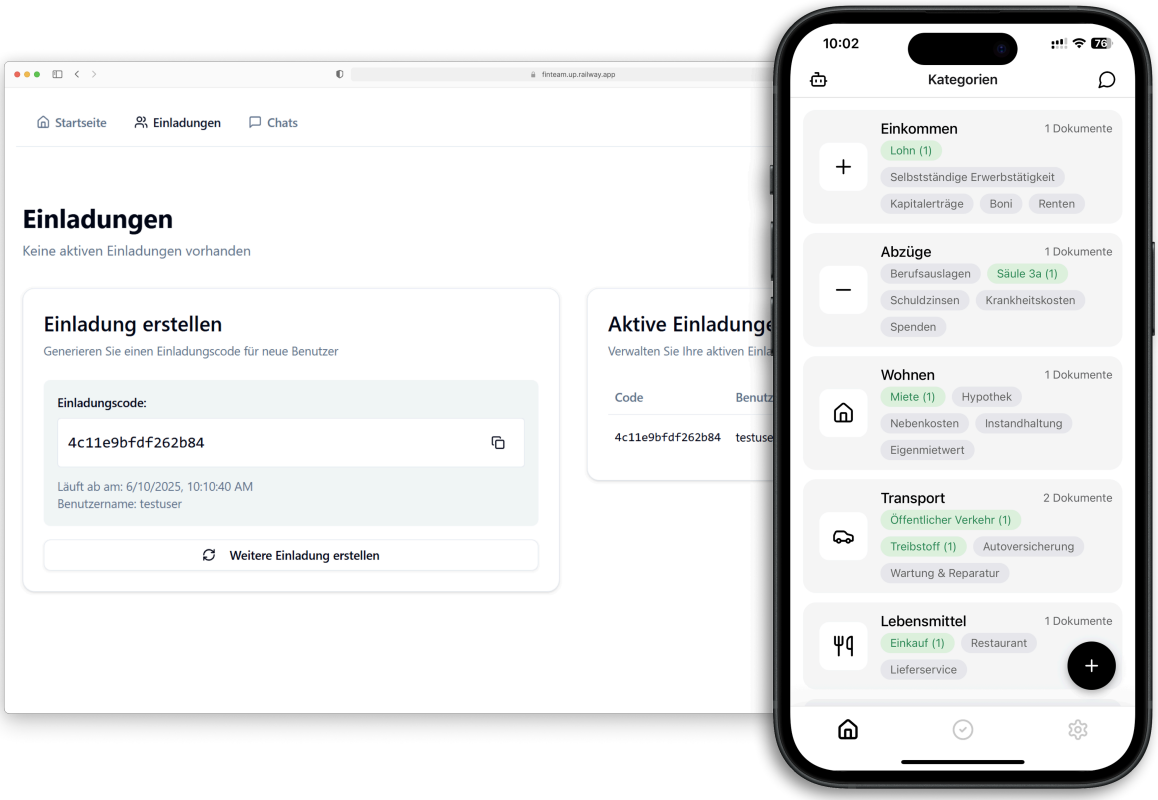
Bachelor thesis
Documentation

# Finteam - A React Native App for Fiduciaries and their clients

Semester: Spring 2025




Version: 1.0
Date: 19-06-2025


| | |
|---|---|
| Project Team: | Noah Fleischmann (noah.fleischmann@ost.ch) |
| | Dominik Rüegg (dominik.rueegg@ost.ch) |
| Project Advisor: | Dr. Marco Lehmann (marco.lehmann@ost.ch) |
| External Expert: | Kirusanth Poopalasingam (kiru@veemg.com) |

School of Computer Science
OST Eastern Switzerland University of Applied Sciences

# Abstract


**Introduction**

In Switzerland, tax filing is often perceived as a cumbersome process, which leads many individuals to delegate this responsibility to fiduciaries. However, this delegation introduces its own inefficiencies, particularly in the form of unstructured communication and incomplete documentation, which impair the ability of fiduciaries to work efficiently.

**Approach and technology**

This thesis presents the development of Finteam, a Minimum Viable Product (MVP) designed to streamline document exchange and communication between clients and fiduciaries. Starting from a basic prototype, the project improved the architecture, added core features, and developed a scalable backend to support future growth. The system consists of three main components: first, a mobile app (built with React Native and Expo) for clients to upload tax documents and interact with an AI assistant. Second, a web app (React + TypeScript) for fiduciaries to manage client submissions and provide feedback. Third, a modular backend (Express.js + TypeScript) that handles authentication, data storage, and an RAG-based tax assistant.

