OST
Eastern Switzerland
University of Applied Sciences

Document Management using Large Language Model

Spring Term 2024

Department of Computer Science

OST - University of Applied Sciences

Campus Rapperswil-Jona

Author(s) Momoko Wymann & Andrew Willi

Advisor Frank Koch
Project Partner Michael Güntensperger
Examiner Mitra Purandare

June 14, 2024

Abstract

Initial Situation

The management and retrieval of documents can be challenging due to the unstructured nature of their content. Traditional search methods, which rely on document names, are often inefficient and ineffective. With the rise of Transformers-based Large Language Models (LLMs), which significantly enhance our everyday tasks, there is an opportunity to improve the search and management of these documents. The integration of LLMs can transform unstructured data into structured metadata, making documents more accessible and organized.

Objective

Within the scope of this project, we aim to build a Single Page Application of this type. We created this prototype application, which is able to store, read, and process PDF documents with unstructured data and generate metadata using an LLM. This metadata improves document search and management, as well as streamlines repetitive tasks such as sending an email reminder to a customer. The prototype is designed to be easily expandable to facilitate the continuous development and implementation of new features.

Conclusion

In this project, we developed a prototype application utilizing React and TypeScript for the frontend while using NestJS with TypeScript for the backend. We integrated external services such as Hugging Face and Zapier. The application enables users to upload PDF documents, extract a suitable title, summary, and tags using a Large Language Model (LLM), and search for documents based on this metadata. Additionally, it includes a feature for sending email reminders to customers for specific files. Designed with scalability in mind, the application allows for the easy addition of new features in the future, such as integrating various file types like pictures, which could utilize the LLM. This proof of concept demonstrates the potential of using LLMs to enhance document management and retrieval.

Glossary

Agile Agile is a project management and product development approach that prioritizes flexibility, collaboration, and customer satisfaction. The methodology is adaptive rather than predictive, with changes and iterations being a normal part of the development process. 66

Backend The part of a computer system or application that is not directly accessed by the user, typically responsible for storing and manipulating data. 12, 16, 17, 58

Blob A Binary Large OBject (Blob) often used to store large amounts of data like images or files. In this context, it's used to represent the α data of a PDF file. 45

Inference API An API that allows for the use of pre-trained models to perform inference on data. 39, 69

Integration A process by which Zapier connects different applications, enabling them to communicate and automate workflows by triggering actions in one app based on events in another. 43

Jira Cloud Jira Cloud, developed by Atlassian, is a cloud-based project management and issue tracking solution. It offers a comprehensive suite of features designed to assist teams in planning, tracking, and managing their projects, tasks, and workflows effectively. Accessible through any web browser, Jira Cloud eliminates the need for self-hosted infrastructure. Key features include customizable issue tracking, agile project management, real-time collaboration, robust reporting and analytics, and seamless integration with popular tools. Jira Cloud is a scalable and flexible solution suitable for teams of all sizes and industries. It facilitates efficient project management and streamlined workflows, thereby enhancing productivity.. 9

Single Page Application In the context of React, a "Single Page Application" (SPA) is a web application that dynamically updates its content without requiring a full page reload. In a React Native SPA, the user interface is built using a single HTML page, and navigation between different views or screens is managed through JavaScript, typically using a library like React Navigation. SPAs provide a seamless and responsive user experience by loading data and updating the UI asynchronously, resulting in faster and more interactive applications. 16

Task In the context of this paper, a task relates to a specific machine learning task that a model is trained to perform. There are 6 different types of tasks: Computer Vision, Natural Language Processing, Audio, Tabular, Multimodal, Reinforcement Learning. In this project, we focus on the Natural Language Processing task. 39

Transformers.js Transformers.js is a JavaScript library that provides a simple and efficient way to use pre-trained transformer models for natural language processing tasks. The library is built on top of the Hugging Face Transformers library, allowing developers to easily integrate state-of-the-art models into their JavaScript applications. Link to Hugging Face. 69

Zero-Shot Classification Zero-shot classification is a task in Hugging Face and a technique used in machine learning to assign a label to a given text without having access to the training data. 39

Management Summary

In many companies, scanned documents are stored as PDFs, which makes searching and managing them difficult. Traditionally, documents can only be searched by file name, which is inefficient and unreliable. This project aims to develop a more intelligent document management and search solution through the use of Large Language Models (LLMs).

The goal was to develop a web application that analyzes business related PDF documents, generates relevant metadata and provides an efficient search function. The development process included the implementation of a frontend for user interaction and a back end for processing and storing the data.

The application was developed using modern web technologies. The frontend was realized with React to provide a user-friendly and responsive web interface. The backend was developed with NestJS and TypeScript to ensure a robust and extensible server architecture. MySQL was used to store metadata and user information, while DigitalOcean serves as an object store for the PDF documents. Hugging Face models were integrated for analysis and metadata generation. In addition, Zapier is used to automate business processes, such as notifications to the accounting department for certain document types. Docker was used for containerization and deployment in both development and production environments, which increases the consistency and scalability of the application.

The developed application offers several key features: automated uploading and processing of documents, advanced search functions and user authentication. PDFs can be uploaded and automatically analyzed, with metadata such as summaries and categories generated and stored in the database. Users can search for documents by text and categories as shown in figure 0.1, significantly improving search efficiency and accuracy. Access to the application is only possible after successful login, see figure 0.2 of the login page, which increases security. Integration with Zapier allows automatic notifications and other processes to be triggered, increasing operational efficiency.

For example, a user can upload a PDF document, which is then analyzed and tagged with metadata. The user can then search for documents and filter them based on the generated metadata.

Future enhancements to the application could include a semantic search function that recognizes similar documents, a comprehensive user management system for managing organizations and their members, and improving performance and usability by paginating search results and optimizing load times.

This application offers an innovative solution for managing and searching PDF documents in companies. The combination of modern web technologies and LLMs significantly increases efficiency and simplifies operation. The security measures and automated processes ensure reliable and effective use in the corporate context.

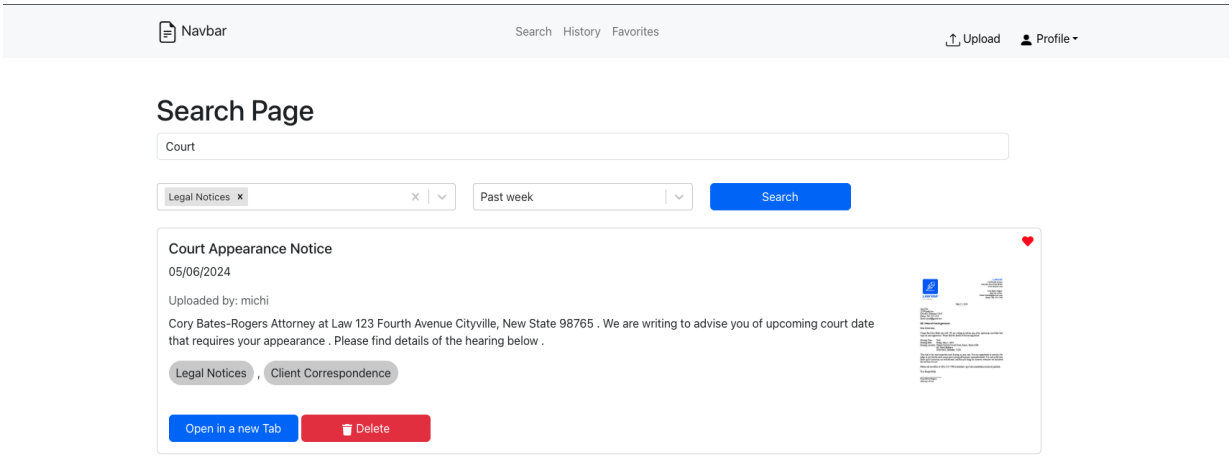

Figure 0.1: The Search Page Where the User can Search for Suitable Documents.

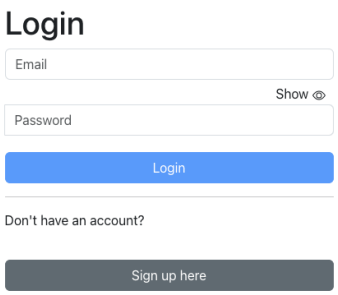

Figure 0.2: The Login Page Where the User can Log In.

Contents

1 Starting Position

As IT enthusiasts, we are dedicated to exploring innovative solutions that automate mundane and repetitive tasks. With the advent of transformers and their ability to simplify complex processes, we are keen to investigate the potential of using LLM technologies. Despite our prior experience with similar technologies, the creation of an interface specifically for transformers presents a novel challenge, which enhances the appeal and educational value of this project.
The objective of this project is to shift the focus from manual document management tasks to automated processes, thereby improving efficiency and accuracy. By leveraging LLM capabilities, we intend to develop an application that not only streamlines document management but also introduces advanced functionalities such as metadata generation, categorization, and automated workflows. This project represents a significant step towards harnessing the power of modern AI technologies to optimize and transform traditional business practices.

2 Conceptual Formulation

The objective of this proposal is to assist small businesses in managing their documents efficiently. To this end, we suggest the development of an application that facilitates the upload of PDF files and generates metadata, such as a suitable title, summary, and categories for improved document searchability. Furthermore, the application triggers automated processes based on the generated metadata, streamlining repetitive tasks such as sending a reminder email to a customer.

3 Requirements

We use Agile Methodologies to keep track of our functional requirements. These are tracked as User-Stories or issues in Jira Cloud. We prioritize the User-Stories and issues in our Jira Cloud Backlog.

3.1 Functional Requirements

All functional requirements are listed as use cases in table 3.1 and figure 3.1 shows the connection between actor and use cases in a use case diagram. These requirements are based on the task assignments.

Actor

The system has different actors with different permissions. There is a normal user, with the aim of uploading documents and easily accessing the stored documents. There is also an administrator with the authorization to delete documents.

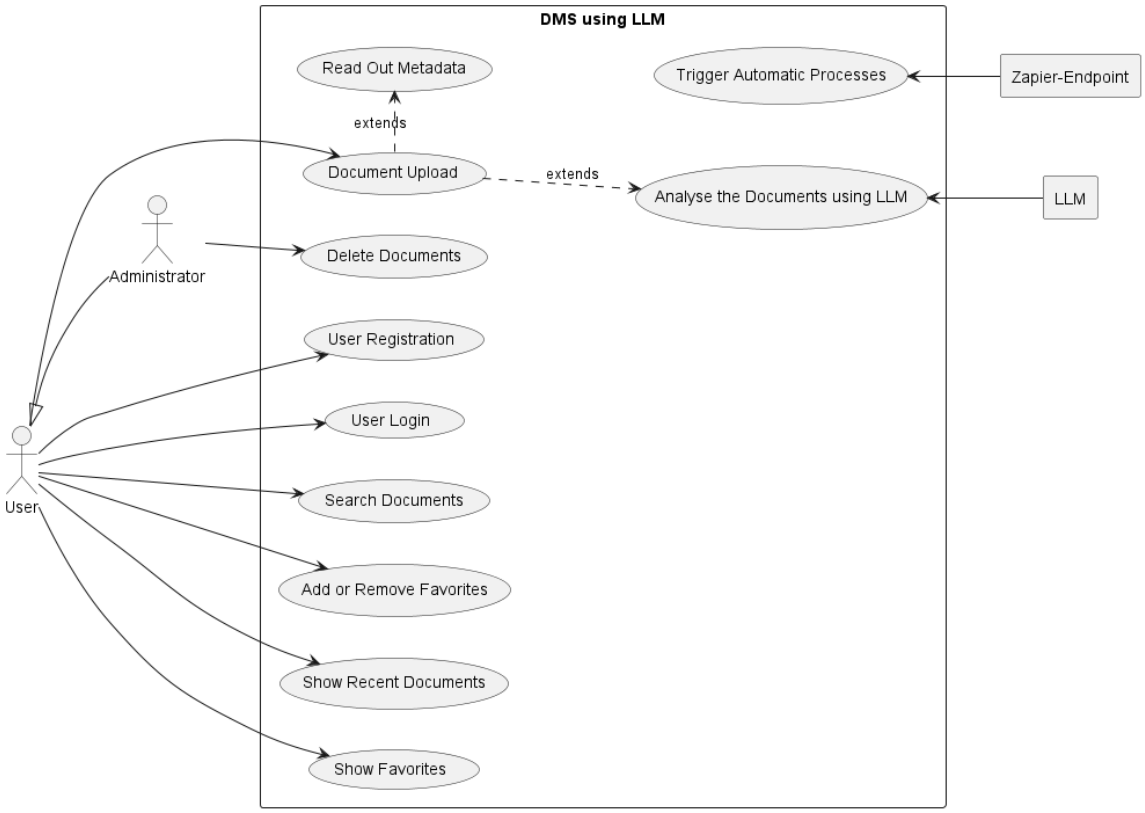

Figure 3.1: Use Case Diagram

3.1.1 Use Case Description

The following table lists the use cases of the system:

#	Actor	Goal	Description
UC1	User	User Registration	The user opens the application and the system provides the registration form. Then the user enters a new username, email and password. The password has to fulfil some acceptance criteria and the email address must not yet be assigned in the system. If the registration is successful, the user is granted access to the application and will be redirected to the home page.
UC2	User	User Login	If the user is already registered, he can go to the login page.There the system provides the login form. Then the user enters their login credentials and the system validates them. If authentication is successful, the user is granted access to the application and will be redirected to the home page.
UC3	User	Document Upload	The user selects the option to upload documents and the system prompts the user to select the documents to be uploaded. The user can also upload documents via drag and drop. The system automatically uploads the selected documents and saves them in the Object Store.
UC4	System	Read Out Metadata	The system extracts relevant informations from the documents that have been uploaded to the object store. The system saves the extracted metadata in the database.
UC5	User	Search Documents	The User is on the search page of the application. The user can enter free text search and is also provided with a selection of tags to refine the search. The system provides the documents that match the search criteria.
UC6	System	Analyse the Documents using LLM	The system uses the LLM to generate summaries, categories and additional metadata for the uploaded documents. The generated summaries, categories and metadata are stored in the database.

Continued on next page

#	Actor	Goal	Description
UC7	System	Trigger Automatic Processes	The system triggers an API call to an n8n or Zapier endpoint after documents have been successfully analysed with the LLM. The endpoint receives the API call and initiates automated processes based on the generated categories of the uploaded documents.
UC8	User	Add or Remove Favorites	The user can add documents to his favorites by clicking on the add favorite button next to the corresponding document. He can also remove documents from his favorites by clicking on the remove favorites button next to the corresponding document.
UC9	User	Show Recent Documents	The User is on the recent page of the application. The system provides the user the documents he has uploaded. The user can also enter free text search and is also provided with a selection of tags to specify the display.
UC10	User	Show Favorites	The User is on the favorites page of the application. The system provides the user the documents he has marked as favorites.
UC11	Administrator	Delete Documents	The administrator can delete the document completely by clicking on the delete button next to the corresponding document.

Table 3.1: Use Cases

3.1.2 Evolution of Functional Requirements

In the course of the project, the functional requirements have been adapted to the current state of the project. The following changes have been made:

- Added User Registration functionality alongside User Login.
- Included management of documents, such as favoriting and viewing.
- Added an administrator role with the ability to delete documents.

3.2 Non-Functional Requirements

Our Non-functional requirements are defined using FURPS+'s definition. Unless otherwise specified, all of the NFR mentioned below are requirements.

NFR #	Type	Description	Acceptance Criteria
NFR 1	Feature	The development team implements features according to the agreed-upon priorities with the customer.	Features implemented according to agreed priority order.
NFR 2	Performance	The Backend should handle 1,000 requests per minute.	Backend maintains performance with 1,000 requests per minute.
NFR 3	Performance	Pages should load within 200ms.	All pages load within 200ms for a responsive user experience.
NFR 4	Usability	The page should look good on the desktop (responsive design is desirable).	The page is aesthetically pleasing on desktop; responsive design is desirable.
NFR 5	Compatibility	The web application should run on Firefox, Chrome, and Safari browsers.	Web application compatible with Firefox, Chrome, and Safari browsers.
NFR 6	Usability	The application should be accessible via the internet using a domain provided by the customer.	Users can access the application via the internet using a domain provided by the customer.
NFR 7	Usability	Three out of four test users should rate the UI of the application (categories: layout, responsiveness, color, content) with a score of at least 8 out of 10 on a PC.	UI rated by test users with at least 8 out of 10 on PC.
NFR 8	Performance	After scanning a document, it should be displayed on the page within 30 seconds.	Document displayed on page within 30 seconds of scanning.
NFR 9	Scalability	The database should be able to manage up to 10,000 documents and 100 users.	The database manages up to 10,000 documents and 100 users without performance degradation.
NFR 10	Reliability	Errors should not cause system failures but display error messages and revert the system to the previous state.	Errors display messages, restore the system, and prevent failures.
NFR 11	Logging	Every error should be logged for monitoring and troubleshooting.	All errors are logged for monitoring and troubleshooting.
NFR 12	Security	Communication between frontend and Backend should be encrypted with SSL certificates.	SSL certificates secure frontend-backend communication.

Continued on next page

NFR #	Type	Description	Acceptance Criteria
NFR 13	Security	Input data should be validated before processing, with no SQL injection vulnerabilities.	Input data validated, no SQL injection vulnerabilities.
NFR 14	Security	The web application should be compliant with data protection regulations.	The web application complies with data protection regulations (e.g., GDPR).
NFR 15	Security	User passwords should be securely hashed in the database.	User passwords hashed securely, not stored in plain text.
NFR 16	Security	Users should only be able to view data they have access to upon login.	Users view only authorized data upon login.
NFR 17	Flexibility	Backend logic should be modular for easy expansion.	Backend logic designed modularly for easy expansion.
NFR 18	Testing	Backend API should be thoroughly tested with appropriate tools.	Backend API thoroughly tested with suitable tools, ensuring functionality and reliability.
NFR 19	Deployment	Implemented functionalities (database, backend, frontend, etc.) should be deployed.	Implemented functionalities (database, backend, frontend, etc.) deployed.

Table 3.2: Non-Functional Requirements

3.3 Optional Requirements

These optional features address various aspects of document management, user interaction and overall system customisation. The table below outlines these optional requirements:

Nr	Description	Type
1	Categorize and tag documents, e.g., as invoices.	Functional
2	Dashboard for users displaying key metrics, such as most viewed PDFs.	Non-Functional
3	Allow manual editing of metadata.	Functional
4	User management for the administration of organizational employees.	Functional

Table 3.3: Optional Requirements

4 Architecture and Design

4.1 Visualizing the Architecture

Based on the Conceptual Formulation and the Requirements, we have created the architectures and designs of the application described in the following chapters. To illustrate these architectures, we have chosen to adopt the C4-Model standard. It is worth noting that only the first three levels of the model have been implemented, as the fourth level is too detailed and would not provide any added value to the reader.

System Context (C4 Model Level 1)

The diagram illustrates how the user and admin interact with the Document Management System, which interacts with two additional software systems. One is the Workflow Service, which polls from our Document Management System and is needed to set up the flow of data between different tools and services. The other is the LLM service, which utilizes API calls to generate metadata for the document.

