
Introduction

1. Initial Situation

Large Language Models (LLMs) have become incredibly popular with the introduction of chatbots such as ChatGPT or Gemini. LLMs are very good at natural language processing (NLP), which means they have the ability to understand and communicate in human language [1]. However, as good as this may sound, LLMs also have their limitations. LLMs are limited to the knowledge used during training, and it is difficult and resource-intensive to keep them up to date and/or to integrate domain-specific knowledge for several reasons briefly outlined below.

The training of these so-called foundation models consists of several steps, including basic training of the model, fine-tuning for specific tasks, and aligning the model to desired behaviors and goals. Each of these steps presents their own challenges, including scaling laws in training [2], fine-tuning to ensure optimal performance [3], and further considerations to align the model [4], [5], making this process time and resource intensive. In addition, LLMs tend to hallucinate when required information is not remembered in the model, leading to inaccurate responses [6]. To address these problems, Retrieval-Augmented Generation (RAG) has been introduced [7].

Retrieval-Augmented Generation (RAG) is a promising technique for extending the capabilities of Large Language Models (LLMs) by integrating information retrieved by a RAG system. With this approach, new data (not seen by the LLM), domain-specific knowledge, or other information can be selectively incorporated into the query sent to the LLM, resulting in more recent, more accurate, and more relevant responses. This information could include current news, internal business documents, weather information, and much more.

1.1. Preliminary Work

Of course, the adoption of RAG systems sounds very promising, and it is, but at the same time it introduces new security risks that need to be addressed. As this is a relatively new topic, it can be quite tedious to get an overview of the risks and mitigation strategies as they are spread across many sources and each risk and mitigation strategy found needs to be evaluated individually to determine if it applies to one's RAG system.

For this reason, in our study thesis [8], we reviewed and organized different risks and mitigation strategies regarding Retrieval-Augmented Generation (RAG) systems. The resulting thesis provides a high-level overview of the actual security risks associated with RAG systems, as well as possible mitigation strategies. Furthermore, it helps to assess whether the mentioned risks and/or mitigation strategies apply to one's own RAG implementation. It therefore reduces the effort required to evaluate the risks and mitigation strategies, encouraging the reader to evaluate their own RAG pipeline accordingly.

The research foundation of our study thesis also provides the basis for our paper, Securing RAG: A Risk Assessment and Mitigation Framework [9]. The paper synthesizes the knowledge from our study thesis and integrates it into a broader view by incorporating additional information and offering a concise overview of the environment. The following excerpt from the abstract offers further insight into the paper.

> *This paper reviews the vulnerabilities of RAG pipelines, and outlines the attack surface from data preprocessing and data storage management to integration with LLMs. The identified risks are then paired with corresponding mitigations in a structured overview. In a second step, the paper develops a framework that combines RAG-specific security considerations, with existing general security guidelines, industry standards, and best practices. The proposed framework aims to guide the implementation of robust, compliant, secure, and trustworthy RAG systems.*
>
> *— [9]*

1.2. Resulting Steps

Due to the positive feedback and strong interest in the topic, we have decided to build upon it and focus our bachelor's thesis on the preliminary work of our study thesis. As a result, certain sections of this thesis, particularly those on related work and the foundational description of RAG systems, are adapted directly from our study thesis.

The goal of this thesis is to develop a scalable and secure RAG-as-a-Service infrastructure including the following key deliverables:

- Design and description of a scalable and secure RAG system architecture.
- Implementation of the core functionalities of the RAG system using an appropriate technology stack.
- Identification and investigation of a related topic of interest.

Furthermore, a workshop for the IEEE Swiss Conference on Data Science [10] will be prepared based on the previous work and the work of this thesis. Therefore, a bonus goal is that the implementation built during this thesis can be reused for our workshop.

2. Related Work

As the field of Retrieval-Augmented Generation (RAG) systems emerges, numerous papers, theses, and surveys have been published recently, reflecting the growing interest and activity in this area. The research diverges into several distinct directions, addressing both the capabilities and the security of RAG systems.

2.1. Capabilities

The concept of RAG was introduced by Lewis et al. in 2020, who proposed a hybrid model that combines pre-trained generative models with non-parametric memory through a retrieval mechanism [7]. The integration of the proposed retrieval process leads to significant improvements in overcoming the common limitations of LLMs, including the ability to incorporate current and domain-specific knowledge into the generation task.

Subsequent research has focused on refining RAG architectures to improve efficiency and accuracy. The concept of RAG has been taken up and developed further by many researchers in several directions. Surveys such as [11], [12], [13] provide a broad overview of the evolution of RAG.

2.2. Security

An important direction of research has focused on RAG attacks, exploring the various ways in which malicious actors can exploit these systems. Several studies have explored this topic in depth, including [14], [15], [16], and [17], which investigate various attack methods such as embedding inversion, knowledge corruption, and manipulation of system responses.

Another important topic, highlighted by [18] and further discussed in [17] and [19], is the disclosure and leakage of retrieval data. This includes unintended sharing of retrieval data with third parties, embedding exposure, and broader data leakage issues within RAG pipelines. These vulnerabilities can compromise sensitive information, posing risks to both individual privacy and organizational security. Research in this area emphasizes the need for robust safeguards to ensure that retrieval data remains protected throughout the system's lifecycle.

A further area of interest is jailbreaking RAG and LLM systems, where the goal is to bypass their intended constraints to perform unauthorized actions. Notable works such as [20], [21], [22], [23] and [24], provide comprehensive analysis and techniques for understanding and preventing these vulnerabilities.

These various research directions, together with legislative initiatives such as the EU AI Act [25] and the US Executive Order on AI [26], have attracted considerable attention and underscored the need for robust tools to ensure the safe, transparent, and fair application of AI systems. Other efforts in this direction include technical frameworks such as the AI Risk Management Framework (AI RMF) [27] and the OWASP Top Ten for LLM Applications [28], both of which go beyond RAG but also include some risks and mitigation strategies that also applies to RAG systems.

Last but not least, our study thesis [8] and our paper [9] that builds on this work present a practical framework tailored to RAG systems that facilitates a critical review of one's own RAG system for security vulnerabilities and mitigation strategies.

3. Retrieval-Augmented Generation (RAG)

The purpose of this chapter is to provide a basic understanding of RAG systems. This knowledge is a prerequisite for the understanding of the whole thesis.

3.1. General Architecture

The general architecture of a RAG system is described below and illustrated in Figure 1 in the form of a pipeline. This pipeline is intended to illustrate the basic idea of RAG and can be extended at almost any step, depending on requirements and the specific implementation. The most common places of extensions are shown as dotted boxes. These components are not mandatory, but are used in many RAG systems to increase security or improve results. The different components are briefly described below in a technology-neutral way.

1) General RAG Pipeline: The general RAG pipeline component represents the entire system, encapsulating all processes from user input to post-processing.
   a) Input Pre-processing: This step is labeled as a supplemental step because input pre-processing is not strictly necessary. However, in many RAG systems, security and response results can benefit greatly if appropriate pre-processing steps on the input are taken. Examples of this step include PII masking, prompt pattern matching, metadata enrichment, and more.
   b) Post-processing: This step is labeled as a supplemental step because post-processing is not strictly necessary. However, in many RAG systems, security and response results can benefit greatly if appropriate post-processing steps are taken. Examples of this step include response verification, adaptive learning, error analysis, and more.

2) Data Ingestion: The data ingestion steps do not belong directly into the pipeline in the sense that these steps do not need to be performed on every user request. These steps only need to be performed when new data is added to the RAG system. This is indicated by the dashed arrow in the figure.
   a) Dataset: The dataset contains all the data that is intended to be used in the RAG process to generate responses based on it.
   b) Data Pre-processing: Pre-processing in this context is the process of transforming the data from the dataset into a suitable form that can be used by the retriever. The data in the appropriate form is then stored in the retrieval datastore. A common procedure is to split the data into chunks and create a vector representation for each chunk so that similar chunks can be quickly found later based on a user prompt.

3) Retriever: The retriever searches for relevant data related to the user's prompt and passes it to the generator.
   a) Retrieval Datastore: The retrieval datastore contains the dataset in an optimized form to support the process of finding the most relevant documents. The retrieval datastore typically contains a vector representation of the data.
   b) Retrieve Most Relevant Documents: This step searches for the most relevant documents in the retrieval datastore and returns them as "retrieved information".
   c) Re-Ranker: This step is labeled supplemental because re-ranking is not strictly necessary. Nevertheless, many RAG systems use this well-known step, which can lead to greatly improved results. Re-ranking involves reordering the initial set of retrieved documents to make them more relevant to the user's prompt.
   d) Create Query: In this step, the prompt and the retrieved information are merged and passed to the generator as a "query". More technically, this can be thought of as the following, where {} represents a placeholder.
   $\text{query} = \{\text{prompt}\} + \{\text{retrieved documents}\}$.

4) Generator: The generator creates the response using the prompt along with the data retrieved from the retriever.
   a) Generate Response (LLM): In this step, the response is generated based on the received query. This is done using a Large Language Model (LLM).

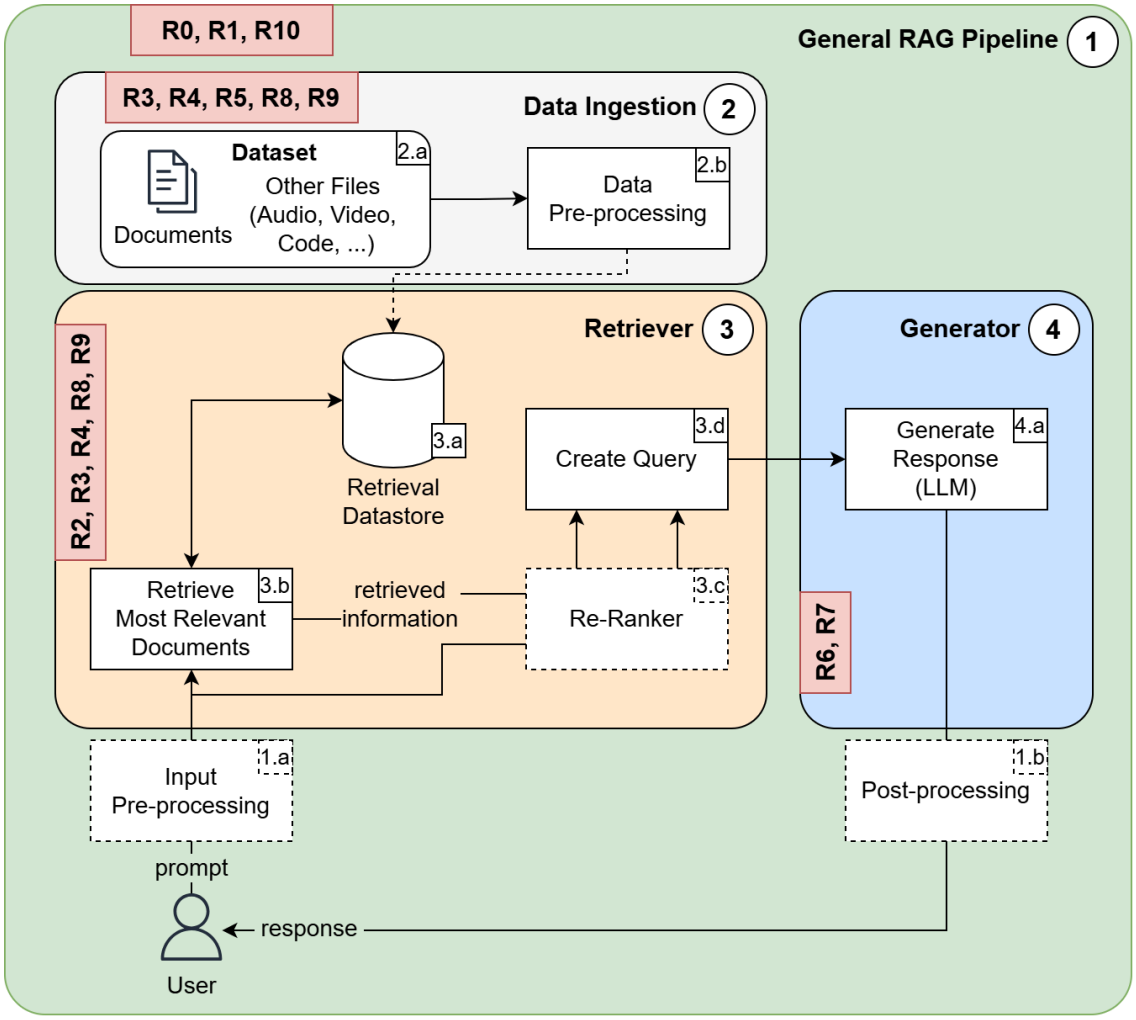

Figure 1: Basic Structure of a RAG System - source: [9]

3.2. Workflow

The basic workflow of a RAG system, as illustrated in Figure 2, can be divided into the following steps [13].

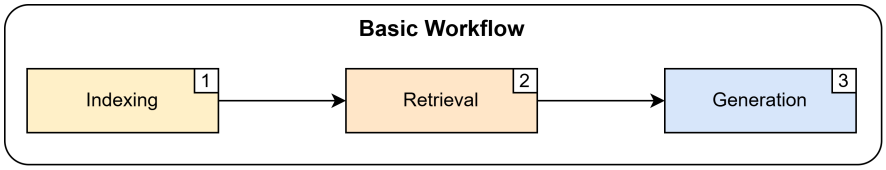

Figure 2: Basic RAG Workflow - adapted from [13]

1) Indexing: This step includes text normalization as well as text segmentation into sentences or paragraphs. This step may include deep learning to generate a semantic vector representation of the text.
2) Retrieval: In this step, the most appropriate documents/paragraphs/phrases are retrieved and prepared for generation. Traditional methods focus on term frequency and are therefore straightforward to implement, but may overlook semantic information. More modern strategies use language models for this task and can therefore capture the semantic meaning more accurately, but are also more complex.
3) Generation: This task generates the response text. This text should be relevant to the query and reflect the information from the retrieved documents.

In a more detailed manner, RAG can also be broken down into the following steps called Pre-Retrieval, Retrieval, Post-Retrieval and Generation [13]. Figure 3 shows some example tasks for each step, further described below. However, these tasks may vary in each specific RAG implementation.

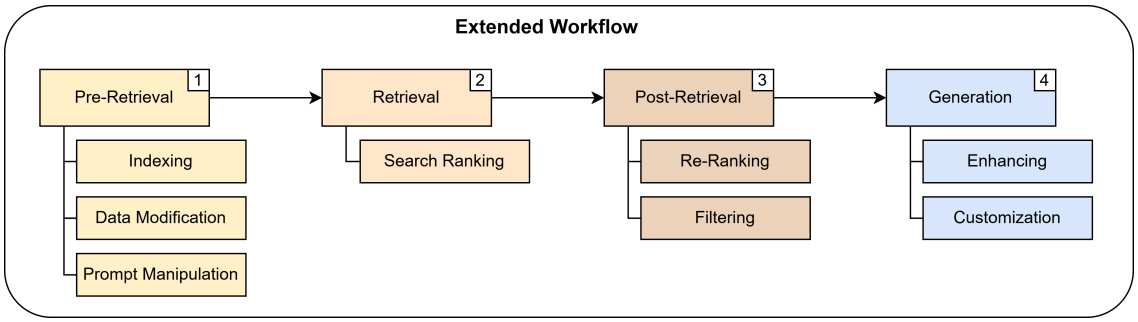

Figure 3: Extended RAG Workflow - adapted from [13]

Here is a brief introduction to each of the steps shown.

1) Pre-Retrieval: This step is essential for successful data and prompt preparation. It can include tasks such as Indexing, Data Modification, and Prompt Manipulation, all of which prepare the system for effective data access.
2) Retrieval: This step includes the search and ranking part. The goal is to search, select and prioritize documents from the datastore. This step should help to improve the quality of the generator's response.
3) Post-Retrieval: This step aims to refine the documents retrieved in the previous steps and to re-evaluate, score and reorganize them. After this step, the documents should be ready for the generation step. Tasks like Re-Ranking and Filtering belong to this step.
4) Generation: This step generates the response text based on the prompt and the retrieved data. It includes steps such as Enhancing, where the retrieved information is merged with the user's prompt, and Customization, which aims to generate user-centric responses.

3.3. Benefits

Some general benefits of RAG systems are as follows. Note that this may vary depending on the exact implementation:

- Improved accuracy: The accuracy of responses can be improved through more recent and domain-specific knowledge. [29]
- Reduce hallucination: LLMs tend to hallucinate due to outdated information or lack of knowledge in a domain. RAG can effectively counter this by passing specific information to the generator. [30]
- Cite sources: Since the information passed to the generator is known, the source can also be added to the response, leading to higher trustworthiness. [30]
- Simpler maintenance: LLMs require retraining whenever new information is added to them. RAG dramatically simplifies this process by only requiring new information to be added to the retriever.

3.4. Approaches

There are several different approaches to implementing a RAG system. Four different approaches are shown in Figure 4 and briefly described below [11].

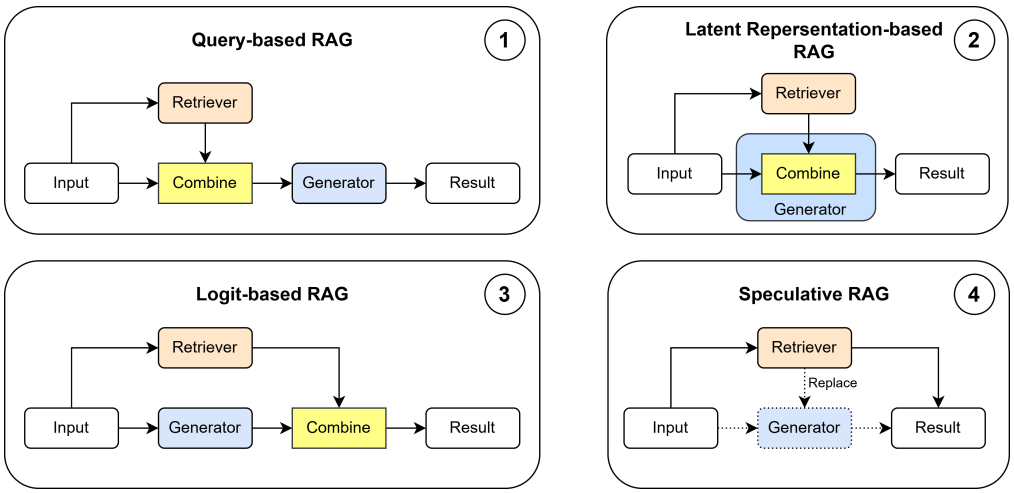

Figure 4: Implementation Approaches for RAG Systems - adapted from [11]

1) Query-based RAG: This is the simplest type of RAG implementation. The user's prompt is concatenated with the retrieved information to form the query (query = {prompt} + {retrieved documents}). This query is then fed into the generator to generate the response. [11]
2) Latent Representation-based RAG: The retrieved information (based on the user's prompt) is transformed into latent representations and fed directly into the internal layers of the generator. This helps to overcome limitations such as the context window, but requires additional training to match the hidden state of the LLM. [11]
3) Logit-based RAG: The retrieval information is only integrated into the probability distribution (logits) at the decoding stage. The logits (from the LLM) are compared against the retrieval information, and the most relevant information is integrated into the response. [11], [31]
4) Speculative RAG: To save resources and minimize response time, retrieval is used instead of generation in certain steps. The retriever and the generator are decoupled, allowing pre-trained models to serve as modular components [11]. In some implementations, the LLM generates multiple hypotheses, which are later evaluated against relevant retrieved information to support or refute each hypothesis. The final response is then generated based on this information [31].

3.5. Use Cases

There are numerous applications for RAG. The following examples are provided for inspiration.

- Question-answering based on internal documents
- Academic research including current literature
- Code generation based on existing code in your domain
- Fact-checking based on self-provided and actual facts
- Medical information systems using own patient data
- Legal information systems including recent court decisions
- and many more ...

Sources: [11], [31]

3.6. Limitations

The following are some limitations of RAG systems:

- Lengthy context: Especially the Query-based RAG is heavily dependent on the context window of the generator. The query plus the retrieved information must not exceed the context window of the generator. [11]

$$\{\text{prompt}\} + \{\text{retrieved information}\} \leq \text{generators context window}$$

- Increased complexity: The complexity of the system inevitably increases with the integration of RAG. The number of hyperparameters increases and more expertise is needed to tune the generator. [11]
- Gap between Retriever and Generator: Implementing the interaction between the retriever and the generator may require very precise design and optimization. [11]
- Overhead: In most cases, an overhead is introduced that leads to an increase in latency. [11]
- Bad/biased data: The quality of the results from the retriever is only as good as the data fed into it. If the data is inaccurate, outdated, or biased, the RAG system will have a difficult time identifying and improving it. This leads to poor retrieval results, which can lead to poor overall results. [11], [32]
- Multiple aspects of data: Traditional RAG systems do not focus on queries that may require the retrieval of multiple documents with significantly different content. [33]
- Complex questioning: Traditional RAG systems may struggle to synthesize and provide comprehensive answers because documents are retrieved and processed individually. [34]
- Imperfect retrieval: Irrelevant, misleading or conflicting information can degrade system performance. [35]
- Hallucination: RAG systems can hallucinate. They may provide convincing yet incorrect answers. [36]
- Assessment: The evaluation and benchmarking of RAG systems is not a trivial matter. [37], [38]

Part II
Planning

1. Requirements

This chapter presents the functional and non-functional requirements for our Retrieval-Augmented Generation (RAG) system. It defines the roles of the primary system actors and outlines the specific actions that each actor performs. The goal is to formally capture the behavior and expectations of the system to guide the subsequent design, implementation, and validation phases.

1.1. Actors

The system involves two primary actors: the User and the RAG System. The User initiates interactions by performing actions such as uploading documents and prompting the chatbot, while the RAG System handles background processes such as document processing, chunk retrieval, and response generation.

- User
  - Login in to the RAG application.
  - Upload PDF file(s).
  - Ask chatbot questions about their PDF file(s).
  - View passed chat messages (chat history).
- RAG System
  - Chunk PDF file(s) and generate embeddings based on PDF chunks.
  - Retrieve relevant chunks from the vector store.
  - Generate responses using retrieved chunks.
  - Provide citations for the generated responses.
  - Store chat history for the user.
  - Return generated responses.

1.2. Use Case Diagram

Figure 5 visualizes the use cases in the form of a UML use case diagram. Further explanations of each use case can be found in the following functional requirements section.

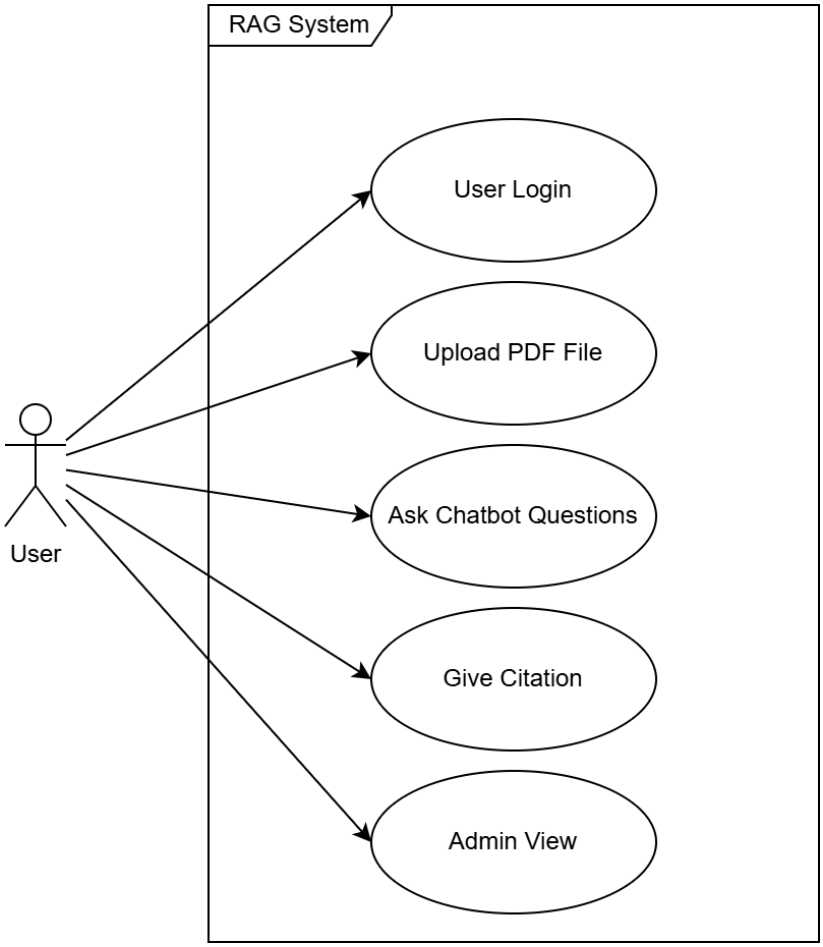

Figure 5: Use Case Diagram - own presentment

1.3. Functional Requirements

This section describes the functional requirements for the RAG system. We will use the brief variant of the use case description, as defined in Craig Larman's template for briefly dressed use cases [39].

1.3.1. FR1: User Login

Main Success Scenario:

- User visits the webapp, sees the login page and can login with correct credentials.

Alternate Scenario:

- If the user has incorrect login credentials the user remains on the login page. An error message is displayed.

1.3.2. FR2: Upload PDF File

Main Success Scenario:

- User opens the upload data function, selects a file, and uploads it. The user confirms the file for further processing and loading it into the vector store with a button click. A success message is displayed.

Alternate Scenario:

- If the user is not logged in, this page is not available.
- If processing takes too long, an error is displayed.

1.3.3. FR3: Ask Chatbot Questions

Main Success Scenario:

- User opens the chatbot function, types in a question, and starts the generation process. The retrieval system extracts the most relevant chunks based on the question and sends them to the LLM service for response generation. The generated response is then presented to the user.

Alternate Scenario:

- If the user is not logged in, this page is not available.
- If a user asks a question unrelated to any of their uploaded PDF files, the system should inform them that no related information exists.

1.3.4. FR4: Give Citation

Main Success Scenario:

- When a user receives a response from the chatbot, the chatbot includes a citation indicating which PDF files(s) were used to generate the response.

Alternate Scenario:

- If the user is not logged in, this page is not available.
- If a user asks a question unrelated to any of their uploaded PDF files, the system should inform them that no related information exists.

1.3.5. FR5: Admin View

Main Success Scenario:

- An authorized user can access the admin view interface and enter a query or word. The retrieval service will then return the most relevant chunks based on the query or word.

Alternate Scenario:

- If the user is not logged in, this page is not available.
- If the user did not upload any PDF files yet, the response will be empty.

1.4. Non-Functional Requirements

The following non-functional requirements are listed and categorized according to ISO/IEC 25010 [40].

1.4.1. NFR1: Horizontal Scaling

Category: Scalability
Description: The system infrastructure shall support horizontal scaling to accommodate increased user load.
Acceptance Criteria:
- Replicas can be set up easily.
- System works as intended, despite the scaling.

Verification Process:
- Each service can be scaled independently and will continue to function as intended.

1.4.2. NFR2: Access Restriction

Category: Security
Description: A user can only retrieve his upload files. A user cannot access files uploaded by another user.
Acceptance Criteria:
- Each user can only retrieve chunks/responses for their own uploaded files.
- The system enforces authentication to ensure users can only access their own uploaded files.

Verification Process:
- User A uploads file X, user B must not be able to retrieve chunks/responses from file X.

1.4.3. NFR3: Implement Local Services

Category: Security
Description: Fully implement all services locally without the need for external API calls.
Acceptance Criteria:
- No external APIs used.

Verification Process:
- The system must function in an offline environment without requiring external API calls.

1.4.4. NFR4: API Documentation

Category: Maintainability
Description: The system includes API documentation for all backend endpoints.
Acceptance Criteria:
- Each endpoint must include at least one descriptive sentence.

Verification Process:
- API documentation is populated and available in the development environment.

1.4.5. NFR5: Modular Architecture

Category: Maintainability
Description: The system code base shall follow modular architectural principles to facilitate updates and maintenance.
Acceptance Criteria:
- System components should be designed and set up to be loosely coupled and easily interchangeable.

Verification Process:
- Each module has well-defined interfaces and can be modified or replaced independently.

2. Testing Concept

Our test concept is based on the four test quadrants from the Agile Testing Methodology [41]. Each quadrant employs a different testing method, as illustrated in Figure 6.

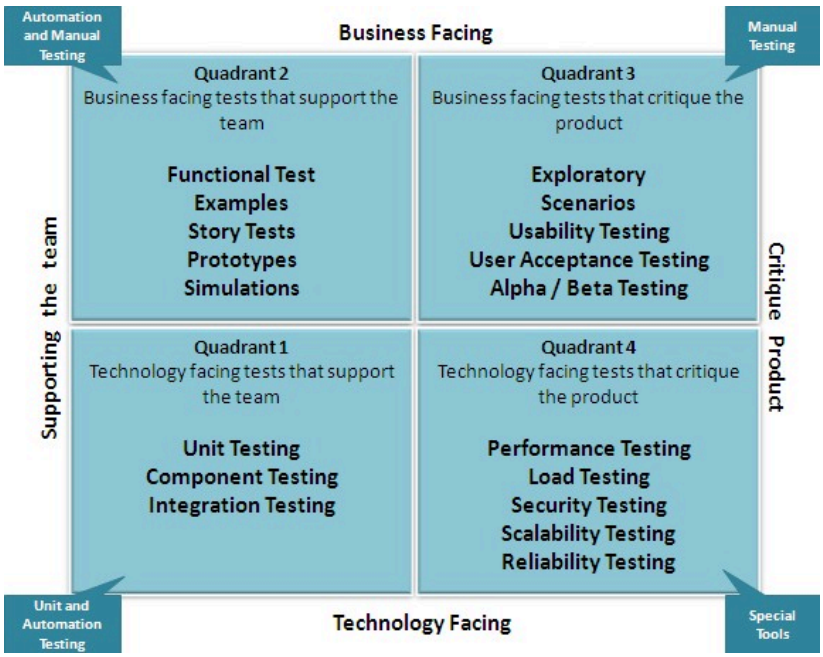

Figure 6: Agile Testing Quadrants - source: [41]

2.1. Quadrant One

The first quadrant focuses on verifying individual code components and the integration of them into the environment [42].