**Conclusion**

During this thesis, the team successfully developed an MVP for the Finteam application. The system replaces unstructured communication with a digital workflow that reduces administrative overhead on both sides. The mobile application enables clients to upload tax documents, while the web platform provides fiduciaries the ability to review submissions and send feedback to the client. This approach eliminates the typical back-and-forth email exchanges that currently affect fiduciaries.

As a secondary but notable feature, the system incorporates an AI-powered tax assistant built on Retrieval-Augmented Generation (RAG) technology integrated into the mobile app. It combines vector- and keyword-based search with reranking and query rewriting techniques to improve response quality. An empirical evaluation of various configurations showed that reranking significantly enhanced the accuracy of responses. While this came with a latency trade-off of approximately 38% compared to vector-only retrieval, which served as a baseline. With this tradeoff, the overall user experience benefited from more relevant answers. Key evaluation criteria included helpfulness, correctness, clarity, and fluency.

Key limitations include the absence of a scalable fiduciary onboarding flow and restricted iOS deployment due to the lack of access to Apple's Developer Program. Future work should focus on automating onboarding, enabling administrative oversight, integrating document parsing to prefill tax forms, and at-rest encryption of user documents to enhance security.

# Management Summary

## Initial Situation

In today's fast-paced and highly connected world, many individuals in Switzerland lack a clear overview of their personal tax situation. As a result, tax filing is often perceived as a frustrating and complex task—one that is postponed for as long as possible. To avoid dealing with the intricacies of tax filing, many people choose to delegate the task to a fiduciary. However, this solution introduces its own set of challenges.

Fiduciaries rely on timely and complete documentation from their clients to carry out their work efficiently. In practice, clients are frequently unaware of which specific documents are required. This leads to lengthy email exchanges or repeated phone calls for clarification. Such inefficiencies result in wasted time, increased workload, and reduced profitability for fiduciaries.

This is where the digital solution proposed in this thesis comes into play. The goal was to develop Finteam, a system designed to optimize communication between clients and fiduciaries by providing a structured interface with clear instructions on which documents need to be submitted. The primary objective was to create a user-friendly communication platform that minimizes unnecessary interaction and increases efficiency on both sides.

The project focused on improving an existing prototype and developing a Minimal Viable Product (MVP). The solution consists of a mobile app for clients to upload and manage their tax documents, and a web-based application for fiduciaries to view and process the submissions in a structured and efficient manner.

Furthermore, a chatbot should be integrated into the system, supported by a Retrieval-Augmented Generation (RAG) approach, to answer common tax-related questions.

## Procedure and Technologies

At the beginning of the project, the system architecture was analyzed and three main components were identified for development:

**Mobile Application for Clients** A cross-platform mobile application developed using React Native [1]. This application enables clients to upload their tax documents, access a structured overview of required documents, and communicate efficiently with their fiduciary or the chatbot.

An initial [2] design was provided, which defined the basic user interface and user experience principles. The existing design language was adopted and adapted as necessary to meet technical requirements and improve usability where needed.

**Backend API, RAG Chatbot and Database** A backend system responsible for handling core functionalities such as authentication, data management, and document storage. It serves as the communication layer between the client application and the fiduciary interface. In addition to that, it hosts a RAG chatbot.

The backend is built using TypeScript [3] and Express.js [4], utilizing a layered architecture with controllers, services, and models. It is connected to two PostgreSQL [5] databases through the Drizzle ORM [6]. One storing the application data, the other the vector embeddings for the chatbot.

**Web Application for Fiduciaries** A web-based platform designed for fiduciaries to manage clients, review submitted documents, and interact with clients by providing comments or feedback through the system. It is built using React.js [7] and TypeScript [8], with Tailwind CSS [9] for styling.

# Results

Over the course of the project, a fully functioning MVP of the Finteam platform was successfully developed. The solution comprises a mobile application for clients, a web-based platform for fiduciaries, and a modular backend infrastructure with an integrated RAG-based AI chatbot.

### Mobile Application

The mobile app allows clients to effortlessly upload tax documents, track their submitted documents, and communicate directly with their fiduciary or the integrated AI chatbot to clarify tax-related questions. This reduces uncertainty about required documents and minimizes the need for follow-up communication.