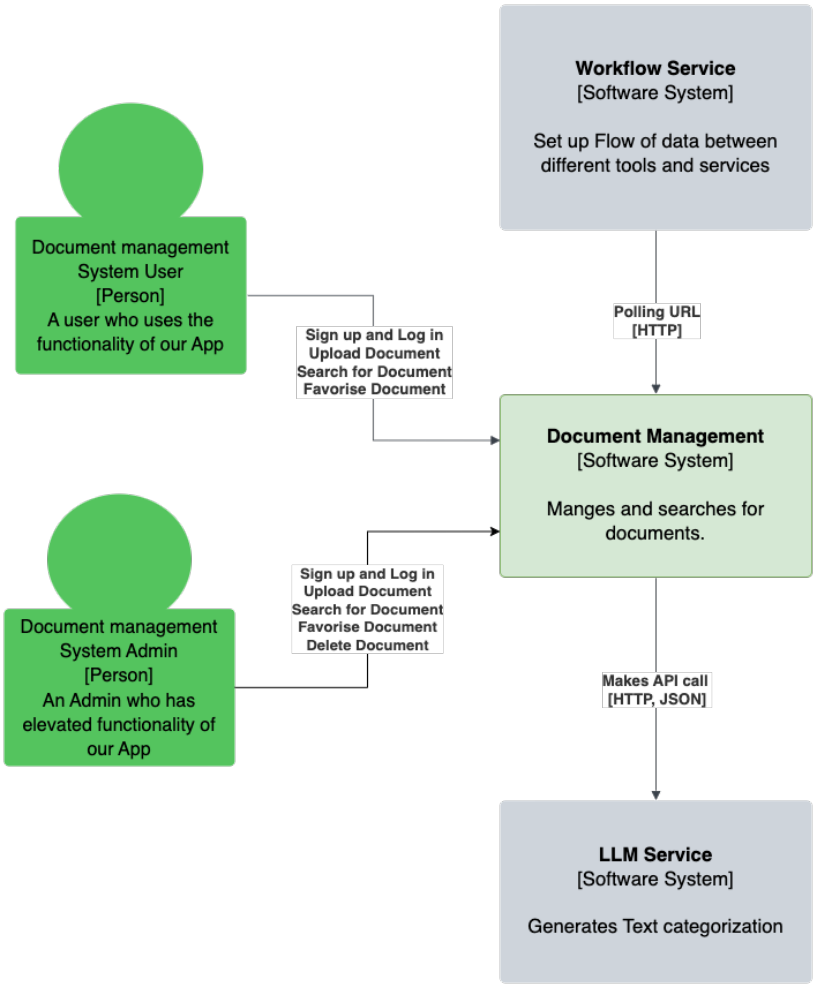

Figure 4.1: C4 Model Level 1

Container Diagram (C4 Model Level 2)

The internal structure of the application is examined in greater detail. As depicted, it consists of four components: the Single Page Application (SPA), the Object Storage, the Database, and the Backend Application. The latter serves as the central hub for the system, managing the communication between the other components.

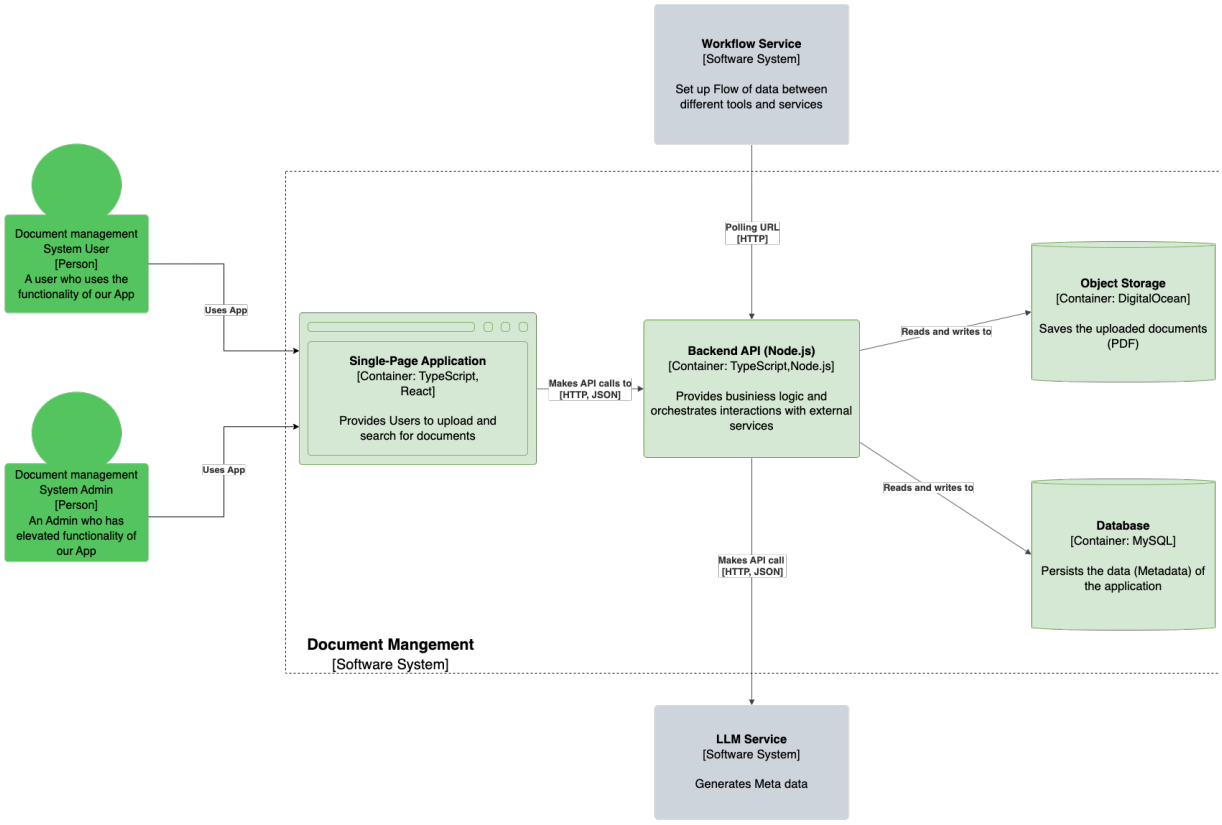

Figure 4.2: C4 Model Level 2

4.2 Architecture in Detail

In the next chapters, we will go into more detail about the architecture of the frontend and backend. We will also provide wireframes of the frontend application to give a visual representation of the application's layout and functionality.

4.2.1 Frontend Architecture

Component Diagram (C4 Model Level 3)

Since the frontend application is designed to be straightforward, we opted for a simple architecture. Users, including general users and administrators, interact with the system through the Index.tsx entry point, which renders the main application component (App.tsx). This component manages top-level routes and global state, delegating specific routing tasks to the Routes component, which handles both authenticated and unauthenticated paths. UI elements are displayed through React components, encapsulated under the Component layer. The service layer, implemented using TypeScript, Axios, and React, manages communication with the Backend, making HTTP JSON calls to a Node.js-based backend API.

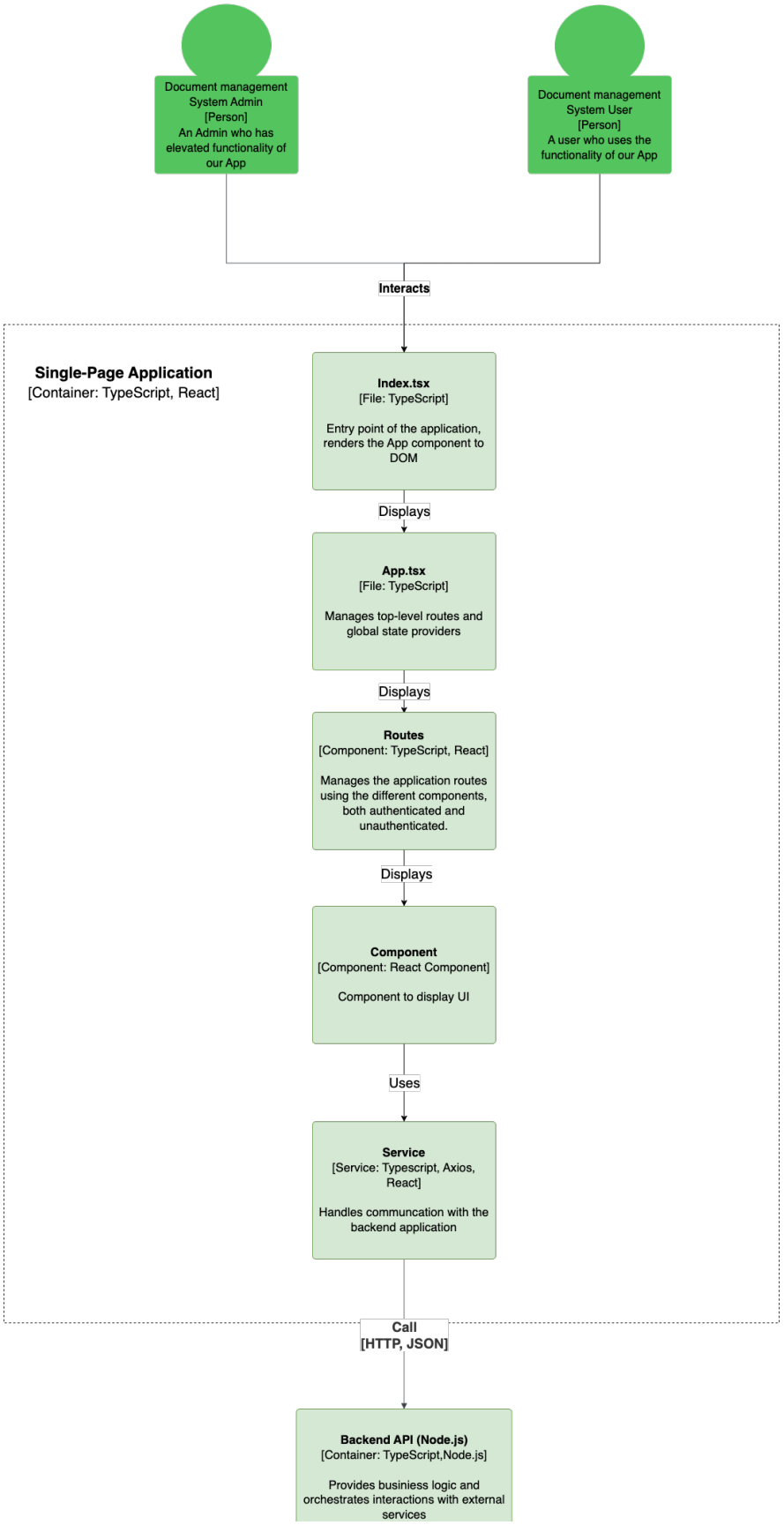

Figure 4.3: C4 Model Level 3

UI Design

This section will present the initial draft at the outset of the project. The design of a user interface is an evolutionary process, and as a result, the final product may differ from the mockups presented here. In order to provide an understanding of the evolution of the design over the course of the project, we will include the initial state of the draft at the beginning of the project. The design of the user interface may undergo modifications in response to feedback from customers and as the implementation process commences. It is important to note that this is a prototype, and as such, it serves as an initial guide for our Document Management System.
The wireframes were created in collaboration with the industry partner, who provided the initial drafts. The wireframes are intended to provide a visual representation of the application's layout and functionality.

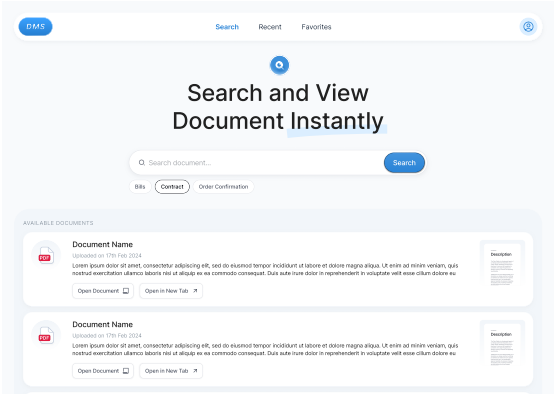

Figure 4.4: Search Page

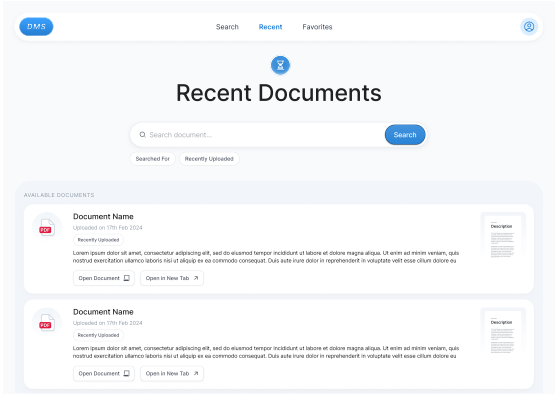

Figure 4.5: Recent Documents Page

The wireframes were then adapted to the application's design. The following wireframes illustrate the design of the application's search and document pages.
The initial two images demonstrate the user interface elements that are presented to the user upon first accessing the system. These include the signup and login pages.

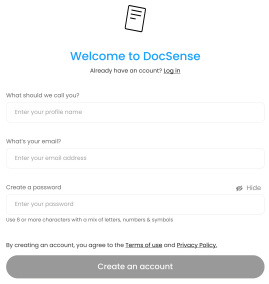

Figure 4.6: Sign Up Page

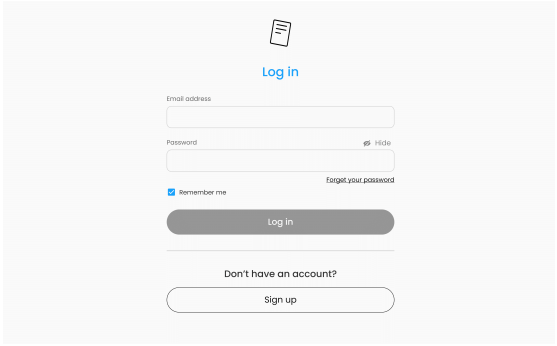

Figure 4.7: Log In Page

The following two images illustrate the design of the search pages. The first example depicts the search page in its absence of text, while the second example depicts the search page with text. We have taken some inspiration from the Google Search Bar on how the filtering should look.
The following two images illustrate Favorites and History pages.
The subsequent image depicts the interface that is displayed when the user clicks the "Upload" button.

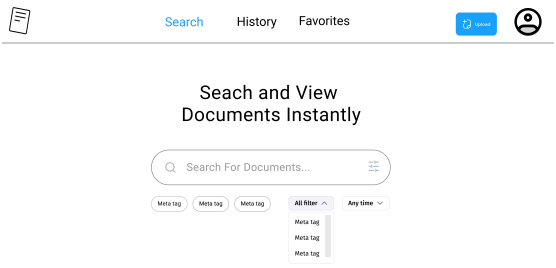

Figure 4.8: Search Without Text

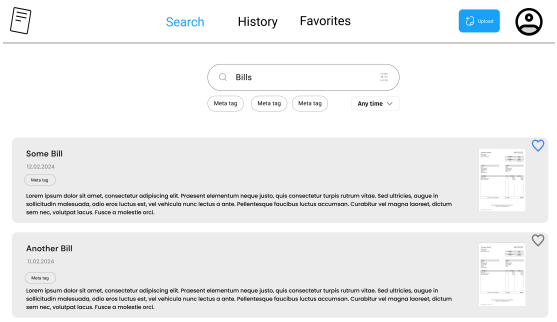

Figure 4.9: Search With Text

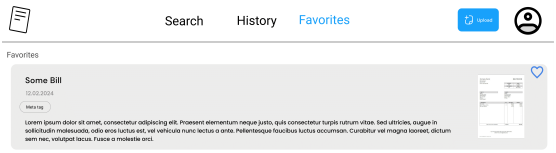

Figure 4.10: Favorites Page

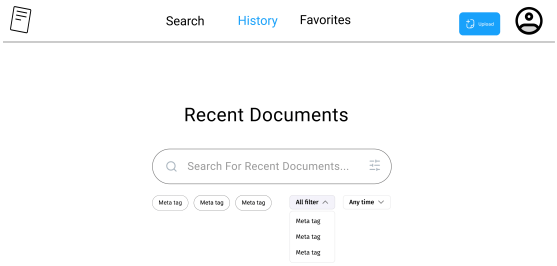

Figure 4.11: History Page

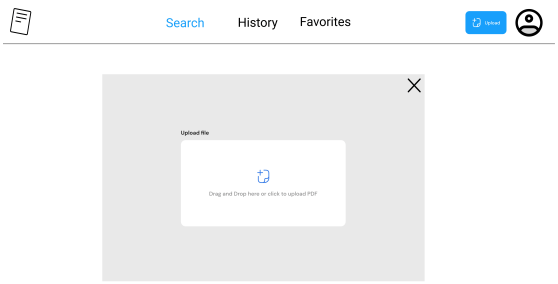

Figure 4.12: Upload Page

4.2.2 Backend Architecture

The backend architecture was designed to create a robust and scalable foundation for application delivery. It is based on a service-oriented architecture that enables a clear separation of responsibilities. The component diagram in Figure 4.13 provides an overview of the structure of the backend.

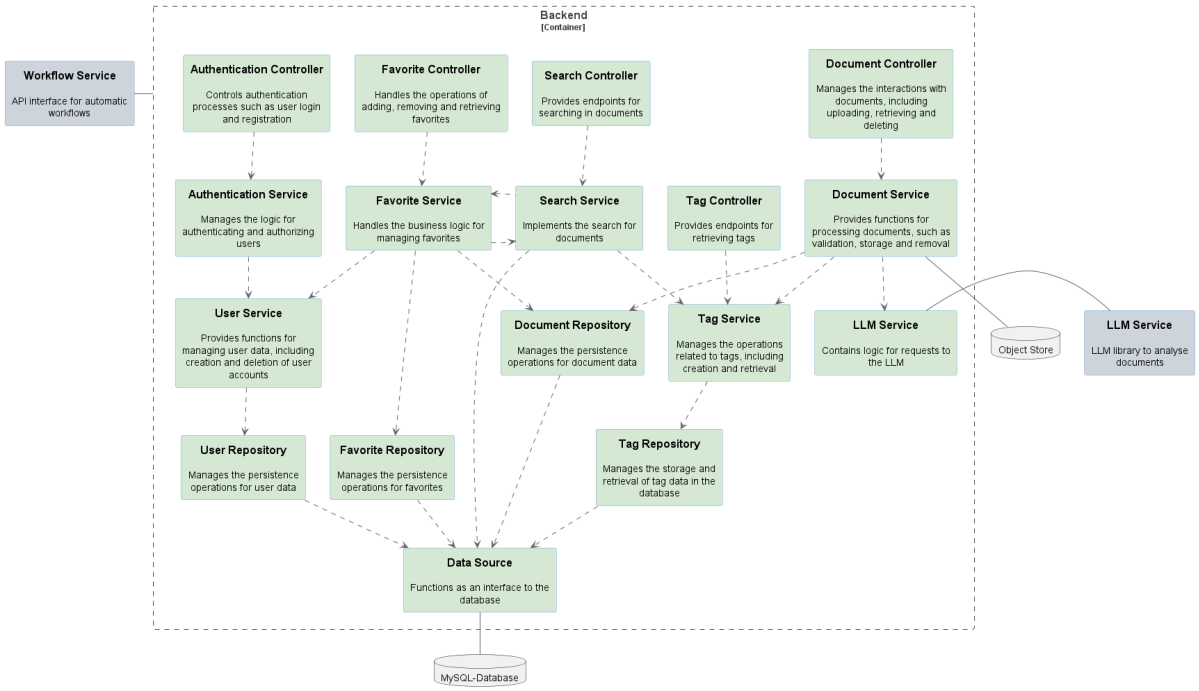

Figure 4.13: C4 Model Level 3

Concepts

RESTful API

The RESTful APIs are provided by the controllers and serve as an interface for external clients, such as the frontend, to interact with the backend. The controllers are responsible for processing HTTP requests and form the endpoints of the RESTful APIs. They receive requests from the frontend and forward them to the corresponding services. The controllers interpret the requests, perform the required actions, and send the corresponding responses back to the frontend.

Data Transfer Object

Data Transfer Objects (DTOs) are defined for the data structures that are exchanged between the controllers and the external clients. They ensure that the data is validated and transferred correctly.

Service-oriented architecture (SOA)

The architecture of the backend follows the principle of service-oriented architecture (SOA), in which individual components function as independent services that perform specific tasks. This enables high cohesion and low coupling between the components, as well as the reusability and scalability of functions. The services contain the business logic of the application and carry out the actual actions requested by the controllers. They are responsible for processing requests, manipulating data, and accessing the database or other external services.

Database access

ORM (Object-Relational Mapping) is used to connect the object-oriented backend with the relational database. Entities are used to represent the structure of the data stored in the database. Each entity corresponds to a table in the database. Database access is centralized through repositories that serve as an interface between the service components and the database. This promotes a clear separation of data access logic and business logic and facilitates the maintenance and expandability of the system.

Dependency Injection

Dependency injection is used to manage the dependencies between components and to improve the flexibility and testability of the code. This allows different implementations of interfaces to be exchanged without requiring changes to the dependent components.

Logging and error handling

Logging and error handling are integral parts of the architecture and are used to identify, monitor, and resolve problems. The use of a logging service and centralized error handling mechanisms improves the maintainability and stability of the system.

Service Components

The backend consists of a number of service components, each of which fulfills specific tasks in the system. These components are loosely coupled and can be developed and tested independently of each other.

- **Authentication Service:** The Auth Service is responsible for managing authentication processes such as user login and registration.
- **User Service:** The User Service manages user data and provides functions for creating and reading user accounts. It interacts with the database to store and retrieve user information.
- **Document Service:** The Document Service offers functions for managing documents, including uploading, downloading, and deleting.
- **Tag Service:** The Tag Service manages tags that are assigned to documents. It enables the creation and search of tags.
- **Favorite Service:** The Favorite Service allows users to mark and manage documents as favorites.
- **Search Service:** The Search Service offers functions for full-text searches in documents. It uses search techniques to identify relevant documents based on users' search queries.
- **LLM Service:** The LLM Service is responsible for creating metadata from documents using Hugging Face API.