Although unit and automation testing would be ideal for products of any scale, the focus of this project is on building a scalable and secure RAG system. Due to time constraints, setting up an automated testing suite was deemed impractical, as it would require significant initial setup work. However, if anyone were to use this RAG system in production, unit and automation testing would be highly recommended.

2.2. Quadrant Two

The second quadrant focuses on the evaluation of the system against business requirements or customer expectations [42].

We will perform functional tests to ensure that all functional requirements are met, the components are functioning properly, and the entire system is operating correctly. For details about these tests, refer to the functional requirements part of the testing section (Part VII, Section 1.1.1.1.) and the test protocols in the appendix (Part IX, Section 5.), (Part IX, Section 6.).

2.3. Quadrant Three

The third quadrant focuses on testing to ensure that user expectations are met [42].

We will conduct an end-user test with some CAS students at the beginning of June. Additionally, the workshop at the SDS2025 Conference [10] will provide us with more user feedback. For details abut these tests, refer to the user test part in the testing section (Part VII, Section 1.5.).

2.4. Quadrant Four

This quadrant focuses on non-functional testing, including security, performance, and scalability testing [42].

We will conduct tests and review the requirements to ensure that the non-functional requirements are met. Additionally, the performance tests (Part VII, Section 1.6.) and the load test (Part VII, Section 1.5.) that are to be conducted fall within this quadrant. For details about this tests, refer to the non-functional requirements part of the testing section (Part VII, Section 1.1.1.2.) and the test protocols in the appendix (Part IX, Section 5.), (Part IX, Section 6.).

3. C4 Model

The C4 model is a framework for visualizing the architecture of software systems. It provides a hierarchical approach to modeling, allowing for different levels of detail in the diagrams. The C4 model consists of four levels: Context, Container, Component, and Code. Each level provides a different perspective on the system's architecture [43].

In this thesis, we will apply components of the C4 model as appropriate and meaningful. However, we will not apply the entire C4 model, as doing so would unnecessarily inflate the scope of our work without providing additional insight. We will primarily use the two levels of context and container to describe our architecture at different levels of detail.

Part III
Architecture

1. Architectural Pattern

This section presents a utility analysis of various potential architectural patterns to aid in selecting the most suitable option for our use case. It also highlights key considerations related to the selected architectural approach.

1.1. Utility Analysis

This section provides an overview of the utility analysis conducted.

1.1.1. Criteria

This section provides a description of each criterion for the utility analysis.

1.1.1.1. Scalability

Definition: Indicates how easily the architecture can be scaled to handle an increased workload.
Focus: How easily can the architecture scale horizontally (adding more nodes) or vertically (adding more resources to existing nodes).
Possible values: Ranges from "Hard to scale" (1) to "Supports robust autoscaling" (5).

1.1.1.2. Ease of Deployment

Definition: Indicates how much effort and complexity is involved in setting up and deploying the RAG system.
Focus: Installation steps, automation tools (CI/CD), required infrastructure, and ongoing maintenance.
Possible values: Ranges from "complex, high effort" (1) to "quick, push-button deployment" (5).

1.1.1.3. Adaptability

Definition: Reflects how straightforward it is to adapt, customize, or restructure the architecture.
Focus: Ease of reconfiguration, modular design, flexibility of adaptions to individual pipeline stages, and ability to introduce new features or components.
Possible values: Ranges from "static, poorly customizable" (1) to "highly flexible and customizable" (5).

1.1.1.4. Data Security

Definition: Evaluates how well the architecture protects data — both at rest and in transit.
Focus: Encryption, access control, compliance with privacy/security standards, isolation of sensitive data.
Possible values: Ranges from "negligible or non-existent safeguards" (1) to "strong controls, end-to-end security measures" (5).

1.1.1.5. Extensibility

Definition: Assesses how easy it is to add new functionality, integrate third-party services, or extend the existing capabilities of the system.
Focus: Plug-in systems, well-defined interfaces/APIs, support for external libraries or extensions.
Possible values: Ranges from "very difficult, requires significant refactoring" (1) to "simple drop-in modules and integrations" (5).

1.1.1.6. Hosting & Portability

Definition: Reflects the ease of running the pipeline in a variety of hosting environments (different cloud providers, on-premises, hybrid) and how easily it can be moved or re-deployed elsewhere.
Focus: Vendor lock-in, portability of services (e.g., containerization, open standards), ability to run it on-premise or in cloud environments.
Possible Values: "Locked-in, extremely difficult to switch environments" (1) to "Highly portable, can be hosted almost anywhere with minimal effort" (5).

1.1.2. Approaches

For this RAG system, three architectural approaches - namely microservices, event-driven, and serverless - were evaluated to determine how well they met our key requirements. These requirements included scalability, ease of deployment, adaptability, data security, extensibility and portability. The comparison table summarizing the findings can be seen below. It provides a basis for choosing an architectural style. The reasoning behind each number can be found below the table.

1.1.3. Comparison Table

Table 1: Classification and Comparison of Defined Criteria Across Architectural Patterns

	Microservices	**Event-Driven**	**Serverless**
Scalability	5	4	5
Ease of Deployment	3	3	5
Adaptability	4	4	2
Data Security	4	3	3
Extensibility	4	4	3
Hosting & Portability	5	4	2
Total	**25**	**22**	**20**

1.1.4. Reasoning

This section explains the reasoning behind each number listed in Table 1.

1.1.4.1. Microservices

- Scalability: 5, Highly scalable due to independent services that can handle increased load by adding more instances of specific services without impacting the rest of the system.
- Ease of Deployment: 3, Deployment can be complex (many moving parts) and requires coordination between teams, tools, and environments to ensure smooth updates and minimal downtime.
- Adaptable Architecture: 4, Generally flexible, but still requires well-defined boundaries to prevent service overlap, ensure maintainability, and allow for clear ownership of responsibilities.
- Data Security: 4, Good if services are secured properly using techniques such as strong authentication, encryption, and strict network segmentation to protect data in transit and at rest.
- Extensible: 4, Easy to add new microservices for new features without disrupting existing services, which allows teams to introduce functionality incrementally and iterate rapidly.
- Hosting & Portability: 5, Containerized microservices (e.g., Docker + Kubernetes) are extremely portable and can run on any public cloud or on-premises, and can be optimized with continuous integration and continuous deployment (CI/CD) pipelines for seamless updates across environments.

1.1.4.2. Event-Driven

- Scalability: 4, Scales well, but depends on robust messaging systems that must handle high throughput and ensure reliable delivery to maintain overall performance.
- Ease of Deployment: 3, Deployment can be intricate (managing event brokers) and can require specialized knowledge of messaging protocols and infrastructure to ensure event consistency.
- Adaptable Architecture: 4, Highly adaptable, new consumers/publishers can be added on demand with minimal disruption.
- Data Security: 3, Security can be fine but depends on secure messaging/event channels and the correct configuration of access controls, encryption, and monitoring to prevent unauthorized data access or message tampering.
- Extensible: 4, Extensible by adding new event streams or handlers that can process and route data efficiently, allowing for easy integration with additional services or modules.
- Hosting & Portability: 4, An event-driven system can be quite portable if an open-source solutions for messaging is chosen (e.g., Apache Kafka, RabbitMQ), and can be deployed on various infrastructures, from on-premises to cloud environments, providing great flexibility.

1.1.4.3. Serverless

- Scalability: 5, Built-in auto-scaling automatically adjusts resources based on workload, eliminating the need for manual provisioning.
- Ease of Deployment: 5, Often easy to deploy (function-based) with minimal operational overhead, allowing developers to focus on code rather than managing containers or servers.
- Adaptable Architecture: 2, Various options but probably also constraints on customization (depends on vendor). Those can limit architectural flexibility and may require creative workarounds to meet specific requirements.
- Data Security: 3, Security largely managed by the provider (depends on trust) and requires careful configuration of identity management, access management, and compliance settings for robust protection.
- Extensible: 3, Extending across multiple functions/services can get complex because each function may require separate configuration, orchestration, and monitoring to maintain overall system coherence.
- Hosting & Portability: 2, Managed serverless platforms (e.g., AWS Lambda, Azure Functions, Google Cloud Functions) often introduce tight vendor lock-in (proprietary runtimes, function triggers, logging, etc.), limiting direct migration between cloud providers and sometimes necessitating significant refactoring when changing environments.

1.1.5. Decision

In a series of discussions and with reference to the comparison table, the decision was made to adopt a microservices architecture. Microservices provide fine-grained control, allowing teams to adapt or update individual services without disrupting the overall system. They also offer significant flexibility in technology choices, which helps minimize vendor lock-in. Moreover, since each service operates independently, debugging and monitoring become simpler because potential issues can be isolated to a single component. This combination of adaptability, independence, and clarity ultimately makes microservices the most suitable choice.

Summarized in form of Y-Template [44]
In the context of the RAG pipeline, facing the need to create a environment which is easily adaptable and scalable, we decided for the architectural pattern microservices and against event-driven or serverless to achieve scalability and adaptability, accepting that this may create extra effort to deploy.

1.2. Further Considerations

This section outlines additional considerations related to the selected architectural pattern.

1.2.1. Microservice Orchestration

To orchestrate the microservices, a dedicated tool is required. Kubernetes was chosen for this purpose due to its widespread adoption in production environments, high extensibility, and robust scalability features. These strengths make it a natural fit for managing microservice architectures. Further details can be found in the infrastructure section.

1.2.2. Communication

The different microservices need to communicate with each other. To enable interaction among the various microservices, communication is established via HTTP messages using FastAPI (further detailed in the Tools section Part III, Section 6.7.1.). Most of the endpoints are implemented as POST endpoints, as they facilitate the transmission of JSON payloads within the request body. These endpoints are intentionally designed to be as generic and reusable as possible, ensuring consistency throughout the system.

A notable exception is the /remove-history endpoint, which uses the DELETE method. With the exception of this endpoint, all HTTP interfaces provided via FastAPI employ the POST method, and no further distinction is made between different HTTP verbs. This design choice is motivated by several considerations [45]:

- It ensures a consistent communication pattern across all microservices and endpoints.
- It supports the transmission of large payloads without size limitations.
- It allows unrestricted use of characters in the transmitted data.

2. RAG Framework

From the outset of the Bachelor's project, it was clear that a framework for the RAG system would be essential, as starting from scratch would be too complex and time-consuming. These two frameworks aim to provide a set of tools and libraries to simplify the development of RAG systems. This section describes the process of comparing the two candidates and reaching a decision.

2.1. LlamaIndex vs. LangChain

The two candidates selected are LlamaIndex and LangChain. The following comparison ensures that the correct tool is used for each task. Note that LangChain is the company name and thus encompasses all of their tools, including LangGraph and LangSmith.

2.1.1. Feature Comparison

Table 2: Comparison Between LlamaIndex and LangChain

Feature	Description	LlamaIndex	LangChain
1) Document Ingestion & Splitting	Reading and splitting documents (PDFs, text files, etc.) into chunks for processing.	✓	✓
2) Custom Indexing Strategies	Tree, graph, keyword, or list-based structures for organizing data.	✓	✗
3) Vector Store Integrations	Ability to use external vector databases (e.g. FAISS, Pinecone, Weaviate) for retrieval.	✓	✓
4) Retrieval QA	Out-of-the-box retrieval-based question-answering (end-to-end).	✓	✓
5) Prompt Templates	Built-in methods to manage prompt templates for LLM calls.	Basic	✓
6) Multi-Step Prompt Chaining	Composing multiple prompts (or LLM calls) into a single, orchestrated workflow.	Basic	✓
7) Agents & Tools	Let an LLM dynamically select and call external tools (e.g., web search, calculators).	✗	✓
8) Conversation Memory	Storing and referencing steps in a conversation.	Basic	✓

Feature	Description	LlamaIndex	LangChain
9) Advanced Reasoning / Branching Logic	Complex chain-of-thought or conditional branching in LLM applications.	✗	✓
10) Hierarchical / Structured Retrieval	Leveraging nested document structures for more precise retrieval.	✓	Partial
11) Graph-Based Linking	Linking document chunks in a directed graph for more nuanced retrieval paths.	✓	✗
12) Data Loading Ecosystem	Wide variety of pre-defined loaders for different file types, APIs, or data sources.	Basic	✓
13) Community & Ecosystem Size	Scale of community-driven extensions, documentation, and integration examples.	Growing	Large
14) Setup of a minimal QA pipeline	The setup of a minimal QA pipeline is easily possible.	✓	✓

Legend: ✓ = fully supported, ✗ = not supported, Basic / Partial = basically / partially supported

2.2. Decision

Table 2 shows that each tool has its own strengths and weaknesses depending on the area of application. For this reason, it was decided to use both in the area where they have their strongest capabilities. So LlamaIndex is used for ingestion and retrieval, and LangChain for the rest, including prompt logic, agentic behavior, advanced chaining, and more. Together, they enable us to create a powerful LLM-driven application.

3. Vector Store

As described in the planning section, we need a local vector store for our RAG system. This local vector store needs to be persistent, flexible, and work within our hosting setup. Based on several criteria, we compared two state-of-the-art options: ChromaDB and Weaviate.

3.1. ChromaDB vs. Weaviate

Weaviate is a vector database that combines object and vector storage to provide semantic search capabilities. It is designed to handle large-scale data and offers features like horizontal scaling, persistence, and user-based access control [46]. ChromaDB is a lightweight, open-source vector database that is easy to set up and use. It is designed for smaller-scale applications and provides basic functionality for storing and retrieving data, including vectors [46].

3.1.1. Feature Comparison

Table 3: Feature Comparison Between ChromaDB and Weaviate

Vector Store	Metadata Filtering	User-Based Access Control	Persistence	Horizontal Scaling	Kubernetes Support (out-of-the-box)
ChromaDB	✓ Yes	✗ No	✓ Yes	✗ No	✗ No
Weaviate	✓ Yes	✓ Yes	✓ Yes	✓ Yes	✓ Yes

3.2. Decision

Based on the comparison shown in Table 3, Weaviate was selected primarily for its superior scalability, production readiness, and out-of-the-box Kubernetes compatibility. A key advantage of Weaviate is its robust horizontal scaling support, which enables efficient handling of large-scale deployments by distributing data across multiple nodes [46]. This makes it ideal for applications that require high availability and consistent performance under heavy workloads. Weaviate's out-of-the-box Kubernetes support is provided through official Helm charts, making it ideal for Kubernetes environments.

ChromaDB, by contrast, is better suited for prototyping and simpler data structures [46]. ChromaDB is ideal for users seeking a simple, straightforward implementation without special requirements.

For this project, Weaviate's scalability, security, and Kubernetes support made it the preferred vector database.

4. Embeddings

One requirement of the RAG system is the local generation of embeddings. These embeddings serve two functions within the system. First, they represent each PDF file chunk as a vector for semantic searches. Second, they are used to find chunks with a given prompt. To accomplish the latter, a vector representation of the prompt is generated and compared to the vector representations of the chunks in the vector store.

Weaviate, as decided in the previous section, supports several ways to generate embeddings:

- Use a third-party provider, such as OpenAI via an API, to generate the embeddings for you. However, this is not an option for this project, since it is no longer local.
- Use the t2v-transformer (text-to-vector) service offered by Weaviate. With this approach, a separate container runs a model that automatically generates embeddings as needed.
- Generate the embeddings independently and then manually pass them to Weaviate.

For this project, we investigated and tested options two and three. A key feature of option two is the internal use within Weaviate. This setup consists of a separate service, and an inline configuration that instructs Weaviate to use this service for embedding generation. In this setup, the retriever sends plain text chunks to Weaviate, which then generates embeddings using the transformer service.

The third option - manually generating embeddings - is implemented using Hugging Face. Hugging Face provides a sentence transformer setup that includes some pre-defined models. The selected model is downloaded on startup and then executed completely locally. The generated embeddings are then manually passed to Weaviate.

Both approaches are functional and represent valid implementation choices. However, the third approach - manual creation of embeddings - was selected based on the performance evaluations detailed in the performance test section (Part VII, Section 1.6.) due to its superior performance.

5. State Management

In a microservices architecture, containers should ideally remain stateless to ensure that each instance of a given service type can process each incoming request. However, certain stateful data, such as chat history, requires some form of state to be maintained. For this reason, this section reviews different methods for managing application state and evaluates the most appropriate approach.

5.1. Strategies

In the context of the RAG system presented in this thesis, two viable strategies for handling states are worth considering.

5.1.1. Strategy 1

"Let the frontend handle the states and send all the necessary data within each request."

Moving the state management to the frontend results in a lighter backend. However, this adds complexity to the frontend, especially since it is no longer stateless. There are two primary options with this strategy:

1) Bind each user or session to a specific frontend instance (sticky sessions), which hurts scalability and flexibility.
2) Build a centralized state management solution directly into the frontend, which adds significant complexity.

The first option is particularly risky because it makes your application dependent on a single frontend instance, increasing the chance of errors and system instability. The second option is also not ideal, as it goes against the principle that logic and state should be managed in the backend whenever possible. For these reasons, this approach is not considered viable.

Finally, it should also be pointed out that this strategy introduces some new possibilities to manipulate the system by providing the option to directly manipulate states within the frontend.

5.1.2. Strategy 2

"Implement a state store solution within the backend to manage states centrally."

The second strategy introduces a high degree of flexibility and adaptability along with great scalability options.

5.1.3. Decision

Based on the problems with strategy one, the bright prospects of strategy two, and to keep the logic in the backend and the frontend as lightweight as possible, the second strategy will be pursued further.

5.2. State Store

Based on the decision above, this section defines the state storage requirements, compares viable solutions, and identifies the solution to proceed with.

5.2.1. Requirements

The state store should store state information such as chat histories, user IDs, and other information needed to respond to requests. However, this data is not critical in the sense that it needs to be protected from loss on any instance. Rather, it is important that the data be available on a fast instance.

5.2.2. Database Products

There are many options to handle state storage in a microservice environment. Two popular approaches often used in production systems are Redis and Apache Cassandra. Both are NoSQL databases designed for scalability and high workloads, they have overlapping use cases, but they differ significantly in architecture and focus. They are briefly described and compared below.

Redis is an in-memory data structure store optimized for performance and low latency, making it ideal for caching, session management, and real-time analytics [47]. In contrast, Apache Cassandra is a distributed database emphasizing availability and fault tolerance, suitable for large-scale transactional systems [48].

Following the CAP theorem, a database can only guarantee two out of the three following guarantees [49]:

- Consistency: Every read operation returns the most recent write or an error if consistency cannot be guaranteed.
- Availability: Every request gets a response, even if some nodes are down.
- Partition Tolerance: The system continues to operate despite network failures.

In this sense, Redis guarantees consistency and partition tolerance (CP), while Apache Cassandra guarantees availability and partition tolerance (AP) [50]. This impressively shows the different use cases that both databases are designed to handle. However, bear in mind that the interpretation of a database in terms of the CAP theorem can vary depending on the argumentation and configuration settings used. This should only provide an initial understanding of the main differences.

5.2.3. Decision

Based on the requirements, Redis is pursued as a better fit because low latency is more important than preventing data loss at any instance. Redis achieves this by storing data in-memory, but also provides several options for persistence. Still, both options have their advantages and disadvantages, so both can be argued as valid approaches.

5.3. Redis

Redis provides several options related to availability and persistence. These options are particularly important because proper configuration of these settings ensures both high availability and persistence of data. In the corresponding subsection, both are explained and discussed in more detail, and appropriate decisions are made for their use within our system.

5.3.1. Availability

One major concern of a state store is the availability. To ensure this, Redis provides several different architectural approaches to address different availability requirements. The different architectural approaches [51] are briefly described below and show in Figure 7.

- Simple Database: a single primary node that handles all operations by itself
- HA[1] Database: one primary node to handle reads and writes, plus one or more replica node(s) to handle reads only
- Clustered Database: multiple primary nodes, each managing a subset of data
- HA Clustered Database: multiple pairs of primary and replica nodes, each managing a subset of data

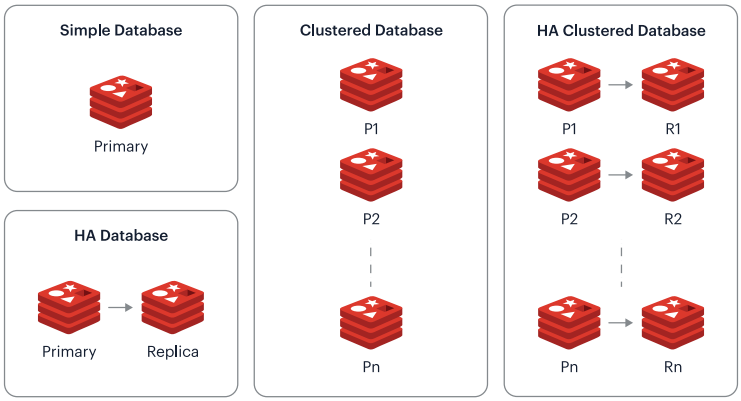

Figure 7: Redis Architecture Approaches - source: [51]

In the context of managing replicated Redis systems, the following Redis approaches exists [52]:

- Redis Sentinel: Supports high availability in Redis by automatically detecting master node failures and replacing them with a replica node. In Redis Sentinel, there is only one master node that handles writes.
- Redis Cluster: Provides a distributed version of Redis by automatically sharding data across multiple master nodes.

Note that in both approaches, the application interacting with Redis must be aware of the underlying architecture (master, replicas, potential sharding). The application is responsible for sending read and write operations to the correct node. For this thesis the Redis Cluster approach is used, as it supports better scaling and performance.

[1] High Availability

5.3.2. Persistence

Redis is designed to be an in-memory database. This means that the data is kept in memory. However, persistence options can be added as needed. The following options are available [53]:

- RDB (Redis Database): Takes point-in-time snapshots of your dataset at specified intervals.
- AOF (Append Only File): Logs each write operation received. At startup, the database can be reconstructed using these entries. The following option can be applied:
  - appendfsync always: append to the AOF at every new command (very slow, very safe)
  - appendfsync everysec: append to the AOF every second (loose at most one second of data)
  - appendfsync no: never fsync, rely on the operating system (in linux normally every 30s)
- No persistence: Persistence can be turned off completely.
- RDB + AOF: Any combination of RDB and AOF is possible.

In summary, RDB is better for backup purposes because it creates snapshots that can be used for disaster recovery. AOF, on the other hand, is better if you do not want to lose data accidentally. In general, it is advisable to use both approaches together.

5.3.3. Decision

The Redis cluster approach is used for this pipeline because of its availability and scalability advantages. Since the system is already set up with a clustered approach, scaling can be done later by simply adjusting the appropriate Helm configuration without any code changes. In addition, speed, especially when writing data, is better compared to the Sentinel approach because more nodes are running to handle requests.

For persistence, only AOF is adopted, so that the current state of the system is retained even if the system is restarted. RDB is not adopted, but can be retrofitted at any time by simply changing the appropriate Helm configuration. This setting is not adopted because backup is beyond the scope of this work. If the system is to be used in production, this must be taken into account.

In summary, a minimal Redis Cluster architecture with three master and three replica nodes is used together with AOF to keep the state of the database persistent. The resulting cluster architecture is described in more detail in the according implementation section (Part V, Section 2.2.6.).

6. Tools

This chapter identifies and briefly describes the key tools used in this thesis.

6.1. GitLab

GitLab is a popular web-based platform designed to streamline the entire software development lifecycle by integrating tools for version control, collaboration, and automation [54]. Built on top of Git, a distributed version control system, GitLab provides a comprehensive solution for source code management, project planning, workflow automation and application deployment.

6.2. Docker

Docker is an open-source platform that automates the deployment of applications within lightweight, portable containers [55]. These containers encapsulate an application and all its dependencies, ensuring consistent behavior across various environments, from development to production. Unlike traditional virtual machines, Docker containers share the host system's kernel, making them more resource-efficient and faster to start.

6.2.1. Docker Compose

Docker Compose is a tool that simplifies the definition and management of multi-container Docker setups [56]. It allows developers to configure application services, networks, and volumes using a single YAML file. With the command "docker compose up" one can build, start, and manage all the services defined in the configuration file.

6.3. Kubernetes (K8s)

Kubernetes is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications [57]. Kubernetes has been developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF). Over the past years, Kubernetes has become the industry standard for the provision and operation of container applications, including microservices architectures and cloud-native applications.

Key features of Kubernetes include the following [58]:

- Container Orchestration: Kubernetes automates the deployment, management, scaling, and networking of containers across a cluster of machines.
- Scalability: Kubernetes enables applications to scale seamlessly to match the current workload by adding or removing container instances.
- Self-healing: Kubernetes automatically restarts failed containers, replaces and reschedules them on healthy nodes, and kills containers that do not respond to defined health checks.
- Load Balancing: Kubernetes distributes network traffic evenly across deployed containers.
- Automated Rollouts and Rollbacks: Kubernetes can manage application updates by incrementally rolling out changes, while monitoring and responding to application health.
- Storage Orchestration: Kubernetes can automatically attach storage systems to containers for seamless data management.

6.3.1. Minikube

Minikube is a lightweight tool for running a single node Kubernetes cluster locally on a single machine [59]. The focus is on developing and testing Kubernetes deployments without the need for a fully sized Kubernetes environment.

6.3.2. Kustomize

Kustomize is an open source configuration management tool that is built directly into kubectl, allowing it to be used natively with Kubernetes [60]. Kustomize offers a template-free, declarative approach to customizing, managing, and bundling Kubernetes configuration files without altering the original ones. This provides the benefit of easy deployment and management of multiple environments while maintaining reusability and consistency.

6.3.3. Helm

Helm is an open source package manager for Kubernetes that simplifies the deployment and management of applications [61]. The packaging format, called Helm charts, bundles all necessary Kubernetes resources (such as deployments, services, and ConfigMaps) into reusable templates. These templates can be parameterized using a values.yaml file, allowing users to customize configurations based on their needs without having to deal with the underlying configurations. Helm simplifies application lifecycle management by supporting version control, rollbacks, and upgrades.

6.3.4. K9s

K9s is a terminal-based user interface tool designed to simplify the management of Kubernetes clusters. K9s provides an intuitive and efficient way to interact with Kubernetes resources, providing a visual and user-friendly extension to the kubectl command line tool [62].

6.3.5. Ingress NGINX

Ingress NGINX is a Kubernetes ingress controller that manages external access to cluster services, using NGINX as a reverse proxy and load balancer [63]. It implements ingress resources by dynamically configuring NGINX to route HTTP/HTTPS traffic based on host names, paths, or TLS settings.

6.3.6. Cert Manager

Cert Manager is a Kubernetes native certificate management tool that automates the issuance, renewal, and injection of TLS certificates from trusted Certificate Authorities (CAs) such as Let's Encrypt [64]. In doing so, Cert Manager ensures secure communication for ingress resources, services, and workloads. Furthermore, it monitors certificate expiration and proactively handles renewals, reducing downtime and manual intervention.

6.3.7. Let's Encrypt

Strictly speaking, Let's Encrypt is not a tool in the sense of an application. However, an introduction to the product makes sense in this chapter because it is used within Cert Manager.

Let's Encrypt is a non-profit certificate authority that provides free TLS certificates to make it easy for websites to encrypt over HTTPS and thus secure their applications [65].

6.4. RAG Frameworks

Throughout the project, different RAG frameworks have been used. These frameworks are briefly described below.

6.4.1. LlamaIndex

LlamaIndex is a lightweight AI framework designed to streamline the loading, structuring, and querying of large text datasets [66]. LlamaIndex enables the creation of specialized indexes — such as list-based, tree-based, or graph-based — that optimize document retrieval for the use in LLM-powered applications. By simplifying document ingestion, chunking, and retrieval strategies, LlamaIndex is particularly effective for building document-focused applications like question answering and summarization.

6.4.2. LangChain

LangChain is a powerful AI framework designed for building applications incorporating Large Language Models (LLMs) [67]. It offers modular components — such as Chains, Agents, Tools, and Memory — that can be seamlessly combined to create complex language-driven workflows. LangChain simplifies integration with document loaders, vector databases, and external APIs, while also supporting multi-step reasoning, conversation memory, and dynamic tool usage. All of this is orchestrated through flexible and robust prompt templates, making LangChain a comprehensive solution for developing and deploying advanced LLM-powered applications.

6.5. Data Storage

This section briefly describes the tools used for state management and data storage.

6.5.1. Weaviate

Weaviate is an open-source, AI-native vector database specifically designed for storing and searching high-dimensional data, such as embeddings generated by machine learning models [46], [68]. It supports efficient similarity and offers native integration with popular embedding models. This makes it well-suited for use cases like semantic search, recommendation systems, and generative AI applications.