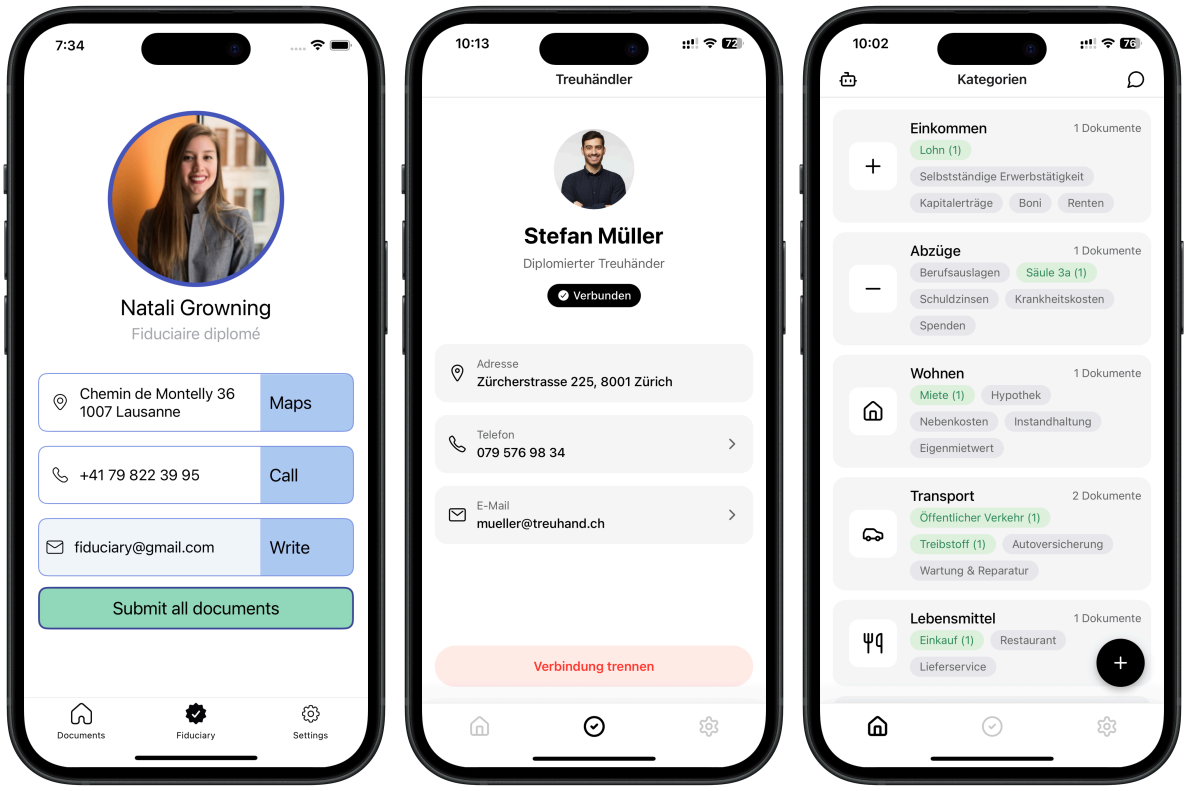



Figure 1: Prototype (left) to MVP (middle, right) transformation

### Web Application for Fiduciaries

The web platform provides fiduciaries with a structured interface to manage clients, review submitted documents, and communicate within the system. This optimizes document management, reduces administrative effort, and helps trustees maintain a clear overview of their work.