The quality of the backend services is ensured by integration and API tests.

4.2.3 Data Model

This section describes the structure and organization of the database used for the DMS application. A relational MySQL database is used and is carefully structured to meet the application's requirements for user management, document storage, tagging, and favorites management. The schema comprises five main tables:

- **Users:** Contains all information about the users of the application, including their credentials, roles, and security tokens. Important fields: user ID, username, role, email, password, refresh token.
- **Documents:** Stores the metadata of all uploaded documents, such as upload date, title, file path, content, and thumbnails. Contains a reference to the user who uploaded the document. Important fields: document ID, upload date, title, file path, content.
- **Tags:** Manages the tags that can be assigned to documents to facilitate categorization and search. Important fields: tag ID, tag name.
- **Document Tags:** Implements a many-to-many relationship between documents and tags. Allows multiple tags to be assigned to a document and vice versa. Important fields: document ID, tag ID.
- **Favorites:** Allows users to mark documents as their favorites and find them quickly. Implements a many-to-many relationship between users and documents. Important fields: document ID, user ID.

These tables work together to support the main functions of the application.

Relationships between the tables

Figure 4.14 shows the schema of the database. A user can upload multiple documents. This is ensured by a foreign key (UploadedBy) in the Documents table, which refers to the UserId in the Users table. The intermediate table DocumentTags creates a many-to-many relationship between documents and tags. This enables flexible categorization of documents. The Favorites table allows users to add documents to their favorites. A many-to-many relationship is also mapped here.

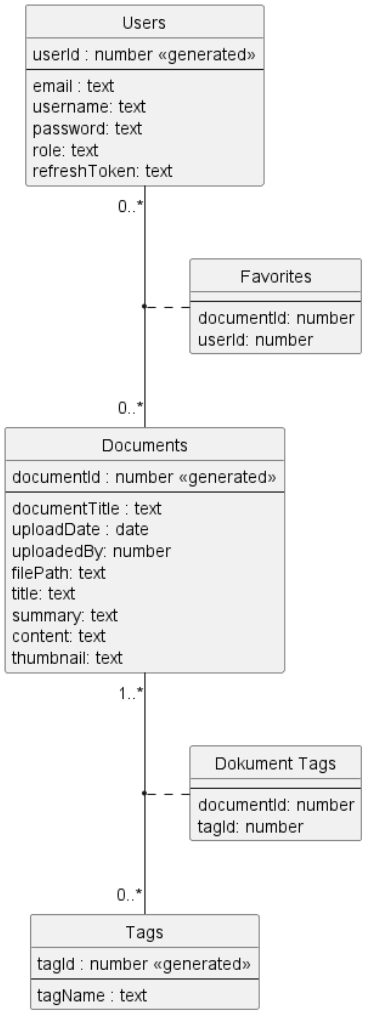

Figure 4.14: Structural design of the database

Design decisions and security aspects

- **Unique identifiers:** Primary keys and unique constraints ensure that each data record is uniquely identifiable.
- **Referential integrity:** Foreign keys guarantee the referential integrity between the tables so that orphaned data records are avoided.
- **Indexing and search:** Full-text indexes on certain fields (such as content in the Documents table) enable efficient search operations.
- **Cascading operations:** Cascading delete operations (ON DELETE CASCADE) ensure that linked data records are automatically removed when a parent data record is deleted.
- **Security:** Sensitive data such as passwords are stored in encrypted form, and security-relevant information such as refresh tokens are contained in the Users table.

5 Implementation

5.1 Technology

The decision-making process was guided by the project requirements, with certain technologies prescribed to ensure a standardized approach and others selected to align with the skills and preferences of the team. Each technology plays a pivotal role in various stages of the project, resulting in a comprehensive and versatile solution. Further details on the rationale behind each technology choice can be found in the sections that follow.

5.1.1 Programming Language

In consideration of the specifications provided by our industry partner, which indicated that the backend must be a Node application and that the frontend could be either React or Angular, we were constrained to consider only two programming languages: JavaScript and TypeScript.
Our decision to select TypeScript was motivated by its status as a superset of JavaScript and its designation as a strongly typed language. Furthermore, we deemed it advantageous to utilize the same language for team collaboration, given the clarity of interfaces and expectations that this approach would afford. Another advantage of TypeScript is the fact that we both have prior knowledge of it, which allows us to leverage its capabilities in our project.

5.1.2 Frontend Framework and Libraries

The main reason for choosing React over Angular was our team's previous experience with the former. This existing familiarity with React significantly reduces the learning curve, allowing us to accelerate the development process. In addition, React's popularity, flexibility and extensive ecosystem reinforce its suitability for our project goals, providing a solid foundation for building a scalable and maintainable frontend solution.
To complement React, we integrated React Bootstrap for its seamless alignment and pre-styled components, ensuring consistency and efficiency in UI design. For the upload functionality, we chose to integrate react-dropzone primarily because of our team's prior experience with the library. For security measures, react-password-checklist was integrated, optimizing password validation with minimal setup.

5.1.3 Backend Framework and Libraries

The framework used and the most important libraries used to develop the backend are listed here.

NestJS

The backend architecture of the application was built using NestJS, an advanced Node.js framework for building efficient, reliable and scalable server-side applications.

- **Modular Architecture:** NestJS promotes a modular architecture, making the codebase more manageable and maintainable. Modules allow the encapsulation of related components, making the application more scalable and understandable.
- **TypeScript Support:** NestJS is built with TypeScript and provides type safety, which helps detect errors during development and improves the overall developer experience.
- **Dependency Injection:** NestJS has a powerful dependency injection system that improves the testability and reusability of components.
- **Integrated Solutions:** NestJS integrates well with other libraries and frameworks and has built-in support for WebSockets, GraphQL and more.
- **Community and Ecosystem:** NestJS has a growing community and ecosystem and offers various plugins and modules that can be easily integrated into the application.

Key Libraries

@nestjs/common
Provides common decorators, interfaces and utilities for creating NestJS applications. These are essential for creating the core building blocks of a NestJS application.

@nestjs/swagger
Integrates Swagger for API documentation. This provides an easy way to generate and manage API documentation, making it easier for developers and customers to understand and interact with the API.

@nestjs/passport and passport-jwt
Implements JWT-based authentication with Passport.js. This provides robust authentication strategies and integrates seamlessly with NestJS. JWT was chosen for its stateless nature and ease of use in modern web applications.

class-validator and class-transformer
Used to validate and transform incoming request data. This ensures data integrity and security by validating user input. These libraries work well with NestJS pipes to handle the validation of request data.

typeorm
An ORM for managing database interactions. It provides a flexible and powerful ORM that supports various databases and makes it easy to perform CRUD operations and manage database schemas.

dotenv
Loads environment variables from an .env file. This simplifies configuration management and ensures that confidential information such as API keys and database credentials are not hard-coded into the application.

5.1.4 Database

The decision to use MySQL as our database was influenced by several key factors. Firstly, our industry partner specifically preferred a relational database, which made MySQL a suitable option. In addition, all team members had previous experience with MySQL, which would streamline our development process.
For the development phase, we chose to implement a dockerized MySQL database to ensure consistency across all development machines. Dockerization makes it easy to spin up and spin down the database environment so that each team member can work with a standardized MySQL instance. In addition, using Docker containers isolates dependencies, which greatly simplifies the management and replication of our development environment.

5.1.5 Docker

Docker is a platform that enables the creation, deployment and execution of applications in containers. The containers enable portability, so they can run on any platform that supports Docker without the need for customization.

The DMS application is deployed on the DigitalOcean app platform, using Docker containers for the backend and frontend. This offers the advantage that the application can be easily scaled by deploying additional container instances.

For local development and testing, docker-compose is used to create an environment with backend, frontend and test database. This offers the following advantages:

- **Consistent development environment:** All developers work in an environment that is very similar to the production environment, reducing the likelihood of environment issues.
- **Easy management of dependencies:** All dependencies, including the database, are managed in containers, which simplifies the setup and maintenance of the development environment.
- **Fast setup:** With a single command (docker-compose up), the entire environment can be started.

5.1.6 External Services

LLM Hugging Face

In selecting a platform for large-scale language model implementation, we evaluated several options, including Google AI Platform, Microsoft Azure NLP, AWS Web Services, OpenAI and OST hosted LLama2. Some platforms offer limited free services, while OST-hosted LLaMA2 is fully functional and free to use. However, our industry partner had data security concerns, which made our choice clearer.
Additionally, we found that the task-based approach of Hugging Face closely matched our requirements for executing specific language-related tasks. This led us to discover that Hugging Face's modular nature allows for flexibility and customization, which was important for our project. Furthermore, the strong community support surrounding Hugging Face played a significant role in solidifying our decision, as it provides valuable support, resources and collaboration opportunities.
Considering all these factors, we concluded that Hugging Face was the best choice for implementing our language models.

Workflow Zapier

In our search for a workflow automation platform, we evaluated several options including Make, Workato, Tray.io, n8n.io and Zapier. Our decision was based on Zapier's extensive features that aligned with our project needs. With over 3,000 app integrations, Zapier offers a vast network of automation capabilities that seamlessly connect to our existing applications. Being completely cloud-based, Zapier eliminates the need for server management and provides a maintenance-free solution. Its easy-to-use interface simplifies setup and use, making automation accessible to non-technical users.
While Zapier does offer a free tier, we found it insufficient for our needs, prompting us to opt for the premium version. We're grateful that our industry partner agreed to this decision, ensuring our workflow automation meets our project's demands effectively.

Object Storage DigitalOcean

Choosing an object storage solution for our project was easy because our industry partner already had a relationship with DigitalOcean. They already had an account and were using DigitalOcean for other projects, so it was a natural choice for us. This existing setup allowed us to integrate with their systems quickly and at no additional cost. In addition, our familiarity with DigitalOcean's infrastructure and services further validated our decision, ensuring that we could manage and navigate the platform efficiently.

5.2 Test Concept

In this section, we will describe the test concept of our project.

5.2.1 Frontend

As part of the non-functional requirements for the bachelor thesis assignment, a usability test is needed where 3 out of 4 testers should give at least 8 out of 10 points.
To evaluate the user experience and functionality of the desktop application through a series of tasks that encompass registration, login, uploading, and searching for documents.
Test environment:

Device: Test-Device (MacBook Pro 2019)

Browser: Firefox, Safari, Brave

Gmail: Provided by us

Files: Provided by us

Participant Profile:

Age: 22-55

Tech-Savviness: Moderate to High

Background Questions:

Can you briefly describe your experience with similar applications or online services?

Have you used applications with similar features before?

Scenario:
Imagine running a small business and dealing with a lot of documents every day - contracts, invoices, bills, and more. Right now, searching through files takes time and can be frustrating. You also don't want to repeat the same task to send the same document to multiple people. You're testing a new system that promises to make document management easier and faster.

Tasks:

1. **Sign up:**
   Create a user account by signing up. Follow the registration process.
2. **Login:**
   Log in using the credentials you just created.
3. **Upload:**
   Now that you are logged in, upload the file we provided you.
4. **Search own uploaded file:**
   Search for the file you just uploaded. (You can use the search bar and use the filter option, which can help you out.)
5. **Search another file:**
   Search a Bank Statement that was sent to you (Jane Customer).
6. **Look at the Gmail:**
   We have provided a Gmail account for you. Check the email and see what you can find.
7. **Favorite a document:**
   Pick any suitable document you want to favorite and look inside your Favorite section. Then you can remove it.
8. **Settings:**
   Now locate where you can log out.
9. **Log in as an admin:**
   Log in with these credentials:
   E-Mail: test@example.com
   Password: strongAdminPW
10. **Delete a document you uploaded:**
    Now you know how to look for a specific document, so please delete the last document you uploaded with the other user.
11. **Finish:**
    Congratulations, you did it! Thank you for participating.

Closing questions (in order of importance):

1. On a scale of 1 to 10, how satisfied are you with the app (layout, responsiveness, color, content)?
    - Layout
      1[ ] 2[ ] 3[ ] 4[ ] 5[ ] 6[ ] 7[ ] 8[ ] 9[ ] 10[ ]
    - Responsiveness
      1[ ] 2[ ] 3[ ] 4[ ] 5[ ] 6[ ] 7[ ] 8[ ] 9[ ] 10[ ]
    - Color
      1[ ] 2[ ] 3[ ] 4[ ] 5[ ] 6[ ] 7[ ] 8[ ] 9[ ] 10[ ]
    - Content
      1[ ] 2[ ] 3[ ] 4[ ] 5[ ] 6[ ] 7[ ] 8[ ] 9[ ] 10[ ]
2. Have you encountered any problems, bugs, or unnatural behavior throughout this test? Any negative feedback?
3. What are some positive feedback you can give us?
4. Please provide any other comments or suggestions you may have.

Consolidated Result Frontend Test

General Impressions
The application is generally seen as simple, very understandable, and has a pleasant design. Participants appreciated its clean layout and overall usability. However, some noted areas for improvement, particularly in handling document uploads and search functionality.

Positive Feedback
Participants praised the application's simple and clean design, noting that it is easy to understand and use. The integration with external APIs and the drag-and-drop feature for document uploads were highlighted as particularly effective. The responsive design that works well across different devices was also appreciated.

Negative Feedback
Several participants pointed out issues with the document tagging system, describing it as overly complicated and confusing without proper guidance. There were also mentions of the loading circle disappearing during uploads, leading to confusion about the upload status. Some participants noted that the search function could be improved by allowing searches without explicitly choosing a tag, and the upload times were longer than expected.

Confusion Points
Participants experienced some confusion regarding specific elements of the application. The purpose of the History feature was unclear to some, and the document tagging system was seen as complex. There was also feedback suggesting that more intuitive guidance and tooltips could help mitigate these issues.

Use of the Application
Overall, participants were happy with the application and found it satisfactory for their needs. They appreciated the drag-and-drop feature and the clean design, although they suggested some improvements to enhance usability further.

Ratings (Scale 1-10)

- **Layout**: Ratings ranged from 8 to 9, indicating a strong positive response to the application's design.
- **Responsiveness**: Ratings varied from 7 to 8, with some noting longer upload times.
- **Color Scheme**: Generally rated around 8, with participants finding the color scheme pleasant.
- **Clarity of Instructions and Content**: Ratings ranged from 6 to 8.5, with suggestions for clearer instructions and better guidance for some features.

General Feedback
Participants were generally very satisfied with the application's simplicity and effectiveness. However, they recommended improvements in certain areas, such as providing more intuitive guidance and reducing the complexity of the document tagging system.

Suggestions for Additional Features
Participants suggested several enhancements, including replacing the heart icon with a more intuitive button, adding checkboxes for easier file searching, integrating more keyboard shortcuts, and implementing a dynamic help system. They also expressed interest in seeing future features like AI-based document sorting to further enhance the user experience.
For further details, please refer to the Usability Test Protocol section in the appendix 9.4.

5.2.2 Backend

Integration Test

Tests are implemented in the backend to test the core functions of the services. The focus is on realistic tests using a test database. They check the functionalities of the various services in cooperation with the database. Integration tests are better suited to testing the backend than pure unit tests, as many critical functions of the services are dependent on the database. Pure unit tests would not be sufficient to ensure the correct functionality of the services. For example, creating, finding or updating entities. Therefore, most tests use real data sources and repositories to test real interactions with the database. Where appropriate, services and external resources are also mocked to reduce complexity, make the tests more efficient and ensure that the tests are stable and reproducible. The tests are carried out with jest with the support of the mock library jest-mock. They cover various use cases and test successful actions as well as error messages in the event of failures. The description of all tests can be found in the appendix in section 9.5.

A test database running in a Docker container is used for the tests. The test database can be set up with the corresponding Dockerfile, whereby an initialization file fills the test database with the test data. The test uses environment variables for the test environment, which are stored in the .env.test file. This allows tests to be carried out under realistic conditions, making the tests stable and reproducible.

API-Tests

The testing of the backend APIs was carried out using Postman. Postman is a powerful tool for the development, testing and documentation of APIs. These tests check the functionality of the various API endpoints to ensure that they work correctly and as expected. Postman enables the creation and execution of API tests, the automation of tests and the monitoring of APIs. The API tests were also executed on the test database. There is no test instance of the object store, which is why the APIs of the document controller interact directly with the real object store. To ensure that the tests do not collide with the production data, a special, very high user ID is used for these tests, which is clearly identifiable as a test ID.

A request was created in Postman for each API, which defines the required parameters, headers and body data. A test script is added to each API request, which checks the response of the API. The test scripts are written in JavaScript and use the Postman testing library to validate various aspects of the API response. A distinction is made between pre-request and post-response scripts. The pre-request scripts are executed before the actual API call and are used to ensure that all the necessary requirements for the test are met.

Most API endpoints are protected by Auth Guards, which ensure that only authenticated users can access certain resources. These endpoints require a valid bearer token, which is generated and set in the pre-request scripts. Post-request scripts are executed after the API call and are used to validate the test results. They check whether the API response meets the expected criteria, such as status code, content or structure. The API tests cover all methods of the controllers described in the API documentation in the appendix in chapter 9.1.

5.3 Deployment

This chapter describes the process and considerations made when deploying the document management system application on the DigitalOcean app platform. Among other things, the architecture, the selection of technologies, and the configuration decisions are discussed.

DigitalOcean

The entire deployment is on DigitalOcean, as our industry partner already had an account there and provided us with an environment in which we could set up all the services we needed. Both the object store and the database are hosted there. The application itself is deployed on the DigitalOcean app platform.

DigitalOcean App Platform

The DigitalOcean App Platform was chosen for the DMS application because it offers a simple, scalable, and secure solution that minimizes administrative effort while ensuring high availability and performance. The platform is particularly suitable for applications that are regularly updated and where the focus is on application development and enhancement rather than infrastructure details.

The DMS application consists of two main components: the frontend and the backend. Both components are packaged in separate Docker containers and deployed as part of a single DigitalOcean app. Docker was used because the containerization of applications allows them to run consistently regardless of the environment. This makes deployment and scaling much easier.

GitHub Integration

The app platform can be easily integrated with GitHub. As a result, all changes in the code repository are automatically deployed to the DigitalOcean App Platform. This enables fast and efficient development and deployment of new features and bug fixes. The app platform monitors the specified GitHub repository and automatically starts a new build and deployment process as soon as changes are made to the defined branch “deployment”.

Security

Environment variables were used to configure the application. Environment variables enable flexible and secure configuration of the application. In particular, security-critical information such as database passwords and API keys are not stored in the code itself but are injected at runtime. This also facilitates the management of different environments such as development, test, and production.

The backend only accepts requests from the frontend for browser-based interactions. This is achieved by the origin setting in the CORS configuration of the backend. This measure prevents unauthorized sources from accessing the backend through browser requests and protects against cross-site request forgery (CSRF) attacks. However, server-to-server requests or requests from other non-browser clients are not affected by CORS settings and are still accepted by the backend.

```
app.enableCors({
    origin: ['https://the-frontend-domain.com'],
    credentials: true,
});
```

Listing 5.1: CORS Configuration in the Backend

The setting shown in listing 5.1 restricts access to the backend only to the specific URL of the frontend, which increases security.

DigitalOcean App Platform provides automatic SSL certificates for each application. This ensures that the communication between the frontend and the backend is secure and encrypted as long as the calls are made over HTTPS.

The database is configured to only accept connections from trusted sources. This setting ensures that only authorized applications and services can access the database. This configuration is set up via the “Trusted Sources” setting in the DigitalOcean database. The IP addresses or domains that are allowed to access the database are specified here. Only the DMS application itself is specified as a trusted source so that only it can access the database.

5.4 Function Implementation

This chapter will describe the most important implementation details and aspects of our project.

5.4.1 Uploading Documents

This section will examine the implementation of one of our core functionalities.

The implementation of the file upload mechanism in the frontend involves a component that leverages the react-dropzone library to facilitate file uploads. The interface for the dropzone only accepts PDF files and supports multiple file uploads, as requested by our industry partner.