6.5.2. Redis

Redis (Remote Dictionary Server) is an open-source, in-memory data store optimized for performance and low latency, making it ideal for caching, session management, and real-time analytics [47]. In this project, Redis is used to store management information in a centralized but scalable way to ensure that containers remain interchangeable because they do not have to manage states themselves.

6.6. Authentication

This section briefly describes the tools used for authentication purposes.

6.6.1. Keycloak

Keycloak is an open source identity and access management solution that provides centralized authentication, single sign-on (SSO), and authorization services [69]. It supports multiple authentication standards including OpenID Connect, OAuth 2.0 and SAML. This provides the ability to federate users with LDAP/Active Directory, social login (e.g. Google, GitHub), and fine-grained authorization policies. Keycloak simplifies security implementation with customizable login themes, session management, and token mapping for applications.

6.6.2. OAuth2 Proxy

OAuth2 Proxy is a reverse proxy that adds OAuth2/OpenID Connect authentication to applications [70]. It intercepts requests, redirects users to an identity provider (e.g. Keycloak) for login, and forwards validated sessions with user attributes (e.g. roles, email). OAuth2 Proxy supports multiple OIDC providers and can be integrated as middleware or a standalone proxy, ideal for securing applications without native OAuth2/OIDC support.

6.7. Python Modules

This section briefly describes some special Python modules that were used during this project.

6.7.1. FastAPI

FastAPI is a modern, high-performance web framework for building APIs [71]. Designed for speed and ease of use, it provides automatic interactive documentation and strong data validation.

6.7.2. Streamlit

Streamlit is an open-source framework for building interactive web applications, particularly for data science and machine learning projects [72]. It is designed to enable quick and easy creation of data-driven front-end applications, allowing developers to focus on logic and functionality rather than underlying web development complexities.

6.7.3. Pip-compile

Pip-compile is a command-line tool from the pip-tools package that helps Python developers to manage dependencies more reliably [73]. It reads a requirements.in file containing direct dependencies and generates a fully resolved requirements.txt file with pinned versions, including transitive dependencies. This improves traceability and prevents errors relating to the release of new versions and version clashes.

6.7.4. Faker

Faker is a Python library that generates fake data for various purposes, such as testing, development, and data anonymization [74]. It provides a wide range of data types, including names, addresses, dates, and more, allowing developers to create realistic datasets without using real user information.

6.8. Code Quality Tools

This section briefly describes the tools used to ensure code quality.

6.8.1. Black

Black is a Python code formatter that enforces PEP 8-compliant styling by automatically restructuring code, eliminating manual formatting decisions, and ensuring consistency across the codebase [75].

6.8.2. Flake8

Flake8 is a Python linter that combines PEP 8-style checking and code error detection into a single tool that promotes readability and reduces logical errors through static analysis [76].

6.8.3. MyPy

MyPy is a static type checker for Python that validates type hints (PEP 484) at compile time, catching type inconsistencies before execution and improving code reliability [77].

Part IV
Infrastructure

1. Environment

This chapter describes the tools and environment used to develop and host our RAG system.

1.1. Docker Compose

For development purposes, we use a local Docker Compose environment. This allows us to easily create and run the individual Docker containers, including the network connection between them. This saves us a lot of time as we can develop and test our system without having to deploy anything. We also have the ability to override options from the Docker files and enable development specific options such as "hot reloading" of our code base.

This approach allows us to have a development-specific configuration for our containers in our Docker Compose environment, while keeping the original Docker file production-specific. This enables us to build the Docker files in a production-ready way from the beginning, without having to have multiple of them.

The following is an excerpt of the file structure used for the Docker Compose environment. Below that is an excerpt of the Docker Compose file showing the enabled development settings for the manager service.

```
.
├── llm-service/
│   └── Dockerfile
├── manager-service/
│   └── Dockerfile
├── ...
└── docker-compose.yml
```
file-structure

```yaml
services:
  manager-service:
    build: ./manager-service
    environment:
      - REDIS_PASSWORD=${REDIS_PASSWORD}
    ports:
      - "5000:5000"
    volumes:
      - ./manager-service:/app
    networks:
      - shared_net
    command:
      - "uvicorn"
      - "manager-service:app"
      - "--host"
      - "0.0.0.0"
      - "--port"
      - "5000"
      - "--reload"
```
YAML

1.2. Kubernetes Cluster

This chapter provides an in-depth overview of the Kubernetes-based hosting environment, an essential component of our infrastructure.

1.2.1. Development

For local testing and development purposes, we run a self-hosted Minkube instance. The advantage of this is that we have full control over the entire setup, including installation and execution. This allows us to easily make adjustments to the system where needed and have better debugging capabilities.

1.2.2. Deployment

In order to test and run our pipeline in a real-world manner and to have more computing power available, we use a three-node Kubernetes setup hosted by the Eastern Switzerland University of Applied Sciences in Rapperswil. This setup is very similar to the one shown in Figure 8.

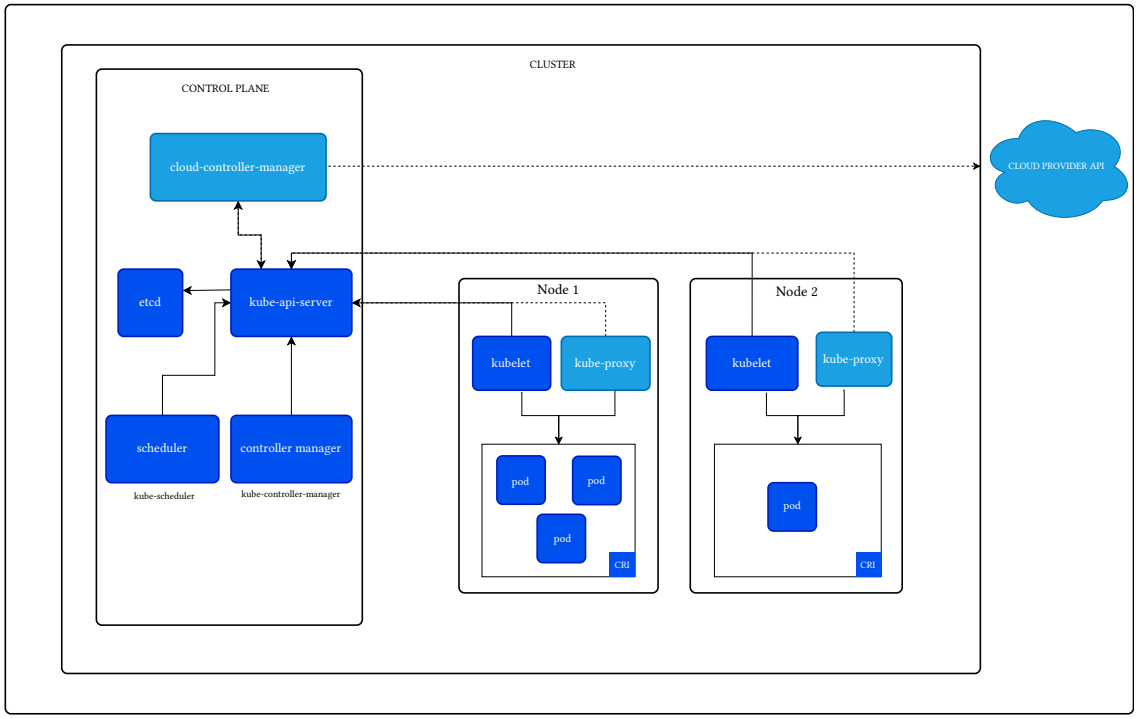

Figure 8: Kubernetes Cluster - source: [78]

The different types of nodes and components are briefly described below.

One of the three nodes in our cluster is dedicated to the control plane and its associated components. This node is responsible for managing global decisions across the cluster and for detecting and responding to cluster events [79]. The key components running on this node include:

- kube-apiserver: exposes the Kubernetes API and thus serves as the frontend of the control plane
- etcd: consistent and high-available key-value store used to store all cluster data
- kube-scheduler: watches for newly created pods and selects a node to run them on based on various factors
- kube-controller-manager: runs various controllers to ensure the desired state of the cluster
- cloud-controller-manager: typically used only in cloud-hosted environments, embeds cloud-specific control logic to connect the cluster to the cloud provider's API

The remaining two nodes are responsible for executing the actual workload of our pipeline. The smallest deployable units on these nodes are the pods, which represent the services within our microservice architecture. The components running on these nodes include:

- kubelet: agent that makes sure that containers are running in a pod and ensures that those pods are healthy
- container runtime: enables Kubernetes to run containers
- kube-proxy (optional): maintains the network rules on the nodes

1.2.3. Configuration File Management

Deploying applications on Kubernetes often involves managing numerous configuration files. Kustomize offers a state-of-the-art solution to streamline this process by enabling efficient organization, customization, and bundling of these files, thereby simplifying both configuration management and deployment.

In our use case, we only have one environment on Kubernetes (prod), so we do not need the overlays folder structure. For the sake of completeness, the overlays folder is created anyway. A snippet of the resulting file structure looks like the following.

```file-structure
k8s/
├── base/
│   ├── deployments/
│   │   ├── kustomization.yaml
│   │   ├── xyz.deployment.yaml
│   │   └── ...
│   ├── secrets/
│   │   ├── kustomization.yaml
│   │   ├── xyz.secret.yaml
│   │   └── ...
│   ├── services/
│   │   ├── kustomization.yaml
│   │   └── xyz.service.yaml
│   └── ...
├── kustomization.yaml
├── xyz.nampespace.yaml
├── ...
└── overlays
```

The following subsections provide brief descriptions of each configuration type.

1.2.3.1. Deployments

Deployments describe the hosting details of each service in the RAG system. It includes configurations such as the deployment name, container image, number of replicas, environment variables, resources, ports, and more.

1.2.3.2. Ingress

The ingress is the only component of the RAG system that directly communicates with the outside world. As such, its configuration defines how services are exposed externally. The ingress also handles encryption settings and acts as the termination point for TLS connections. Since all incoming requests must pass through the ingress, it also serves as the entry point for the authentication workflow. Depending on the outcome of this workflow, the ingress either forwards the request to the RAG system or rejects it.

1.2.3.3. Issuer

Strictly speaking, the issuer configuration file contains two different configurations, the issuer and the certificate. Since these configurations are closely related, including them in the same configuration file makes the configuration more convenient and easier to understand.

The issuer defines the details of the Certificate Authority (CA) to receive valid certificates from. In our system, this is Let's Encrypt (for more information about Let's Encrypt, see the according Tools section (Part III, Section 6.3.7.)).

The certificate configuration, on the other hand, defines details about the certificate itself. For example, which issuer to use, how long the certificate is valid, and after how many days the certificate should be renewed.

1.2.3.4. Secrets

Some services require secrets for security reasons or to function properly. These secrets can be API keys or passwords, for example. Of course, these secrets are never pushed directly to the repository for security reasons. For this reason, the corresponding folder in the repository contains only one Kustomize configuration file and no other configuration files. All secrets must be base64 encoded to work properly.

1.2.3.5. Services

Each deployment also has a service configuration. This configuration defines the exposed port within the Kubernetes system, as well as the destination port inside each container.

1.2.3.6. Namespace

Kubernetes has namespaces to logically separate different systems or components. In our case, the three most important namespaces are the following:

- RAG pipeline (mvp, basic): contains all the microservices used within the deployed pipeline
- Authentication (auth): contains the components used within the authentication workflow (except the ingress)
- Ingress (ingress-nginx): contains the components used for the ingress to work properly

There are other namespaces, of course, but only these three are directly affected by our configuration.

1.2.3.7. Kustomize

The kustomization configuration file belongs to Kustomize and bundles all the configuration files together for easy deployment. This configuration specifies each file that needs to be deployed to Kubernetes. In addition, some settings, such as the version of the Docker images to deploy, are overridden at the top level of the kustomization configuration. This makes it easier to change the version to deploy.

Each folder containing configuration used for deployment has a separate kustomization configuration file. This keeps the structure clean.

To deploy the system to Kubernetes using Kustomize, all you need is the following command. The -k options stand for Kustomize. Of course, you must be in the correct directory for the command to work properly.

```cli
kubectl apply -k ./base
```

1.2.4. Tag Management

It is highly recommended to deploy container images based on version tags rather then with the “latest” tag, as this ensures reproducibility and simplifies debugging. This approach prevents many problems from occurring in the first place.

Kubernetes deployments often involve numerous configuration files, making tag management cumbersome. Kustomize simplifies this by allowing you to define image tags in a central kustomization.yaml file. Using the newTag field, you can override image tags from the original files. This creates a single point of control for updating image versions during deployment, streamlining the process.

The following is an excerpt of the described configuration structure:

```yaml
images:
  - name: <image name>
    newTag: <tag>
```

1.2.5. Ingress Setup

To make the system publicly accessible, an ingress controller is required. This system uses ingress-nginx, which functions as both a reverse proxy and load balancer. The ingress is configured using the following command, as recommended by Jan Untersander from the Eastern Switzerland University of Applied Sciences.

Note: The ingress-nginx version was updated to the latest version during the thesis due to a critical security vulnerability. More information can be found under the following CVE ID: CVE-2025-1974.

```cli
helm \
  upgrade -i --atomic --cleanup-on-fail --timeout 10m0s --create-namespace --repo https://kubernetes.github.io/ingress-nginx \
  ingress-nginx \
  ingress-nginx \
  --version 4.11.2 \
  --namespace ingress-nginx \
  -f helm-nginx-ingress-values.yaml
```

1.2.6. Certificate Manager Setup

To enable issuer configuration, a certificate manager is required. This system uses cert-manager from Jetstack, following the recommendation of Jan Untersander from the Eastern Switzerland University of Applied Sciences. It is set up using the following command.

Note: The appropriate repository must be added to Helm before the Helm chart can be used. This process is not described here.

```cli
helm upgrade -i \
  cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --create-namespace \
  --version v1.17.0 \
  --set crds.enabled=true
```

1.2.6.1. Container Registry Access

Since the container registry is private, a secret is required to access it. The secret is created using the following command, along with an access token issued via GitLab.

```cli
kubectl create secret docker-registry gitlab-token-auth \
  --docker-server=registry.gitlab.ost.ch:45023 \
  --docker-username=<token_username> \
  --docker-password=<token>
```

1.3. Repository

Each pipeline increment (MVP, basic) has its own GitLab repository. This avoids confusion between increments and simplifies the management of code, pipelines, and container images. It also makes it easy to treat increments as standalone systems.

1.3.1. Access

Access to the repositories is via an SSH key, which simplifies the authentication process, particularly when working with multiple repositories.

1.4. Container Registry

Kubernetes requires a container registry to retrieve the images needed for deployment. For simplicity and scalability, we use GitLab's container registry, as our project is already hosted on GitLab and it provides a wide range of integration and extension options.

1.4.1. Tag Convention

The following two tags are added to each container image upon build for the stated purpose:

- latest: always represents the latest container and is overwritten when a new container image is pushed
- <version>: represents the version of the container as described in the GitLab tag, and remains in the registry until deleted (does not get overwritten)

This concept ensures easy container image management and version control. It allows for easy system updates and rollbacks to previous versions during hosting.

1.4.2. Naming Convention

As our RAG system is built up incrementally, we potentially have many different container images. To ensure clarity and avoid mix-ups, we enforce a strict naming convention for our containers. GitLab defines the initial part of the container name as follows [80]:

```
1 <registry server>/<namespace>/<project>                    filename-template
```

Building on this, our naming convention defines the last part of the container names as follows:

```
1 /<service type>-service:<tag>                              filename-template
```

An example using our naming convention for the retrieval service is shown below:

```
1 <gitlab naming convention>/retrieval-service:latest        filename-template
```

1.4.3. Build

The entire build process is managed via a GitLab pipeline. The process is as follows:

1. login to the container registry
2. build the container image (using the version tag)
3. tag the container image with “latest”
4. push both container images to the registry

This process only starts if a tag exists on GitLab. Otherwise, it is ignored. This approach prevents the unnecessary use of resources and ensures that updates or new versions are created intentionally. Furthermore, this makes the process more predictable, which is beneficial for hosting.

The following snippet illustrates the GitLab CI build pipeline, which builds a container image and pushes it to the registry. The placeholder <service_type> should be replaced with the specific service being built.

```yaml
build_<service_type>-service:
  image: docker:24.0.5
  stage: build
  services:
    - docker:24.0.5-dind
  variables:
    SERVICE_NAME: "<service_type>-service"
    IMAGE_TAG_VERSION: $CI_REGISTRY_IMAGE/$SERVICE_NAME:$CI_COMMIT_TAG
    IMAGE_TAG_LATEST: $CI_REGISTRY_IMAGE/$SERVICE_NAME:latest
  script:
    - echo "$CI_REGISTRY_PASSWORD" | docker login $CI_REGISTRY -u $CI_REGISTRY_USER
      --password-stdin
    - docker build -t $IMAGE_TAG_VERSION ./$SERVICE_NAME
    - docker tag $IMAGE_TAG_VERSION $IMAGE_TAG_LATEST
    - docker push $IMAGE_TAG_VERSION
    - docker push $IMAGE_TAG_LATEST
  only:
    - tags
```

A successful build process on GitLab looks like shown in Figure 9: first, linting and type checking are performed, and then the container images are built.

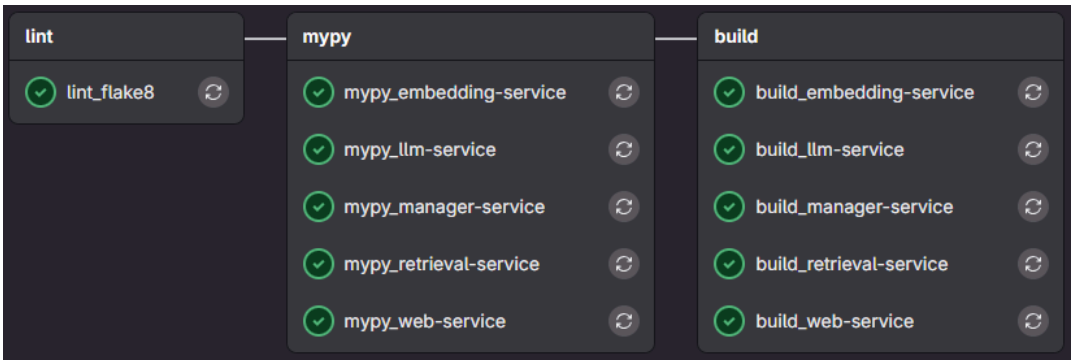

Figure 9: GitLab CI Pipeline - own presentment

1.4.4. Access Token

The container registry must remain private and inaccessible to the public. To allow secure access, authentication is required. This is achieved using an access token that is limited to pushing and pulling images from the registry, ensuring minimal necessary permissions.

1.5. Code Quality Tooling

To support the writing of high-quality Python code, the following tools were used throughout the project to highlight potential errors and enforce the writing of clean code.

1.5.1. Mypy

Mypy is employed as a static type checker to verify return types, parameter types, and other type annotations. It is used both locally on each developer's environment and within the GitLab pipeline, ensuring comprehensive type checking across the entire codebase. The project follows Mypy's default configuration.

1.5.2. Black

Black is used as an automatic code formatter to enforce Python coding standards. However, it is only applied locally, as formatting changes within the GitLab pipeline are not recommended as they can lead to unintended and unpredictable behavior.

1.5.3. Flake8

Flake8 is used as a linter to enforce consistent coding style by issuing warnings when style rules are violated. It runs both locally in developer environments and globally within the GitLab pipeline to ensure coding standard compliance throughout the codebase. The Flake8 configuration includes the following customizations to ensure compatibility with Black to enable an automated formatting approach.

```
1 [flake8]                                                    config-file
2 extend-ignore = E203, E704, E501
```

- E203 (Whitespace before ':') - Ignored due to conflicts with Black's formatting, especially in slicing (e.g., a[1: 5] vs. a[1:5]).
- E704 (Multiple statements on one line (def)) - Allowing single-line function definitions as Black does on occasion.
- E501 (Line too long (> 88 characters)) - Disabled because enforcing this policy within Black is currently marked as a preview feature and may result in unintended behavior. Also, wrapping long lines in e.g. templates can result in less readable code.

2. Authentication

The authentication component is a critical part of the overall system because it protects all subsequent components and processes. This section describes each component of the authentication system and how they interact.

2.1. Fundamentals

This section identifies and explains the key standards used in the authentication process.

2.1.1. OAuth2

OAuth2 (Open Authorization 2.0) is an authorization framework which enables third-party applications to access user resources hosted on a service without exposing credentials [81]. The following types of tokens are involved in the OAuth2 framework:

- Access Token: Grants temporary access to a protected resource (short-lived)
- Refresh Token: Obtains new access tokens without requiring re-authentication (long-lived)

2.1.2. OpenID Connect

OpenID Connect (OIDC) is an identity layer based on OAuth2 that enables secure authentication and exchange of user profile information between applications [82]. The following token and core functions are involved in OIDC:

- ID Token: JSON Web Tokens (JWTs) that contains verified user identity claims (e.g. email, name)
- UserInfo Endpoint: A RESTful API to retrieve additional user attributes using the OAuth2 access tokens
- Single Sign-On (SSO): Enables seamless authentication across applications

2.1.3. Tokens

In this section the different tokens used within the authentication process are described in a more detailed manner.

2.1.3.1. Access Token

An access token is a credential that grants the client temporary access to a protected resource on behalf of a user [83]. Typically, it is a JWT or opaque string issued by an authorization server, such as Keycloak, after the user grants permission. This token contains information about the granted permissions and scope, and is presented to the resource server to access specific APIs or services. These tokens are short-lived to minimize the security risks associated with token compromise.

2.1.3.2. Refresh Token

A refresh token is a credential used to obtain new access tokens without requiring the user to re-authenticate [84]. This token is issued along with the access token and has a longer lifetime. When the access token expires, the client application can use the refresh token to request a new access token from the authorization server. Refresh tokens are intended for use only with the authorization server and should be stored securely to prevent unauthorized access.

2.1.3.3. ID Token

An ID token is a JSON Web Token (JWT) that serves as a proof of authentication [83]. This token is issued by the identity provider (e.g. Keycloak) upon successful user authentication. It contains information about the user's identity, such as their name and email address. The ID token is intended for the application and should not be used to access protected resources. It is primarily used to convey information about the authenticated user to the application, enabling a personalized user experience.

2.2. Security Considerations

Since authentication is a broad topic, there are many possible security considerations. This chapter identifies and describes the most important security considerations associated with our RAG system.

2.2.1. Access Token Lifespan

The lifespan of the access token is a critical security aspect. As long as the access token is valid, successful requests can be made to the corresponding protected resource without entering any credentials. This is possible even if the session on the identity provider (Keycloak) has been revoked. Therefore, this lifespan must be short enough to prevent misuse. In general, values between 5 and 10 minutes are recommended. Our system uses 10 minutes.

2.2.2. Refresh Token Lifespan

Another critical security consideration is the lifespan of the refresh token. A refresh token is used to generate a new access token once the previous one has expired. In other words, as long as the refresh token is valid, more access tokens can be issued. However, if the session has been revoked by the identity provider, no more access tokens can be issued with the refresh token, and access to the resource is withdrawn.

Our system uses an idle time of 8 hours. After this time a new cookie must be generated as shown in the description of workflow 4 (Part IV, Section 2.4.4.). Further, the session maximum time is set to 24 hours. This means that the identity provider will invalidate the session after one day, regardless of whether any requests have been made in the meantime. After this time a complete new session must be initiated as shown in the description of workflow 1 (Part IV, Section 2.4.1.).

The refresh token lifespan is the shorter of the two times introduced above. In this case, it is 8 hours.

2.2.3. Refresh Token Revocation

Refresh token revocation is an option to harden the system against refresh token abuse. This is done by revoking refresh tokens after they have been used and issuing a new one. The old token can no longer be used. This option is used in our RAG system to enhance security.

2.2.4. Session Validation

Session validation checks each request to see if the token has been invalidated by the identity provider. This happens, for example, when a session is revoked. This option is not currently used in our system due to time limitations, but should be carefully considered in a production system.

2.3. Components

The three main components involved in the authentication process are identified and briefly described below. In addition, the interaction of the components is shown in Figure 10.

2.3.1. Ingress

In technical terms the ingress is a reverse proxy. It is the only point of the system with direct contact to the outside world. All requests to the system must go through this component. Consequently, it is the first component involved in the authentication process.

2.3.2. Keycloak

Keycloak is our identity provider (IdP). It is responsible for handling the user accounts with their credentials, the groups, the login workflow, as well as providing the login functionality including the login page. This component is a fully self-hosted state-of-the-art identity provider.

2.3.3. OAuth2 Proxy

The OAuth2 Proxy is the intermediary between the ingress and Keycloak. It handles the session management and sets and validates the needed cookies for the client. The great advantage of this component is that the RAG system does not need to be aware of any authentication measures and steps. The only thing the RAG system needs is the user information via the x-header option to distinguish different users. This makes the system less error-prone and as lean as possible.

2.3.4. Components Diagram

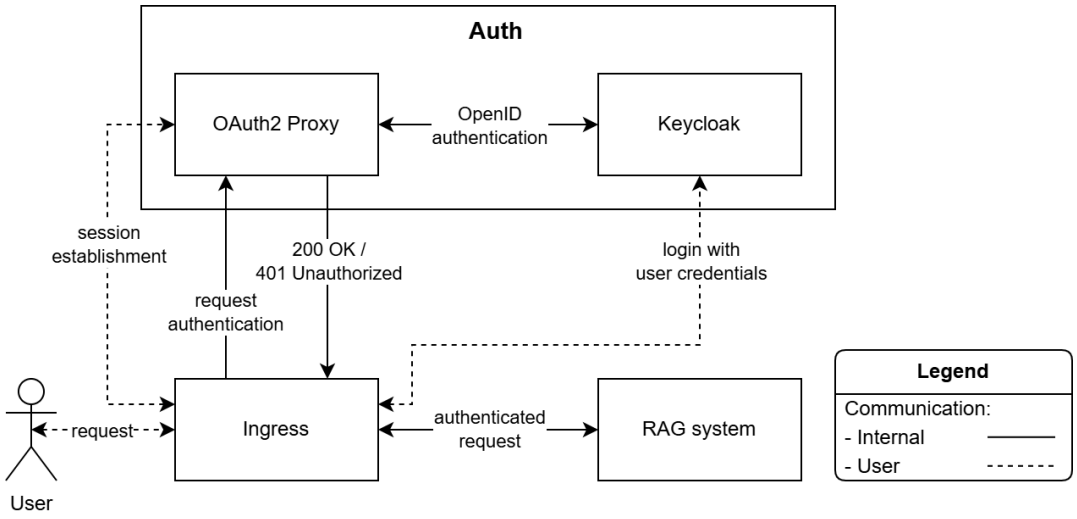

Figure 10: Authentication Architecture - own presentment

Note: OAuth2 Proxy uses Redis to store session information and Keycloak uses Postgresql to store user information and other configuration. Both components are omitted in Figure 10 because they do not add any valuable information to the understanding of the authentication workflow.

Disclaimer: Unfortunately, the workflow of the OAuth2 Proxy is not documented in great detail. The following workflows are thus assembled based on the information found and conducted experiments. The real workflow can thus slightly vary.

2.3.4.1. Interaction between the Components

The ingress is the only point of the system that is accessible from the outside. Therefore, every request must go through it. The ingress is instructed to authenticate every request going to the RAG system. This authentication request is then further handled by the OAuth2 Proxy.

The OAuth2 Proxy receives all authentication requests. If a client sends a session cookie along with the request, it is forwarded to the OAuth2 Proxy. Based on this information, it decides whether the session is still active and access to the RAG system can be granted, or whether the session is no longer active and re-authentication via Keycloak must be performed. If no session cookie is sent to the OAuth2 Proxy, it will also initiate an authentication via Keycloak.

When an authentication via Keycloak is initiated, the user is redirected to the Keycloak service. Keycloak presents the user with a login page where he can authenticate himself to the system.

After successful authentication, a valid session is created in both Keycloak and the OAuth2 Proxy. The OAuth2 Proxy then generates a session cookie and sends it to the browser. Until the session is expired (or revoked), requests with this session cookie can be authenticated directly within the OAuth2 Proxy.

Of course, this is a simplified description of the workflow. The various stages of the authentication workflow are described in more detail in the following section.

2.4. Workflow

In this section, the following authentication workflows are shown and described in detail (titles are sometimes abbreviated later):

- Workflow 1: Initial Login / Resource Access with Expired Session
- Workflow 2: Resource Access with a valid Access Token
- Workflow 3: Resource Access with an expired Access Token but a valid Refresh Token (token refresh)
- Workflow 4: Resource Access with expired Access and Refresh Tokens
- Workflow 5: Logout Procedure

Note: Each request is terminated at the ingress. To keep the figures as readable as possible, this fact is omitted in the following figures.

Disclaimer: All of these workflows are shown as successful procedures. In most cases, if something goes wrong, a simple re-authentication is all that is needed.

2.4.1. Workflow 1: Initial Login / Expired Session

This workflow covers the initial login scenario, when a session is created for the first time, as well as when a session has expired or been revoked in both the OAuth2 Proxy and Keycloak. Each step is briefly described below and illustrated in Figure 11.