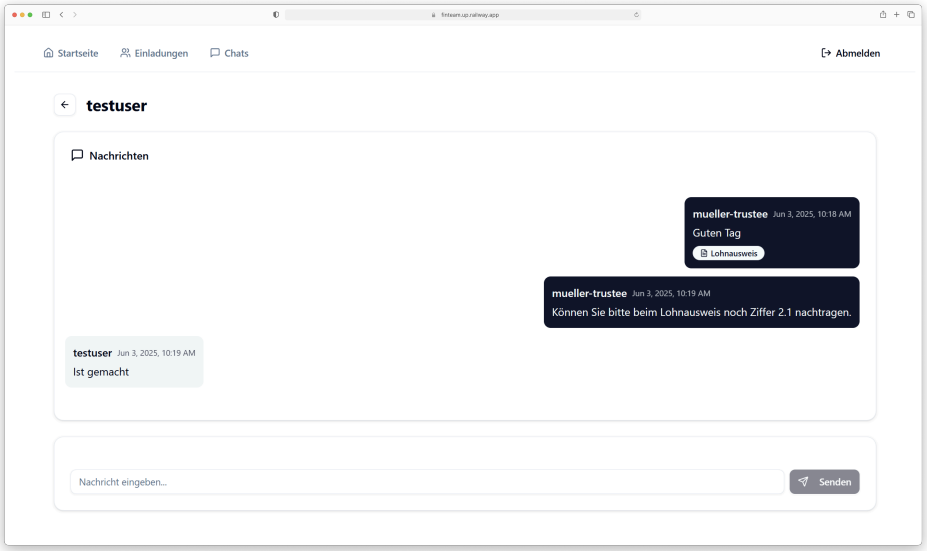



Figure 5: Web app chat interface

**Backend and Chatbot** The backend follows a modular, layered architecture to support scalability and maintainability. Input validation is handled, and data is stored in PostgreSQL databases. A Retrieval-Augmented Generation (RAG) chatbot is integrated into the system to answer common tax-related questions. Various retrieval configurations were implemented and evaluated, with the Rerank setup achieving the highest response quality based on a structured metric evaluation.

Together, these components form a modern, scalable foundation for a production-ready tax support platform.

## Conclusion

The Finteam project has clearly demonstrated that a modern, digital solution can optimize communication between fiduciaries and their clients. The developed platform effectively combines best practices in full-stack development with state-of-the-art AI integration. By using TypeScript, modular design, and structured workflows, the team has delivered a robust and extensible MVP.

Key areas for improvement include implementing a secure, self-service onboarding process for fiduciaries, developing an administrative dashboard for system oversight, and automating the pre-filling of tax forms using structured document data. Additionally, enabling iOS deployment remains a priority once access to the Apple Developer Program is available.

In summary, the project shows that with the right combination of modern technology, AI capabilities, and structured development processes, significant efficiency gains can be achieved in traditionally manual and communication-intensive areas such as tax consulting.

# Contents

# Part I

# Documentation

# Chapter 1

# Introduction

## 1.1. Project Goal

The primary objective of this project is to develop a cross-platform mobile application using React Native [1] that facilitates the seamless transfer of tax documents and tax-related data from individual users to fiduciaries. This application will enable users to conveniently capture and submit their tax information via their smartphones. On the fiduciary side, a complementary web application will provide access to the submitted data, allowing fiduciaries to download documents and add comments, thereby enhancing communication efficiency and reducing processing time between the fiduciary and the client.

In addition to the core data exchange functionality, this project aims to design and implement a RAG-based chatbot to assist users by answering common questions related to taxation. Various RAG approaches will be explored and systematically evaluated to determine the most effective method for integrating knowledge retrieval and natural language generation in the context of tax advisory support.

## 1.2. Background

The digitalization of financial and administrative processes has become increasingly important in recent years, particularly in the domain of tax management. Traditional methods of exchanging tax documents and information between taxpayers and fiduciaries often involve manual handling, physical paperwork, and inefficient communication channels. These factors can lead to delays, errors, and increased workloads for both parties.

Mobile applications have the potential to streamline this interaction by enabling users to capture and submit tax data directly from their smartphones in a secure and user-friendly manner. Cross-platform development frameworks such as React Native [1] facilitate the creation of applications that can operate on both iOS and Android devices, ensuring broad accessibility and ease of use for a diverse user base.

Moreover, advancements in artificial intelligence and natural language processing have opened new possibilities for automated support through chatbots. RAG is an emerging technique that combines the retrieval of relevant documents from a knowledge base with generative language models to produce accurate and contextually relevant responses. Applying RAG to tax advisory services greatly enhances user support by providing instant answers to complex tax-related questions.

In this context, the development of a mobile application integrated with a RAG-based chatbot presents an innovative approach to modernizing tax communication workflows. Evaluating

different RAG methodologies within this project aims to identify the most effective solutions for providing the best answers.

## 1.3. General Conditions

This documentation references multiple code files from a GitHub repository [10]. Access to the repository is restricted and only available to members of the organization. The project advisor can be asked to grant access.

The work was carried out as part of a bachelor thesis with a time budget of 360 hours per person, i.e., a total of 720 working hours. Participants are awarded 12 ECTS credits for their work.

# Chapter 2

# Requirement Analysis

The requirements engineering process was based on initial swimlane flowcharts created in Miro [11]. These diagrams served as a conceptual starting point and were analyzed and refined to create structured and clearly defined requirements.