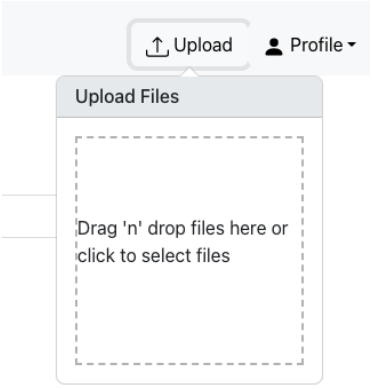

Figure 5.1: Uploading Documents

The document upload process in the backend is designed to ensure maximum data integrity and consistency. The use of database transactions and the careful sequencing of processing steps ensures that all data is stored correctly and consistently. This is shown in listing 5.2

Database Transactions

The upload process begins with the creation of a database transaction. All changes to the database and uploading to the Object Store are performed within this transaction. This is done by using a QueryRunner that initializes a transaction. The transaction ensures that all subsequent operations are either fully completed or, in the event of an error, fully rolled back to avoid inconsistent data states. Several important steps are carried out within this transaction:

- **Check for duplicates:** A check is made to see whether a document with the same name already exists in the user's object store. If this is the case, the transaction is aborted.
- **Extraction of PDF content:** The text of the PDF document is extracted and analyzed. This includes creating a summary, generating a title and classifying tags using LLMs.
- **Creation of thumbnail:** A thumbnail for the preview is created.
- **Creation of the document entity:** A new document entity is created and filled with the extracted metadata and the thumbnail.

If a step fails, all previous changes are undone.

Thumbnail creation

The creation of thumbnails is also an essential part of the document upload process. Thumbnails provide a visual preview of the document, making it easier for users to quickly identify the document they are looking for without having to open it fully. This improves the user experience and reduces loading times and server load as less data needs to be transferred. With the pdf2pic library, the first page of the PDF file is converted into an image. Using the first page as a thumbnail provides a representative preview of the document. The generated thumbnail is converted into a Base64 format, which is stored in the database and can easily be used as an image source in the frontend. The creation of thumbnails requires several dependencies and tools to convert PDF files into images and edit them. These dependencies are installed and configured during the build process.

- **GraphicsMagick:** A powerful image editing tool used to create thumbnails.
- **Ghostscript:** An interpreter for PostScript and PDF files that is required to process PDF files.
- **pdf2pic:** A Node.js library used to convert PDF pages into images.

Uploading to the Object Store

Once the document entity has been successfully saved in the database, the PDF document is uploaded to the DigitalOcean Object Store. Each user has their own folder, identified by the user ID, so that a clear distinction can be made between the documents of the different users. This process takes place as the last step within the same transaction. The upload is performed by the AWS SDK for JavaScript.

```
 async uploadFile(buffer: Buffer, filename: string, userId: number):
Promise<{ success: boolean, message: string, filename: string }> {
    //start the transaction
    const queryRunner = this.dataSource.createQueryRunner();
    await queryRunner.connect();
    await queryRunner.startTransaction();

    try{
        const exists = await this.checkDocumentInObjectStore(filename,
userId);
        if(exists) {
            throw new HttpException(`Document with name ${filename} already
exists.`, HttpStatus.BAD_REQUEST);
        }

        //creation of the metadata, the thumbnail and the document entity
        await queryRunner.manager.save(document);

        const params = {
            Bucket: this.bucketName,
            Key: `${userId}/${filename}`,
            Body: buffer,
        };
        await this.s3.upload(params).promise().then(() =>
queryRunner.commitTransaction());
        //successfully end the document upload
    } catch (error) {
        await queryRunner.rollbackTransaction();
        //unsuccessfully end the document upload
    } finally {
        await queryRunner.release();
    }
 }
```

Listing 5.2: Document Upload Implementation

5.4.2 LLM

We are utilizing Inference API Task to summarize, tag, and generate titles for the documents. We use the Axios library to send a POST request to the specific API. All the APIs have an Apache Apache License 2.0, which can be used for commercial and private use as well as distribution, stated here Apache License 2.0.

Summarization

For our `summarizeText` function, we are using Falconsai, a fine-tuned T5 model, which takes a text input that can handle lengthy documents. We also have found other working models such as BART and Google Pegasus.

Tagging

For the tagging process, we adopted a methodology similar to the `summarizeText` function. We utilize the Deberta V3 model for our tagging tasks. These models employ a Zero-Shot Classification approach for document tagging, which allows multiple tags to be assigned to a single document. Additionally, we provide the `labels` parameter to the `classifyText` function, currently encompassing ten possible tags. The response includes scores for each label. Labels with scores above a predefined threshold are selected as applicable tags for the document. This method enhances the flexibility and accuracy of the tagging process.

Title Generation

Similarly, for the generation of titles, we employ a dedicated function, `generateTitle`, which automates the creation of document titles based on the content. We utilize the Czearing T5 model, a fine-tuned version of T5. Additionally, we have explored Alpaca, a fine-tuned GPT-Neo model. This function interacts with an external API to generate appropriate titles.

5.4.3 Searching

This section will examine another core functionality of our project.

In the frontend, the `SearchPage` component is the core of the search functionality, as well as our home page, after logging in. It integrates several key React components and services to provide a seamless user experience. The component structure includes:

- **Header**: A navigation bar that includes links to various sections and a file upload feature.
- **Search Bar**: An input field where users can enter search terms.
- **Tag Filter**: A multi-select dropdown allowing users to filter search results by tags.
- **Time Range Filter**: A dropdown to filter documents based on their upload date.
- **Document List**: A dynamically generated list of search results displayed using React Bootstrap cards.

The component uses state management to handle search terms, selected tags, time ranges, and document data. It also includes authentication checks, ensuring that only authenticated users can access the search functionality. More visual details are provided in the appendix here.

Search Service for API Calls

The `SearchService` class manages the communication between the frontend and the backend API. It extends the `AxiosBaseService` to handle HTTP requests, providing methods for searching documents, fetching recent documents, and performing search operations on recent documents.

```
class SearchService extends AxiosBaseService {
    public async search(accessToken: string, tags: string[], searchText:
   string, timeRange: string): Promise<DocumentResponseDto> {
        try {
            const response: AxiosResponse = await this.instance.get('/search', {
                params: { tags, text: searchText, timeRange },
                headers: { 'Authorization': `Bearer ${accessToken}` }
            });
            return response.data;
        } catch (error) {
            ...
        }
    }
}
```

Listing 5.3: Code for Search Service in Frontend

This method constructs a GET request with the specified parameters and authorization header, retrieving the search results from the backend.
The backend class `SearchService` is used to search for documents based on the criteria received. The search service contains the main logic for searching for documents and accesses the data source directly, performing complex queries to find relevant documents. A dynamic query builder is used to create an SQL query that is tailored to the specified criteria. Since the query contains complex joins and conditional logics, no repositories are used as in the other services, which allows finer control over the executed SQL queries. The use of parameterized

queries prevents SQL injection attacks, as the parameter values are not directly embedded in the SQL query.

```
@Injectable()
export class SearchService {
    constructor(private dataSource: DataSource, private favoritesService:
   FavoritesService, private tagsService: TagService) {}

    async searchDocuments(tags: string[], searchText: string, userId: number,
   timeRange: TimeRange | null, recent: boolean): Promise<DocumentDto[]> {
        const timeCondition = this.getTimeRange(timeRange);
        ...
        const queryBuilder = this.dataSource.createQueryBuilder()
            .select('d')
            .addSelect('GROUP_CONCAT(t.TagName) AS tags')
            .addSelect(`    //relevant sorting
                CASE
                WHEN COUNT(t.TagName) = :totalTags AND MATCH(d.Content) AGAINST
   (:searchText IN NATURAL LANGUAGE MODE) > 0 THEN 1
                WHEN MATCH(d.Content) AGAINST (:searchText IN NATURAL LANGUAGE
   MODE) > 0 THEN 2
                WHEN COUNT(t.TagName) = :totalTags THEN 3
                ELSE 4
                END`, 'relevance')
            .from(Document, 'd')
            .leftJoin('DocumentTags', 'dt', 'd.DocumentId = dt.DocumentId')
            .leftJoin('Tags', 't', 'dt.TagId = t.TagId')
            .leftJoinAndSelect('d.uploadedBy', 'u')
            .where(tagQueries)
            .groupBy('d.DocumentId')
            ...

        ...
        //enables recent search in the same function to reduce code duplication
        if(recent) {
            queryBuilder.andWhere('d.UploadedBy = :userId');
        }
        //parameter transfer for parameterized queries
        queryBuilder.setParameters({ totalTags, tags, searchText, userId });

        const results = await queryBuilder.getMany();
        return await this.createDocumentDto(results, userId);
    }
}
```

Listing 5.4: Search Service Implementation

The `searchDocuments` method as shown in listing 5.4, creates a query based on tags, search text, and time range, ensuring precise and relevant search results. The method thus processes user-specific document retrievals and integrates tag-based filtering.

5.4.4 Automatic Process

In order to optimize the use of the Zapier Workflow service, we have integrated three distinct Zaps following the initial presentation of the Zap. A Zap consists of a trigger (an event in one app that starts the Zap) and actions (events completed by Zapier in other apps). This decision was influenced by our industry partner's preference to avoid using the MySQL application, which would require another program to access the DigitalOcean database. As a result, three Zaps were implemented: our custom integration provided by our industry partner, one using webhooks, and one using the MySQL application.
While the latter two Zaps are turned off, they can be published and shared publicly, while the initial Zap is private and accessible only to individuals authorized to utilize the custom integration. The subsequent figure illustrates the Zaps.

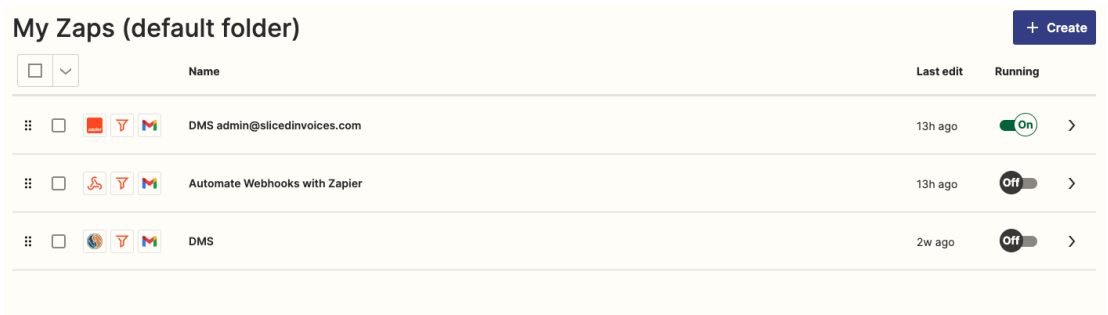

Figure 5.2: My Zaps

Implementation of the Integration

Before we can implement the integration, we need to head over to the Zapier platform website, which access was given to us by our industry partner. Authentication requires users to provide their email and password, which Zapier uses to authenticate against the DMS through a token exchange endpoint. The trigger is configured to poll the endpoint at intervals, checking for new or updated documents. It proceeds only if the document summary contains "admin@slicedinvoices" and the tags include "Invoice." Upon triggering, an email is sent via Gmail containing the document details, tagged "Invoice is due."

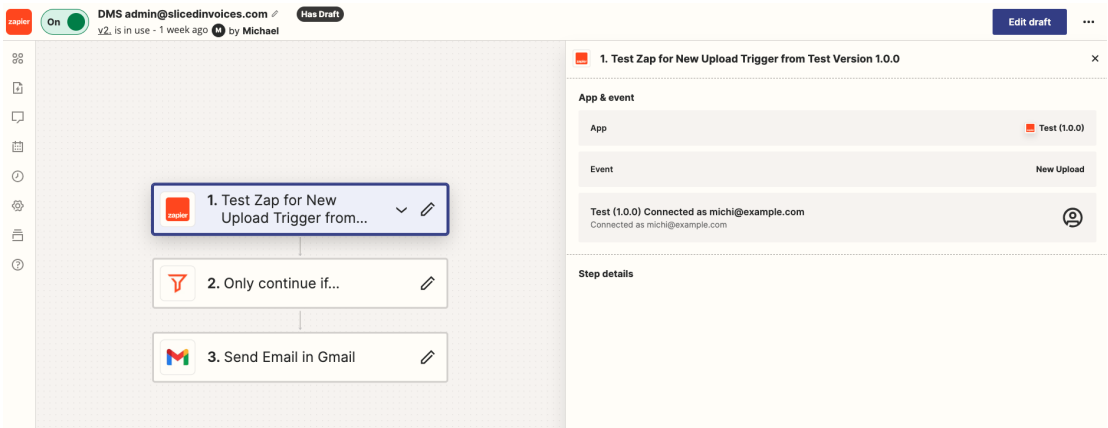

Figure 5.3: Zap with Custom Integration

Validation and testing involve initiating a GET request with appropriate headers and parameters to ensure relevance and security. The returned data is parsed to filter and map documents meet-

ing specified criteria, while expected data formats are tested to ensure consistency and accuracy in the trigger's output.

5.4.5 View

Prior to enabling the functionality for viewing individual documents in a new tab, it was necessary to create a foundational framework for displaying all documents. For this purpose, we selected the Card Component from React Bootstrap.

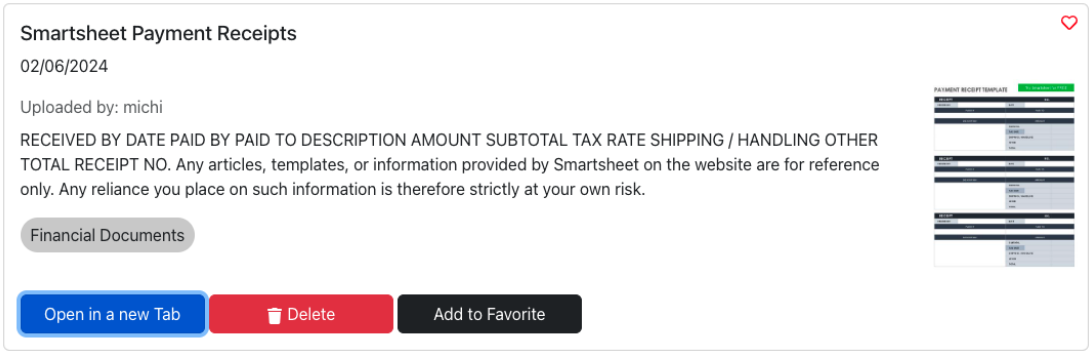

Figure 5.4: CardView of Documents

The Card Component, as visualized in the figure above, provides a comprehensive display of document metadata, including the title, summary, tags, and a preview of the document. The preview functionality is particularly noteworthy, as it sets the conversion path and options, and then converts the PDF buffer to a JPEG image. This process is designed to handle any potential errors and returns a base64-encoded string that can be used to render a thumbnail preview for each document.
The user may initiate a click action on the card, thereby triggering the display of the document in a new tab. Upon clicking the button, the system fetches the document in the form of a Blob, subsequently creating a new Blob object containing the retrieved PDF data. This is then utilized to generate a URL object, which is subsequently opened in a new tab using the `window.open` method with the `_blank` target, as shown below:

```
const openDocument = async (documentId: number) => {
    const pdf = await
   fileService.getDocument(localStorage.getItem('accessToken') || '',
   documentId);
    const file = new Blob([pdf], { type: 'application/pdf' });
    const fileURL = URL.createObjectURL(file);
    window.open(fileURL, '_blank');
};
```

Listing 5.5: Code for Opening a Document

5.4.6 Delete

The delete button, prominently displayed in the Card Component (as shown in Figure 5.4), allows users to remove documents from the system. When a user clicks the delete button, a confirmation dialog is presented to prevent accidental deletions. Upon confirmation, the system triggers the document deletion process, which is handled by the backend service. The backend service ensures that the document is deleted from both the database and the object store, thereby preventing any residual data from being left behind. This dual deletion mechanism provides a robust and reliable way to manage document deletion.

Subsequently, the document list is refreshed by invoking the `search` function.
The backend service for deleting a document involves the following steps:

1. An authentication check is conducted to verify that the requesting user possesses the necessary permissions to execute the deletion.
2. A document existence check is performed to confirm that the document is present in the database.
3. The document is deleted from both the database and object store by removing its corresponding record and file respectively.

```
async deleteDocument(documentId: number, userId: number): Promise<void>  {
    const user = await this.userRepository.findOneBy({userId});
    if (!user || user.role !== 'admin') {
        throw new HttpException('Access denied. Admin role required.',
   HttpStatus.FORBIDDEN);
    }
    const document = await this.documentRepository.findOne({
        where: {documentId},
        relations: ['uploadedBy'],
    });
    if (!document) {
        throw new HttpException('Document not found.', HttpStatus.NOT_FOUND);
    }

    const queryRunner = this.dataSource.createQueryRunner();
    await queryRunner.connect();
    await queryRunner.startTransaction();

    try {
        await queryRunner.manager.delete(Document, documentId);
        await this.deleteDocumentInObjectStore(document.uploadedBy.userId,
   document.filePath).then(() => queryRunner.commitTransaction());
        ...
    } catch (error) {
        await queryRunner.rollbackTransaction();
        ...
    } finally {
        await queryRunner.release();
    }
}
```

Listing 5.6: DeleteDocument Function in the Backend

5.4.7 Favorites

As depicted in Figure 5.4, users can add a document to their favorites by clicking the favorite button on the document card. Favorited documents are visually distinguished within the interface, and users can easily access these documents from a dedicated favorites section within the application. On the top right, there is a heart icon, which indicates whether the document is favorited (Full Heart) or not (Outlined Heart). As long as the document is favorited, the favorite button is not visible.

5.5 Code Documentation

For readability and consistency, we decided to use TSDoc. TSDoc is a TypeScript documentation standardization proposal that looks like this:

```
export class Statistics {
  /**
   * Returns the average of two numbers.
   *
   * @remarks
   * This method is part of the {@link core-library#Statistics | Statistics
    subsystem}.
   *
   * @param x - The first input number
   * @param y - The second input number
   * @returns The arithmetic mean of `x` and `y`
   *
   * @beta
   */
  public static getAverage(x: number, y: number): number {
    return (x + y) / 2.0;
  }
}
```

Listing 5.7: TSDoc Example

5.6 Security

5.6.1 Authentication:

The authentication strategy is based on a combination of stateless and stateful JSON Web Tokens (JWT). A JWT is a compact, URL-secure token format that contains information about an entity, in this case, the user. This application uses two types of tokens: access tokens and refresh tokens.

Access Tokens Access tokens are stateless and offer high session security with low latency, as no database access is required for validation. They contain all the necessary information to verify the identity and authorizations of the user. As access tokens do not require any server-side session information, they are scalable and reduce the load on the server. A major disadvantage is that they cannot be easily revoked once they have been created. If a token is compromised, it remains valid until it expires. To minimize the risk, access tokens have a very short lifespan of around 15 minutes. However, due to the short lifespan, the user would have to log in repeatedly, which does not provide a good user experience.

Refresh Tokens In contrast to access tokens, refresh tokens are stateful and are stored in the database on the server side. They have a longer lifespan and are used to generate new access tokens without the user having to log in again. As refresh tokens are stored on the server side, they can be easily revoked, for example, when the user logs out. This increases security, as compromised refresh tokens can be deactivated. The automatic renewal of access tokens using

refresh tokens provides a seamless user experience as the user does not need to be constantly re-authenticated.

This hybrid session management solution combines the benefits of stateless JWT-based sessions with the revocation capabilities of stateful sessions:

- **Low latency and high security:** The short lifetime of access tokens minimizes the risk that compromised tokens can be misused.
- **Ease of use:** Refresh tokens ensure that the user experience is not compromised by automatically generating new access tokens.

Implementation of the Auth Guards

Auth Guards in the backend ensure that only authenticated users can access protected resources by checking and validating the JWTs. The NestJS authentication library, which is based on the @nestjs/passport library and passport-jwt, was used to implement the guards. There are two different guards, the AccessTokenGuard and the RefreshTokenGuard. This ensures that the refresh token is only used to renew the access token. These guards check whether the corresponding JWT in the Authorization header of the request is present and valid. The AccessTokenGuard also uses asynchronous local storage to store user-specific data such as the user ID in the context of the request. Listing 5.8 shows how the guards are used to protect the routes. The @ApiBearerAuth() decoration indicates that the endpoint requires JWT authentication and the @UseGuards(AccessTokenGuard) decoration applies the guard to a specific route but can also be applied to the entire controller.

```
@ApiTags('Authentication')
@Controller('auth')
export class AuthController {
    constructor(private authService: AuthService) {}

    @ApiBearerAuth()    //endpoint requires JWT authentication
    @UseGuards(AccessTokenGuard)    //guard is applied on this route
    @Get('testAuth')
    async getProfile(@Request() req: any) {
        return req.user;    //retrieve user-specified data from the request
    }

    //more endpoints

}
```

Listing 5.8: AccessTokenGuard Controlls the Access to the Routes

Authentication Process

The following sequence diagram 5.5 shows the authentication process, which encompasses user registration, login, resource access, token refresh, and logout. The user initiates the registration process by entering their details, which the client application transmits to the authentication service. This service then verifies the email address and creates a new user account, generating access and refresh tokens. These tokens are returned to the client and contain information such as the user ID, the issuance date, and the expiry date. When logging in, the user's login information is validated and tokens are issued if correct. The tokens are stored on the client side, and to access resources, they are included as bearer <token> in the Authorization header for requests. For each request, the token provided is checked by the service to ensure that it is valid and has not expired. When tokens are approaching expiration, the client initiates a request for new tokens to maintain the session. Finally, upon logout, the client instructs the authentication service to invalidate the refresh token, thereby terminating the user session.