1. A user sends a HTTPS request to a protected resource (RAG system) via a browser.
2. The request is intercepted by the ingress and forwarded to the OAuth2 Proxy for authentication purpose.
3. The OAuth2 Proxy checks the request for an existing session cookie. As no cookie is found (initial access / expired session) the OAuth2 Proxy initiates the authorization workflow.
4. The browser is redirected to the Keycloak login page.
5. The user enters his credentials and authenticates himself within Keycloak.
6. Upon successful login, Keycloak issues an authorization code and redirects the browser back to the OAuth2 Proxy including the authorization code.
7. The OAuth2 Proxy now directly communicates with Keycloak to exchange the authorization code for an access token, a refresh token and an ID token.
8. The OAuth2 Proxy validates the received token and extracts the user information.
9. The OAuth2 Proxy generates and sets the session cookie in the browser, stores the necessary information in its database, and redirects the browser to the originally requested URL.

Addition: If a user enters an incorrect password, they will receive an error message and be able to try logging in again. This is not explicitly shown in the figure below for simplicity.

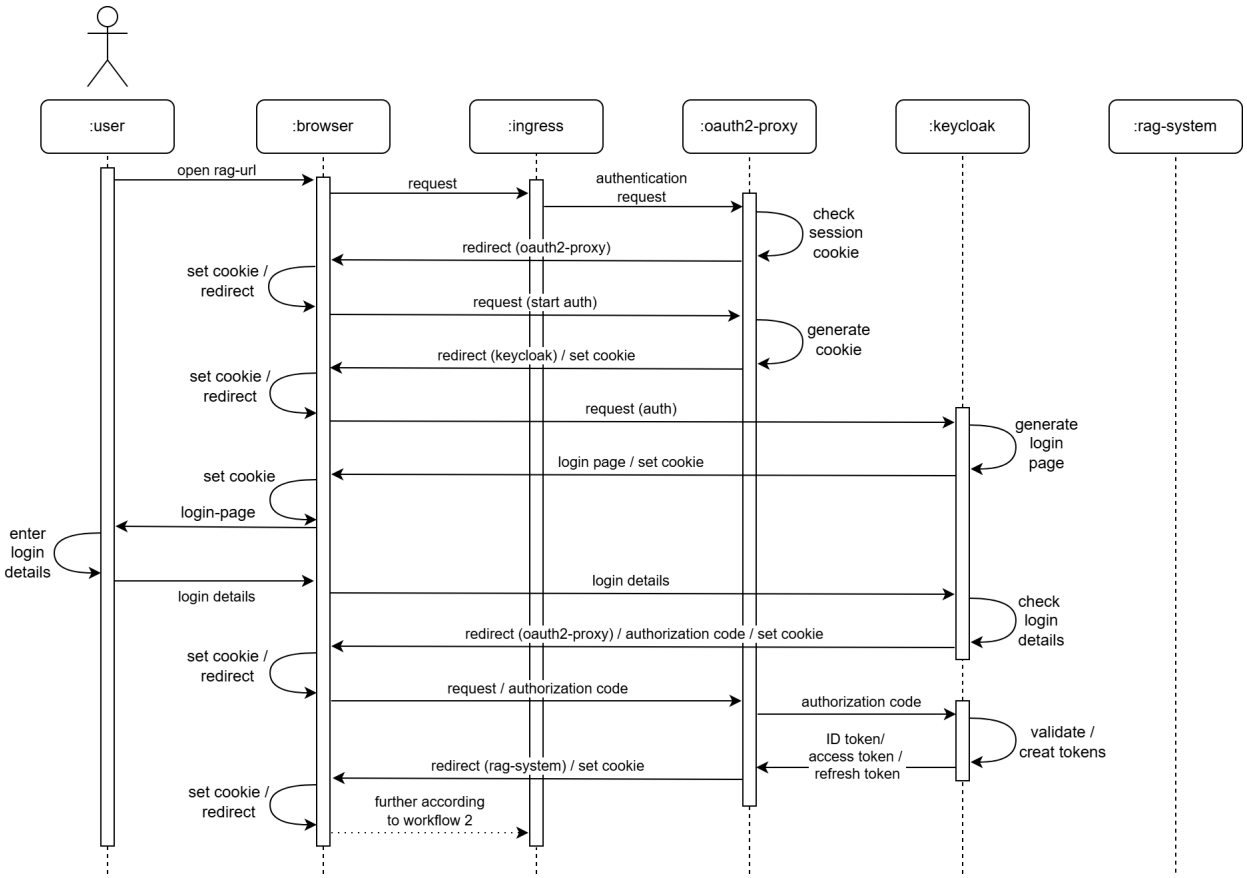

Figure 11: Authentication Workflow of Initial Login / Expired Session - own presentment

2.4.2. Workflow 2: Valid Access Token

This workflow applies to most user requests, as the access token remains valid for a specified period following successful authentication or token refresh. As long as the token is valid, this workflow is followed. Each step is briefly described below and illustrated in Figure 12.

1. A user sends a HTTPS request to a protected resource (RAG system) via a browser.
2. The request is intercepted by the ingress and forwarded to the OAuth2 Proxy for authentication purpose.
3. The OAuth2 Proxy checks the request for an existing session cookie. Since a session was previously established and is still active, a session cookie was sent with the request.
4. The session cookie is checked and found to be valid and not expired.
5. The OAuth2 Proxy responds with a 200 OK response.
6. Based on this response, the ingress knows that the request from the user has been successfully authenticated and forwards it to the RAG system.
7. The RAG system performs its intended function and sends the response back to the browser.

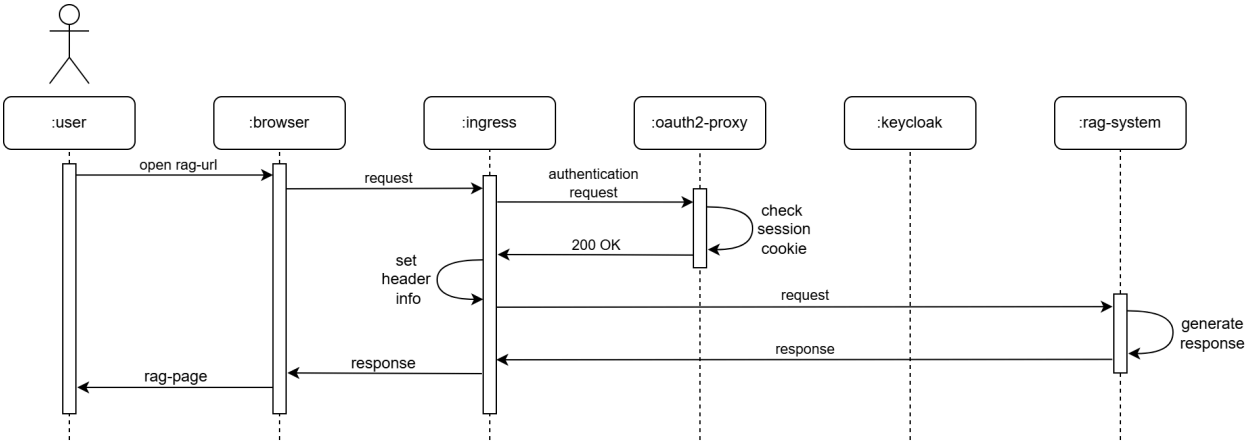

Figure 12: Authentication Workflow of Resource Access with a Valid Access Token - own presentment

2.4.3. Workflow 3: Token Refresh with Valid Refresh Token

When a session or token expires, there are different ways to handle it. The concrete handling depends heavily on the configuration. Our configuration uses refresh tokens. This means that the OAuth2 Proxy tries to refresh the session with Keycloak periodically on request after a certain amount of time. If everything is still okay, this happens without any user interaction. If something has changed or the token is no longer valid within Keycloak, the user will need to re-authenticate. Using refresh tokens has several advantages, including fewer redirects for better performance and potentially faster detection of revoked sessions for better security.

In this project, the OAuth2 Proxy is configured to validate and renew access and refresh tokens. After ten minutes, once the access token expires, the proxy uses the refresh token to renew these tokens. This ensures that, if a user or session is revoked or deactivated, access to protected resources is withdrawn within ten minutes at most. Validation is automatically triggered on any request if more than ten minutes have passed since the last check.

This workflow is illustrated and described in Figure 13. However, it will not be described in detail because it is largely the same as workflow 2 (Part IV, Section 2.4.2.). The only difference from workflow 2 is that the OAuth2 Proxy requests a new access token and, in our configuration, a new refresh token with the existing refresh token. It also restarts the timer for the access token until the next refresh. Of course, this is only the successful case. If the refresh token were not valid, a re-authentication would be requested.

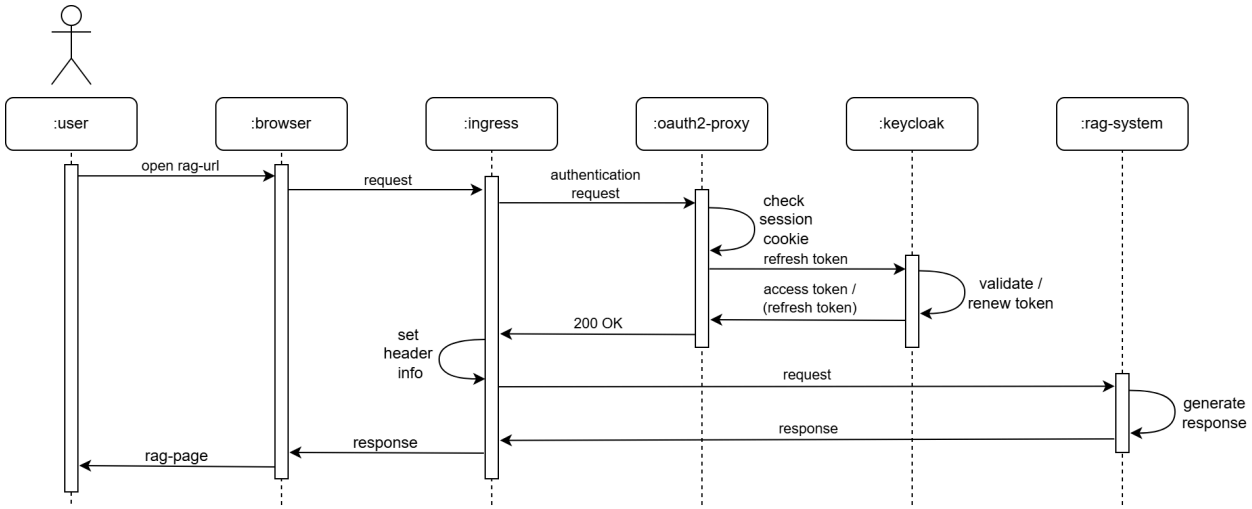

Figure 13: Authentication Workflow of Resource Access with a Token Refresh - own presentment

2.4.4. Workflow 4: Expired Access and Refresh Token

If the access and refresh tokens within the OAuth2 Proxy have expired, the system must start the authentication workflow from scratch as described in workflow 1 (Part IV, Section 2.4.1.). However, if the user is still authenticated within the identity provider, the login page will not be displayed and the user will only see some redirects. The rest is handled automatically. Since the steps are mostly the same as in workflow 1, they will not be described in detail here again, but only the differences will be highlighted. Furthermore, the entire workflow is illustrated in Figure 14.

The only difference to workflow 1 is that the user does not have to enter his credentials because the browser already has a valid session cookie for Keycloak. Since this session is still active, the authentication works without any user interaction.

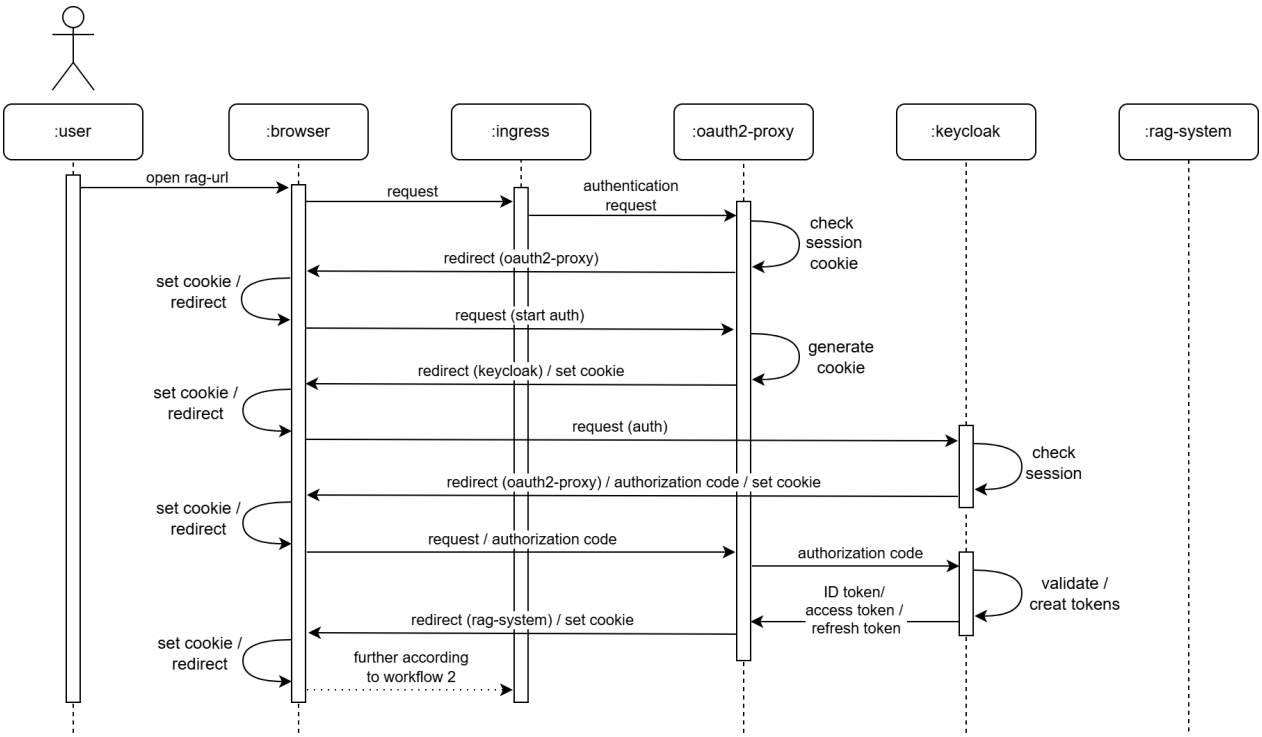

Figure 14: Authentication Workflow of Resource Access with Expired Tokens - own presentment

2.4.5. Workflow 5: Logout

The logout process starts when a user clicks the logout button on the RAG system's webpage. This sends a logout request to the OAuth2 Proxy, which removes the corresponding session cookie and redirects the browser to the Keycloak logout endpoint. Keycloak then destroys the SSO session and invalidates all tokens associated with that session. After that, Keycloak sends a redirect request to the base URL and the whole process starts over from scratch as described in workflow 1 (Part IV, Section 2.4.1.).

The entire workflow is illustrated in Figure 15.

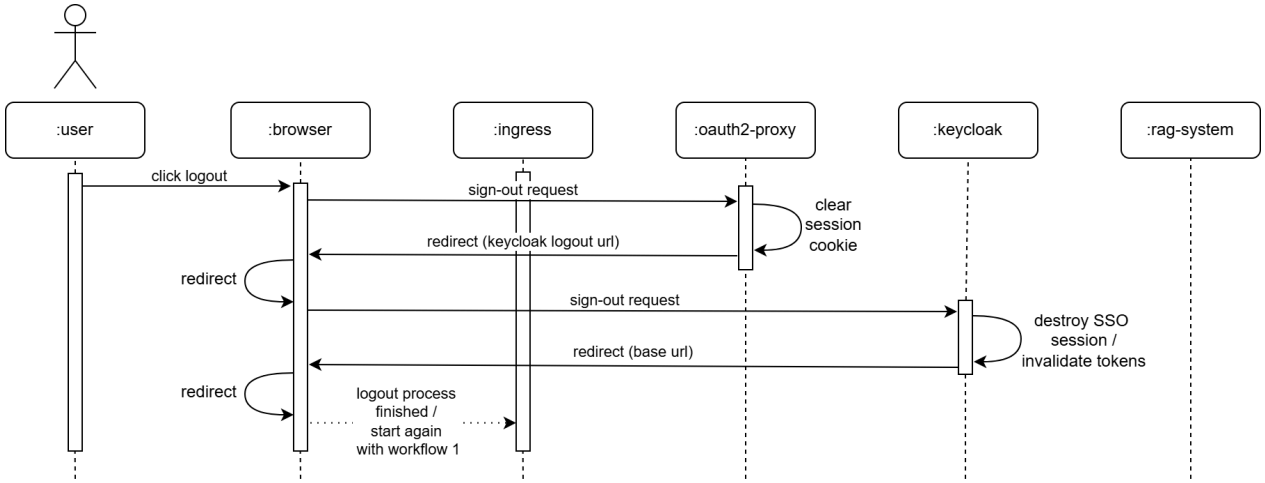

Figure 15: Authentication Workflow of Logout - own presentment

Part V
Implementation

1. MVP Pipeline

In order to gain practical experience in building a Retrieval-Augmented Generation (RAG) pipeline, the first iteration of the project focuses on developing a Minimal Viable Product (MVP). This MVP represents a fully functional, but simplest version of the pipeline. All of the relevant architectural decisions for the MVP have been made and documented in Part III, Section 1.1.5..

Since this pipeline is an incremental step in the project, not all aspects are covered in detail here. Additional information can be found in the descriptions of subsequent development increments.

1.1. System Architecture

Figure 16 visualizes the different components of the MVP pipeline and how they interact. This diagram provides a high-level overview of the components and their interactions. Details such as scaling have been intentionally omitted to keep the diagram concise. Descriptions of the components and their interactions can be found below the figure.

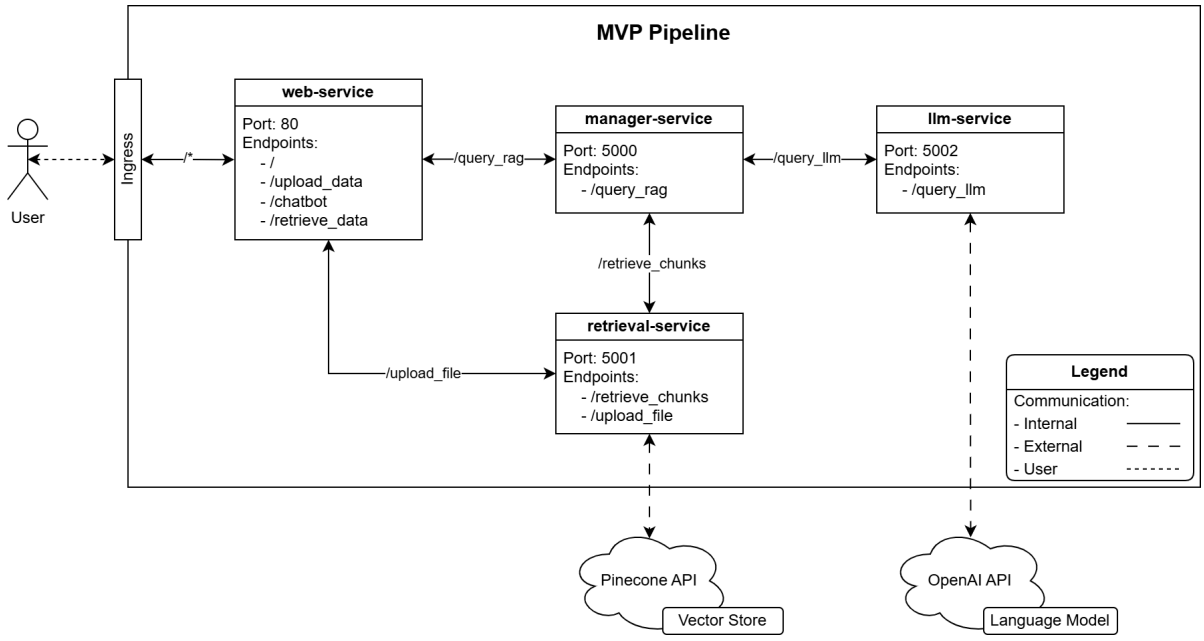

Figure 16: Architecture Diagram of the MVP Pipeline - own presentment

To clarify the role and interaction of each component within the pipeline, a brief description of each is provided below. In addition, the custom components are described in more detail in the corresponding components section (Part V, Section 1.3.).

1.1.1. Components

Interface:

- Ingress: The ingress component is the sole point of contact between the system and the user. Implementation-wise, it is a reverse proxy that handles, redirects, and distributes user requests accordingly.

Internal:

- Web Service: A user interface that allows interaction with the system. Functionality includes uploading PDF files, retrieving data, and answering questions based on the uploaded data.
- Manager Service: The central component of the pipeline that orchestrates all the actions required to generate responses.
- Retrieval Service: Responsible for handling file uploads, extracting relevant chunks, vectorizing and storing data, and retrieving the most relevant data.
- LLM Service: Processes requests to the Large Language Model (LLM).

Cloud:

- Vector Store: Stores vectorized data and retrieves data based on queries.
- Language Model: Generates a response based on the given query.

1.1.2. Connections

Internal:

- Ingress / web-service: Forwards user requests to the web-service.
- web-service / manager-service: Requests to query the RAG system are sent to the manager-service.
- web-service / retrieval-service: Sends PDF files for ingestion purpose to the retrieval-service.
- manager-service / llm-service: Requests to query the language model are sent to the llm-service.

External:

- retrieval-service / Vector Store: Add new data to the vector store and retrieve data based on search queries.
- llm-service / Language Model: Request generation of LLM based response based on a query.

User:

- User / Ingress: Interactions with the system by sending requests and receiving responses.

1.2. Workflow

In this section, the two different workflows regarding the MVP pipeline are visualized as a sequence diagram in Figure 17 and described further below.

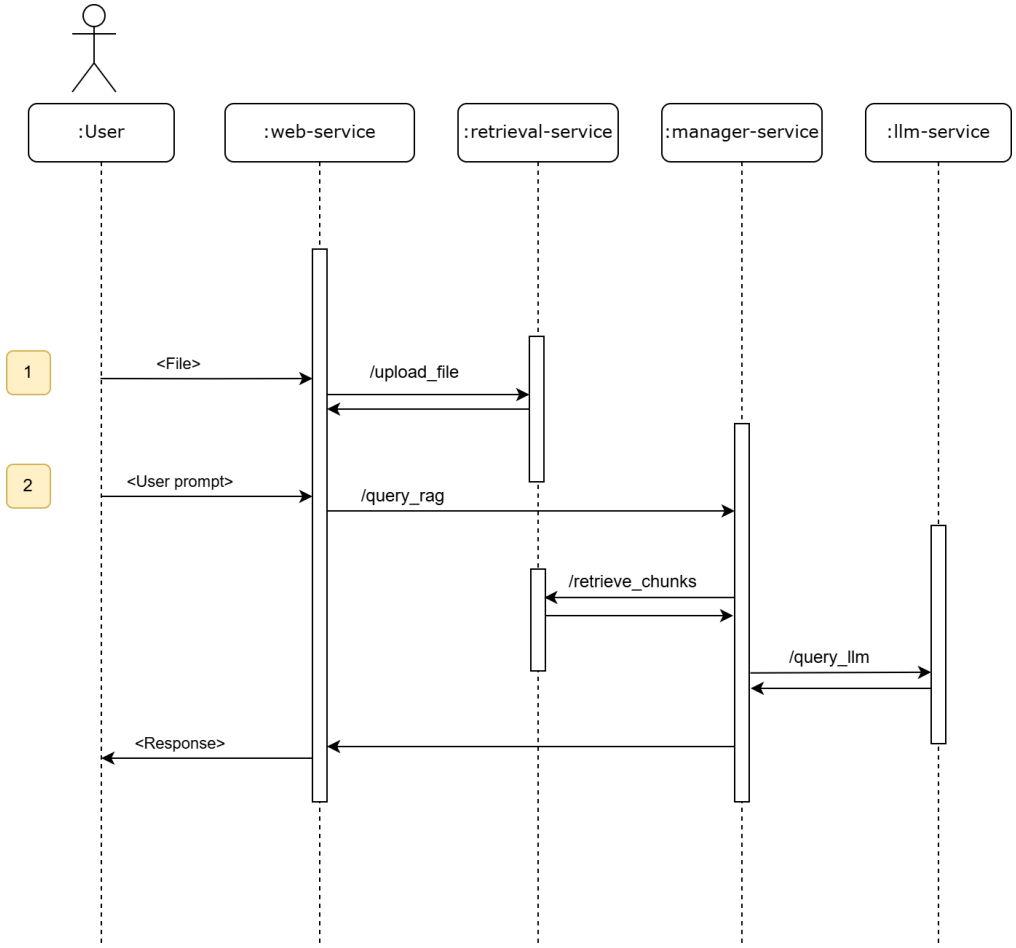

Figure 17: Sequence Diagram of the MVP Pipeline - own presentment

1.2.1. Workflow 1: File Upload

The file upload process begins when the user selects and uploads a file through the frontend interface. Once uploaded, the file is transmitted to the retrieval service, which is responsible for processing the data. This includes splitting the file into smaller segments (chunks), vectorizing the content for efficient retrieval, and storing the processed data in a vector database.

1.2.2. Workflow 2: Prompt Processing

When a user submits a prompt to the chatbot through the web interface, it is first sent to the manager service. The manager service then interacts with the retrieval service to get the most relevant chunks of stored data. The retrieval service then retrieves these chunks from Pinecone, a cloud-based vector store (omitted in the figure). The retrieved data, along with the user's prompt, is then sent to the LLM service. Using the capabilities of OpenAI, the LLM service generates a contextually relevant response, which is then returned to the user via the web interface.

1.3. Components

In this chapter the different components of the MVP pipeline are described in a more detailed manner.

1.3.1. Web Service

The web service, which was built using Streamlit, provides a web interface for interacting with the system. It offers the following features:

- Upload File: Allows users to upload a PDF file, which is then processed into chunks and stored in the vector database.
- Chatbot: Enables users to ask questions about an uploaded document and receive context-aware responses.
- Admin View: Provides insight into the retrieval process by displaying the most relevant chunks for a given query.

This simple yet effective frontend makes it easy for users to interact with the system while also providing transparency into how data is being processed.

1.3.2. Manager Service

The manager service is responsible for bringing the various services together and handling the entire RAG process. This starts with accepting requests, retrieving documents via the retrieval service, creating a query for the LLM, generating a response via the llm service, and returning the response. All of these steps together make up the most basic RAG pipeline.

Combining these steps into a pipeline is achieved through the LangGraph framework. The workflow is illustrated in Figure 18.

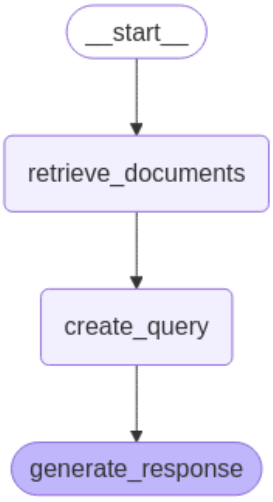

Figure 18: LangChain Graph of the MVP Pipeline - own presentment

1.3.3. Retrieval Service

The retrieval service manages document chunking, embedding generation, interaction with the vector store, and document retrieval. It processes PDF files by extracting their text and dividing it into meaningful chunks. Embeddings for these chunks are generated using OpenAI services. The resulting data is stored in a cloud-based vector database provided by Pinecone, which offers a free tier suitable for small-scale applications, making this solution cost-effective. As the project progresses, the plan is to transition the vector storage to an in-house system.

The retrieval service exposes two endpoints:

1. Upload Endpoint: Accepts a PDF file, processes it into chunks, and stores them in the vector database.
2. Query Endpoint: Retrieves the most relevant chunks from the vector store based on a query provided by the user.

Figure 19 shows an example of retrieving the most relevant chunks for a given word using the Postman tool.

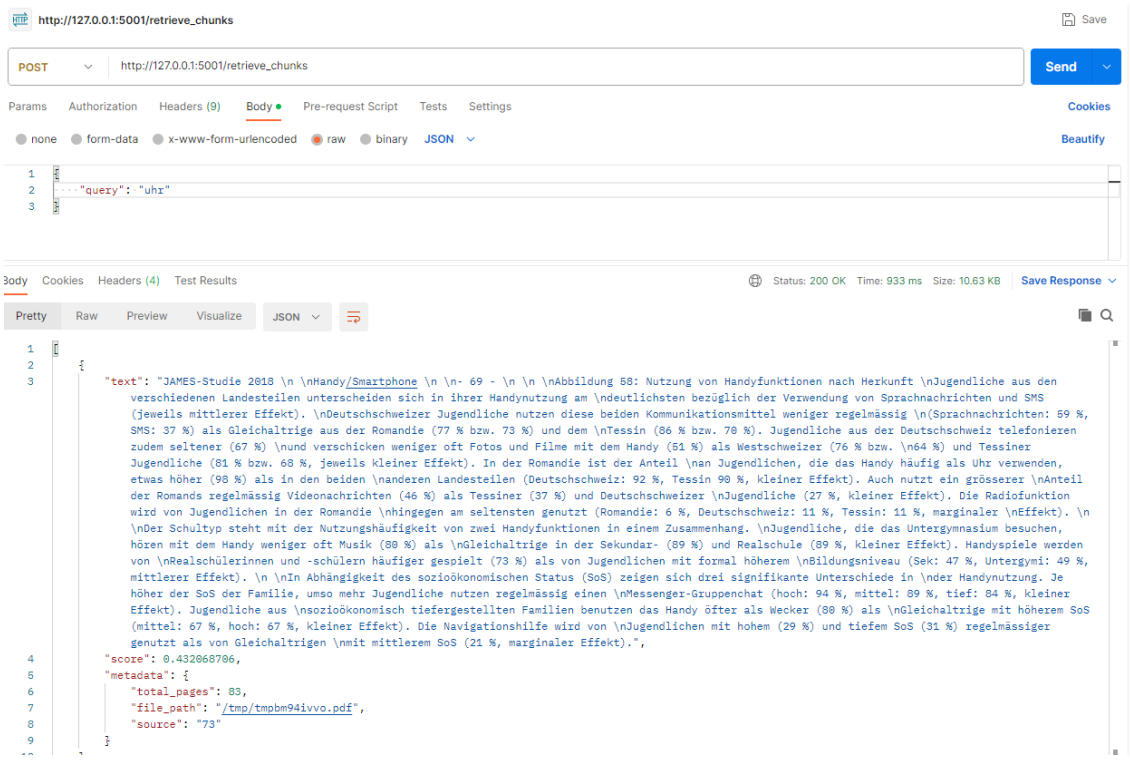

Figure 19: Retrieve the Most Relevant Chunks for “uhr” - own presentment

1.3.4. LLM Service

The LLM Service is responsible for managing queries to a Large Language Model (LLM). In this most basic pipeline, the service receives incoming queries and forwards them to OpenAI’s API for processing. The generated response from the LLM is then returned to the requester.