Based on this analysis, both Functional Requirements (FRs) and Non-Functional Requirements (NFRs) were identified. These requirements form the basis for the design and implementation of the system and ensure that both the expected functionalities and quality characteristics are taken into account.

## 2.1. Flowcharts

To support the development of the core functionalities, several flowcharts were designed. These visual representations served to identify and structure the fundamental processes of the application's behavior. It is important to mention that the user always interacts with a mobile app, while the fiduciary exclusively uses the web application. This distinction reflects the assumption that users primarily manage their tasks on mobile devices, whereas fiduciaries typically work in an office environment using a laptop or desktop computer.

The key processes identified during this phase include:

1. **User registration** – enabling private individuals to create an account and access the platform.
2. **Fiduciary registration** – allowing fiduciaries to register and gain access to the platform.
3. **Fiduciary invites customer (user)** – allowing fiduciaries to invite customers (users) to see and review their data.
4. **Document scanning and forwarding** – allowing users to scan or upload tax-related documents via the mobile application and forward them to their assigned fiduciary.
5. **Communication between fiduciary and user** – enabling fiduciaries to request changes or additional documents and allowing users to respond accordingly.

### User registration

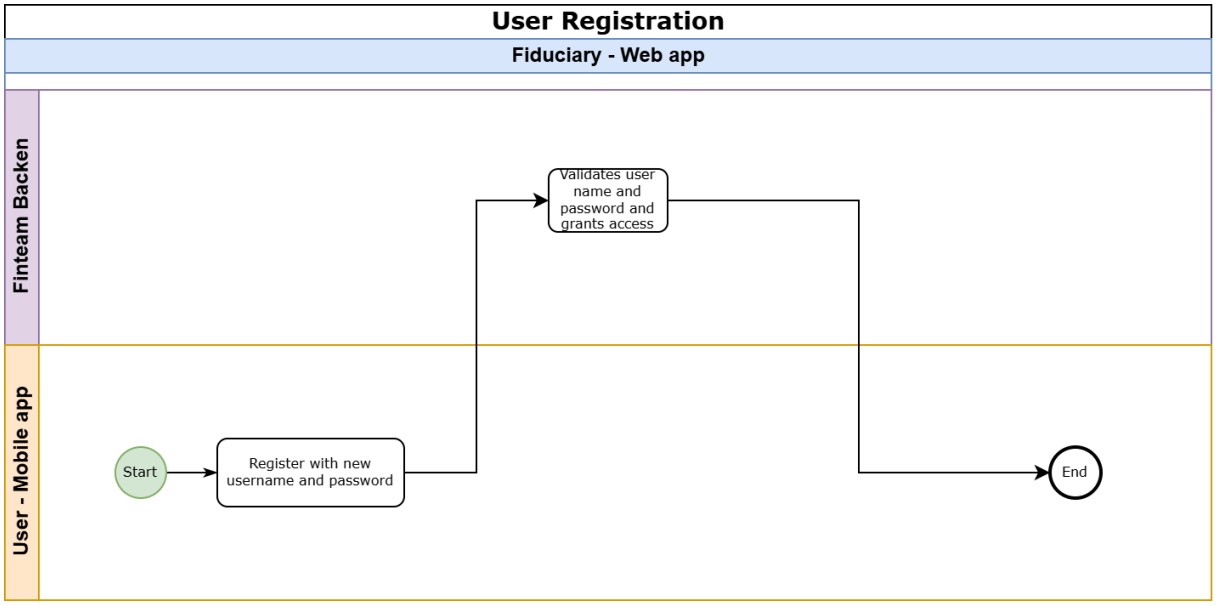



Figure 6: User Registration

The user registration process follows a standard industry practice, utilizing a login system based on a username and password. Due to the limited scope of the project, the registration does not require an email address. This design choice avoids the need for integrating external email services, which would introduce additional complexity and infrastructure requirements for sending verification or confirmation emails. There is also no self-service password reset, as this functionality would require the use of an external communication method like email to verify user identity.