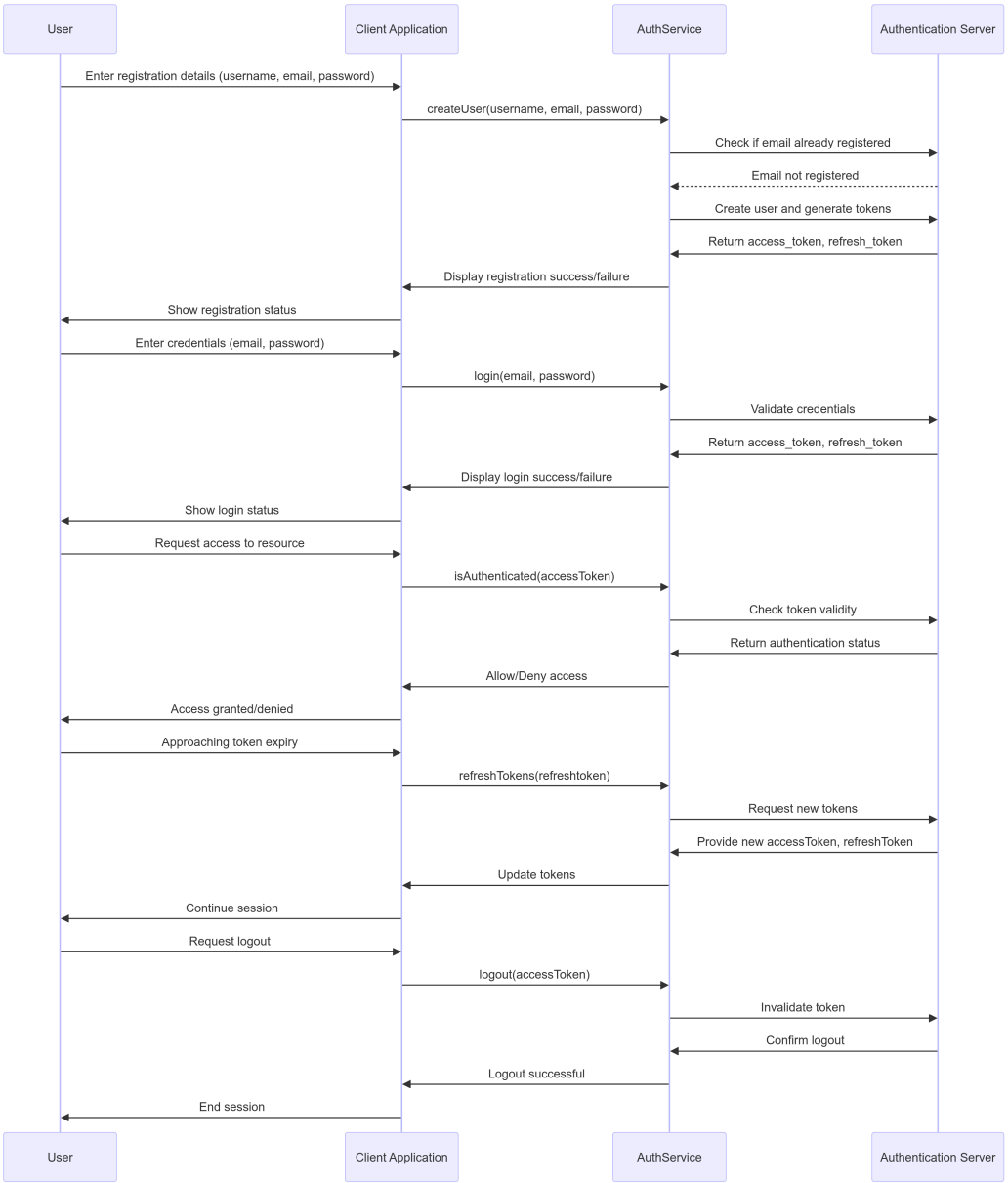

Figure 5.5: Authentication in the Frontend

5.6.2 Password Hashing

To ensure the secure storage of passwords, all passwords are hashed. For the implementation, bcrypt was chosen, which is a proven choice for password hashing in web applications. bcrypt uses an adaptive hash function that automatically adjusts the number of rounds to achieve a certain delay, making it more resource intensive and thus making brute force attacks more difficult. This delays the loading of pages, but increases security.

5.6.3 Input Validation:

Backend

Validation of the data sent to the backend is crucial to ensure the integrity and consistency of the data and to ensure that only correctly formatted and valid data is processed. A combination of DTO validation and pipes is used to enable robust and secure processing of incoming data in the backend. This is implemented through the use of 'class-validator' and 'class-transformer' in DTOs as well as user-defined pipes.

Frontend

In addition to the backend validation, validation for email, password and username has been implemented in the frontend. This helps avoid unnecessary requests and optimizes the response time in the user interface.

5.6.4 Security Measures LLM:

Security is a critical aspect of our project, particularly in conjunction with the implementation of the Hugging Face LLM. However, due to the agreement with our industry partner and the constraints imposed by limited time and resources, we were unable to implement a comprehensive security solution. At the outset of our project, the decision was made to utilize self-generated business-related PDFs. Nevertheless, it is of the utmost importance to delineate a potential framework for future security measures to ensure that the model is accessible only locally and securely.

- Access Control: Implemented through basic authentication to ensure only authorized users can access the model.
- Local Access: Restricted API access to localhost to prevent external access.
- Environment Isolation: Containerized the application using Docker for better isolation and resource management.
- Encryption: Ensure data in transit is encrypted using TLS/SSL when needed.
- Monitoring and Logging: Plan to implement logging and monitoring to track access and detect unauthorized attempts.

By following these steps, we can enhance the security of the Hugging Face transformer model, ensuring that it operates securely within a local environment and is accessible only to authorized users.

6 Result

In this project, we developed an application that runs on modern web browsers, enabling users, especially small businesses, to manage their PDF documents efficiently with the help of LLM. Users must register and log in to the application to upload the desired documents and can search for them.
The backend of this application is hosted on DigitalOcean and managed by the publisher of the application. Therefore, we were able to fulfill all previously defined functional requirements. Due to time constraints, it was not possible to implement most of the optional requirements except one: "Categorize/tag documents, e.g., as invoices."

Fulfilled Functional Requirements:

The following table contains the completed functional requirements as stated in the task formulation 9.3 with deviations as mentioned in this section 3.1.2

Functional Requirement	Result
User Registration	Users can register an account in the system.
User Login	Users can log into the system.
Document Upload	Users can upload multiple documents to the system.
Read Out Metadata	System displays metadata extracted from documents.
Search Documents	Users can search for documents with tag and time range filter.
Analyse the Documents using Large Language Model	System generates summaries, categories, title and additional metadata for documents.
Trigger Automatic Processes	System triggers process to send an Email reminder to a customer.
Add or Remove Favorites	Users can mark or unmark documents as favorites.
Show Recent Documents	System displays the most recently uploaded documents.
Show Favorites	System displays all documents marked as favorites.
Delete Documents	Admin users can delete documents from the system.

Table 6.1: Implemented Requirements and Expected Results

Fulfilled Non-Functional Requirements:

Non-Functional Requirement	Result
The development team implements features according to the agreed-upon priorities with the customer.	Features implemented according to agreed priority order.
The page should look good on the desktop (responsive design is desirable).	In the Usabilty test most of the user were happy how the page looks on desktop.
The web application should run on Firefox, Chrome, and Safari browsers.	Web application was tested on Usablity Test with Firefox, Chrome, and Safari browsers.
The application should be accessible via the internet using a domain provided by the customer.	Users can access the application via the internet using a domain provided by the customer.
Three out of four test users should rate the UI of the application (categories: layout, responsiveness, color, content) with a score of at least 8 out of 10 on a PC.	UI rated by 3 test users with at least 8 out of 10 on PC.
After scanning a document, it should be displayed on the page within 30 seconds.	One document is displayed on page within 30 seconds of scanning.
The database should be able to manage up to 10,000 documents and 100 users.	Locally tested on a docker container database.
Errors should not cause system failures but display error messages and revert the system to the previous state.	Errors display messages, restore the system, and prevent failures.
Every error should be logged for monitoring and troubleshooting.	All errors are logged for monitoring and troubleshooting.
Communication between frontend and backend should be encrypted with SSL certificates.	SSL certificates secure frontend-backend communication.
Input data should be validated before processing, with no SQL injection vulnerabilities.	Input data validated, no SQL injection vulnerabilities.
The web application should be compliant with data protection regulations.	The web application complies with data protection regulations.
User passwords should be securely hashed in the database.	User passwords hashed securely, not stored in plain text.
Users should only be able to view data they have access to upon login.	Users view only authorized data upon login.
Backend logic should be modular for easy expansion.	Backend logic designed modularly for easy expansion.
Backend API should be thoroughly tested with appropriate tools.	Backend API tested with Postman.
Implemented functionalities (database, backend, frontend, etc.) should be deployed.	Implemented functionalities (database, backend, frontend, etc.) deployed on DigitalOcean.

Table 6.2: Non-Functional Requirements

Non-Functional Requirements that were not fulfilled:

In the course of this work, we could not test **NFR 2: The backend should handle 1,000 requests per minute** This was not prioritized highly by us, as the application is intended to be an initial prototype that demonstrates functionality. Furthermore, DigitalOcean Spaces provides the necessary resources and capacities to meet the specified requirement. Nevertheless, capacity is always a matter of cost, as additional storage space is associated with higher expenses.

NFR 3: Pages should load within 200ms is not possible for pages that handle passwords, e.g., the login and sign-up pages. As **NFR 15: Secure Password Storage** is more important, this requirement was not fulfilled with today's hardware. Therefore, this requirement is only partially fulfilled.

7 Conclusion

As mentioned in the Result 6 section, we developed a foundation for an extendable service that enables users to efficiently manage their PDF documents. We successfully met all requirements, with the exception of three specified by our industry partner. Consequently, the application we developed serves as a robust foundation for a future application that can significantly simplify the management of PDF documents.
All in all, we are satisfied with the results of this project and were happy to use modern technologies. We would like to see more features in the future and more fleshed-out capabilities. This project provided us with valuable experience, taking a project from scratch to a complete prototype.
However, there are some areas that need to be addressed in the future.

7.1 Needs to Be Addressed

If we are to continue developing this service and implement new features. These are our recommendations, based on our knowledge of the application, as listed here:

- **Security:** As mentioned in this document, security concerns should be addressed and the application should be tested for vulnerabilities. Using external services is always a risk and should be monitored.
- **Testing:** Comprehensive testing is essential before deploying the application to production. Before further development, we need to extend our testing efforts, including additional unit tests, fixing existing tests, API testing, frontend testing, and more usability tests.
- **Pagination in the Frontend:** It would be beneficial to implement pagination in the frontend to improve the user experience when browsing through documents. This would make it easier to navigate through the documents and find the desired information.
- **Workflow Service:** For further development, we would like to add the fact, that Zapier is a paid service, so that it can be integrated with the application.

7.2 Future Vision

In our opinion, the application has great potential to extend its capabilities and implement more features:

- **Various file types:** Adding the ability to upload various file types, such as images, would be a great addition to the application. This would mean integrating another LLM aside from the Natural Language Processing but with Computer Vision.
- **Version control:** Implementing a version control system to keep track of changes to documents. This feature would allow users to revert to previous versions, compare different versions, and manage document revisions more efficiently.
- **Advanced Search Filters:** Enhancing the search functionality by adding advanced filters such as date range, document type, and author to make it easier to find specific documents.
- **Settings:** Enhancing the user settings to change the password or username and other settings.
- **Optional Requirement:** Manual editing of metadata and user management for the administration of organizational employees.

8 Project and Time Management

8.1 Project Schedule

Figure 8.1 shows the rough schedule at the beginning of the project. In the end, it deviated due to some factors that were unknown at the beginning of the project and some risks that were realised

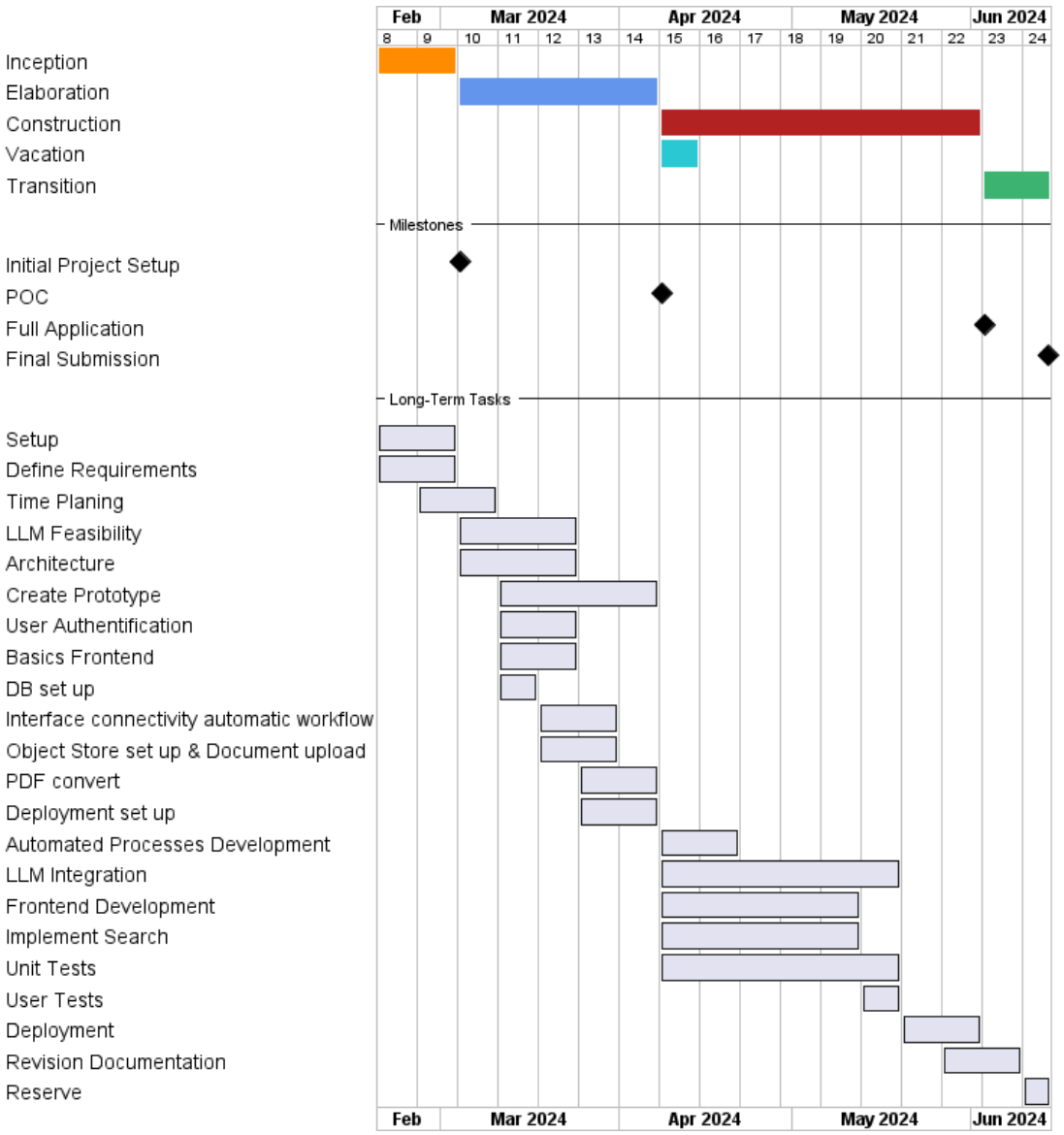

Figure 8.1: Project schedule

Phases

As Scrum+ is used for this project, the four-phase framework included in RUP will be adopted.

- **Inception**: The first phase of a project involves setting up the necessary environments, tools, and resources for the team to begin working. This includes the setup of Jira, the creation of the GitLab repository for the documentation and source code, and assigning roles to the team members. Additionally, the project's requirements are identified and documented. This phase is completed once the 'Initial Project Setup' milestone has been reached.
- **Elaboration**: In this phase, the architecture of the application is defined. A first prototype is created to ensure the compatibility of the technologies and validate the operational processes. The prototype consists of all the components from the application to test the flow between them. This includes the basics of the frontend and the Backend with user authentication, setup of the database, and the Object Store, including document upload. It also includes the connection to the LLM and automatic processes API. The LLM feasibility is also checked in this phase as well as the setup of the deployment. The elaboration phase is completed with the milestone 'POC'.
- **Construction**: The next phase is the primary stage of development. During this phase, the team writes the business logic and implements the UI. This also encompasses the integration of the LLM and the configuration of the automatic processes. Comprehensive testing is also performed to ensure that everything functions as intended. The objective of this phase is to produce a fully functional and automatically deployable software system using CI/CD. This phase is completed with the milestone 'Full Application'.
- **Transition**: The final phase will be used to finish the documentation and has an additional built-in buffer to compensate for any delays. The project and phase will be concluded with the milestone "Final Submission".

Milestones

- **Initial Project Setup**: This milestone marks the completion of the initial setup phase, where all necessary environments, tools, and resources are in place, and project requirements are documented.
- **POC (Proof of Concept)**: This milestone is reached upon completing the elaboration phase, where a working prototype demonstrates the feasibility and compatibility of the system components.
- **Full Application**: This milestone signifies the end of the construction phase, with the development of a fully functional application, including business logic, UI, LLM integration, and automated processes.
- **Final Submission**: This milestone marks the end of the transition phase, where all documentation is finalized, and the project is submitted for evaluation.

Long-Term Tasks

- **Setup**: Establishing the development environment, including setting up Jira for project management, GitLab for version control, and assigning roles to team members.

- **Define Requirements**: Identifying and documenting the requirements of the project to ensure a clear understanding of the objectives and deliverables.
- **Time Planning**: Creating a detailed schedule for the project, including all phases, milestones, and tasks to ensure timely completion.
- **LLM Feasibility**: Assessing the feasibility of integrating LLMs into the application, ensuring they meet the project's requirements.
- **Architecture**: Designing the overall architecture of the application, including the backend, frontend, database, and object storage.
- **Create Prototype**: Developing a prototype to validate the integration of different components and ensure the system works as intended.
- **User Authentication**: Implementing secure user registration, login, and access control features.
- **Basic Frontend**: Creating a simple frontend to display the results of the summarization and other basic functionalities.
- **DB Setup**: Setting up the database to store user data, document metadata, and other relevant information.
- **Interface Connectivity Automatic Workflow**: Ensuring seamless connectivity between the frontend, backend, and other components, and establishing automated workflows.
- **Object Store Setup & Document Upload**: Configuring the object storage for document uploads and managing metadata storage in the database.
- **PDF Convert**: Implementing functionality to convert documents to PDF format as needed.
- **Deployment Setup**: Setting up the deployment on DigitalOcean and ensuring the application is ready for production.
- **Automated Processes Development**: Developing automated processes for various tasks, enhancing efficiency and reducing manual intervention.
- **LLM Integration**: Integrating LLMs into the application for tasks such as text summarization, title generation, and tagging.
- **Frontend Development**: Enhancing the frontend with React Bootstrap to make it more user-friendly, appealing, and responsive.
- **Implement Search**: Implementing advanced search capabilities to quickly locate and filter documents.
- **Unit Tests**: Writing unit tests to ensure individual components work as expected and identifying any issues early in the development process.
- **User Tests**: Conducting user testing to gather feedback and make necessary improvements to the application.
- **Deployment**: Deploying the fully functional application using DigitalOcean
- **Revision Documentation**: Finalizing and refining the project documentation, ensuring it is comprehensive and up-to-date.
- **Reserve**: Allocating additional time to address any unforeseen issues or delays, ensuring the project stays on track.