2. Basic Pipeline

The basic pipeline is an incremental development built upon the MVP pipeline. There are several major improvements to the basic pipeline over the MVP pipeline, as noted and described below.

- Authentication: Each request to the system is now authenticated before being passed to the pipeline. See the authentication section (Part IV, Section 2.) for more details.
- User awareness: The system is now aware of the logged-in user. In this sense, it can distinguish between different users and handle requests accordingly.
- Local vector store: The storage, processing, and retrieval of documents are now handled completely locally, without any request to an external service.
- Chat message history: The system is now able to handle chat histories. These histories are included in the whole RAG process and are independent for each user.

In order to implement these enhancements, several new services and modifications to existing services had to be implemented. These changes are described in the following sections.

2.1. System Architecture

Figure 20 visualizes the different components of the Basic pipeline and how they interact. This diagram is intended to give a high-level overview of the components and how they interact. Further details, such as scaling, are intentionally omitted to keep the diagram clear. Each component and their interactions are described below the figure.

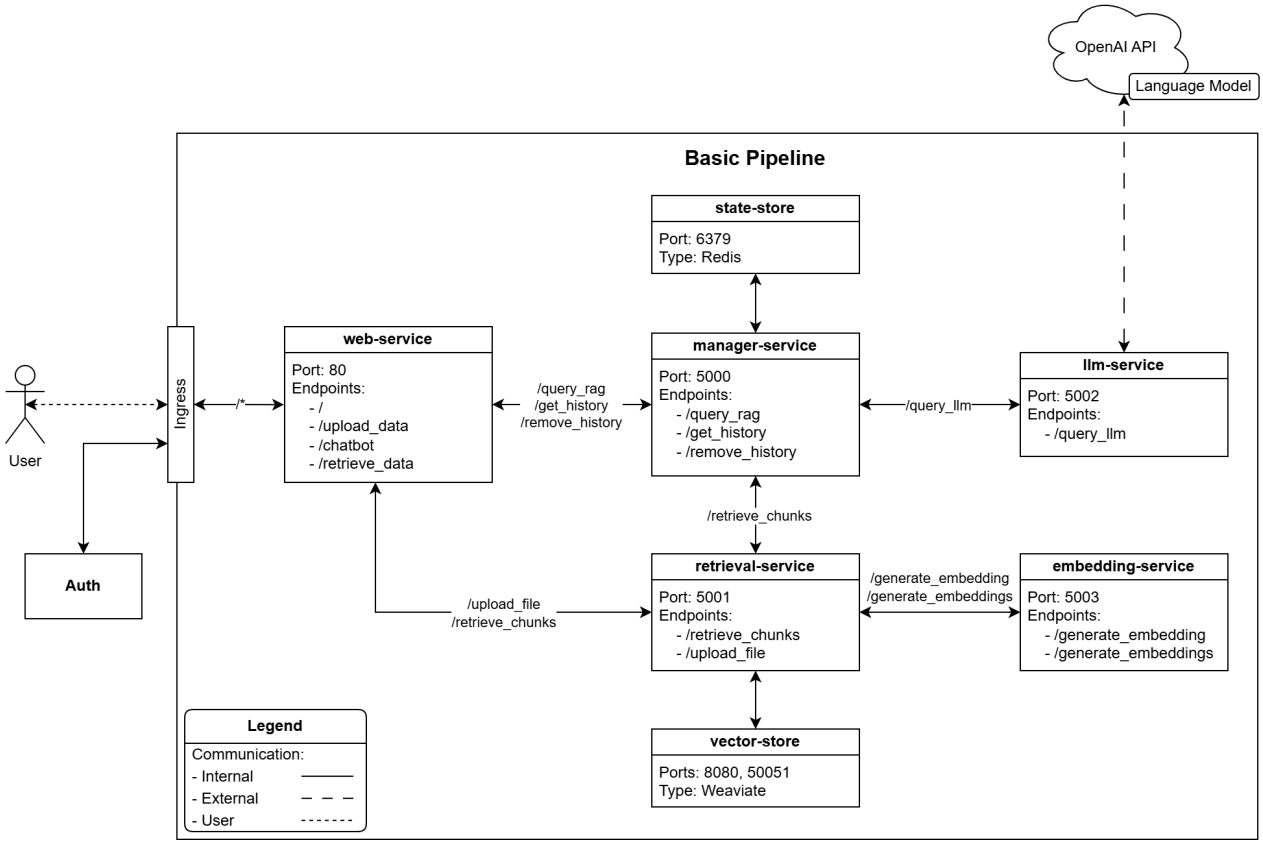

Figure 20: Architecture Diagram of the Basic Pipeline - own presentment

To clarify the role and interaction of each component within the pipeline, a brief description of each is provided below. For completeness and to facilitate a comprehensive understanding of the overall archi-

tecture, all components are described - even those that remain unchanged from the MVP setup. In addition, the custom components are described in more detail in the corresponding components section (Part V, Section 2.2.).

2.1.1. Components

Interface:

- Ingress: The ingress component is the sole point of contact between the system and the user. Implementation-wise, it is a reverse proxy that handles, redirects, and distributes user requests accordingly. In addition, the ingress is now responsible for initiating and responding to the authentication workflow.
- Auth: The auth container, including several components, is responsible for handling the entire authentication workflow, as described in detail in the corresponding section (Part IV, Section 2.).

Internal:

- Web Service: A user interface that allows interaction with the system. Functionality includes uploading PDF files, retrieving data, and answering questions based on the uploaded data. It now also includes chat history functionalities.
- Manager Service: The central component of the pipeline that orchestrates all the actions required to generate responses. Now, the service also manages each user's chat history and incorporates it into the process as needed.
- Retrieval Service: Responsible for handling file uploads, extracting relevant chunks, storing data, and retrieving the most relevant data. The service now interacts directly with the local vector store instead of an external API.
- LLM Service: Processes requests to the Large Language Model (LLM).
- Embedding Service: Generates all the embeddings needed within the system. This includes data chunks during the ingestion process, as well as user prompts during RAG querying and use of the admin view.
- State Store: Stores the user's chat history.
- Vector Store: Stores the user's documents, including a vector representation for each chunk of data.

Cloud:

- Language Model: Generates a response based on the given query.

2.1.2. Connections

Interface:

- Ingress / Auth: Manages the communication required for the authentication.

Internal:

- Ingress / web-service: Forwards authenticated user requests to the web service.
- web-service / manager-service: Requests to query the RAG system are sent to the manager-service. Additionally, the manager-service can be used to retrieve or remove a specific user's chat history.
- web-service / retrieval-service: Sends PDF files for ingestion purpose and requests to retrieve appropriate chunks for the admin view (debugging).
- manager-service / llm-service: Requests to query the language model are sent to the llm-service.
- manager-service / state-store: Add, retrieve, and remove message data.
- retrieval-service / embedding-service: Retrieve embeddings for prompts or batches of data.
- retrieval-service / vector-store: Add new data to the vector store and retrieve data based on embedded search queries.

External:

- llm-service / Language Model: Request generation of LLM-based response based on a query.

User:
- User / Ingress: Interactions with the system by sending requests and receiving responses, including completing authentication actions.

2.1.3. API Endpoints

The four microservices manager-, retrieval-, llm- and embedding-service provide different API endpoints for interacting with each service. The web service also provides endpoints, but since the focus is on API endpoint and the web endpoints are easily accessible through a web page, they are not discussed further here.

Table 4 lists all API endpoints, including the path, the related service, the method, and a brief description of each endpoint. This table provides only a brief overview of the endpoints. For more details, refer to the Swagger documentation for each service. The Swagger documentation is available at each service URL under the /docs path. Screenshots of the Swagger documentation for each API endpoint are also included in the appendix (Part IX, Section 7.).

Table 4: API Endpoints in the Basic Pipeline

Path	Service	Method	Description
/query_rag	Manager Service	POST	Query the RAG (Retrieval-Augmented Generation).
/get_history	Manager Service	POST	Get the message history for a defined owner_id.
/remove_history	Manager Service	DELETE	Remove the message history for a defined owner_id.
/upload_file	Retrieval Service	POST	Upload a file to the system. The file is preprocessed and stored in the vector store.
/retrieve_chunks	Retrieval Service	POST	Retrieve the most relevant chunks based on a user prompt.
/query_llm	LLM Service	POST	Get a response from the LLM based on a query (user prompt + chat history + retrieved chunks).
/get_embedding	Embedding Service	POST	Get an embedding vector for a single prompt/query.
/get_embeddings	Embedding Service	POST	Get a list of embedding vectors for multiple chunks of data.

2.2. Components

The following subsections describe the components of the basic pipeline in more detail.

2.2.1. Web Service

The web service provides the user interface of the system. It allows users to upload files, retrieve chunks, ask questions, and receive answers based on the uploaded documents. The web service is built using Streamlit, a Python library that makes it straightforward to create interactive web applications.

2.2.1.1. Chat History

An enhancement introduced in the transition from the MVP to the basic pipeline is the addition of chat history support within the frontend. When a user accesses the chat page, the chat history is retrieved from the manager services and displayed accordingly. A dedicated button in the sidebar allows users to delete their chat history and start with a new chat.

Figure 21 illustrates an example of a search query using the chat history feature of the chatbot.

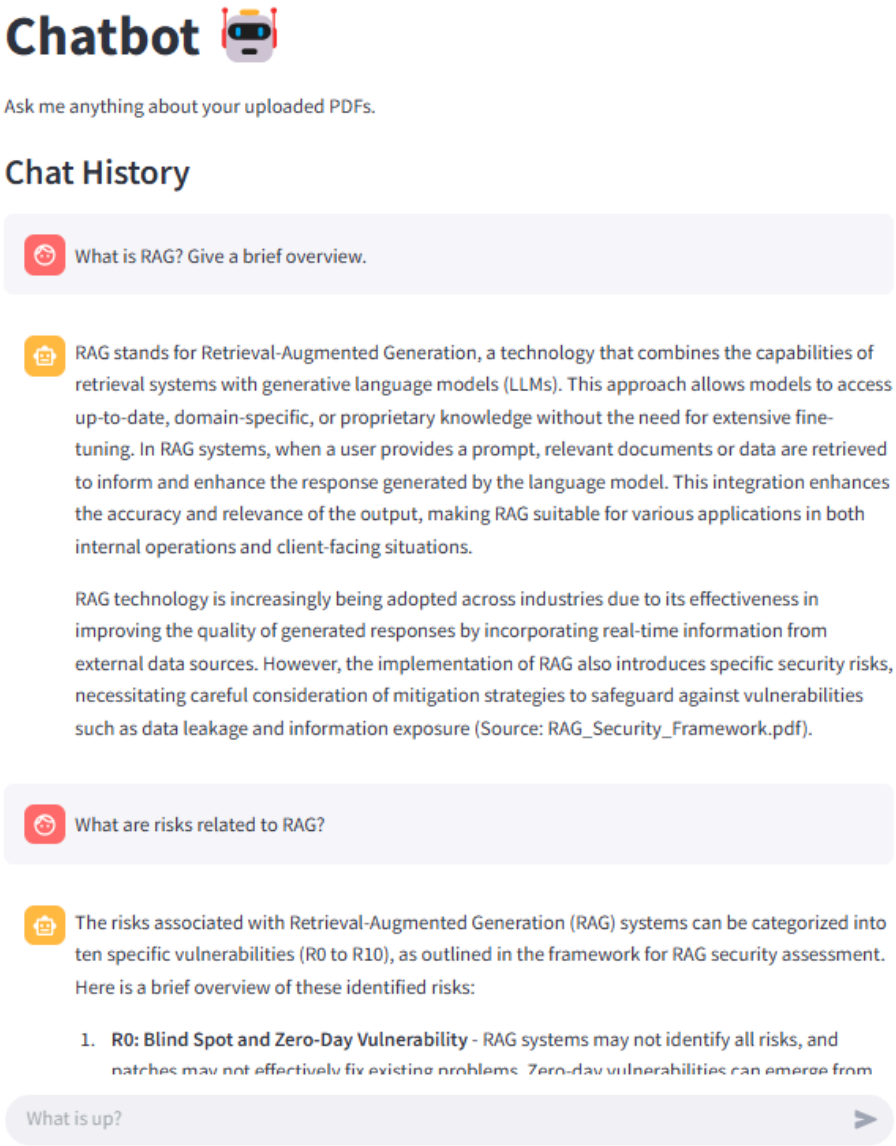

Figure 21: Chat History (excerpt of web service) - own presentment

2.2.2. Manager Service

The manager service is the central component of the pipeline. It orchestrates the entire process, from receiving user queries to generating responses. This section highlights and describes the major changes in the manager service. Three major improvements to the manager service have been adopted. These are the ability to handle chat message histories, user awareness, and the implementation of the citation feature.

2.2.2.1. Chat Message History

The ability to handle chat message histories means that the manager is now able to store message histories and also to include the entire chat message history in the query process so that follow-up questions can be handled and answered in a meaningful way. The chat message history is stored in a Redis database, which is described in the vector store section (Part V, Section 2.2.6.).

2.2.2.2. User Awareness

The term user awareness means that the manager service is now able to distinguish between requests from different users. This results in the ability to retrieve only chunks of documents uploaded by that user. In addition, chat message histories are also user-aware and can only be received by the specific logged-in user.

2.2.2.3. Citation Support

An additional enhancement is the citation feature, which allows the system to reference the source of retrieved content. Specifically, the filename of each document chunk is included in the request sent to the LLM. The LLM is then instructed to incorporate these filenames as citations, indicating the sources used to generate the response.

2.2.2.4. Workflow

In connection with these improvements, several steps of the workflow had to be adapted and some completely new steps had to be added. Each step is briefly described below and the relationship between them is illustrated in Figure 22.

- create_history_aware_retrieval_query: Takes the user prompt and the entire chat history for the specific logged-in user and generates a single retrieval query via an LLM to retrieve the most promising chunks from the system regarding the user query and the chat history.
- retrieve_documents: Based on the retrieval query created, the most promising chunks for the specific logged-in user are retrieved from the vector store via the retrieval service.
- create_query: Creates the query for the LLM to answer the user prompt, including system instruction, the user prompt itself, the retrieved documents, and the chat history of the specific user.
- generate_response: The LLM service is queried to generate the response.
- get_formatted_message_history: Formats the chat message history for use in the frontend.
- add_to_chat_history: Stores the user prompt and the response from the LLM in the state store.

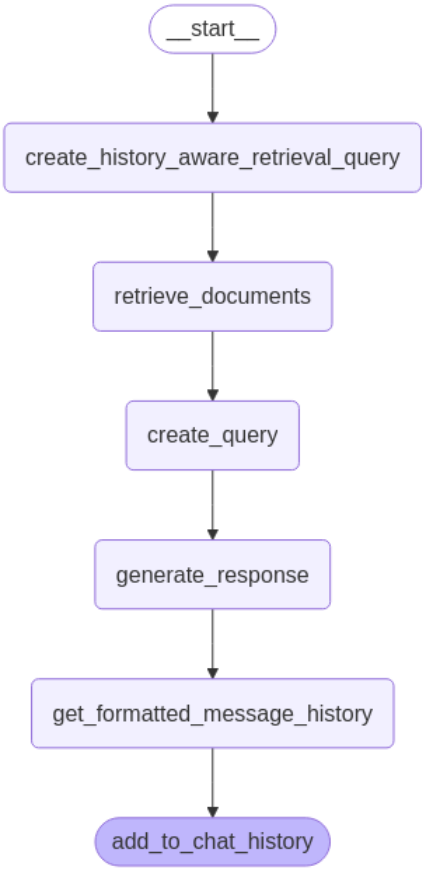

Figure 22: LangChain Graph of the Basic Pipeline - own presentment

2.2.2.5. Endpoints

Two new endpoints have been added to the service. These are the /get_history endpoint and the /remove_history endpoint. The first is used to get the chat history for a specific user for use within the frontend, the second is used to remove the chat history for a specific user to start a new chat from scratch.

2.2.3. Retrieval Service

The retrieval service is responsible for processing incoming requests and interacting with the underlying vector store.

The functionalities of the retrieval service include:
- Handling file uploads.
- Extract text from PDFs.
- Chunk the text into smaller, manageable pieces.
- Store the chunks and their corresponding embeddings in the vector store.
- Retrieve filtered results based on metadata from the vector store. The filter includes the owner_id to ensure that users can only access their own documents.

2.2.3.1. Owner-Based Filtering

To ensure data isolation, each stored document chunk includes an owner_id in the metadata. During the retrieval process, the results are filtered to retrieve only documents where the owner_id matches the user ID of the logged-in user. This ensures that users can only access their own documents, maintaining privacy and security.

2.2.3.2. Citation Support

In order to determine which document a chunk belongs to, the original filename of each document chunk is stored in the vector store. This is accomplished by adding a property field to the vector store that contains the original filename. Thus, when a chunk is retrieved, it can be traced back to its original file.

2.2.3.3. Workflow

The following subsections describe the workflows of the two API endpoints, and illustrate them with a figure.

2.2.3.3.1. File Upload

As illustrated in Figure 23, the process starts with a user uploading a PDF via the /upload_file endpoint. The file is temporarily saved and processed by a PDF extractor, which extracts information from the PDF. The text is then split into smaller, more manageable pieces, called chunks.

These text chunks are then passed to the embedding service to generate vector embeddings. The resulting embeddings, along with their associated text and metadata, including the associated filename and a user-specific owner_id, are sent to the vector store via the Weaviate connector.

The metadata of document chunks is stored to support secure retrieval and features such as document citation.

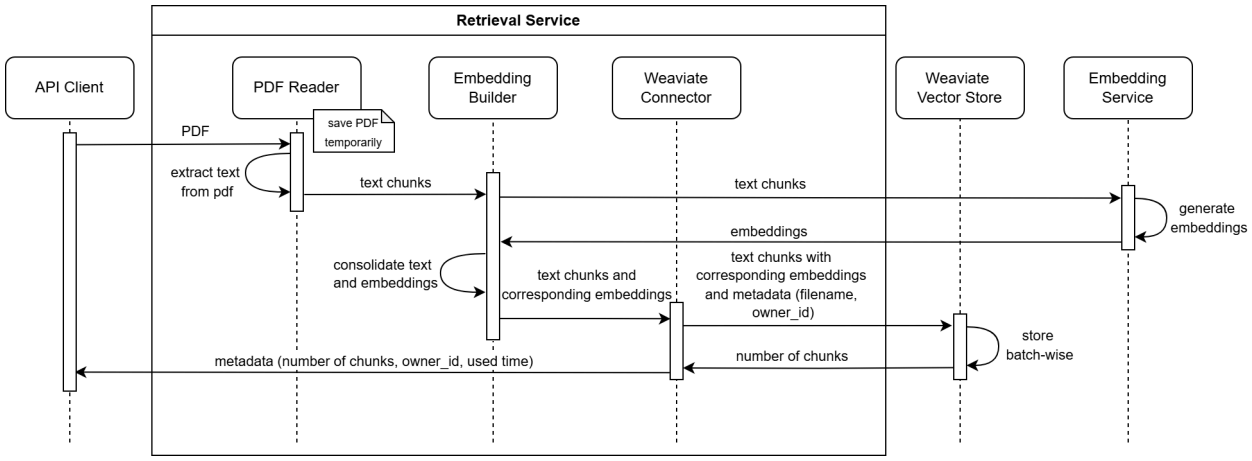

Figure 23: Workflow of Uploading a File - own presentment

2.2.3.3.2. Chunk Retrieval

As shown in Figure 24, the process starts with the client submitting a textual prompt to the /retrieve_documents endpoint.

Upon receiving the request, the retrieval service uses the embedding service to generate the embedding representation of the prompt. This embedding vector is then passed to the Weaviate connector. The connector constructs a query that searches for the top-k most similar vector representations stored in the vector store.

To enforce data privacy, the query includes a filter condition that restricts results to chunks associated with the same owner_id as the requesting user. It also specifies the desired number of chunks to search. The vector store performs a semantic similarity search using the provided prompt embedding, number of chunks, and filter, and returns the filtered top-k matching chunks. The retrieval service then forwards these results to the client.

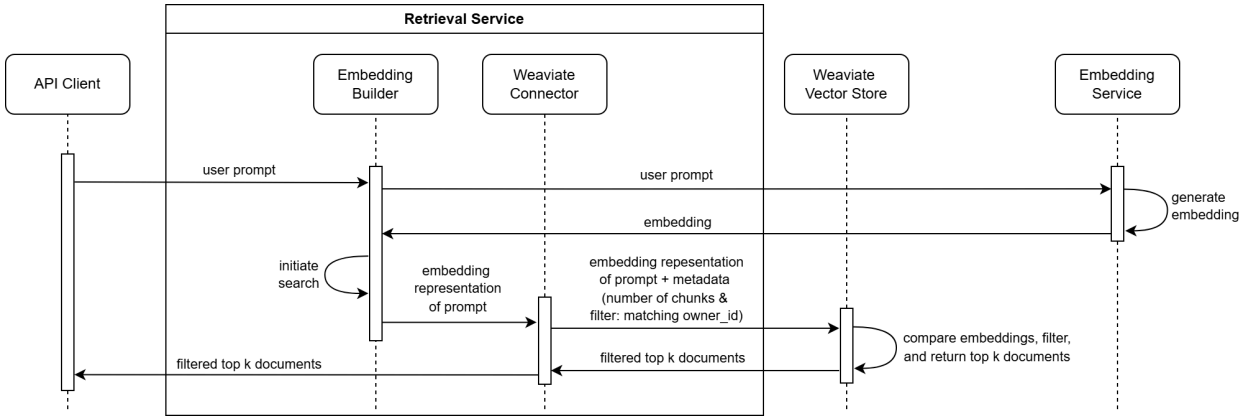

Figure 24: Workflow of Retrieving Chunks - own presentment

2.2.4. LLM Service

The LLM Service remained exactly the same as in the MVP pipeline, as we did not have the time to set up and experiment with local hosting of the LLM, nor did we have the hardware to host the system. For this reason, the service is not discussed again in this section.

2.2.5. Embedding Service

The embedding service was added to the system during the implementation of the basic pipeline. This ensures a strict division of responsibilities regarding the microservices. The implementation allows us to scale the system in a more fine-grained manner, resulting in better utilization of the hardware. It also resulted in significant speed improvements to the overall RAG system.

The service introduced two new endpoints to the system. The first (/generate_embedding) is responsible for generating an embedding representation of a single prompt or query to search the vector store. The second (/generate_embeddings) is responsible for generating the embedding representation of an entire batch of data during data ingestion.

As an embedding model, the sentence transformer model, all-MiniLM-L6-v2, from the Hugging Face platform was used. This model uses a 384-dimensional vector space [85] to map sentences or paragraphs.

2.2.6. State Store

As reasoned in the architecture section (Part III, Section 5.2.) Redis is used to store state information. Furthermore, the corresponding Redis section (Part III, Section 5.3.) argues for a minimal six-node architecture. The concrete implementation of this architecture is described in more detail below.

2.2.6.1. Architecture

The aforementioned six-node cluster architecture is shown in Figure 25. As there are three master nodes, the data within the database is sharded across these three nodes. Each of the three master nodes is responsible for keeping the current state of the data for the slots it is responsible for [86]. The three replicas, on the other hand, contain a replication of the data from the corresponding master node. Write operations can only be performed on master nodes, while read operations can be performed on both master and replica nodes.

This architectural approach results in higher read performance and ensures that data is available even if a master node fails. In the event of a master node failure, the appropriate replica can take over the master role to enable write operations. All of this coordination is done through the so-called gossip protocol, where each node communicates with every other node. All nodes collectively decide how to establish or change the role of each node.

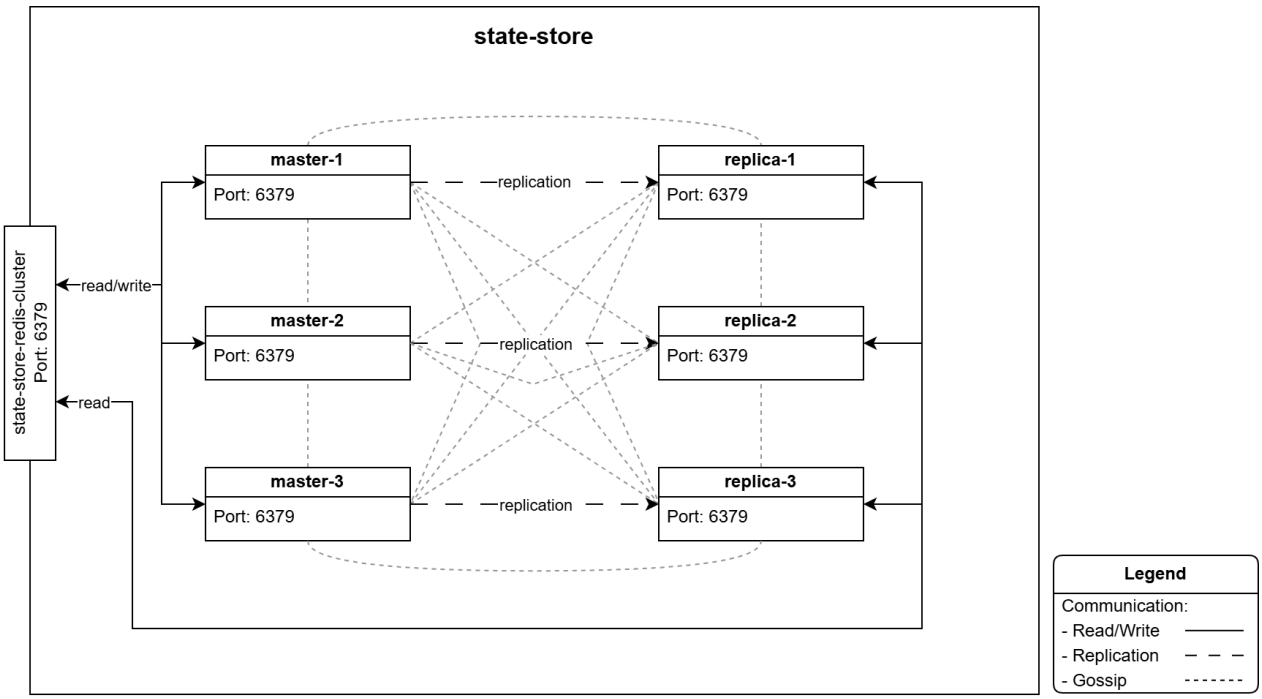

Figure 25: State Store Architecture - own presentment

2.2.6.2. Client

Under the hood, Redis uses a system of hash slots. Each master node is responsible for a subset of these hash slots. The client knows which node is responsible for which hash slot and can thus compute the correct master to send the data to. On the other hand, it can compute which master or replica has the desired data and send the read request to the appropriate node. For this project, the Python client redis-py is used to handle this logic.

Figure 26 provides a visual representation of the aforementioned description.

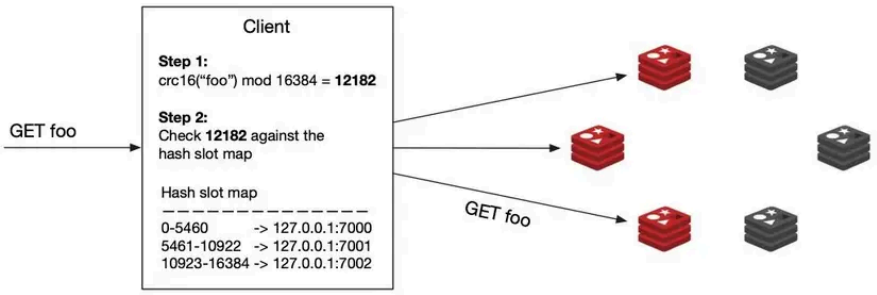

Figure 26: Redis Client - source: [87]

2.2.7. Vector Store

The vector store is implemented using a local instance of Weaviate, an open-source vector database that supports both internal and external vectorization. Both approaches have been evaluated and compared (Part III, Section 3.). Weaviate provides a robust and flexible platform for storing data, including high-dimensional vector representations and the ability to efficiently search the dataset.

2.2.7.1. Design and Configuration

- Vector Source: The system uses pre-generated vectors that are created by the embedding service during document ingestion.
- Schema: Each document chunk in Weaviate includes the fields text, owner, filename and of course the vector representation.
- Filtering: Queries can include conditions using Weaviate's filter option to limit results based on the owner_id, for example.