### Fiduciary registration

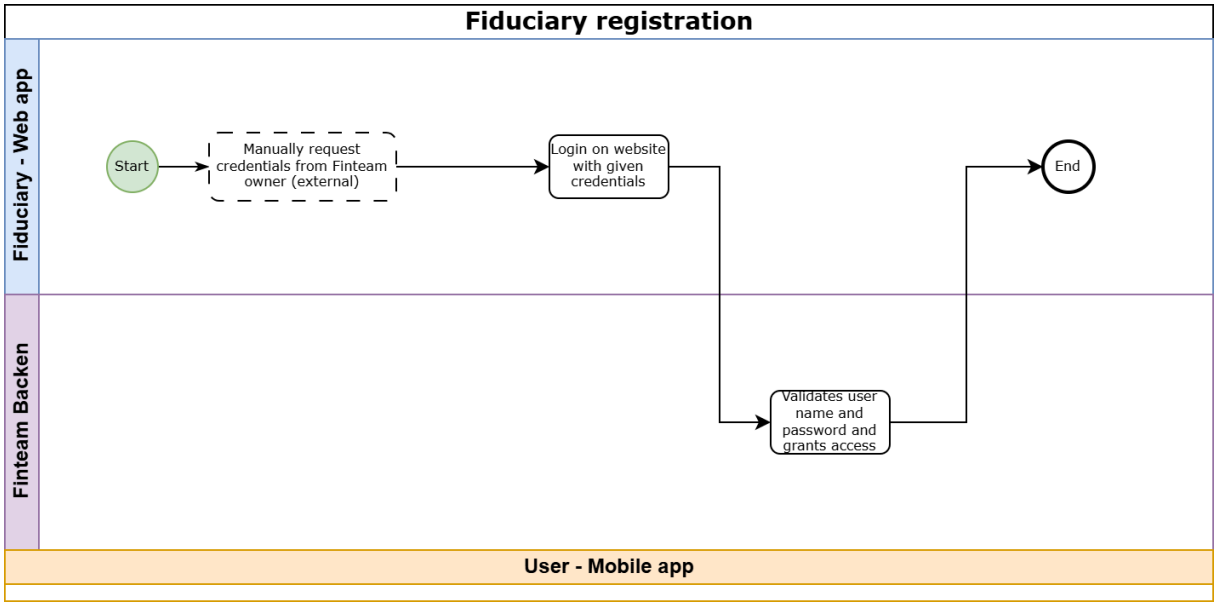



Figure 7: Fiduciary registration

In the initial phase of the platform, the project team decided to manually create accounts for fiduciaries. This approach offers several advantages. Most importantly, it allows the platform operator to verify the legitimacy of each fiduciary individually. By requesting appropriate documentation or credentials, the operator can ensure that only verified fiduciaries gain access to the system. This controlled onboarding helps keep the number of fiduciaries manageable during the early stages and significantly enhances security. It reduces the risk of unauthorized individuals posing as fiduciaries to gain access to sensitive client data.