8.2 Project Organization

8.2.1 Roles

To better divide the project workload, tasks are split into multiple roles among the project members. The roles are defined as shown in table 8.1, but are only intended as rough guidelines.

Task	Lead
Scrum Master	Andrew Willi
Backend Setup	Momoko Wymann
Frontend Setup	Andrew Willi
Frontend Design	Andrew Willi
External Service	Andrew Willi
Database	Momoko Wymann
Prototype	Both
Deployment	Momoko Wymann
Meeting minutes	Momoko Wymann
Meeting agenda	Andrew Willi
Documentation	Both

Table 8.1: All roles with the assigned members

8.2.2 Code Repository

To share the source code and the documentation, a monorepo GitLab repositoriy is used:

```
DMS/
├── code
│   ├── backend/
│   ├── frontend/
│   └── database/
└── documentation/
```

This structure allows us to work on the backend and frontend separately, while having the documentation in a separate folder.

8.2.3 Jira Board

The board is divided into five categories: To Do, In Progress, Awaiting Review, Done, and Meeting. A new issue is initially placed in the To Do category. Once a developer begins working on an issue, it is moved to the In Progress category. It then proceeds to the Awaiting Review stage. When the issue is successfully merged into the main branch, it is marked as Done. The meeting has its own category, making it easier to locate, because there is going to be a weekly occurrence.

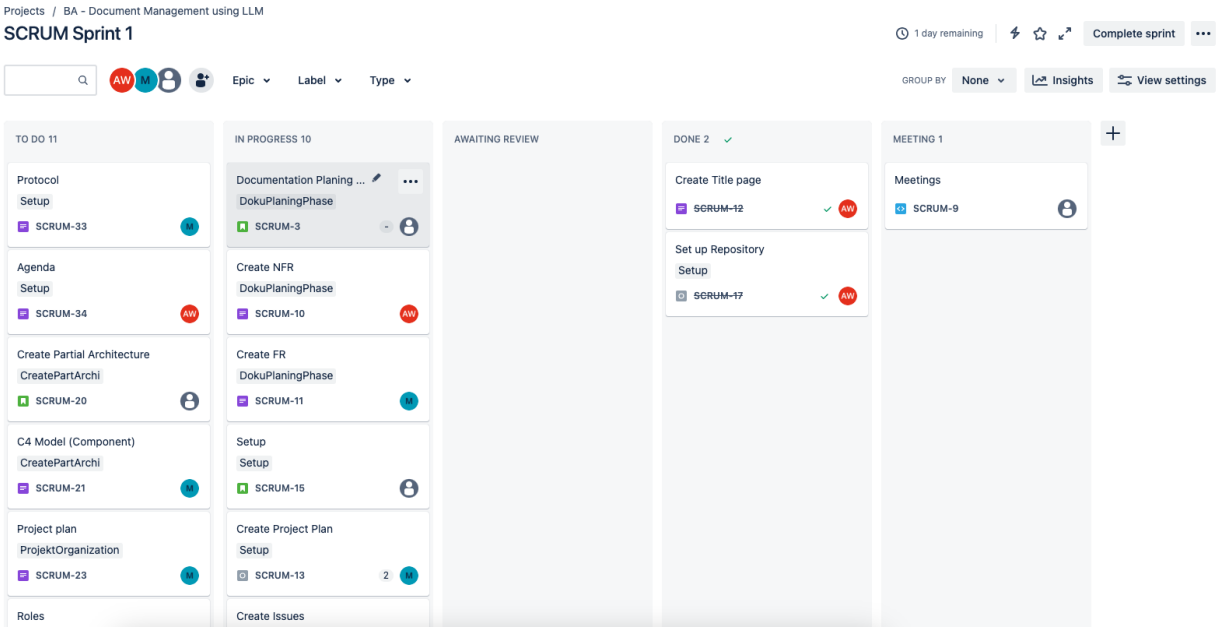

Figure 8.2: Jira Board

8.2.4 Issue Management

Our challenge revolves around reconciling the structure of Jira, which features Epics and customizable issues, with the desired Scrum hierarchy comprising Epics, Stories, and Tasks. To address this, we have introduced a Story issue as the cornerstone. Under this Story, we can have tasks. Additionally, we have elevated Meetings to a superior level, signifying their overarching importance within our workflow. This adaptation enables us to adhere to Scrum principles while utilizing Jira's customizable framework.

Epic

In the Planning phase, each process step is identified and defined as an Epic. An Epic consists of multiple user stories (green icon) and can extend beyond the sprint boundaries.

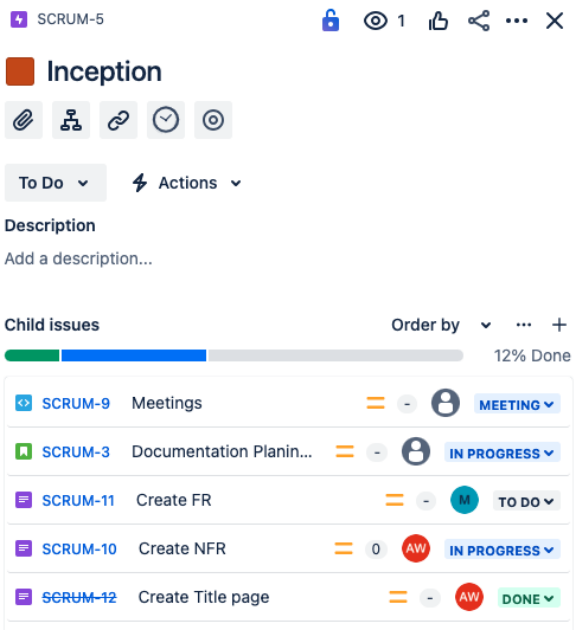

Figure 8.3: Epic

User Story

A user story describes a single functionality that needs to be implemented. It consists of individual tasks that must be completed for successful completion. The scope of a User Story should not exceed that of a sprint. In our case it is the issue with the green icon.

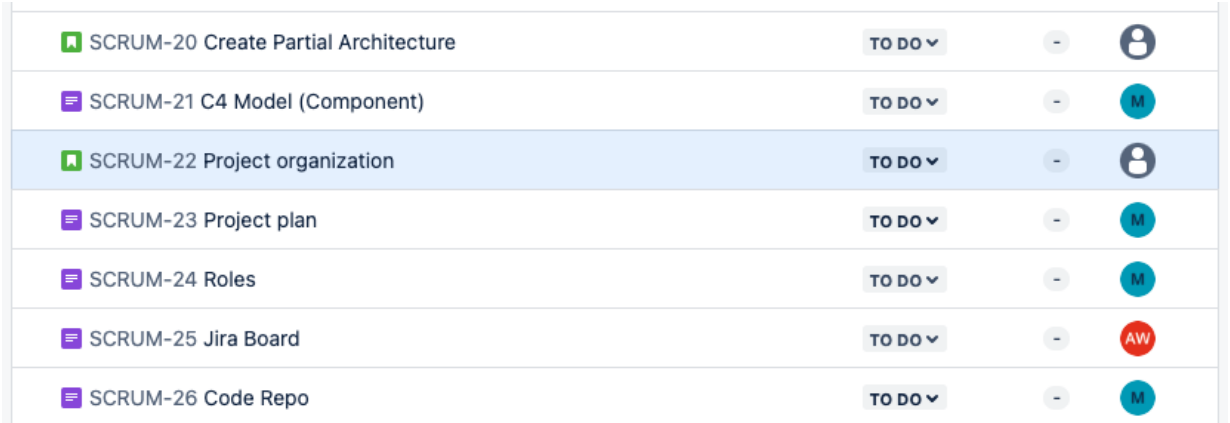

Figure 8.4: User Story

Task

A Task is an indivisible unit of work that should be completed within a few hours and is part of a User Story. It should be self-contained and can be tested using unit testing. External dependencies should be avoided.

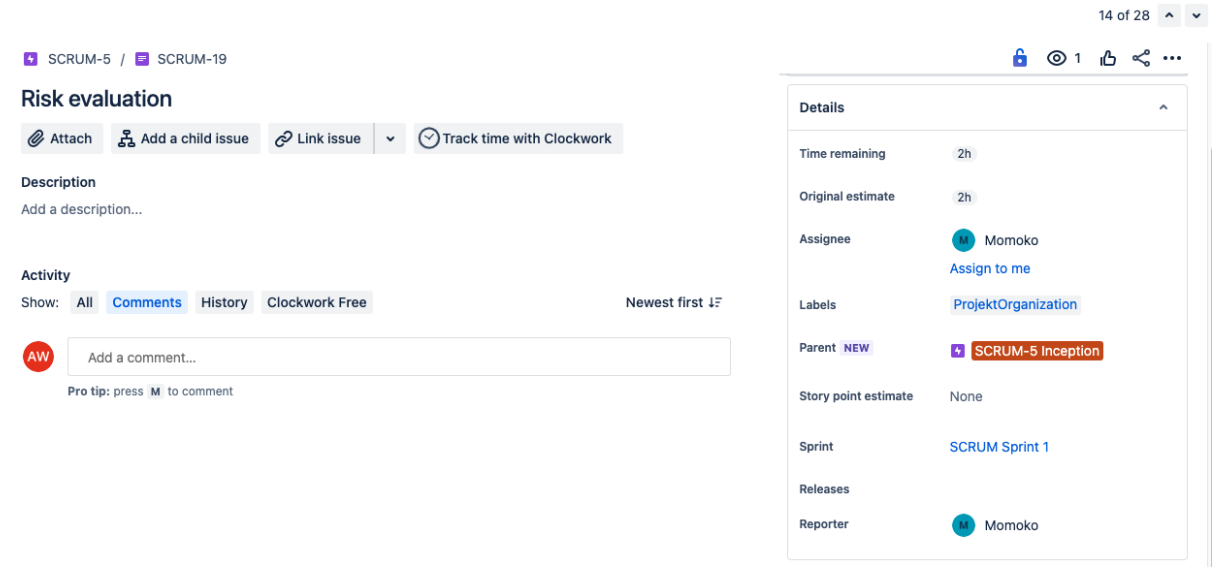

Figure 8.5: Tasks

Meeting

Meetings is about organizing and documenting our meeting time.

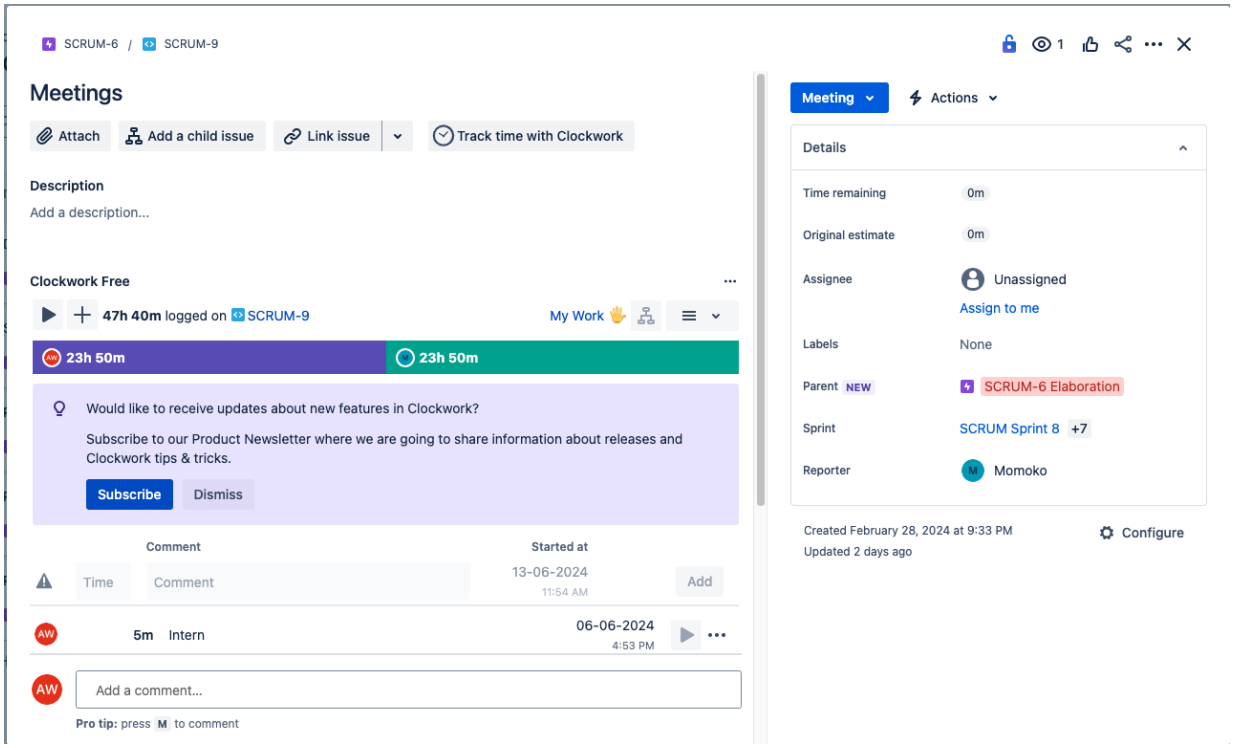

Figure 8.6: Meeting

8.2.5 Branching Strategy

We decided on a very simple branching strategy. We have the following four branches:

- **Main Branch:** Represents the stable, production-ready code. It contains the latest released version of the software.
- **Development Branch:** The main integration branch, where ongoing development work is merged. It's the collective work of all the feature branches.
- **Feature Branches:** Each feature or user story has its own branch, which branches off from the development branch. These branches encapsulate changes related to specific features, and are merged back into the development branch when complete.

In the case of a hotfix, we temporarly create a hotfix branch and after successfully fixing the bug we merge it into the main branch.

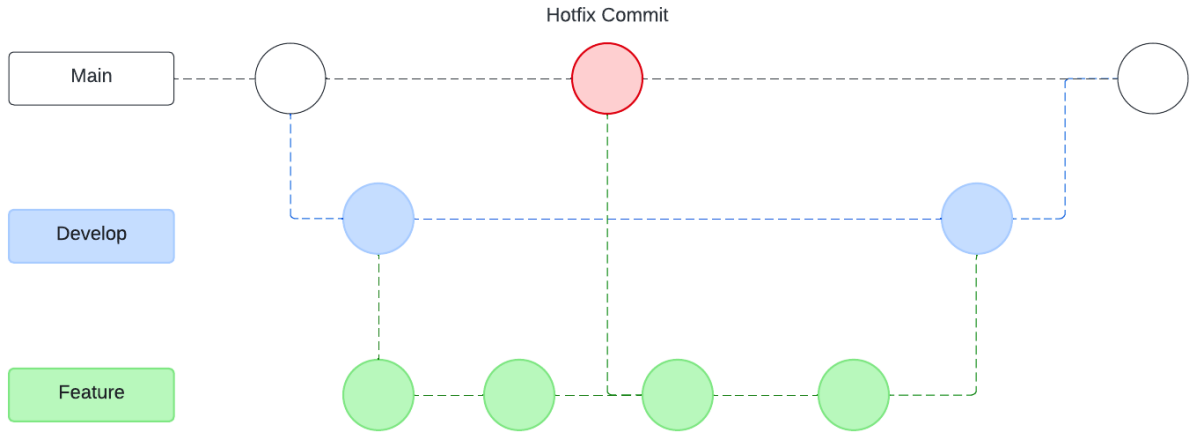

Figure 8.7: Git Branch

8.2.6 Defintion of Done

This list will serve as a checklist when we want to verify that something is done. It will also ensure that quality standards are met with a clearly defined process.

Sprint

- DoD of each task is met
- Sprint Scope completed
- All tests pass
- Application deployed

User Stories

- Project builds without errors
- Tests written according to test concept
- Code written according to defined guidelines
- NFR stay unviolated
- Assumptions of User Story met (justified if not)
- Configuration, Architecture or Build changes documented

Tasks

- Changes to Documentation reviewed
- Code reviewed by the other member
- Time Spent on Tasks documented
- Decisions accepted by Team
- Assumptions of Task met (justified if not)

8.3 Risk Management

Table 8.2 shows the risks that were identified during the inception phase. The risk matrix was utilized to evaluate the identified risks.

No.	Risk	Mitigation	Probability	Severity	Exposure
1	Technical complexity when implementing external services and integrating it with other systems.	Conduct a thorough analysis and evaluation of the LLMs in advance to select the most suitable model. Also conduct a POC to verify the feasibility of the integration before the development.	Possible	Critical	High
2	Team member is unavailable, due to illness or other reasons.	Communicate through channels to distribute tasks between other team members.	Possible	Moderate	Medium
3	Delays in development due to unforeseen complexities or changes in project requirements.	Include buffer times in the project schedule and use Agile project management to accommodate changes and feedback.	Likely	High	High
4	Inadequate handling of PDF data leading to data corruption or loss.	Implement rigorous testing phases, including unit tests, integration tests, and end-to-end tests to ensure data integrity and handling are maintained throughout.	Possible	High	High
5	Security vulnerabilities in the application, particularly in user data handling and document storage.	Adopt a security-first design philosophy, conduct regular security audits, and ensure encryption of sensitive data both in transit and at rest.	Likely	Critical	High

No.	Risk	Mitigation	Probability	Severity	Exposure
6	Difficulty in achieving the non-functional requirements related to system performance.	Use performance engineering practices from the start of the project, including performance testing and optimization activities.	Possible	High	Medium
7	Insufficient scalability to handle increased number of documents or users as the organization grows.	Design the system architecture for scalability from the outset, consider cloud solutions that scale automatically, and perform load testing.	Unlikely	High	Medium

Table 8.2: Initially identified risks

Cost Analysis

The following table presents the cost analysis of the identified risks. The cost of the measures is calculated in hours. The calculated damage is rounded up to half an hour.

No.	Cost of Measure (Hours)	Max Damage (Hours)	Probability (%)	Weighted Damage (Hours)	Priority
1	20	30	10	3	High
2	10	15	10	1.5	Medium
3	15	25	20	5	High
4	18	20	10	2	High
5	25	30	20	6	High
6	12	20	10	2	Medium
7	8	20	5	1	Medium

Table 8.3: Cost Analysis Based on Risk Assessment

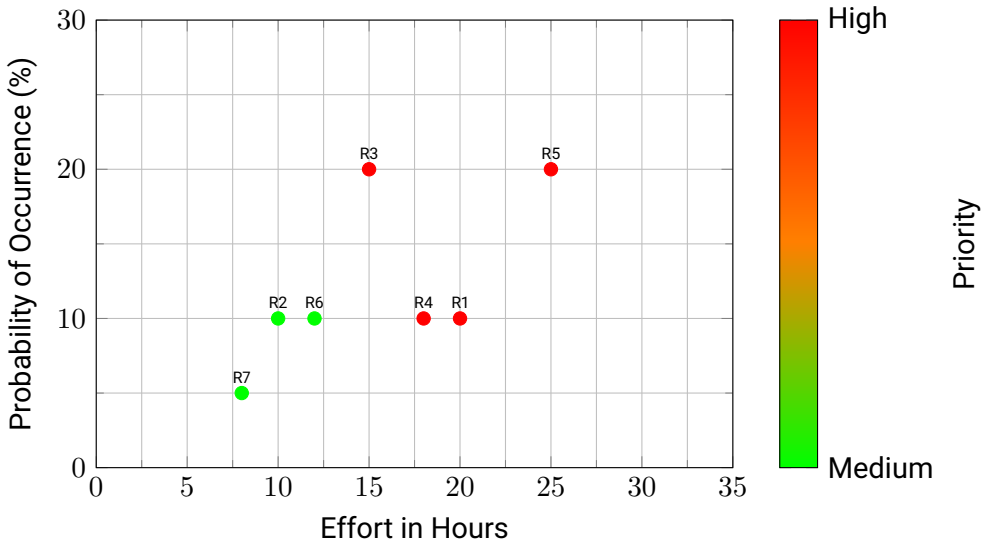

Figure 8.8: Updated Risk Matrix based on the Priority of Identified Risks from Cost Analysis.