2.2.7.2. Architecture

In our system, we use a two-node Weaviate cluster running on Kubernetes. Weaviate is leaderless, meaning that all nodes can accept writes and reads from the client [88]. It also allows for sharding and replication of data, resulting in high performance and availability.

Each Weaviate node is represented by a pod in the Kubernetes cluster. Data is organized into collections, each of which is internally divided into shards. A shard represents a portion of the data in a collection. Weaviate uses consistent hashing to distribute these shards across the nodes. By default, a collection has at least one shard. In the current deployment, data is sharded between the two nodes. In addition to sharding, replication can be enabled. When a replication factor is configured, the same shards are replicated across multiple nodes [89].

When data is ingested into the vector store, the Weaviate connector sends the data to one of the nodes. Internally, Weaviate determines the appropriate shard and routes the data to the correct node, ensuring proper data placement. Each object consists of both semantic vector representations and structured metadata (e.g. owner, filename, text).

During query operations, the node contacted by the Weaviate connector performs a distributed vector search. It sends the corresponding embedding vector to all nodes that hold a relevant shard of the target collection. Each node computes its local similarity scores against the data and returns the top-k candidates, which are then merged and ranked by the initiating node before being returned to the caller.

This architecture allows for:

- Horizontal scaling by adding more nodes.
- Fault tolerance through replication.
- Efficient vector-based retrieval across distributed data.

Figure 27 illustrates the interactions between the retrieval service, using the Weaviate client SDK, and the Weaviate cluster.

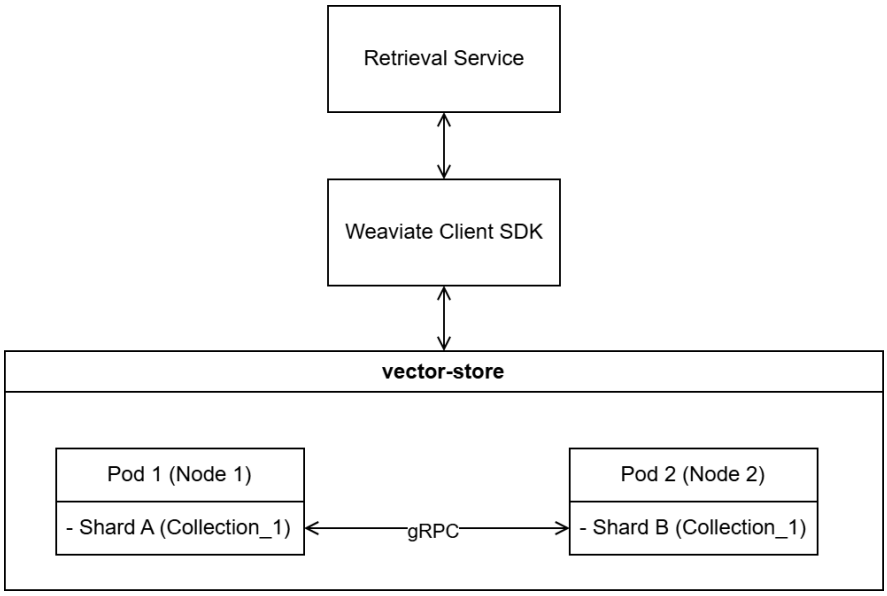

Figure 27: Weaviate Component Diagram - own presentment

2.2.7.3. Performance Notes

As described in detail in the performance testing section (Part VII, Section 1.6.), using externally generated embedding vectors improves data storage speed compared to relying on Weaviate's built-in vectorization service (although using the same embedding model). For this reason, we generate the embeddings using our embedding service and pass them to the vector store instead of letting Weaviate generate them directly.

Part VI
Further Topic

1. Shared Hosting

This section deals with the use case where multiple RAG systems that should be completely isolated (e.g. different companies) are hosted in a single Kubernetes environment. There are several considerations in this case.

Figure 28 shows a scenario where three different RAG systems - each serving a different company - are deployed within a single shared Kubernetes environment. Despite the shared infrastructure, the systems are completely isolated from each other, with no cross-system access allowed. However, there are three specific components where consolidation across system boundaries is found to be beneficial. These common components are identified and discussed in detail below.

- Auth: The authentication component can be effectively shared across all three companies. Separation between organizations is maintained within the authentication system itself by assigning a distinct realm to each company. This design enables each company to operate with its own dedicated authentication endpoint and configuration, thereby preserving strict isolation while enabling resource sharing. Consolidating this component leads to resource savings without compromising the security or autonomy of individual systems.
- Large Language Model (LLM): Consolidating the LLM across systems offers substantial benefits, primarily due to the high computational and memory demands associated with these models. Hosting separate instances for each company would result in considerable resource consumption and increased operational costs. However, for such a shared setup to be viable, it is essential that the LLM does not retain request data beyond what is strictly necessary, and that strong isolation mechanisms are in place to separate client requests. When these conditions are met, consolidation of the LLM is not only feasible, but highly recommended from a resource and cost efficiency perspective.
- Embedding Model: The argumentation for consolidating the embedding model is exactly the same as for the LLM, except that the embedding model usually requires much fewer resources than an LLM. While the performance and cost gains from consolidation are less pronounced, sharing this component still provides measurable benefits in most scenarios. As a result, consolidation is generally recommended, although with a lower priority than the LLM.

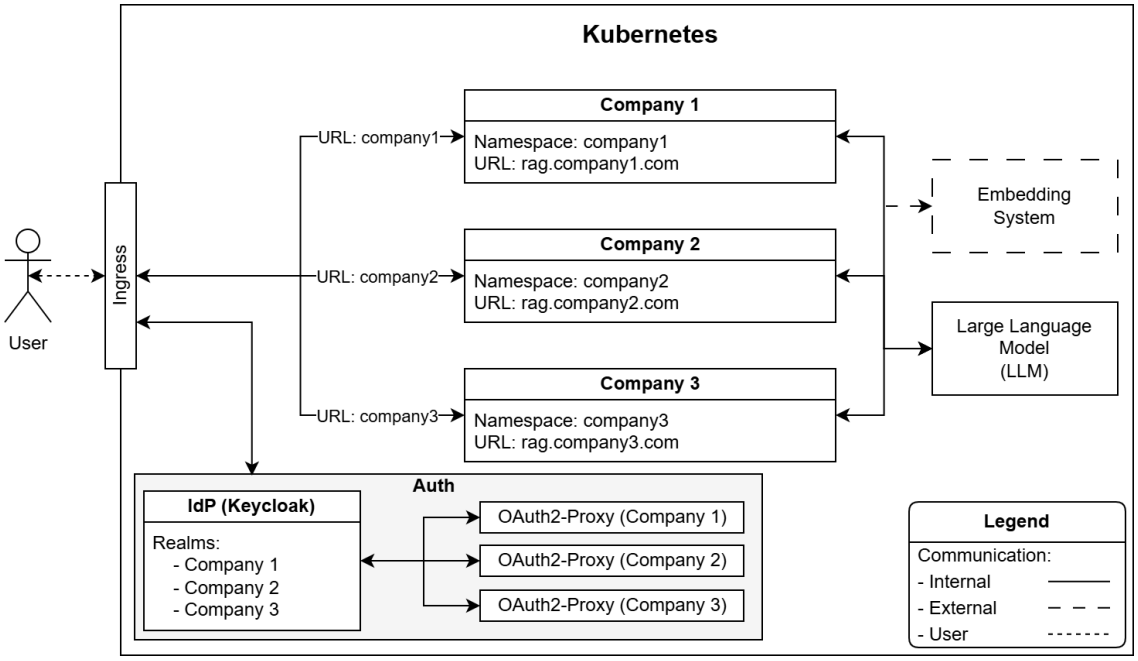

Figure 28: Architecture of Three RAG Systems Hosted in a Shared Environment - own presentment

In conclusion, this shared hosting strategy enables significant resource and cost savings compared to deploying fully independent systems for each organization.

2. Security Considerations

Since this thesis focuses solely on developing the core functionality of a RAG system, not every risk in the following RAG Risk and Mitigation Matrix (Figure 29) is addressed by an appropriate mitigation strategy. Additionally, not every risk requires a mitigation for every use case.

Security was a top priority throughout the planning and development processes of this thesis. The system was designed to be easily adaptable. For this reason, appropriate mitigation strategies can be easily added to the system. Nevertheless, some major mitigations have been incorporated into the core system. This section discusses these strategies.

	Secure Retrieval				System Hardening			Disclosure Prevention	Other Mitigations				
	M0: Anonymization	M1: Pseudonymization	M2: Synthetic Data	M3: Access Limitation	M4: System Instructions	M5: Input Validation	M6: Evaluation	M7: Self-hosted AI-models	M8: Adding Noise	M9: Distance Threshold	M10: Summarization	M11: Re-Ranking	M12: Exposure Minimization
General													
+ R0: Blind spot and zero-day vulnerability	x	x	x	x		x							x
+ R1: System Limitation							x						
Vectorization and Retrieval													
+ R2: Retrieval Data Leakage	x		x	x	x	x	x		x	x	x	x	x
+ R3: Embedding Inversion Attack	x	x	x					x	x				
+ R4: Membership Inference Attack (MIA)	x		x	x	x	x	x		x	x	x		x
+ R5: Retrieval Data Disclosure during embedding	x	x	x					x					x
Operational Risks													
+ R6: Retrieval Data Disclosure during prompting	x	x	x			x		x	x	x	x	x	x
+ R7: Prompt Disclosure	x	x				x		x					
Data Manipulation													
+ R8: Knowledge Corruption Attack				x	x	x	x			x	x	x	
+ R9: Indirect Prompt Injection				x	x	x	x			x	x	x	
+ R10: Indirect Jailbreak Attack				x	x	x	x			x	x	x	

Figure 29: RAG Risks and Mitigation Matrix - source: [9]

2.1. Risks

This section uses the RAG Risk and Mitigation Matrix (Figure 29) to discuss the risks in our core system.

- General: The newest versions of tools and best practices (where applicable) were used during implementation to prevent general risks from happening. However, a production-ready system requires several additional measures, such as proper patch management, vulnerability analysis, and system evaluation.
- Vectorization and Retrieval: The entire vectorization process and storage of vectorized data is handled entirely locally. Thus, a significant step has been taken to mitigate these risks. Furthermore, a restriction is in place that only allows users to retrieve their own documents, which also helps counteract some of these risks effectively.
- Operational Risks: Our system does not currently address this risk because the LLM that generates responses is hosted by a third party. However, the system can easily be adapted to incorporate a locally hosted LLM. This has not yet been implemented due to time and hardware restrictions.
- Data Manipulation: A restrictive authentication system ensures that only authorized individuals can access the system, thereby counteracting these risks. Furthermore, users can only retrieve their own documents, which also helps mitigate these risks.

2.2. Mitigation Strategies

This section uses the RAG Risk and Mitigation Matrix (Figure 29) to discuss the mitigation strategies applied to our core system.

- Secure Retrieval: Since only the core functionality of the system has been implemented, it is unclear what data will be used later on. For this reason, mitigations M0-M2 cannot be properly implemented because this information is lacking. However, M4 access limitation is implemented through an authentication system to restrict users to retrieving only their own documents.
- System Hardening: System instructions are in place, but they are not properly hardened. However, these instructions need to be adapted to each use case, as there is no one-size-fits-all solution. For this reason, it made no sense to harden this step yet. The other two mitigations (M5 and M6) are not currently in place, but they can easily be added.
- Disclosure Prevention: The embedding model is hosted completely locally, which implements this mitigation strategy. However, due to time and hardware constraints, the LLM used to generate responses is still hosted by a third party. Thus, this mitigation strategy is not fully implemented.
- Other Mitigations: The disclosure threshold and exposure minimization are in place to a certain degree, as only the top k documents are retrieved, and access to the system is restricted to authorized users. The other mitigations are not currently in place, but they can easily be added.

3. Workshop

As mentioned in the task definition, another goal of this thesis is that the RAG system built can be used for our workshop Hacking RAG: Exploring Risks and Implementing Mitigations at the IEEE Swiss Conference on Data Science (SDS2025) [10].

3.1. Hosting Setup

The hosting setup for this workshop is very similar to the shared hosting setup described in the shared hosting section (Part VI, Section 1.). The main considerations are briefly described below, and the hosting setup is shown in Figure 30.

- The language model is hosted by a third party for various reasons, including time and hardware constraints.
- All three instances use the same embedding system to save resources. This design choice was already mentioned in the shared hosting section.
- Each of the tasks is located in a different Kubernetes namespace and is therefore logically isolated. Each of them runs all of the necessary components of the RAG system, with the exception of the embedding system, which is hosted in a consolidated fashion. For this reason, we can tailor each system to the purpose of the specific task without interfering with the other systems.
- The same authentication system is used by all three tasks, but with different realms, one for each task system. This, along with a separate OAuth2 proxy instance, keeps the authentication handling completely isolated.

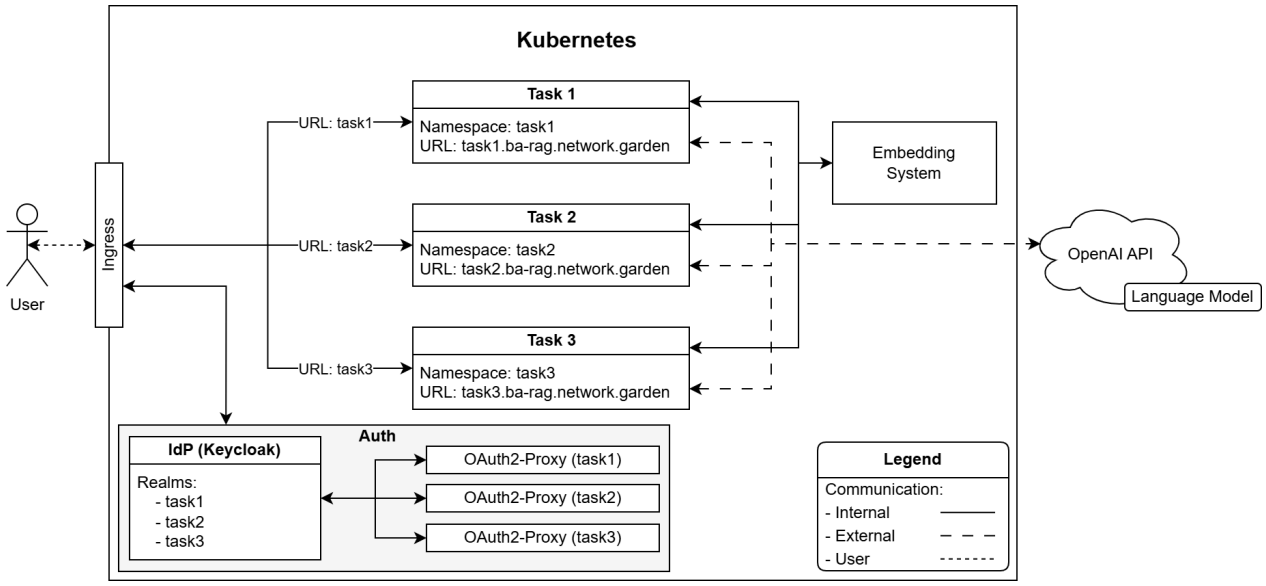

Figure 30: Architecture of the Workshop Infrastructure - own presentment

3.2. Synthetic Data

Throughout the workshop, data is needed for various tasks. We decided to use synthetic patient data to simulate sensitive and thus critical data. Several approaches were considered to create this data.

The first option was to use an existing pre-generated synthetic dataset. One interesting dataset we found was Synthea from the Mitre Corporation [90]. It contains synthetic information about U.S. patients, including demographics, medical history, and insurance information. This dataset has a wide range of attributes that are suitable for testing and developing corresponding applications. However, since the workshop is being held in Switzerland, we would prefer to use Swiss data structures. During our research, we could not find any pre-generated synthetic datasets based on these structures.

Given the reasons above, the decision was made to create our own synthetic patient data. The Faker library, a Python package for generating fake data, was used to produce realistic-looking information, including names, addresses, phone numbers, blood type and insurance details.

The following is an example of the generated data:

```
Patient Name: Simona Bucher
Date of Birth: 1980-10-15
Patient ID: 578046874
Address: Itenstrasse 93, 8610 Uster
Phone Number: +41 77 683 41 84
AHV Number: 756.9091.6998.54
Emergency Contact: Samuel Bucher, +41 63 833 50 40
Blood Type: AB+
Allergies: Nickel
Current Medications: Paracetamol
Past Medical History: Anxiety
Primary Physician: Dr. Joav Ferreira
Insurance Provider: Atupri Versicherung
```

3.3. Workshop Procedure

This section describes the procedure for the workshop, including the three different tasks.

The workshop begins with an introduction to Retrieval-Augmented Generation (RAG) to ensure all participants have a solid foundational understanding of RAG. It then provides a brief overview of the key risks associated with RAG systems and their potential mitigations. Participants will then apply what they've learned by completing three practical tasks. The workshop concludes with a short wrap-up session to summarize the key takeaways.

3.3.1. Introduction

To ensure that all participants start with a shared foundational understanding, the workshop begins with a basic overview of Retrieval-Augmented Generation (RAG), similar to the content provided in Part I, Section 3..

In addition, a Jupyter notebook demonstrating a simple RAG pipeline is provided. This interactive resource allows attendees to explore the individual components and see the workflow in action, offering a hands-on introduction to the concept.

Together, these elements are designed to equip participants with the necessary background knowledge for the upcoming theoretical section and practical tasks.

3.3.2. Risks and Mitigations

The workshop briefly touches on the key risks and mitigation strategies associated with RAG systems, as researched in our study thesis and elaborated on in our published paper. These insights provide a foundation for understanding the challenges of deploying RAG in real-world scenarios.

Both sources are introduced during the session, and direct links are provided so that participants can explore them further, even after the workshop ends. This enables continued learning and reference beyond the scope of the event.

Referenced materials:
- Paper (Securing RAG: A Risk Assessment and Mitigation Framework): [9]
- Study Thesis (Analysis of Risks and Mitigation Strategies in RAG): [8]

3.3.3. Tasks

The hands-on portion of the workshop consists of three tasks designed to give participants practical experience with various RAG-related attacks and corresponding mitigation strategies. The tasks are structured progressively — Task 1 involves minimal security considerations, while Task 3 introduces more complex and layered challenges.

To simulate realistic scenarios for Tasks 1 and 2, the system requires pre-existing data. The impact of the tasks is heightened when this data is treated as sensitive. To support this, we generated synthetic patient data, described in detail in the Synthetic Data section (Part VI, Section 3.2.).

Each task is briefly outlined in the following subsections. Complete details and step-by-step instructions can be found in the following Git repository: https://gitlab.ost.ch/sds_workshop/hacking-rag. Note that these files will be updated after this thesis is submitted.

3.3.3.1. Task 1 - Retrieval Data Leakage

Task 1 serves as an entry-level challenge accessible to all workshop participants. The goal is to simulate a retrieval data leakage attack on a RAG system that lacks specific security measures.

To facilitate this exercise, synthetic patient data is preloaded into a shared vector store. Participants are tasked with probing the system to extract as much information as possible from the data — mirroring a real-world scenario where sensitive information could be inadvertently exposed.

This task introduces the concept of information leakage via retrieval mechanisms and sets the stage for more advanced attacks and mitigations in the subsequent exercises.

3.3.3.2. Task 2 - Membership Inference Attack (MIA)

In Task 2, the RAG system employs the fundamental mitigation strategy of reinforcing system instructions to minimize the disclosure of unintended information. Consequently, the data leakage approach from Task 1 becomes ineffective.

The focus now shifts to a Membership Inference Attack (MIA). Instead of extracting explicit data, participants must determine which data entries were used to populate the retrieval store. To facilitate this, attendees are provided with a set of candidate data samples and are challenged to identify which ones are present in the system and which are not.

This task introduces a more subtle and sophisticated type of privacy threat, encouraging participants to think critically about how even indirect signals can compromise data confidentiality.

3.3.3.3. Task 3 - Knowledge Corruption Attack

In Task 3, the RAG system introduces a second layer of defense named access control. This mitigation strategy ensures that each user can only retrieve documents they have personally uploaded, creating a degree of data isolation across users. Consequently, the data inferring approach from Task 2 becomes ineffective.

Despite these protections, the system is still vulnerable to knowledge corruption attacks. In this task, participants are challenged to manipulate the system's understanding of a topic by introducing specially crafted documents. These documents can target the LLM's general knowledge, reinterpret meanings introduced via system instructions, or override existing information in the retrieval datastore.

Participants are provided with multiple sub-tasks of varying difficulty, which allow them to explore different techniques for subtly or overtly altering the system's responses.

This final task highlights a different aspect of RAG security. Not the leakage of existing data, but the manipulation of the system's knowledge and behavior. While the attack is demonstrated in an isolated environment with user-specific access, it can just as easily be envisioned in a less controlled setting, where malicious inputs could alter shared knowledge, potentially impacting responses for a wide range of users. This scenario underscores the risk of knowledge corruption in collaborative or multi-user RAG systems.

3.4. Conclusion

To wrap up the workshop, the key concepts and takeaways are briefly summarized to reinforce what participants have learned. This includes a recap of RAG fundamentals, the explored security risks, and the practical experience gained through the hands-on tasks.

Finally, participants are once again directed to the provided resources for further study and exploration, enabling them to deepen their understanding and apply these insights beyond the workshop setting.

Part VII
Evaluation

1. Testing

As described in the test concept, several tests were performed. This section documents the various types of tests, as well as their findings and results.

1.1. Preparation

To ensure high quality testing, the following sections define the policy and strategy by which testing should be planned and executed. It also defines the actions to be taken if the tests do not perform as intended.

1.1.1. Software Requirement Specification

The requirements for the thesis are defined in the corresponding requirements section (Part II, Section 1.). For this reason, the requirements are not listed in detail here, but in tabular form to be used throughout the testing phase and to achieve good test coverage.

The last column "Tests" in the following two tables describes whether the requirement is tested with specific test cases or not. If not, the requirement is simply reasoned about, but not tested in the sense of a test case. This may be the case if there is no trivial way to test the case, or if the reasoning simply makes more sense.

1.1.1.1. Functional Requirements

Table 5: Overview of Functional Requirements and Corresponding Test Coverage

Name	Requirement Defintion	Description	Tests
FR1: User Login	Part II, Section 1.3.1.	User login works as intended.	Yes
FR2: Upload PDF File	Part II, Section 1.3.2.	User can upload a PDF file.	Yes
FR3: Ask Chatbot Questions	Part II, Section 1.3.3.	User can ask the chatbot questions and get meaningful answers related to the uploaded documents.	Yes
FR4: Give Citation	Part II, Section 1.3.4.	The chatbot response includes the source of the information.	Yes
FR5: Admin View	Part II, Section 1.3.5.	It is possible to get more insight through an admin view.	Yes

1.1.1.2. Non-Functional Requirements

Table 6: Overview of Non-Functional Requirements and Corresponding Test Coverage

Name	Requirement Defintion	Description	Tests
NFR1: Horizontal Scaling	Part II, Section 1.4.1.	Increased workload can be addressed through successful horizontal scaling.	No
NFR2: Access Restriction	Part II, Section 1.4.2.	A user can only access his own uploaded files and no others.	Yes
NFR3: Implement Local Services	Part II, Section 1.4.3.	The services are fully implemented locally and do not interact with external APIs to accomplish their task.	No
NFR4: API Documentation	Part II, Section 1.4.4.	Endpoint APIs are documented.	Yes
NFR5: Modular Architecture	Part II, Section 1.4.5.	The system is built using a modular architecture.	No

1.1.2. Test Strategy

There are two types of tests that are performed to verify that the functional and non-functional requirements are implemented and work as intended.

The first is functional testing, where individual functionalities are tested to see if they work as intended. The second is end-to-end testing, where the system, including all components and their interactions, is tested as a whole. Both are considered final testing in our case. Of course, we have tested our product extensively during implementation. However, only the final tests are documented in detail.

As noted already, for some requirements, reasoning makes more sense than enforcing a strict test case. These cases are described in the reasoning section.

In addition, a combined load and end-user test is conducted to assess the system's overall usability and performance under load conditions. These test is less constrained and more freely described.

Lastly, some performance tests are conducted without regard to functional and non-functional requirements. These tests are also less constrained and more freely described.

In summary, the following tests are conducted, along with a link to the associated section.

- Functional Tests: Part VII, Section 1.2.
- End-to-End Tests: Part VII, Section 1.3.
- Reasoning: Part VII, Section 1.4.
- Load and End-User Test: Part VII, Section 1.5.
- Performance Tests: Part VII, Section 1.6.

Note: Unless otherwise stated, the basic pipeline is used for all tests.

1.1.3. Test Policy

For each test of functional and non-functional requirements, the following information and results shall be provided, at minimum::

- Number (F: Functional, E: End-to-End)
- Name
- Preconditions (if applicable)
- Test Scenario
- Expected Result
- Was result as expected?
- Date
- Findings
- Success (Yes/No)
- Improvements/Reasoning (not strictly necessary if Success is yes)

1.2. Functional Tests

Functional testing in this thesis refers to the testing of individual functionalities. Each functionality of the system is tested individually to show that each works as intended. Since these tests are heavily based on the requirements, the naming is also based on the requirements.

Table 7 lists the functional tests that were performed. The table only lists whether the tests were successful and the main results. The detailed test protocols can be found in the appendix (Part IX, Section 5.). If more than one test has been run for the same scenario, only the most recent test will appear in this list. All tests performed are listed in the appendix.

Table 7: Overview of Functional Test Cases and Their Outcomes

Name	Protocol	Note	Success
F1: User Login with Valid Credentials	Part IX, Section 5.1.	Worked as intended.	Yes
F2: User Login with Invalid Credentials	Part IX, Section 5.2.	Worked as intended.	Yes
F3: User Logout	Part IX, Section 5.3.	Worked as intended.	Yes
F4: Upload PDF File	Part IX, Section 5.4.	An AxiosError sometimes occurred during file uploads (F4.1). This issue was resolved by implementing sticky sessions (F4.2). More information can be found in the following subsection.	Yes
F5: Ask Chatbot Question - Typical Case	Part IX, Section 5.5.	Worked as intended.	Yes
F6: Ask Chatbot Question - No Related Information	Part IX, Section 5.6.	Worked as intended.	Yes
F7: Give Citation	Part IX, Section 5.7.	Worked as intended.	Yes
F8: Admin View	Part IX, Section 5.8.	Worked as intended.	Yes
F9: API Documentation	Part IX, Section 5.9.	Worked as intended.	Yes

1.2.1. Fix F4 Sticky Sessions

As briefly mentioned in Table 7, test F4 revealed a problem with streamlit's file upload implementation on distributed systems like we use. To solve this problem, sticky sessions were introduced for the web service. The rest of the web service was originally designed to be completely stateless and does not require sticky sessions. But since we are using this streamlit component, we are forced to introduce sticky sessions.

In any case, the stateless implementation results in a more stable system, since the failure of a single pod can only affect the file upload function, which typically takes only a few seconds to a few minutes. All other functions continue to work on another pod without any problems or interruptions.

1.3. End-to-End Tests

End-to-end testing focuses on the functionality of the entire system. These tests can be used to show whether or not the entire system works as intended.

Table 8 lists the end-to-end tests that were performed. The table only lists the main results and whether the tests were successful. The detailed test protocols can be found in the appendix (Part IX, Section 6.). If more than one test has been run for the same scenario, only the most recent test will appear in this list. All tests performed are listed in the appendix.

Table 8: Overview of End-to-End Test Cases and Their Outcomes

Name	**Protocol**	**Note**	**Success**
E1: Access Restriction	Part IX, Section 6.1.	All steps worked as intended.	Yes
E2: Chat History	Part IX, Section 6.2.	All steps worked as intended.	Yes

1.4. Reasoning

This section justifies whether or not each functional and non-functional requirement that cannot be tested in a meaningful way based on functional or end-to-end tests is satisfied. Whether or not tests are performed for the requirement is described in the software requirement specification section (Part VII, Section 1.1.1.). Table 9 summarizes the reasoning conducted and the results obtained.

Table 9: Overview of Reasoning Scenarios and Their Outcomes

Name	**Reasoning**	**Note**	**Success**
R1: Horizontal Scaling	Part VII, Section 1.4.1.	This requirement is fully met.	Yes
R2: Implement Local Services	Part VII, Section 1.4.2.	This requirement is not fully met because the LLM is still hosted by a third-party.	80-90%
R3: Modular Architecture	Part VII, Section 1.4.3.	This requirement is fully met.	Yes

As shown in Table 9, the reasoning R2: Implement Local Services is not fully satisfied. However, this does not conflict with the original task definition but rather with the additional requirements we defined ourselves. For this reason, the implementation of a locally hosted LLM is not included in this thesis due to time constraints, but it is proposed as a future step in the corresponding chapter.

1.4.1. R1: Horizontal Scaling

This section argues whether or not the horizontal scaling requirement is met, as it is more useful to show and to argue that it works than to perform strict test scenarios.

Horizontal scaling affects the entire system, including all components. Therefore, each system component is tested separately and the results are described here.

Given that Kubernetes is used to orchestrate our microservice containers, the system inherently supports horizontal scaling. Since our microservices are designed to be stateless or manage states appropriately using external stores or session handling mechanisms, scaling should be straightforward and reliable. However, correctly implementing these patterns can be challenging in practice. Therefore, this test is designed to validate the system's correct behavior under horizontal scaling conditions.

1.4.1.1. Test Setup

In order to provide a concise justification, a small test setup is still necessary to determine whether the requirement is met.