### Fiduciary invites customer (user)

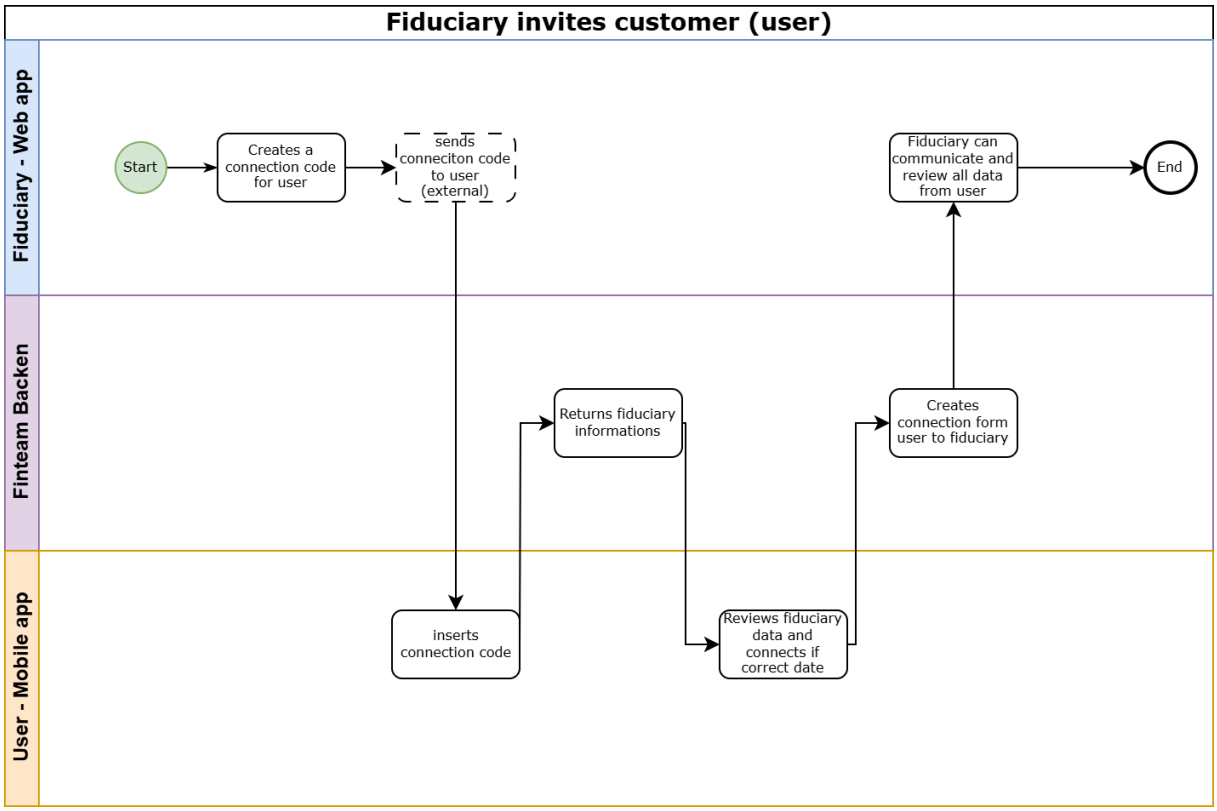



Figure 8: Fiduciary invites customer (user)

Regarding the connection process between fiduciaries and customers, the project team assumes that the app will initially be used primarily with existing clients of fiduciaries. In such cases, the customer and fiduciary are already in contact outside the application. To establish a digital connection within the app, the fiduciary can generate a unique connection string via the web application. This string can then be shared with the customer through external communication channels such as email or messaging apps. Upon receiving the connection string, the customer can enter it into the mobile application, where the fiduciary's profile information will be displayed. The customer can then choose to confirm or deny the connection request. Once the connection is accepted, the fiduciary gains access to the customer's data.