Technical complexity when implementing external Services and integrating it with other systems

During the development phase, the complexity of integrating external services became evident, particularly while constructing the prototype. Although we conducted a thorough analysis for the LLM in advance, integrating Transformers.js proved to be more challenging than anticipated.
Integrating LLMs into our software required various strategic approaches. We chose to fine-tune a model using the transformers library, compatible with frameworks such as PyTorch, TensorFlow, and JAX. The specific challenge was to integrate this capability within a React and Node.js environment, leading us to the Transformers.js library. Although designed for browser use, Transformers.js also supports server-side inference on Node.js. The intention was to implement this during the build phase, but significant hurdles were encountered, including a limited choice of models and slow response times.
To address these issues, we implemented a web worker following a tutorial from the same library. This solution allowed the model to load in the background, preventing the user interface from being blocked. However, it introduced a critical problem: it delayed key user interactions, particularly during the login and sign-up processes. The model loaded immediately upon application startup, resulting in a responsive application only after the initial loading period, which was excessively long.
Given these complexities, we consulted with our industry partner to reassess our approach: whether to continue fine-tuning the model in-house or switch to an external API offering a wider choice of models. In-house fine-tuning required significant computing resources and time, which were beyond our current capacity and project scope. These discussions underscored the need to balance optimizing system performance with incorporating advanced functionality, complicating the development process. Ultimately, our partner agreed to use an external API.
The risk was mitigated by changing the LLM technology to the Inference API instead of the Transformers.js, which was accepted by our industry partner.
Actual Severity: Moderate
Countermeasure success: It was partial successful. We ensured at the beginning that we were trying to do something possible. However, the problem was really visible at the implementation, but thankfully it was at the time for the prototype, which gave us enough time to change the LLM technology.

Team member is unavailable, due to illness or other reasons.

The risk was realized during the construction phase. Momoko Wymann was sick for five days. The risk was mitigated by Andrew Willi taking over the task of Momoko Wymann.
Actual Severity: Moderate
Countermeasure success: It was successful

8.4 Time Management

Framework Condition

This project is part of the bachelor thesis which is required for the eligibility of the bachelor. The planned time budget for this project is 360 hours per person and equals to 12 ECTS.

Resources

The project is expected to last for 16 weeks, with a planned effort of 360 hours per person. Of this, 45 hours are completed in a block week at the end of the project. This means that each person has an average of approx. 22.5 hours per week available to contribute to the project.

Time evaluation

The project is divided into 8 sprints, each lasting 2 weeks, except for Sprint 4, which is 3 weeks long due to a holiday from 08.04.2024 to 12.04.2024. The sprints are as follows:

- Sprint 1: 19.02.2024-03.03.2024
- Sprint 2: 04.03.2024-17.03.2024
- Sprint 3: 18.03.2024-31.03.2024
- Sprint 4: 01.04.2024-21.04.2024
- Sprint 5: 22.04.2024-05.05.2024
- Sprint 6: 06.05.2024-19.05.2024
- Sprint 7: 20.05.2024-02.06.2024
- Sprint 8: 03.06.2024-14.06.2024

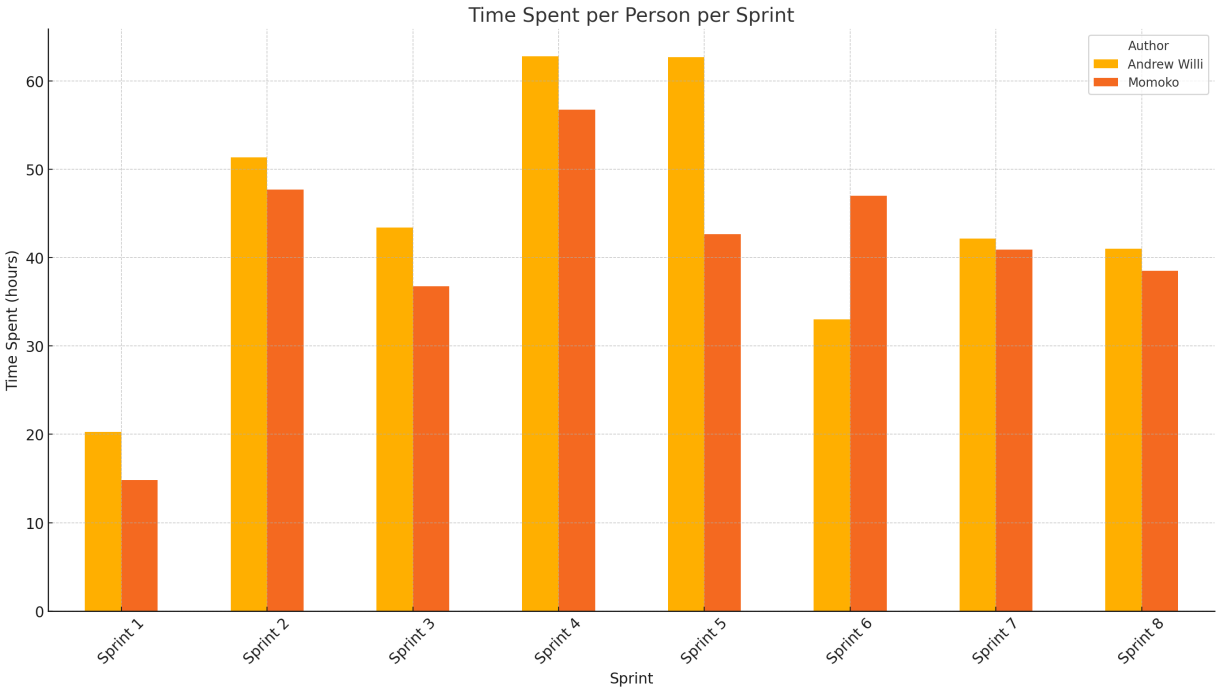

Figure 8.9: Time Spent per Person

The total workload per team member is as follows:

- Andrew Willi: 367.08 hours
- Momoko Wymann: 370.08 hours

The pie chart below displays the distribution of the workload between the most important areas:

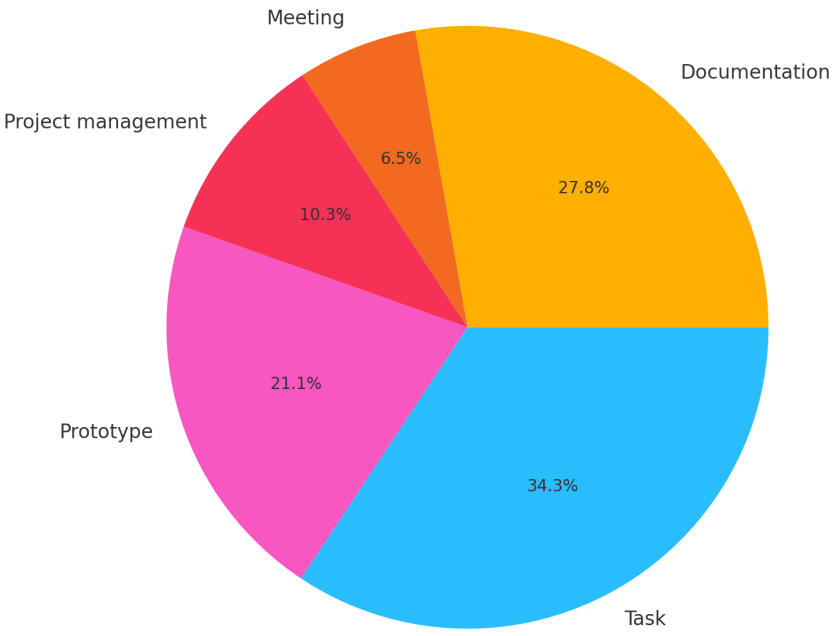

Figure 8.10: Pie Chart Distribution of Working Hours per Area

In this bar chart, the workload per sprint is displayed to showcase in which sprint the the most important area was implemented.

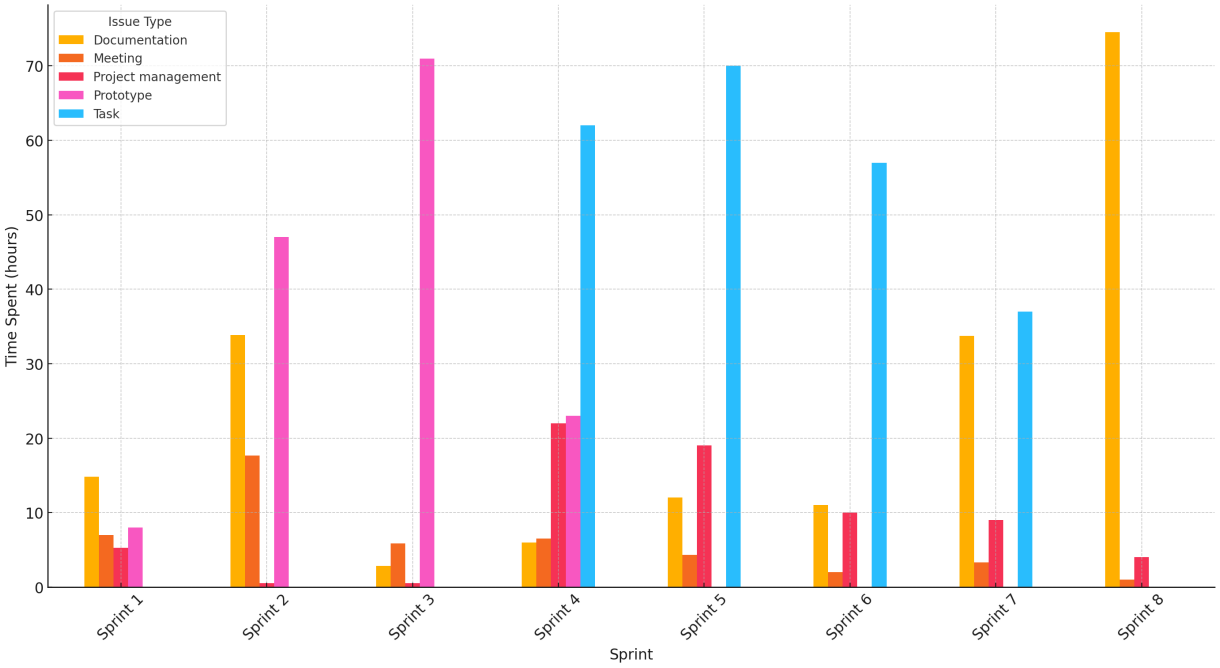

Figure 8.11: Issues per Sprint

List of Figures

List of Tables

Listings

9 Appendix

9.1 API Documentation

The backend APIs can be inspected with swagger at the backend-url on path /api.

9.1.1 Auth Controller

Login

Endpoint: /auth/login
This endpoint is used to authenticate a user by providing their email and password.

Request Body

- **email (string, required):** The email of the user.
- **password (string, required):** The password of the user.

Response The response contains an access and a refresh token which can be used for further authenticated requests.

- **access_token (string):** The access token for authenticated requests.
- **refresh_token (string):** The refresh token to obtain a new access token.
- **username (string):** The username of the autheticated user.
- **role (string):** The role of the autheticated user. Can be 'user' or 'admin'.

Logout

Endpoint: /auth/logout

This endpoint is used to log out the authenticated user. For authorization, the access token is required in the request header in form of a bearer token.

Request Body There is no request body for this request.

Response The response for this request is expected to be null.

Register

Endpoint: /auth/register

This endpoint allows the client to register a new user.

Request Body

- **username (string, required):** The username of the user.
- **email (string, required):** The email of the user.
- **password (string, required):** The password of the user.

Response The response contains an access and a refresh token which can be used for further authenticated requests.

- **access_token (string):** The access token for authenticated requests.
- **refresh_token (string):** The refresh token to obtain a new access token.

Refresh

Endpoint: /auth/refresh

This endpoint sends an HTTP GET request to refresh the authentication token. For authorization, the refresh token is required in the request header in form of a bearer token.

Request Body There is no request body for this endpoint.

Response The response contains an access and a refresh token which can be used for further authenticated requests.

- **access_token (string):** The access token for authenticated requests.
- **refresh_token (string):** The refresh token to obtain a new access token.

Test Auth

Endpoint: /auth/test

This endpoint is used to test the authentication. For authorization, the access token is required in the request header in form of a bearer token.

Request Body There is no request body for this endpoint.

Response The response contains information about the authorized user and the access token.

- **sub (number):** The user id of the user.
- **username (string):** The username of the user.
- **iat (unix timestamp):** The issued at field indicates the time at which the token was created.
- **exp (unix timestamp):** The expired time field indicates the time at which the token expires and therefore becomes invalid.

9.1.2 Document Controller

For authorization, the access token is required in the request header in form of a bearer token for all the endpoints.

Upload

Endpoint: /files/upload

The request should include a form-data body type with the files to be uploaded. It supports the upload of multiple files at once.

Request Body

- **file (PDF-file, required):** The file(s) to be uploaded.

Response The response indicates if the upload of each document was successul or not. The response is an array with the following properties for each document:

- **success (boolean):** Indicates whether the file upload was successful.
- **message (string):** Provides any additional information or error message.
- **filename (string):** The name of the uploaded file.

Download

Endpoint: /files/download/:documentId

This endpoint retrieves a file for download based on the provided id.

Request Parameters No request body parameters are required for this request, but the document id has to be provided as path parameter.

- **documentId (number, required):** The id of the file to be downloaded.

Response The response for a successful request is a PDF file with a status code of 200.

- **status code (number):** Status code of the response.
- **message (string):** Provides any additional information or error message.

Delete

Endpoint: /files/delete/:documentId

This endpoint is used to delete a file based on the provided document id.

Request Parameters No request body parameters are required for this request, but the document id has to be provided as path parameter.

- **documentId (number):** The id of the file to be deleted.

Response The response indicates if the deletion of the document was successful.

- **status code (number):** Status code of the response if the request fails.
- **message (string):** Provides any additional information or error message.

Search Controller

For authorization, the access token is required in the request header in form of a bearer token for all the endpoints.

Search

Endpoint: /search/search

This endpoint is used to search for documents based on tags, text, and time range.

Request Parameters

- **tags (string, optional):** The tags to search for.
- **text (string, optional):** The text to search for.
- **timeRange (string, optional):** The time range for the search. Valid values are 'Past hour', 'Past 24 hours', 'Past week', 'Past month', 'Past year'.

Response The response contains an array of documents that match the search criteria.

- **documents (array of DocumentDto):** An array with the metadata of documents that match the search criteria.

Search Recent

Endpoint: /search/searchRecent

This endpoint is used to search for documents, recently uploaded by the authenticated user, based on tags, text, and time range.

Request Parameters

- **tags (string, optional):** The tags to search for.
- **text (string, optional):** The text to search for.
- **timeRange (string, optional):** The time range for the search. Valid values are 'Past hour', 'Past 24 hours', 'Past week', 'Past month', 'Past year'.

Response The response contains an array of recent documents that match the search criteria.

- **documents (array of DocumentDto):** An array with the metadata of recent documents that match the search criteria.

Get Favorite Documents

Endpoint: /search/favorites

This endpoint is used to retrieve the favorite documents of the authenticated user.

Request Parameters There are no request parameters for this endpoint.

Response The response contains an array of the user's favorite documents.

- **documents (array of DocumentDto):** An array with the metadata of the user's favorite documents.

Get Recent Documents

Endpoint: /search/recent

This endpoint is used to retrieve the recent documents of the authenticated user.

Request Parameters There are no request parameters for this endpoint.

Response The response contains an array of the user's recently uploaded documents.

- **documents (array of DocumentDto):** An array with the metadata of the user's recent documents.

9.1.3 Tags Controller

For authorization, the access token is required in the request header in form of a bearer token for all the endpoints.

Get All Tags

Endpoint: /tags/all

This endpoint is used to retrieve all tags available in the system.

Request Parameters There are no request parameters for this endpoint.

Response The response contains an array of all tags.

- **tags (array):** An array of all tags.

9.1.4 Favorites Controller

For authorization, the access token is required in the request header in form of a bearer token for all the endpoints.

Add Favorite

Endpoint: /favorites/:documentId

This endpoint is used to add a document to the user's favorites.

Request Parameters

- **documentId (number, required):** The id of the document to be added to favorites.

Response

- **message (string):** A message indicating the document has been added to favorites.

Remove Favorite

Endpoint: /favorites/:documentId

This endpoint is used to remove a document from the user's favorites.

Request Parameters

- **documentId (number, required):** The id of the document to be removed from favorites.

Response

- **message (string):** A message indicating the document has been removed from favorites.

Get All Favorites by User

Endpoint: /favorites/allByUser

This endpoint is used to retrieve all favorite documents for the authenticated user.

Request Parameters There are no request parameters for this endpoint.

Response

- **documents (array of DocumentDto):** An array containing all metadatas of the favorite documents of the user.

9.1.5 Log Controller

For authorization, the access token is required in the request header in form of a bearer token for all the endpoints.

Log Error

Endpoint: /log

This endpoint is used to log an error message from the frontend.

Request Body

- **errorMessage (string, required):** The error message to be logged.

Response

- **status (string):** The status of the log operation, indicating that the log has been saved.

Bad Responses

If an inquiry to the backend is not successful, an HTTP Exception is returned in general with the following content.

- **statusCode (number):** The HTTP status code that indicates the type of error.
- **error (string):** A brief description of the error. This typically corresponds to the HTTP status text, such as 'Bad Request,' 'Unauthorized,' 'Forbidden,' 'Not Found,' or 'Internal Server Error.'
- **message (string):** A more detailed message explaining the error. This can provide additional context or specific details about why the error occurred.