The testing process is different for the components that we designed and developed than for third-party components. See below for more details.

1.4.1.1.1. Proprietary Components

The test for each custom component will be as follows:

1. Verify that the component has only one replica and scale it to one replica if not.
2. Perform actions on the system that depend on this component.
   a. Verify that the action is performed in the expected manner.
   b. Check if the log contains any errors.
3. Scale the component to two replicas.
4. Perform actions on the system that rely on this component.
   a. Verify that the action is performed in the expected manner.
   b. Check if the log contains any errors.
   c. Check that each replica has been used at least once during the execution of the action.

In the test setup above, if everything works as intended for each component and no errors are shown, the requirement for this component is considered to be met.

1.4.1.1.2. Third-Party Components

The third-party components, are not tested in as much detail as the self-developed components. All third-party systems were installed using Helm charts, which support horizontal scaling out of the box. Therefore, the horizontal scaling requirement is only reasoned, but not explicitly tested. Since all products are used extensively by many people, they are thoroughly tested during production.

- Weaviate: For Weaviate we use a sharded two replica approach. The system is scaled horizontally. Due to the sharding, any problems with scaling would be quickly noticed. The requirement for this component is met.
- Redis: For Redis we use a minimal six node architecture with three master nodes and three replica nodes. The system is scaled horizontally. Due to the sharding, any problems with scaling would be quickly noticed. The requirement for this component is met.
- Keycloak: For Keycloak we tested a two replica approach. The system worked as intended, including the horizontal scaling.
- OAuth2 Proxy: For the OAuth2 Proxy we tested a two replica approach. The system worked as intended, including the horizontal scaling.

1.4.1.2. Results

Table 10: Reasoning Results for Each Component

Name	Type	Note	Success
LLM Service	Proprietary	Worked as intended.	Yes
Manager Service	Proprietary	Worked as intended.	Yes
Retrieval Service	Proprietary	Worked as intended.	Yes
Web Service	Proprietary	Worked as intended.	Yes
Redis	Third-Party	See reasoning above.	Yes
Weaviate	Third-Party	See reasoning above.	Yes
Keycloak	Third-Party	See reasoning above.	Yes
OAuth2 Proxy	Third-Party	See reasoning above.	Yes

Based on the results shown in Table 10, the horizontal scaling requirement is met.

1.4.2. R2: Implement Local Services

As shown in the description of the basic pipeline (Part V, Section 2.), all services are fully implemented locally, except for one, the language model. First, some specialties about the locally implemented services are mentioned, then it is explained why the language model is not yet implemented locally.

Notable local implementations:

- Vector Store: The vector store has shifted from an externally hosted alternative to a local one, from the MVP to the basic pipeline. It took some effort, but it works pretty well in the end.
- Embedding creation: The embedding model is hosted and used entirely locally. This means that all embeddings are created locally and no data needs to be sent to an external source.

As mentioned above, the language model was not changed from externally hosted to internally hosted throughout the project. There are several reasons for this. The first is that there was not enough time to set up the local language model. The second is that the cluster on which the system currently runs does not have the resources to host a reasonably good language model.

For this reason, the requirement to implement all services locally is not fully met. However, as noted above, all but one service is successfully implemented locally. Thus, the requirement is met for about 80 to 90 percent.

1.4.3. R3: Modular Architecture

As shown in the description of the basic pipeline (Part V, Section 2.), the whole system is implemented using a modular microservices approach. Each microservice can be easily adapted, extended and replaced. This is proven by the transition from the MVP pipeline to the basic pipeline, where some components were adapted and some completely new ones were added to the system. Therefore, this requirement is fully met.

1.5. Load and End-User Test

The load and end-user test is conducted in a combined manner, using a single test scenario as outlined below.

The test takes place during a lecture by Prof. Dr. Marco Lehmann at the Eastern Switzerland University of Applied Sciences, involving a group of CAS students. After a brief introduction to the system, the participants are asked to complete workshop task three (Part VI, Section 3.3.3.3.). This task is specifically chosen because it utilizes most of the system's features, generating a higher system load, which is ideal for testing purposes.

The goal of this test is to determine whether the system can handle concurrent user requests without issues, whether users are able to complete the tasks successfully, and whether they understand how to use the system.

1.5.1. Results

The test was conducted on June 3, 2025, with 10 participants. The results and findings are detailed below.

1.5.1.1. Load Results

During the test, the Kubernetes system was continuously monitored to observe its behavior under increased user load. Simultaneously, several manual system tests were carried out to verify that the system functioned as intended. Throughout these tests, the system performed flawlessly, with no noticeable delays.

Additionally, the Kubernetes metrics did not indicate any unusual spikes in user load; all values remained within expected parameters. This suggests that the system is well-prepared to handle higher workloads during our upcoming workshop.

The visualizations Figure 31 and Figure 32 illustrate the system in idle mode and under load, respectively. As shown, the metrics stayed within the normal range.

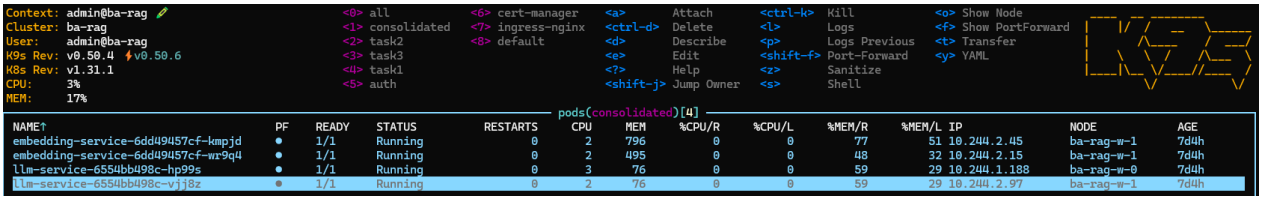

Figure 31: Consolidated Pods in Idle Mode - own presentment

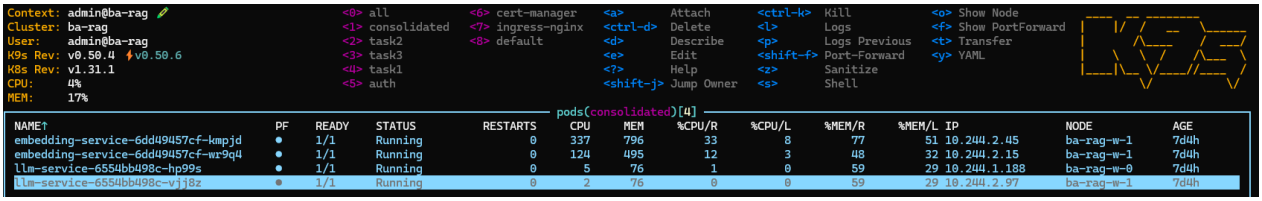

Figure 32: Consolidated Pods under Load - own presentment

1.5.1.2. End-User Results

The users were able to understand and navigate the system effectively, with no major issues encountered related to the system or the task. While a few minor uncertainties or problems arose, none were significant. This outcome indicates that both the system and the task are well-prepared for the upcoming workshop, and no substantial redesign is necessary.

1.6. Performance Tests

In order to improve the RAG system and gain insight into speed improvements from code changes, and to compare embedding options, several manual performance tests were conducted.

1.6.1. Test Setup

The performance tests were done during the basic pipeline implementation. In general, a timeout of 120s was set, since 2 minutes is a long time for a user to wait. An exception was test three, which involved a detailed comparison of large files. The tests were performed using both the web frontend and postman.

The following durations were recorded during the tests:

- time_create_temporary_pdf: The time taken to create a temporary PDF file from the uploaded document.
- time_extract_text_from_pdf: The duration needed to extract text from the PDF file.
- time_generate_embeddings: The time required to generate the embeddings.
- time_write_to_vector_store: The time taken to store the processed data into the vector database.
- time_total: The total time taken for the whole process.

Figure 33 shows an example of a test conducted via the frontend.

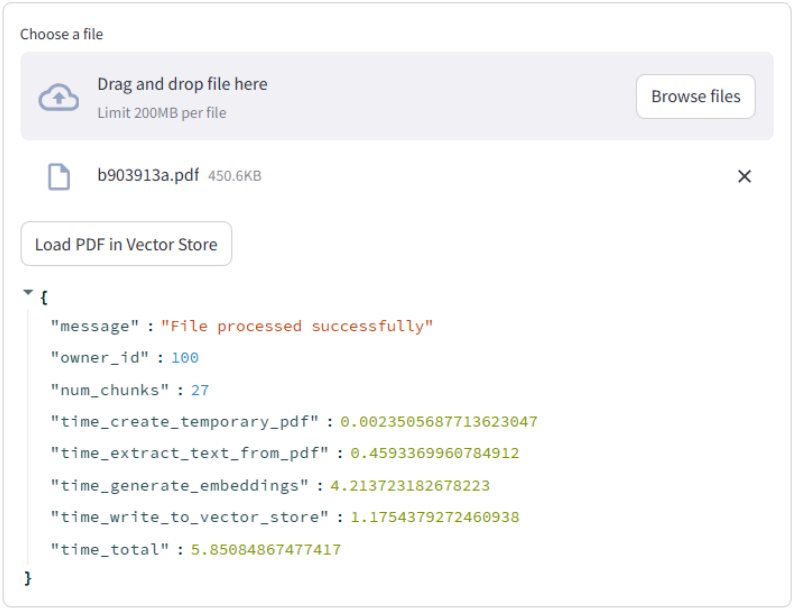

Figure 33: Recorded Test Durations per Method - own presentment

1.6.2. Performance Test 1: Comparison of Performance With and Without Batching

Weaviate, the local vector store, offers several methods for sending data to the database. This test uses files of different sizes to compare two approaches:

- Variant A: Batch processing - 100 chunks are sent together to the vector store.
- Variant B: Single chunk processing - Each chunk is sent individually to the database.

Note: This test is solely intended to compare these two approaches. The timing results should not be used to estimate the overall system performance, as significant enhancements have since been made that greatly improve overall performance. See subsequent tests for more information.

Table 11: Results of Performance Test 1

File Size	Variant A (Batch_size = 100)	Variant B (No Batch_size)
0.8MB	9.58s	5.77s
4.6MB	43.37s	40.95s
10.4MB	113.28s, 102.89s	Timeout (> 120s)
15.8MB	Timeout (> 120s)	Timeout (> 120s)

The test results shown in Table 11 indicate that Variant B performs slightly more efficient on smaller files. However, as the file size increases, Variant A proves to be the better choice. Further analysis revealed that dynamic batching (the .dynamic() function in Weaviate) determines the optimal batch size based on factors such as memory usage and latency. This means that the batch size is not fixed at 100, but rather is determined automatically to optimize performance.

In conclusion, batch processing is generally more efficient than sending chunks individually. Thus, this approach has been adopted.

1.6.3. Performance Test 2: Comparison of PDF File Readers

This test compares two types of file readers. Specifically, pdfplumber and PyMuPDFReader. Both extract text from PDFs. This test measures the time it takes to do so.

Table 12: Results of Performance Test 2

File Size	Variant A (pdfplumber)	Variant B (PyMuPDFReader)
10.4MB	80.58s, 83.08s	2.77s, 3.54s

Table 12 indicates that pdfplumber is much slower than PyMuPDFReader. This is because it analyzes tables, footers, headers, etc., in much greater detail. PyMuPDFReader does not analyze these elements with such detail and is thus more than twenty times faster. Since performance is more important than very high extraction accuracy for our purposes, we decided to use PyMuPDFReader.

1.6.4. Performance Test 3: T2V-Transformer Service vs. Hugging Face Setup

This test compares the performance of embedding creation. Two options are compared: One option is to generate embeddings using the T2V-Transformer service, which is managed by Weaviate. The other option is to manually generate the embeddings using a Hugging Face embedding model and pass them to the vector store. The results of the comparison are shown in Table 13.

Note: Both options use the same embedding model to create the embeddings.

Table 13: Results of Performance Test 3

File Size	Variant A (T2V-Transformer Service)	Variant B (Hugging Face Embeddings)
15.8MB	291.27s, 287.90s	52.73s, 55.13s

Although the T2V-Transformer service is directly integrated into Weaviate and used as an "in-house" vectorizer, it is significantly slower. Manually creating the embeddings with the Hugging Face embedding model and passing them to Weaviate is much faster. This is somewhat surprising since the T2V-Transformer service is directly integrated into Weaviate. For this reason, the option to generate the embeddings via the Hugging Face setup was chosen. More implementation details can be found in the architecture section (Part III, Section 4.).

1.6.5. Performance Test 4: Outsource Embedding Generation

This test evaluates the impact on performance of decoupling the embedding generation process from the retrieval service. Specifically, it compares two configurations.

- Variant A: Embeddings are generated by a dedicated, external service.
- Variant B: Embeddings are generated internally as part of the retrieval service.

Table 14: Results of Performance Test 4

File Size	Variant A (Separate Embedding Service)	Variant B (Integrated in Retrieval Service)
15.8MB	24.79s, 25.63s	52.73s, 55.13s

As shown in Table 14, substantial performance improvements are indicated when generation is offloaded to a separate service. In variant A, processing time was approximately halved compared to variant B, so variant A was chosen. Although this significant performance boost cannot be conclusively explained, it is likely due to the parallelism or other optimization techniques enabled by decoupling. This architectural separation is especially advantageous when handling larger files or scaling the system.

2. Limitations / Next Steps

As stated in the statement of task, this thesis involves the development of some core functionalities of a scalable and secure RAG-as-a-Service infrastructure, as the time for this thesis is limited. However, the system can be adapted and extended in many different ways. This chapter identifies limitations and suggests appropriate next steps to extend and refine the work started in this thesis.

2.1. Implement Local LLM

As previously discussed, the LLM is not currently implemented locally. However, deploying a locally hosted LLM is a recommended next step for developing a secure RAG system, as it allows data to remain within internal system boundaries. Keep in mind, though, that hosting a capable LLM requires substantial computational resources.

2.2. Implement Patch Management

The infrastructure and its components rely on numerous dependencies that need regular updates, particularly when security vulnerabilities are identified. Effective patch management is complex and demands careful planning. It involves maintaining a record of current dependency versions and regularly checking for available updates. The logging system should generate alerts when critical vulnerabilities are detected that require immediate action. Additionally, it's essential to verify that all dependencies remain compatible after updates. To achieve this, a robust testing solution must be in place, as described in the next section.

Currently, there is no patch management in place. However, it is highly recommended that one be introduced for a production system.

2.3. Implement Automated Testing

Automated testing is essential in environments where the system or its dependencies are frequently updated. At a minimum, tests should be run before deploying any changes to the production environment to ensure expected behavior and identify any deviations. Currently, however, only manual testing is in place, as outlined in the testing section. For a production environment, the introduction of automated testing to improve reliability, efficiency, and the early detection of potential issues is highly recommended.

2.4. Implement Logging Solution

Currently, components are logged to the console using basic print statements. While this may suffice during development, it is not suitable for a production environment. A proper logging solution should be implemented to enable more effective monitoring and troubleshooting. This would allow for structured logging, log level management, and centralized log aggregation, making it easier to quickly identify and respond to errors in the production system.

2.5. Improve API Error Messages

The API endpoints currently provide functional error messages, but they lack sufficient detail. Enhancing these messages with more specific and descriptive information would improve transparency, making it easier for developers to understand the cause of errors and streamline the debugging process.

2.6. Improve System Prompts

The initial prompts were created with a primary focus on functionality and have not undergone thorough optimization due to time constraints. However, prompt optimization plays a vital role in building AI

systems that are reliable, predictable, and secure. Therefore, allocating time and effort to systematically refine and improve these prompts is highly recommended.

2.7. Improve Document Chunking

The current document chunking method is based on a simple heuristic that splits documents into chunks of a fixed size. While this approach is straightforward, it may not be the most effective for all types of documents. More sophisticated chunking methods, such as those based on semantic analysis or natural language processing techniques, could be explored to improve the quality of the chunks and enhance the overall performance of the system.

A test conducted during workshop preparation revealed that the current chunking method can lead to misleading information. For instance, when a document containing patient information was uploaded, some of the patient's information was split into two chunks. As a result, the LLM could not answer questions about the patient correctly. Therefore, it is recommended to implement a more sophisticated chunking method that takes into account the context and semantics of the text. This would help ensure that the chunks are meaningful and coherent, leading to better performance in information retrieval and question answering.

2.8. Implement UBAC (User-Based Access Control) in the Vector Store

Weaviate supports User-Based Access Control (UBAC). Introducing this feature would allow for more granular control over who can access specific data within the vector store. This would enhance security and ensure that sensitive information is only accessible to authorized users. Due to time constraints, however, this feature was not included in the current version of the system. Implementing it in a production environment is recommended to enhance security and data protection.

2.9. Implement Secret Management

Currently, secrets are managed locally on each developer's machine as well as in the Kubernetes secret store for the hosting. While this approach is sufficient for development purposes, a production environment requires a more centralized and secure solution. In order to properly handle and protect sensitive information, it is highly recommended that a robust secret management system is implemented.

2.10. Enhance Kubernetes Hardening

Hardening Kubernetes is essential for securing the system. While there are many areas to consider, the two below highlight important measures for our use case. Note that this is not an exhaustive list, and there are many more options and best practices.

2.10.1. Implement Network Isolation for Shared Hosting

In the current shared hosting environment, systems are only isolated at the logical level. However, from a network standpoint, components remain accessible to one another. This means that if an attacker escapes a container, they could access systems belonging to other hosted environments. To prevent such lateral movement and enhance overall security, strict network-level isolation is highly recommended in a production setting.

2.10.2. Implement Internal Network-Level Encryption

Currently, internal network traffic within the system is not encrypted. To enhance security and harden the overall system, it is recommended to implement network-level encryption for all internal communication. It is important to note, however, that this would increase resource consumption within the Kubernetes environment.

Part VIII
Conclusion & Outlook

1. Conclusion

The objective of this thesis was to design and develop a secure and scalable RAG-as-a-Service infrastructure. This included designing and describing the system's overall architecture, implementing its core functionalities, and analyzing a related topic of interest in greater depth rather than focusing on supplementary features. Finally, the resulting infrastructure was intended to be ready for use during our workshop at the IEEE Swiss Conference on Data Science (SDS2025) on June 26, 2025, at the Circle Convention Center, Zurich Airport.

To support our decision-making process and ensure reasonable design and tool choices, we performed an extensive evaluation of different architectural approaches, existing frameworks, and tools. Based on this evaluation, we designed a microservice architecture that builds on Kubernetes and separates responsibilities into different containers. Regarding RAG frameworks, we used LangChain and LlamaIndex based on their respective strengths.

Subsequently, the infrastructure, including all the containers, was developed based on this design. The core components of the RAG pipeline were implemented in a scalable and modular manner. This means that they are either stateless or handle states accordingly, and their interfaces are well-defined. Throughout the process, we incorporated insights from our previous research, which was conducted as part of our study thesis and accompanying paper, to ensure a secure foundation. We implemented a robust authentication system that uses state-of-the-art protocols, such as OAuth 2.0 and OpenID Connect, to establish a critical layer of security. Additionally, all external communication is encrypted to protect data in transit. Thanks to its microservice architecture, the system is flexible and can be expanded with features tailored to specific use cases.

As the chosen related topic, the preparation of our workshop, including its supporting infrastructure, was addressed. This involved implementing a three-system approach that demonstrates the system's ability to operate in an as-a-Service manner. Additionally, tasks focusing on security-related topics and mitigation strategies were planned for the workshop participants. The RAG infrastructure was also analyzed for potential risks and corresponding mitigation measures, which further underscores the security focus of this thesis.

Based on the given description, the RAG infrastructure developed, along with all supplementary tasks, fulfills the objectives of this thesis. Moreover, the infrastructure is fully operational and suitable for use in the workshop setting. As a result, all primary and supplementary objectives have been successfully achieved.

2. Outlook

This thesis focused on implementing the core functionalities of the RAG infrastructure. As a result, the system remains open to future extensions and adaptations, depending on specific use cases and requirements. Known limitations and recommended next steps are outlined in the relevant chapter and will not be repeated here.

The upcoming workshop is expected to provide valuable insights and direct feedback, offering an opportunity to assess the infrastructure's readiness and its potential for real-world application.

Further developments are on the horizon — stay tuned.

Part IX
Appendix

1. Bibliography

[1] H. Naveed *et al.*, "A Comprehensive Overview of Large Language Models." [Online]. Available: https://arxiv.org/abs/2307.06435

[2] J. Kaplan *et al.*, "Scaling Laws for Neural Language Models." [Online]. Available: https://arxiv.org/abs/2001.08361

[3] E. J. Hu *et al.*, "LoRA: Low-Rank Adaptation of Large Language Models." [Online]. Available: https://arxiv.org/abs/2106.09685

[4] N. Lambert, L. Castricato, L. von Werra, and A. Havrilla, "Illustrating Reinforcement Learning from Human Feedback (RLHF)," *Hugging Face Blog*, 2022.

[5] P. G. Sessa *et al.*, "BOND: Aligning LLMs with Best-of-N Distillation." [Online]. Available: https://arxiv.org/abs/2407.14622

[6] Z. Ji, T. Yu, Y. Xu, N. Lee, E. Ishii, and P. Fung, "Towards Mitigating LLM Hallucination via Self Reflection," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, H. Bouamor, J. Pino, and K. Bali, Eds., Singapore: Association for Computational Linguistics, Dec. 2023, pp. 1827–1843. doi: 10.18653/v1/2023.findings-emnlp.123.

[7] P. Lewis *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." [Online]. Available: https://arxiv.org/abs/2005.11401

[8] L. Ammann and S. Ott, "Analysis of Risks and Mitigation Strategies in RAG." [Online]. Available: https://eprints.ost.ch/id/eprint/1255/

[9] L. Ammann, S. Ott, C. R. Landolt, and M. P. Lehmann, "Securing RAG: A Risk Assessment and Mitigation Framework." [Online]. Available: https://arxiv.org/abs/2505.08728

[10] S. 2025, "Hacking RAG: Exploring Risks and Implementing Mitigations." Accessed: Apr. 17, 2025. [Online]. Available: https://sds2025.ch/hacking-rag-exploring-risks-and-implementing-mitigations/

[11] P. Zhao *et al.*, "Retrieval-Augmented Generation for AI-Generated Content: A Survey." [Online]. Available: https://arxiv.org/abs/2402.19473

[12] Y. Gao *et al.*, "Retrieval-Augmented Generation for Large Language Models: A Survey." [Online]. Available: https://arxiv.org/abs/2312.10997

[13] Y. Huang and J. Huang, "A Survey on Retrieval-Augmented Text Generation for Large Language Models." [Online]. Available: https://arxiv.org/abs/2404.10981

[14] M. Anderson, G. Amit, and A. Goldsteen, "Is My Data in Your Retrieval Database? Membership Inference Attacks Against Retrieval Augmented Generation," in *Proceedings of the 11th International Conference on Information Systems Security and Privacy*, SCITEPRESS - Science, Technology Publications, 2025, pp. 474–485. doi: 10.5220/0013108300003899.

[15] W. Zou, R. Geng, B. Wang, and J. Jia, "PoisonedRAG: Knowledge Corruption Attacks to Retrieval-Augmented Generation of Large Language Models." [Online]. Available: https://arxiv.org/abs/2402.07867

[16] Z. Chen *et al.*, "Black-Box Opinion Manipulation Attacks to Retrieval-Augmented Generation of Large Language Models." [Online]. Available: https://arxiv.org/abs/2407.13757

[17] Y.-H. Huang, Y. Tsai, H. Hsiao, H.-Y. Lin, and S.-D. Lin, "Transferable Embedding Inversion Attack: Uncovering Privacy Risks in Text Embeddings without Model Queries," in *Proceedings of the 62nd*

Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, 2024, pp. 4193–4205. doi: 10.18653/v1/2024.acl-long.230.

[18] S. Zeng *et al.*, "The Good and The Bad: Exploring Privacy Issues in Retrieval-Augmented Generation (RAG)." [Online]. Available: https://arxiv.org/abs/2402.16893

[19] J. X. Morris, V. Kuleshov, V. Shmatikov, and A. M. Rush, "Text Embeddings Reveal (Almost) As Much As Text." [Online]. Available: https://arxiv.org/abs/2310.06816

[20] G. Deng, Y. Liu, K. Wang, Y. Li, T. Zhang, and Y. Liu, "Pandora: Jailbreak GPTs by Retrieval Augmented Generation Poisoning." [Online]. Available: https://arxiv.org/abs/2402.08416

[21] Z. Wang, J. Liu, S. Zhang, and Y. Yang, "Poisoned LangChain: Jailbreak LLMs by LangChain." [Online]. Available: https://arxiv.org/abs/2406.18122

[22] Z. Xu, Y. Liu, G. Deng, Y. Li, and S. Picek, "A Comprehensive Study of Jailbreak Attack versus Defense for Large Language Models." [Online]. Available: https://arxiv.org/abs/2402.13457

[23] K. Hu *et al.*, "Efficient LLM Jailbreak via Adaptive Dense-to-sparse Constrained Optimization." [Online]. Available: https://arxiv.org/abs/2405.09113

[24] Y. Zeng, Y. Wu, X. Zhang, H. Wang, and Q. Wu, "AutoDefense: Multi-Agent LLM Defense against Jailbreak Attacks." [Online]. Available: https://arxiv.org/abs/2403.04783

[25] European Parliament, Council of the European Union, "The EU Artificial Intelligence Act (EU AI Act)." 2024.

[26] J. R. B. Jr., "Executive Order on the Safe, Secure, and Trustworthy Development and Use of Artificial Intelligence." Oct. 30, 2023.

[27] "NIST Artificial Intelligence Risk Management Framework (AI RMF 1.0)." 2023. doi: https://doi.org/10.6028/NIST.AI.100-1.

[28] Open Worldwide Application Security Project (OWASP), "OWASP Top 10 for Large Language Model Applications (2025)." Accessed: Jan. 06, 2025. [Online]. Available: https://genai.owasp.org/resource/owasp-top-10-for-llm-applications-2025/

[29] P. AI, "Unlocking the Power of Retrieval Augmented Generation with Added Privacy: A Comprehensive Guide." Accessed: Sep. 20, 2024. [Online]. Available: https://www.private-ai.com/en/2024/05/23/rag-privacy-guide/

[30] J. G. @ Merge, "5 benefits of retrieval-augmented generation (RAG)." Accessed: Sep. 20, 2024. [Online]. Available: https://www.merge.dev/blog/rag-benefits

[31] CloudCraft, "What is RAG (Retrieval Augmented Generation)?." Accessed: Sep. 20, 2024. [Online]. Available: https://www.cloudraft.io/what-is/retrieval-augmented-generation

[32] S. Besen, "The Practical Limitations and Advantages of Retrieval Augmented Generation (RAG)." Accessed: Sep. 20, 2024. [Online]. Available: https://towardsdatascience.com/the-limitations-and-advantages-of-retrieval-augmented-generation-rag-9ec9b4ae3729

[33] M. Besta *et al.*, "Multi-Head RAG: Solving Multi-Aspect Problems with LLMs." [Online]. Available: https://arxiv.org/abs/2406.05085

[34] D. Edge *et al.*, "From Local to Global: A Graph RAG Approach to Query-Focused Summarization." [Online]. Available: https://arxiv.org/abs/2404.16130

[35] F. Wang, X. Wan, R. Sun, J. Chen, and S. Ö. Arık, "Astute RAG: Overcoming Imperfect Retrieval Augmentation and Knowledge Conflicts for Large Language Models." [Online]. Available: https://arxiv.org/abs/2410.07176

[36] C. Niu *et al.*, "RAGTruth: A Hallucination Corpus for Developing Trustworthy Retrieval-Augmented Language Models." [Online]. Available: https://arxiv.org/abs/2401.00396

[37] J. Saad-Falcon, O. Khattab, C. Potts, and M. Zaharia, "ARES: An Automated Evaluation Framework for Retrieval-Augmented Generation Systems." [Online]. Available: https://arxiv.org/abs/2311.09476

[38] P. Laban, A. R. Fabbri, C. Xiong, and C.-S. Wu, "Summary of a Haystack: A Challenge to Long-Context LLMs and RAG Systems." [Online]. Available: https://arxiv.org/abs/2407.01370

[39] C. Larman, "UML and Patterns - Use Cases." Accessed: Apr. 02, 2025. [Online]. Available: https://craiglarman.com/wiki/downloads/applying_uml/larman-ch6-applying-evolutionary-use-cases.pdf

[40] "ISO/IEC 25010." Accessed: Apr. 02, 2025. [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[41] "Testing quadrants." Accessed: Apr. 02, 2025. [Online]. Available: https://tryqa.com/what-are-test-pyramid-and-testing-quadrants-in-agile-testing-methodology/

[42] T. Team, "Agile Testing Methodology: Life Cycle, Techniques, & Strategy." Accessed: May 20, 2025. [Online]. Available: https://www.testrail.com/blog/agile-testing-methodology/

[43] C4 Model, "C4 Model." Accessed: Jan. 06, 2025. [Online]. Available: https://c4model.com/

[44] OST, " Modulbeschreibung - Application Architecture ." Accessed: Feb. 26, 2025. [Online]. Available: https://studien.ost.ch/allModules/public/28236_M_AppArch.html

[45] IONOS Redaktion, "GET vs. POST – die beiden wichtigsten HTTP-Requests im Vergleich." Accessed: May 01, 2025. [Online]. Available: https://www.ionos.de/digitalguide/websites/web-entwicklung/get-vs-post/

[46] Zilliz, "Weaviate vs. Chroma." Accessed: Mar. 24, 2025. [Online]. Available: https://zilliz.com/comparison/weaviate-vs-chroma

[47] IBM, "What is Redis Explained?." Accessed: Mar. 17, 2025. [Online]. Available: https://www.ibm.com/think/topics/redis

[48] Apache, "Apache Cassandra Documentation." Accessed: Mar. 17, 2025. [Online]. Available: https://cassandra.apache.org/_/index.html

[49] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002, doi: 10.1145/564585.564601.

[50] M. Abu Kausar, M. Nasar, and A. Soosaimanickam, "A study of performance and comparison of nosql databases: Mongodb, cassandra, and redis using ycsb," *Indian J. Sci. Technol*, vol. 15, pp. 1532–1540, 2022.

[51] Redis, "Redis Enterprise Cluster Architecture." Accessed: Mar. 20, 2025. [Online]. Available: https://redis.io/technology/redis-enterprise-cluster-architecture/

[52] C. Kong, "Redis Sentinel vs Redis Cluster: A Comparative Overview." Accessed: Mar. 20, 2025. [Online]. Available: https://medium.com/@chaewonkong/redis-sentinel-vs-redis-cluster-a-comparative-overview-8c2561d3168f

[53] Redis, “Redis persistence: How Redis writes data to disk.” Accessed: Mar. 20, 2025. [Online]. Available: https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/

[54] GitLab Inc., “GitLab - Software.Faster..” Accessed: Mar. 07, 2025. [Online]. Available: https://about.gitlab.com/

[55] I. Docker, “Docker: Accelerated Container Application Development.” Accessed: Jun. 02, 2025. [Online]. Available: https://www.docker.com/

[56] I. Docker, “Docker Compose.” Accessed: Jun. 02, 2025. [Online]. Available: https://docs.docker.com/compose/

[57] The Kubernetes Authors, “kubernetes.” Accessed: Feb. 25, 2025. [Online]. Available: https://kubernetes.io/

[58] The Kubernetes Authors, “Kubernetes - Overview.” Accessed: Feb. 25, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/overview/

[59] The Kubernetes Authors, “Minikube Documentation.” Accessed: Feb. 27, 2025. [Online]. Available: https://minikube.sigs.k8s.io/docs/

[60] Kustomize, “Kubernetes native configuration management.” Accessed: Mar. 07, 2025. [Online]. Available: https://kustomize.io/

[61] Red Hat, “What is Helm?.” Accessed: Mar. 14, 2025. [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-helm

[62] Imhotep Software LLC, “k9s - Kubernetes CLI To Manage Your Clusters In Style!.” Accessed: Mar. 07, 2025. [Online]. Available: https://k9scli.io/

[63] Kubernetes Community, “Ingress-NGINX Controller for Kubernetes.” Accessed: Apr. 23, 2025. [Online]. Available: https://github.com/kubernetes/ingress-nginx

[64] Jetstack, “cert-manager: Kubernetes Certificate Management.” Accessed: Apr. 23, 2025. [Online]. Available: https://cert-manager.io/docs/

[65] Let's Encrypt, “Let's Encrypt: Free SSL/TLS Certificates.” Accessed: Apr. 23, 2025. [Online]. Available: https://letsencrypt.org/

[66] LlamaIndex, “LlamaIndex: The framework for context-augmented LLM applications.” Accessed: Apr. 23, 2025. [Online]. Available: https://docs.llamaindex.ai/

[67] LangChain, “LangChain: Build context-aware reasoning applications.” Accessed: Apr. 23, 2025. [Online]. Available: https://www.langchain.com/

[68] Weaviate, “Weaviate.” Accessed: Apr. 23, 2025. [Online]. Available: https://weaviate.io/

[69] Keycloak Project, “Keycloak: Open-Source Identity and Access Management.” Accessed: Apr. 23, 2025. [Online]. Available: https://www.keycloak.org/

[70] OAuth2 Proxy Contributors, “OAuth2 Proxy: Authentication Reverse Proxy.” Accessed: Apr. 23, 2025. [Online]. Available: https://github.com/oauth2-proxy/oauth2-proxy

[71] Tiangolo, “FastAPI.” Accessed: May 01, 2025. [Online]. Available: https://fastapi.tiangolo.com/

[72] Streamlit, “Streamlit: The fastest way to build and share data apps.” Accessed: May 28, 2025. [Online]. Available: https://streamlit.io/

[73] Jazzband, “pip-compile CLI — pip-tools documentation.” Accessed: May 30, 2025. [Online]. Available: https://pip-tools.readthedocs.io/en/latest/cli/pip-compile/

[74] Faker, “Faker: A Python package that generates fake data.” Accessed: Jan. 06, 2025. [Online]. Available: https://faker.readthedocs.io/en/master/

[75] Black Development Team, “Black: The uncompromising code formatter.” Accessed: Apr. 23, 2025. [Online]. Available: https://github.com/psf/black

[76] Trunk.io, “Flake8: The Python Linter for Code Quality and Style.” Accessed: Apr. 23, 2025. [Online]. Available: https://trunk.io/linters/python/flake8

[77] Mypy Development Team, “Mypy 1.15.0 documentation.” Accessed: Apr. 23, 2025. [Online]. Available: https://mypy.readthedocs.io/

[78] Kubernetes Authors, “Kubernetes - Cluster Architecture.” Accessed: May 16, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/

[79] The Kubernetes Authors, “Kubernetes - Cluster Architecture.” Accessed: Feb. 27, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/

[80] GitLab, “Use container images from the container registry.” Accessed: Feb. 28, 2025. [Online]. Available: https://docs.gitlab.com/user/packages/container_registry/

[81] Auth0, “What is OAuth 2.0?.” Accessed: Apr. 09, 2025. [Online]. Available: https://auth0.com/intro-to-iam/what-is-oauth-2

[82] OpenID, “How OpenID Connect Works.” Accessed: Apr. 09, 2025. [Online]. Available: https://openid.net/developers/how-connect-works/

[83] A. Chiarelli, “ID Token vs Access Token: What is the Difference?.” Accessed: Apr. 14, 2025. [Online]. Available: https://auth0.com/blog/id-token-access-token-what-is-the-difference/

[84] IBM, “OAuth 2.0 refresh tokens.” Accessed: Apr. 14, 2025. [Online]. Available: https://docs.verify.ibm.com/verify/docs/oauth-20-refresh-tokens

[85] H. Face, “sentence-transformers/all-MiniLM-L6-v2.” Accessed: May 06, 2025. [Online]. Available: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

[86] P. Boteju, “Deep dive into Redis Clustering.” Accessed: Mar. 28, 2025. [Online]. Available: https://medium.com/@pubuduboteju95/deep-dive-into-redis-clustering-1d71484578a9

[87] Redis, “Redis Cluster and Client Libraries.” Accessed: Apr. 29, 2024. [Online]. Available: https://redis.io/learn/operate/redis-at-scale/scalability/redis-cluster-and-client-libraries

[88] Weaviate, “Cluster Architecture.” Accessed: Apr. 30, 2025. [Online]. Available: https://weaviate.io/developers/weaviate/concepts/replication-architecture/cluster-architecture

[89] Weaviate, “Replication Architecture.” Accessed: Apr. 30, 2025. [Online]. Available: https://weaviate.io/developers/weaviate/concepts/replication-architecture

[90] MITRE Corporation, “Synthea: An Open Source Synthetic Patient Generator.” Accessed: Jan. 06, 2025. [Online]. Available: https://synthea.mitre.org/downloads

2. List of Figures

3. List of Tables

4. Use of AI Tools

The following AI tools were used throughout the project for the stated purposes in accordance with the actual AI regulations of the Eastern Switzerland University of Applied Sciences.

- ChatGPT: ChatGPT has been used for inspiration, research, rewriting text passages, and as a source of inspiration for coding and prototyping.
- PerplexityAI: PerplexityAI has been used for research and as a source of inspiration for coding and prototyping.
- DeepL: DeepL has been used to rewrite and refine text passages.

5. Functional Test Protocols

5.1. F1: User Login with Valid Credentials

Table 15: Test Protocol for Functional Test F1

Type	**Note**
Number	F1
Name	User Login with Valid Credentials
Preconditions	User has valid credentials for the RAG system.
Test Scenario	1. Tester opens a Chromium-based browser and switches to incognito mode. 2. Tester goes to the RAG system URL. 3. The login page should be displayed. 4. Tester logs in with valid credentials. 5. Browser redirects to the RAG system website.
Expected Result	User is logged in and the RAG system website is displayed.
Was result as expected?	Yes, as expected. Since this was the first login of this user and not already defined within the identity provider, the user had to enter email, first name and last name.
Date	07.05.2025
Findings	Worked as expected.
Success	Yes
Improvements / Reasoning	-

5.2. F2: User Login with Invalid Credentials

Table 16: Test Protocol for Functional Test F2

Type	**Note**
Number	F2
Name	User Login with Valid Credentials
Preconditions	None
Test Scenario	1. Tester opens a Chromium-based browser and switches to incognito mode. 2. Tester goes to the RAG system URL. 3. The login page should be displayed. 4. Tester enters invalid credentials in the login field.
Expected Result	An error message should appear and the login page should still be displayed.
Was result as expected?	As expected, the error message "Invalid username or password." was displayed.
Date	07.05.2025
Findings	Worked as expected.
Success	Yes
Improvements / Reasoning	-

5.3. F3: User Logout

Table 17: Test Protocol for Functional Test F3

Type	**Note**
Number	F3
Name	User Logout
Preconditions	F1 - User Login with Valid Credentials (user is logged in).
Test Scenario	1. Tester clicks on the logout button on the main page of the RAG system website.
Expected Result	A redirect should be started and the browser should eventually return to the login page.
Was result as expected?	As expected, the redirect was started and the browser returned to the login page.
Date	07.05.2025
Findings	None
Success	Yes
Improvements / Reasoning	-

5.4. F4: Upload PDF File

5.4.1. First Attempt

Table 18: Test Protocol for Functional Test F4.1

Type	**Note**
Number	F4.1
Name	Upload PDF File
Preconditions	• F1 - User Login with Valid Credentials (user is logged in). • PDF file is available for upload.
Test Scenario	1. Tester goes to the “Upload Data” section of the RAG system website. 2. Tester clicks the “Browse Files” button and uploads a test pdf file. 3. Tester clicks the “Load PDF in Vector Store” button. 4. The system should successfully confirm that the action is complete. 5. Tester goes to the “Retrieve Data” section and makes sure that the file is properly added to the system and accessible via an appropriate query.
Expected Result	• The system should successfully confirm that the action is complete. • The file should be properly added to the system and accessible via an appropriate query.
Was result as expected?	No, when choosing a file to upload, the system displayed an error message: AxiosError: Request failed with status code 400. The file had to be chosen again. The system then displayed the success message with the number of chunks, owner_id, filename and timestamps. The file was correctly added to the system and could be accessed via an appropriate query.
Date	07.05.2025
Findings	Research told us that this is a known open issue of streamlit when using distributed systems like Kubernetes. The problem is that the request is not always handled by the same pod, and then this error can occur.
Success	No
Improvements / Reasoning	We will use sticky sessions for the web service to avoid this issue.

5.4.2. Second Attempt

Table 19: Test Protocol for Functional Test F4.2

Type	**Note**
Number	F4.2
Name	Upload PDF-File
Preconditions	• F1 - User Login with Valid Credentials (user is logged in). • PDF file is available for upload.
Test Scenario	1. Tester goes to the "Upload Data" section of the RAG system website. 2. Tester clicks the "Browse Files" button and uploads a test pdf file. 3. Tester clicks the "Load PDF in Vector Store" button. 4. The system should successfully confirm that the action is complete. 5. Tester goes to the "Retrieve Data" section and makes sure that the file is properly added to the system and accessible via an appropriate query.
Expected Result	• The system should successfully confirm that the action is complete. • The file should be properly added to the system and accessible via an appropriate query.
Was result as expected?	Yes, worked as expected.
Date	12.05.2025
Findings	None
Success	Yes
Improvements / Reasoning	-

5.5. F5: Ask Chatbot Question - Typical Case

Table 20: Test Protocol for Functional Test F5

Type	**Note**
Number	F5
Name	Ask Chatbot Question - Typical Case
Preconditions	• F1 - User Login with Valid Credentials (user is logged in). • F4 - Upload PDF-File successfully completed (chunks are in the vector store).
Test Scenario	1. Tester goes to the "Ask Chatbot" section of the RAG system website. 2. Tester asks a question about the uploaded PDF file. 3. The system should display the generated response in a chat manner. 4. Tester asks a follow-up question about the uploaded PDF file. 5. The system should display a meaningful generated response in a chat manner. 6. Tester goes to the Home section and returns to the "Ask Chatbot" section. 7. The system should load the chat history. The previously asked question and their response should be displayed.
Expected Result	• The system should display the generated response in a chat manner. • The system should load the chat history. The previously asked question and their response should be displayed.
Was result as expected?	The steps worked as expected. The system displayed the generated response in chat manner. The system loaded the chat history and displayed the previously asked question and their response.
Date	07.05.2025
Findings	None
Success	Yes
Improvements / Reasoning	-

5.6. F6: Ask Chatbot Question - No Related Information

Table 21: Test Protocol for Functional Test F6

Type	**Note**
Number	F6
Name	Ask Chatbot Question - No Related Information
Preconditions	• F1 - User Login with Valid Credentials (user is logged in). • Chatbot history is empty.
Test Scenario	1. Tester goes to the “Ask Chatbot” section of the RAG system website. 2. Tester asks a question unrelated to the uploaded PDF file.
Expected Result	The system should inform the user that no related information is available therefore no response can be generated.
Was result as expected?	Yes, as expected. The bot responded: I cannot provide an answer about ... as the context and chat history do not contain relevant information on that topic.
Date	07.05.2025
Findings	None
Success	Yes
Improvements / Reasoning	-

5.7. F7: Give Citation

Table 22: Test Protocol for Functional Test F7

Type	**Note**
Number	F7
Name	Give Citation
Preconditions	• F1 - User Login with Valid Credentials (user is logged in). • F4 - Upload PDF-File successfully completed (chunks are in the vector store). • Chatbot history is empty.
Test Scenario	1. Tester goes to the “Ask Chatbot” section of the RAG system website. 2. Tester asks a question about the uploaded PDF-file.
Expected Result	• The system should show the generated response in a chat manner. The response contains the citation of used the PDF-file.
Was result as expected?	Yes, as expected. The system displayed the generated response in chat manner. The response contained the citation of the PDF-file used. At the end there is the citation in brackets, e.g. (Source: climate_zurich.pdf).
Date	07.05.2025
Findings	None
Success	Yes
Improvements / Reasoning	-

5.8. F8: Admin View

Table 23: Test Protocol for Functional Test F8

Type	**Note**
Number	F8
Name	Admin View
Preconditions	• F1 - User Login with Valid Credentials (user is logged in). • F4 - Upload PDF-File successfully completed (chunks are in the vector store).
Test Scenario	1. Tester goes to the “Retrieve chunks (admin view)” section of the RAG system website. 2. Tester asks for chunks about a certain word.
Expected Result	The system should show the top 5 relevant chunks with their metadata.
Date	07.05.2025
Was result as expected?	Yes, as expected. The system showed the top 5 relevant chunks with their data (text, owner, filename, score).
Findings	None
Success	Yes
Improvements / Reasoning	-

5.9. F9: API Documentation

This test is done locally (dev environment) as API documentation is not available on the deployed version of the system.

Table 24: Test Protocol for Functional Test F9

Type	**Note**
Number	F9
Name	Give Citation
Precondition	None
Test Scenario	1. Tester opens a chromium-based browser and switches to incognito mode. 2. Tester opens the different urls with /docs. a. Tester opens the manager swagger API under localhost:5000/docs. b. Tester opens the retrieval swagger API under localhost:5001/docs. c. Tester opens the LLM swagger API under localhost:5002/docs.
Expected Result	The system should display the API documentation for each.
Was result as expected?	Yes, as expected. The system displayed the API documentation for each.
Date	07.05.2025
Findings	None
Success	Yes
Improvements / Reasoning	-

5.9.1. Reasoning

Since API documentation reveals details about service endpoints, it should not be publicly accessible. As a security measure, the API documentation is not reachable in the production deployment of the system.

6. End-to-End Test Protocols

6.1. E1: Access Restriction

Table 25: Test Protocol for End-to-End Test E1

Type	Note
Number	E1
Name	Access Restriction
Preconditions	• User has valid credentials for the RAG system. • Empty chat history.
Test Scenario	1. Tester opens a chromium-based browser and switches to incognito mode. 2. Tester logs in as described in C1 (Part IX, Section 5.1.). 3. Tester goes to the "Upload File" section of the RAG system website. 4. Tester uploads a PDF-file. 5. Tester opens the Home screen and loges out of the system. 6. Tester opens a chromium-based browser and switches to incognito mode. 7. Tester logs in (with a different user) as described in C1. 8. Tester goes to the "Ask Chatbot" section of the RAG system website. 9. Tester asks a question about the uploaded PDF-file. 10. The system should inform the user that no related information is available (for this user), therefore no response can be generated. 11. Tester goes to the "Retrieve Data" section of the RAG system website. 12. Tester types in a reasonable query related to the uploaded file. 13. The system cannot retrieve information from the PDF-file uploaded by the other user. 14. Tester opens the home screen and loges out of the system.
Expected Results	• Every step of the test scenario could be successfully completed. • The system should inform the user that no related information is available (for this user), therefore no response can be generated. • The system cannot retrieve information from the PDF-file uploaded by the other user.
Was result as expected?	Yes, every step of the test scenario could be successfully completed. The system informed the user that no related information is available.
Date	07.05.2025
Findings	No issues were found.
Success	Yes
Improvments / Reasoning	-

6.2. E2: Chat History

Table 26: Test Protocol for End-to-End Test E2

Type	**Note**
Number	E2
Name	Chat history
Preconditions	• User has a valid credentials for the RAG system. • Empty chat history.
Test Scenario	1. Tester opens a chromium-based browser and switches to incognito mode. 2. Tester logs in as described in C1 (Part IX, Section 5.1.). 3. Tester goes to the “Upload File” section of the RAG system website. 4. Tester uploads a PDF-file. 5. Tester opens the Chatbot section of the RAG system website. 6. Tester asks a question about the uploaded PDF-file. 7. Tester opens the Home screen and loges out of the system, closes the browser. 8. Tester opens a chromium-based browser and switches to incognito mode. 9. Tester logs in again with a the same user as described in C1. 10. Tester goes to the “Ask Chatbot” section of the RAG system website. 11. The system should load the previous chat history. The questions and response earlier generated should be displayed. 12. Tester asks a follow-up question about the uploaded PDF-file. 13. The system should display the chat history, including the newly generated response, in a chat-like manner. 14. Tester opens the Home screen and loges out of the system.
Expected Results	• Every step of the test scenario could be successfully completed. • The system should load the previous chat history. The questions and response earlier generated should be displayed.
Was result as expected?	Yes, every step of the test scenario could be successfully completed. The system loaded the previous chat history after the user logged in again.
Date	07.05.2025
Findings	The citation was not only at the end of the response, but also in the middle of the response. This is not a problem, of course.
Success	Yes
Improvments / Reasoning	-

7. Swagger Documentation Screenshots

According to the documentation, the API endpoints for the retrieval, manager, embedding, and LLM services are automatically documented using Swagger. This provides a clear overview of the available features, making it easy to explore and interact with the endpoints. The screenshots below display the Swagger documentation for each service, all captured from the basic pipeline.

7.1. Retrieval Service

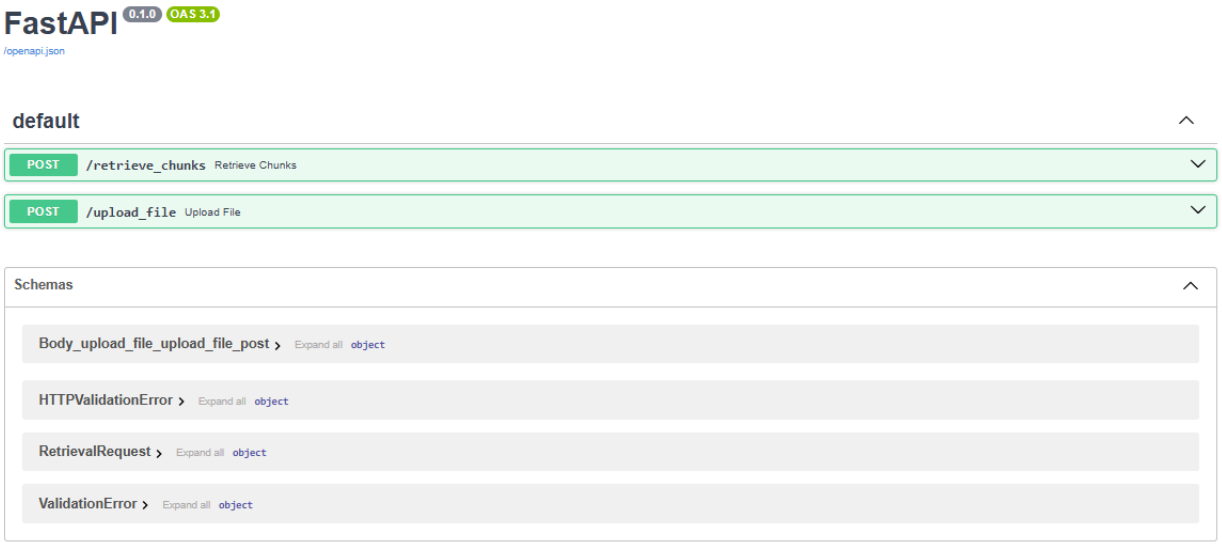

Figure 34: Swagger Documentation of the Retrieval Service - own presentment

7.2. LLM Service

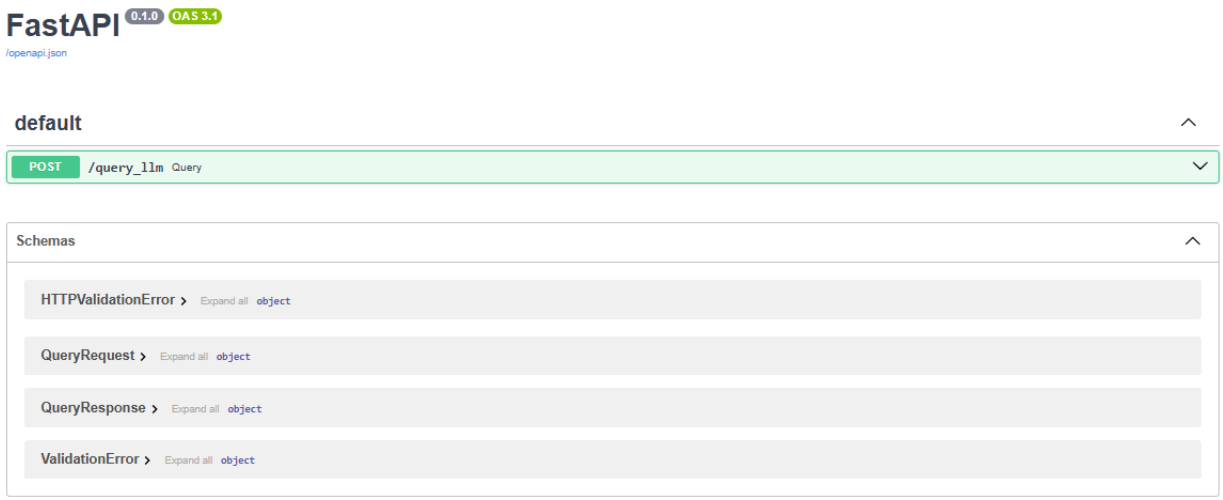

Figure 35: Swagger Documentation of the LLM Service - own presentment

7.3. Manager Service

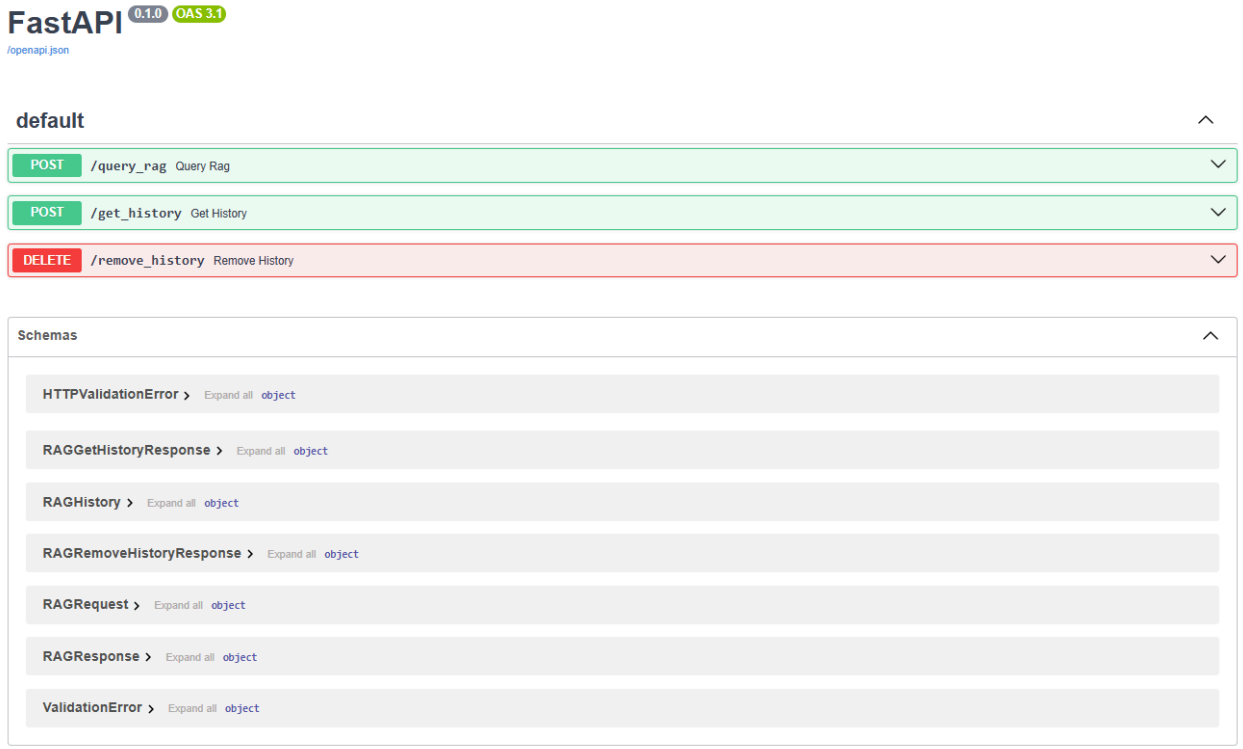

Figure 36: Swagger Documentation of the Manager Service - own presentment

7.4. Embedding Service

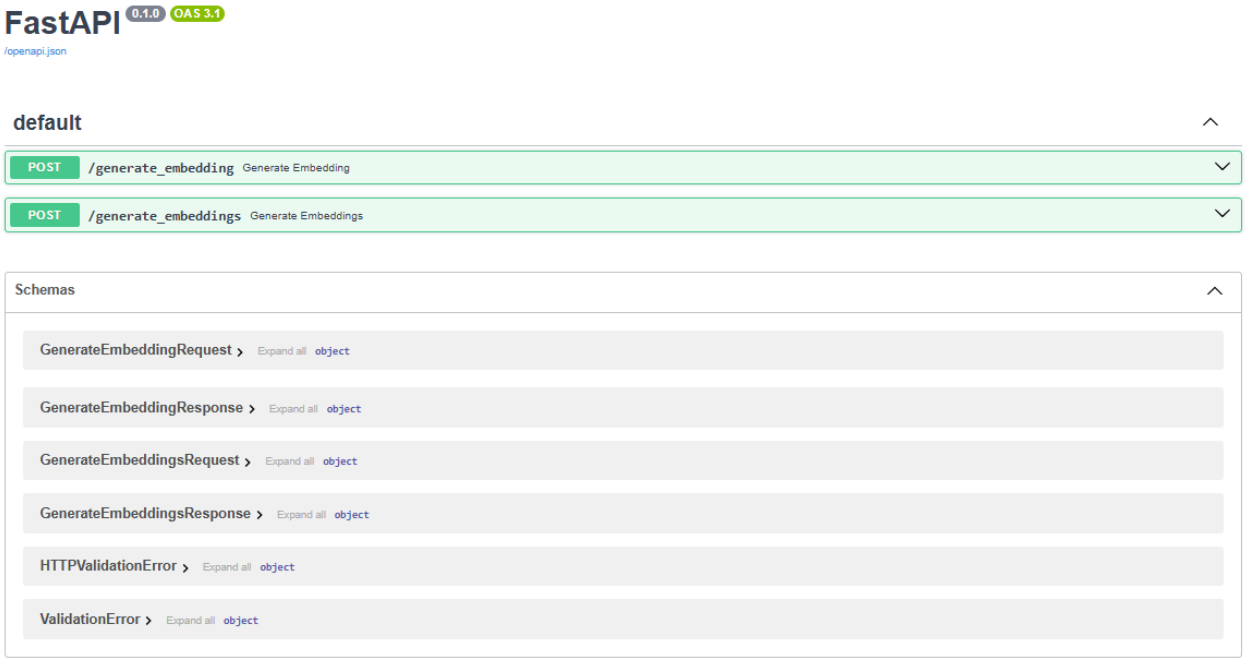

Figure 37: Swagger Documentation of the Embedding Service - own presentment