9.2 Screenshots

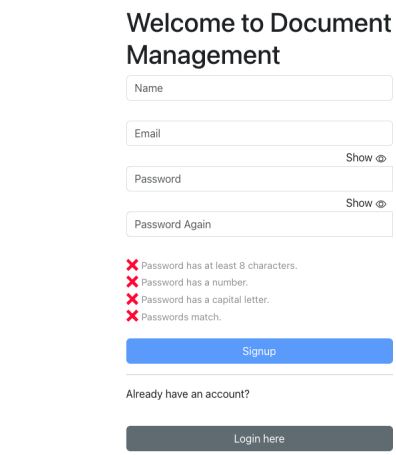

Figure 9.1: Register Page

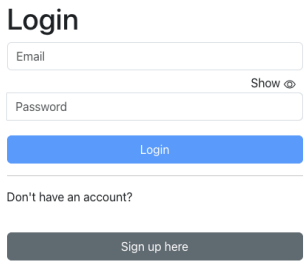

Figure 9.2: Login Page

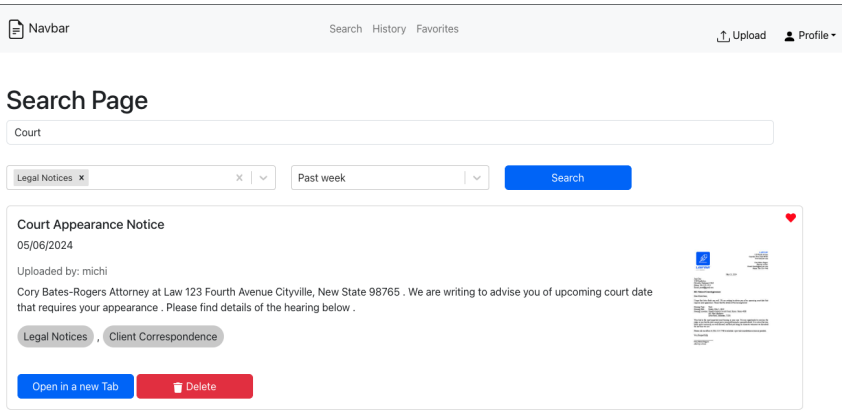

Figure 9.3: Search Page

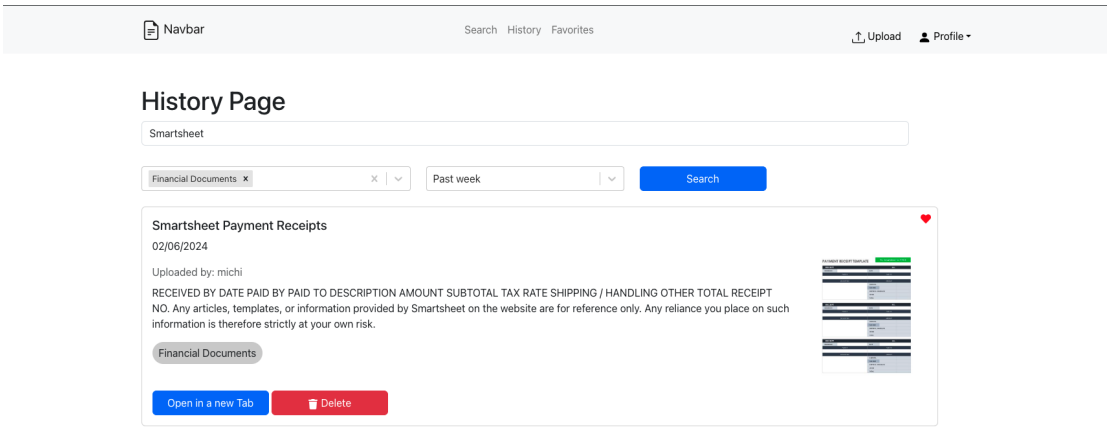

Figure 9.4: History Page

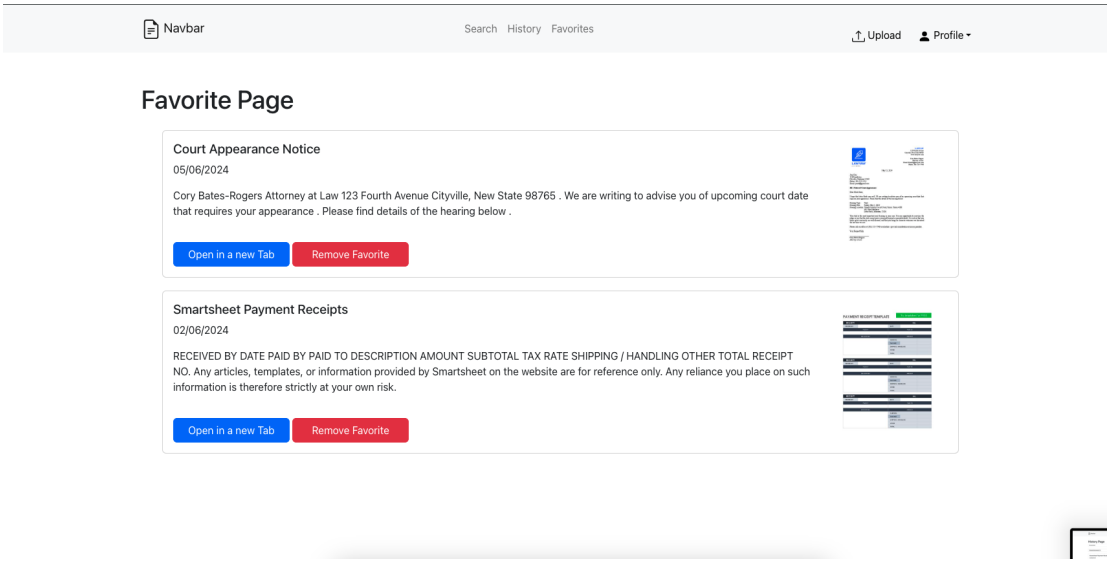

Figure 9.5: Favorite Page

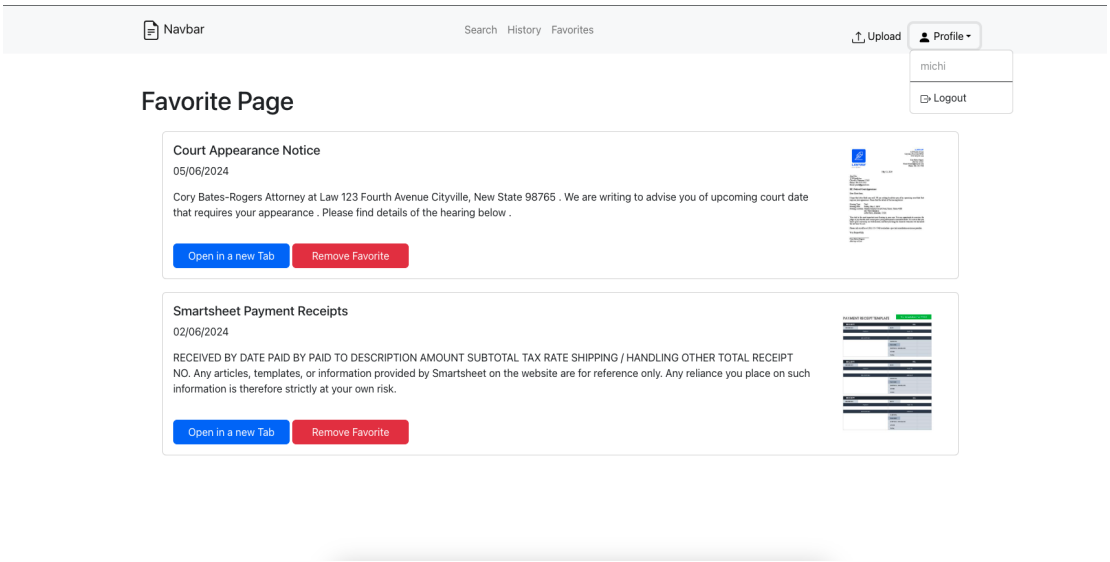

Figure 9.6: Profile View

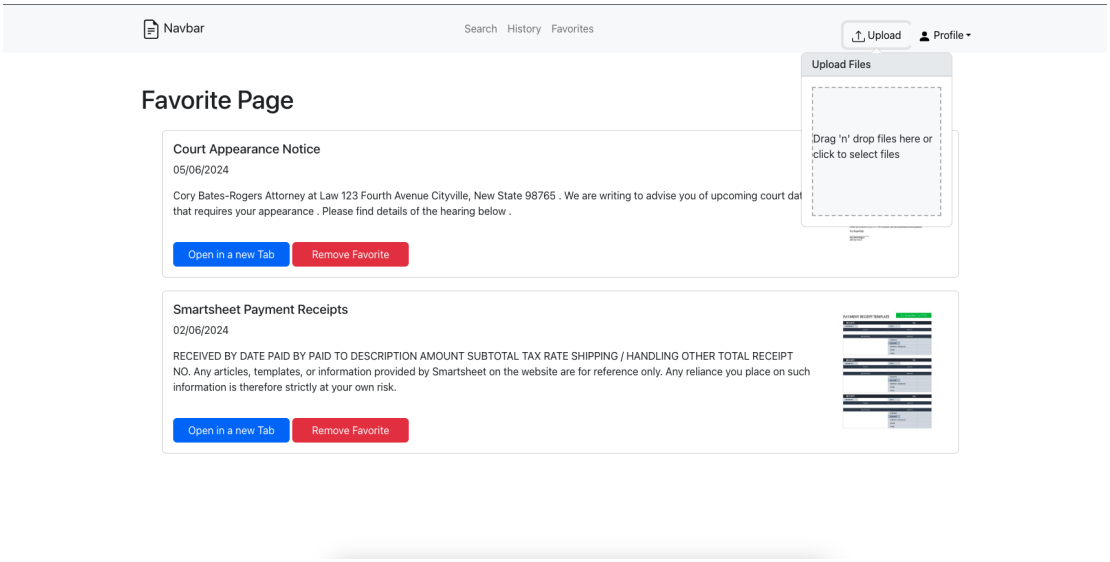

Figure 9.7: Upload Zone

9.3 Task Definition

Document Management using Large Language Model

1. Beteiligte Personen

- Studierende: Momoko Wymann, Andrew Willi
- Industriepartner: AdaptIT GmbH, Michael Güntensperger
- Experte: Hansjörg Huser
- Gegenleserin: Mitra Purandare
- Betreuer: Frank Koch

2. Problembeschrieb

Gescannte Dokumente werden meist als PDFs gespeichert. Eine Suche ist lediglich auf Basis des Doku- mentennamens möglich. Mit Hilfe eines LLM soll ein neuer Ansatz für die Suche und Verwaltung von Doku- menten gefunden werden.

3. Aufgabenstellung

Die zu entwickelnde Applikation soll bei einem betriebswirtschaftlichen Hintergrund als PDF gespeicherte Dokumente mit unstrukturierten Daten (z.B. Kundenbriefe, Rechnungen, . . .) einlesen und mittels eines LLM (z.B. auf Hugging Face) aus den Inhalten Metadaten generiert (z.B. Zusammenfassungen, Tags, Katego-rien). Beispiele für Kategorien wären Rechnung, Vertrag, ... Die Metadaten werden in einer Datenbank abgelegt, die Dokumente selbst in einen Object Store hochgeladen.

Auf einem Frontend soll eine auf den Metadaten basierende Suche bzw Navigation angeboten werden um auf die Dokumente zuzugreifen.

Technische Umgebung

- Frontend: React / Angular
- Backend: Node.js
- Datenbank: MySQL
- Object Store für PDF's
- Hoster: Z.B. DigitalOcean

Funktionale Anforderungen

- User-Login
- Automatisiertes Hochladen von Dokumenten
- Dokumente auslesen und in DB speichern
- Suche nach Dokumenten auf Basis von Metadaten erlauben
- Inhalt der Dokumente mithilfe eines LLMs zusammenfassen

- Aufgrund der generierten Kategorien sollen nach dem Hochladen via API-Call auf einen n8n- oder Zapier-Endpoint automatisierte Prozesse angestossen werden können, wie z.B. eine Notification an die Buchhaltung beim Erhalt einer Zahlungserinnerung.

Optionale Anforderungen

- Dokumente Kategorisieren / Tagen z.B. als Rechnung
- Dashboard für User mit wichtigsten Kennzahlen (z.B. meist angesehene PDF's)
- Aufgrund der generierten Kategorien sollen nach dem Hochladen via API-Call auf einen n8n- oder Zapier-Endpoint automatisierte Prozesse angestoßen werden können, wie z.B. eine Notification an die Buchhaltung beim Erhalt einer Zahlungserinnerung.
- Manuelles Bearbeiten von Metadaten ermöglichen
- Usermanagement für die Verwaltung der Mitarbeitenden einer Organisation

Nicht-Funktionale Anforderungen

- Das Entwicklerteam implementiert die Features gemäß den mit dem Kunden vereinbarten Prioritäten.
- Das Backend sollte 1'000 Requests pro Minute verarbeiten können.
- Jede Seite sollte nicht länger als 200ms für das Laden benötigen.
- Die Seite soll mindestens auf dem Desktop gut aussehen (Responsive wäre wünschenswert).
- Die Web-Applikation sollte auf Firefox, Chrome und Safari laufen.
- Via Internet sollte auf eine vom Kunden zur Verfügung gestellte Domain zugegriffen werden können.
- Drei von vier Test-Usern sollten das UI (Kategorien: Layout, Responsiveness, Colour, Content) der Applikation mit einem PC mit einer Note von mindestens 8 von 10 bewerten, mit 10 als bester Note.
- Nach dem Einscannen des Dokuments soll es nicht länger als 30 Sekunden dauern, bis dieses auf der Seite angezeigt wird.
- Die Datenbank soll bis zu 10'000 Dokumente und 100 Benutzende managen können.
- Errors sollen keine Systemfehler erzeugen, stattdessen eine Fehlermeldung anzeigen und das System auf den vorherigen Zustand zurücksetzen.
- Jeder Error soll im System geloggt werden.
- Jede Kommunikation zwischen Front- und Backend soll mit einem SSL-Zertifikat verschlüsselt werden.
- Daten, die in Eingabefelder abgefüllt werden, sollen zuerst validiert werden, bevor diese durch das System verarbeitet werden. SQL Injection Tests der Eingabefelder dürfen keine Verletzlichkeiten zeigen.
- Die Webapplikation soll Datenschutz-konform umgesetzt werden.
- User-Passwörter werden nicht in Plain-Text in der Datenbank gespeichert.

- Wenn sich ein User in die Web-Applikation einloggt, werden ihm nur Daten angezeigt, auf die er Zugriff haben soll.
- Businesslogik im Backend soll modular aufgebaut werden, so dass sie erweitert werden kann.
- Das Backend-API soll durch ein API-Testing Tool überprüft werden.
- Implementierte Funktionalitäten (Datenbank, Backend, Frontend,...) sollen deployed werden.

4. Zur Durchführung

Mit dem Betreuer finden Besprechungen gemäss Absprache statt. Die Besprechungen sind von den Studie-renden mit einer Traktandenliste vorzubereiten und die Ergebnisse in einem Protokoll zu dokumentieren. Die Kommunikation mit dem Betreuer findet primär über E-Mail statt.

Für die Durchführung der Arbeit ist ein Projektplan zu erstellen. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten. Abweichungen vom Projektplan sind rechtzeitig mit dem Betreuer zu besprechen.

5. Dokumentation und Abgabe

Siehe «Leitfaden für Bachelor- und Studienarbeiten Version 1.2» Abschnitt 5.5 "Umfang und Form der Abgabe".

6. Termine

Siehe veröffentlichte «Termine BA FS24».

7. Bewertung

Siehe «Leitfaden für Bachelor- und Studienarbeiten Version 1.2» Abschnitt 6 "Bewertung". Rapperswil, den 12.02.24

Frank Koch

9.4 Usability Test Protocol

Participant 1:

- **Name:** Lara
- **Test date:** 13.05.2024
- **Background:** A 26 year old teacher with moderate Tech-Savviness and has to manage a lot of documents for her job. She uses mainly Apple devices and uses the Finder application to manage her documents. If it needs to be on the cloud she uses Google Drive or the iCloud.

Ratings:

- **General satisfaction rating (scale 1-10):** 8.25
- **Layout and design:** 9
- **Responsiveness rating:** 7.5
- **Colour scheme thoughts:** 8.5
- **Clarity of instructions and content:** 8

Negative Feedback:
After uploading and clicking somewhere, the loading circle was gone. So it was bit confusing, if the upload was successful or not.

Positive Feedback:
Very simple and understandable application. The color scheme is very pleasant and the instructions are very clear.

Additional Comments or Recommendations:
She mentioned that instead of the heart, maybe a button similar to the "Open in new Tab" Tab would be better.

Conclusion:
Lara was happy with the application. She was able to upload her documents and she liked the drag and drop feature.

Participant 2:

1. **Name:** Jaqueline
2. **Test date:** 15.05.2024
3. **Background:** Jaqueline is a 55 year old Photoghrapher/Graphic Designer with moderate tech-savviness, Jaqueline relies heavily on technology uses mainly Apple devices for various design software.

Ratings:

1. **General satisfaction rating (scale 1-10):** 8
2. **Layout and design:** 8.5
3. **Responsiveness rating:** 8

4. **Colour scheme thoughts:** 8
5. **Clarity of instructions and content:** 7.5

Negative Feedback:
No negative Feedback.

Positive Feedback:
Overall design is nice and neat. The color scheme is ok.

Additional Comments or Recommendations:
The purpose of History was firstly not clear for her.

Conclusion: She was happy to use the application

Participant 3:

1. **Name:** Alex
2. **Test date:** 15.05.2024
3. **Background:** Alex is a 26 year old computer science Student with high tech-savviness. He uses Windows and Apple devices.

Ratings:

1. **General satisfaction rating (scale 1-10):** 7
2. **Layout and design:** 8
3. **Responsiveness rating:** 7
4. **Colour scheme thoughts:** 8
5. **Clarity of instructions and content:** 6

Negative Feedback:
Searching should be possible without explicitly choosing the tag. Alex tried to upload another PDF document, to see what would happen and the loading time was a bit longer than anticipitated.

Positive Feedback:
Overall design looks solid.

Additional Comments or Recommendations:
The History could have been a checkbox to search through only his uploaded files instead of a history component. It would be nice to have some kind of function to display the latest uploaded files.

Conclusion: There are some improvements to be done but mainly the application is fine.

Participant 4:

1. **Name:** Florian
2. **Test date:** 15.05.2024

3. **Background:** Florian is a 32-year-old IT architect who is heavily involved in backend development and is currently studying for his PhD. He mainly uses Linux-based systems and is familiar with various software development environments.

Ratings:

1. **General satisfaction rating (scale 1-10):** 8
2. **Layout and design:** 9
3. **Responsiveness rating:** 7
4. **Colour scheme thoughts:** 8
5. **Clarity of instructions and content:** 8.5

Negative Feedback:
Florian mentioned that while the interface is mostly intuitive, some elements like the document tagging system felt overly complicated. He noted that the options could be overwhelming for less tech-savvy users without proper tooltips or guidance.

Positive Feedback:
Florian praised the application for its robust performance, particularly appreciating the seamless integration with external APIs which facilitated easy document handling and retrieval. He also highlighted the application's responsive design, which scaled well across different screen sizes and resolutions.

Additional Comments or Recommendations:
Florian recommended integrating more keyboard shortcuts to enhance productivity for power users. He also suggested that the application could benefit from a more dynamic help system that adapts to the user's actions and offers context-specific assistance.

Conclusion:
He looks forward to seeing how additional features like AI-based document sorting could further enhance the user experienc

9.5 Testing Backend

Integration Tests

All integration test of the backend are listed below.

AuthService

login

- **Successful login:** Checks whether tokens and user information are returned on successful login
- **Invalid login:** Checks whether an UnauthorizedException is thrown if the login data is invalid.

createUser

- **create new user:** Checks whether a new user is created and tokens are returned.
- **Email already registered:** Checks whether an HttpException is thrown if the email is already registered.

verifyToken

- **valid token:** Checks whether true is returned if the token is valid.
- **Invalid token:** Checks whether false is returned if the token is invalid.

AuthController

signIn

- **Protection against SQL injection attacks in the password:** Checks whether the application is protected against SQL injection attacks in the password field.
- **Protection against SQL injection attacks in the email:** Checks whether the application is protected against SQL injection attacks in the email field.

DocumentService

The text and thumbnail extractions as well as the S3 interactions are mocked. Some services and the datasource are also mocked.

uploadFile

- **Successful upload of a file:** Checks whether a file is successfully uploaded and the document is saved.
- **Error message for existing document:** Checks whether an error message is returned if the document already exists.
- **Rollback on failed S3 upload:** Checks whether the transaction is rolled back if the upload to the S3 object store fails.

deleteDocument

- **Successful deletion of a document:** Checks whether a document is successfully deleted.
- **Error message if administrator rights are missing:** Checks whether an error message is returned if the user does not have administrator rights.
- **Error message if document does not exist:** Checks whether an error message is returned if the document to be deleted does not exist.
- **Rollback on failed deletion:** Checks whether the transaction is rolled back if the delete operation in the database fails.

SearchService

searchDocuments

- **Empty result list if no match:** Checks whether an empty result list is returned if no documents match.
- **Result list with matching tags:** Checks whether the search returns documents with matching tags
- **Result list without tags:** Checks whether the search returns all documents if no tags are specified.
- **Better match with more precise tag match:** Checks whether documents with better tag matches are returned first.
- **Better match with more accurate text match:** Checks whether documents with better text matches are returned first.
- **Documents with all matching tags:** Checks whether only documents containing all specified tags are returned

getFavorites

- **Empty result list with no favorites:** Checks whether an empty result list is returned if there are no favorites.
- **Result list with favorites:** Checks whether a list of favorites is returned.

getRecent

- **Empty result list if no recently viewed documents:** Checks whether an empty result list is returned if there are no recently viewed documents.
- **Result list with recently viewed documents:** Checks whether a list of recently viewed documents is returned.

SearchController

search

- **Protection against SQL injection attacks in tags:** Checks whether the application is protected against SQL injection attacks in the tags field.

- **Protection against SQL injection attacks in text:** Checks whether the application is protected against SQL injection attacks in the text field.
- **Protection against SQL injection attacks in the time range:** Checks whether the application is protected against SQL injection attacks in the time range field.

searchRecent

- **protection against SQL injection attacks in tags:** Checks whether the application is protected against SQL injection attacks in the tags field.
- **Protection against SQL injection attacks in text:** Checks whether the application is protected against SQL injection attacks in the text field.
- **Protection against SQL injection attacks in the time range:** Checks whether the application is protected against SQL injection attacks in the time range field.

FavoritesService

isFavorite

- **Check whether document is a favorite:** Checks whether a document is marked as a favorite.

addFavorite and removeFavorite

- **addFavorite:** Checks whether a document is successfully added as a favorite.
- **remove favorite:** Checks whether a document is successfully removed as a favorite.

TagService

findAll

- **Return of all tags:** Checks whether all tags are returned.

creatTagIfNotExist

- **create new tag:** Checks whether a new tag is created if it does not yet exist.
- **Return existing tag:** Checks whether an existing tag is returned if it already exists.

findTagsByDocumentId

- **return tags for a document:** Checks whether the tags for a given document are returned.
- **Empty array for missing tags:** Checks whether an empty array is returned if no tags are found.

UserService

findByEmail

- **find user by email:** Checks whether a user can be found by their email address.

- **Return null if user does not exist:** Checks whether null is returned if no user with the given email is found

findById

- **find user by ID:** Checks whether a user can be found by their ID.
- **Return null if user does not exist:** Checks whether null is returned if no user with the given ID is found.

createUser

- **create new user:** Checks whether a new user is created.

updateToken

- **Updating the user refresh token:** Checks whether the refresh token of a user is updated.

clearRefreshToken

- **Deletion of the user refresh token:** Checks whether the refresh token of a user is deleted.

API Tests

The Collection of the API tests in Postman are located in the documentation folder in the file: Backend api.postman_collection.