### Document scanning and forwarding

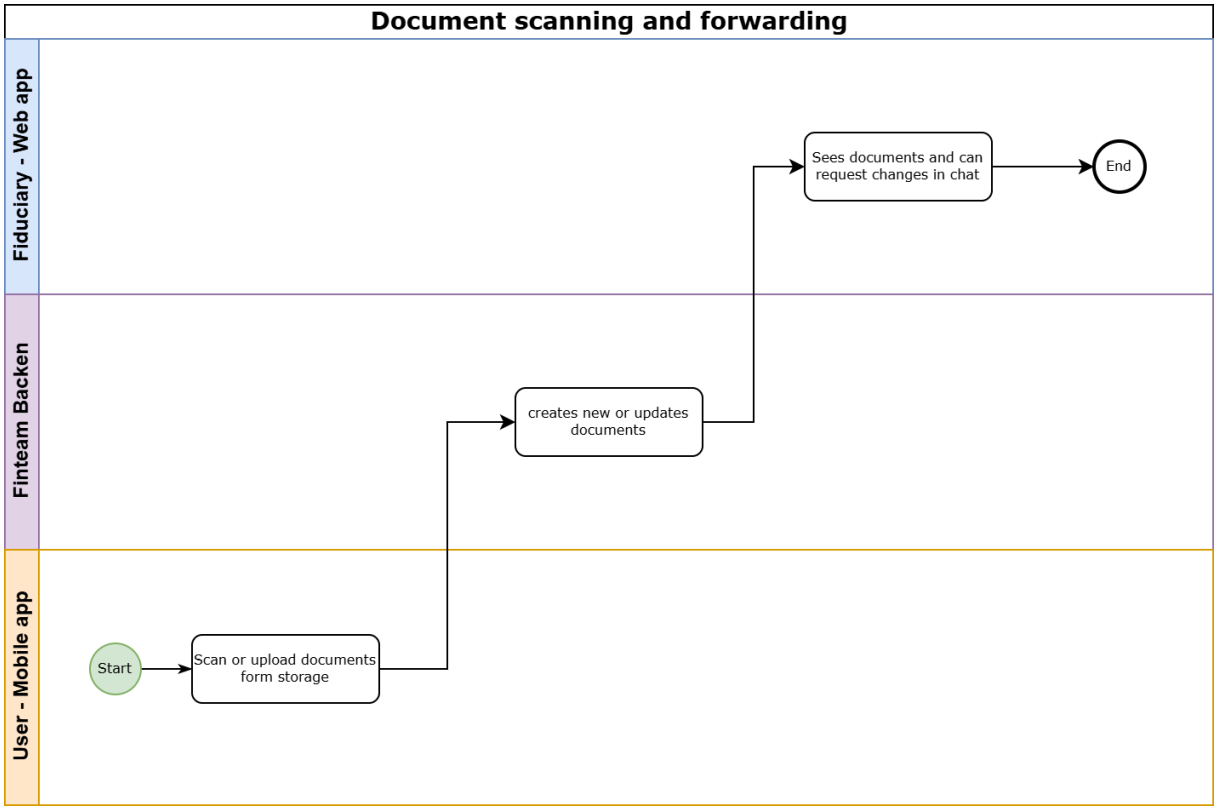



Figure 9: Document scanning and forwarding

The document scanning process has been designed to be simple and user-friendly. Customers can either scan physical documents using their smartphone camera or upload existing files stored on their device. Once uploaded, the data is securely stored in the backend of the system. If the customer is already connected to a fiduciary, the uploaded documents immediately become accessible to the fiduciary through the web application. However, the fiduciary is not actively notified since email integration is not part of the MVP.

**Communication between fiduciary and user**

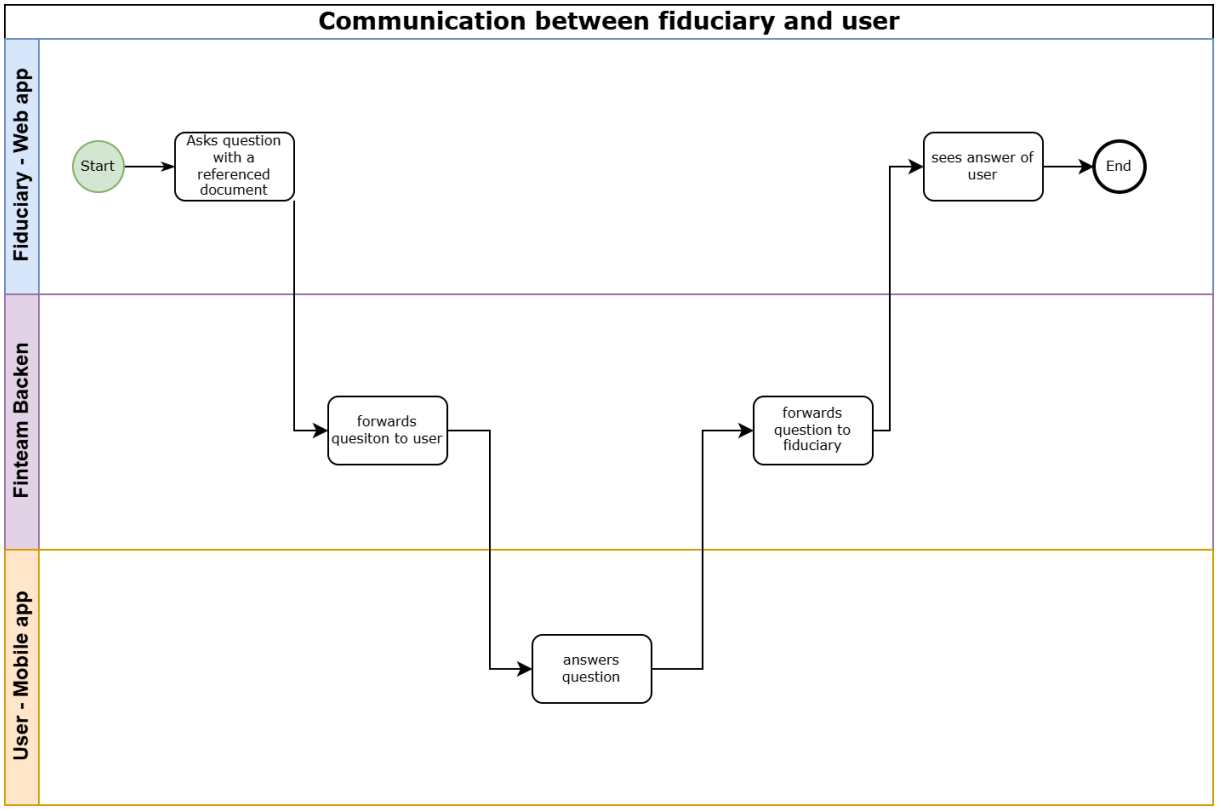



Figure 10: Communication between fiduciary and user

Communication between fiduciaries and customers is enabled through an integrated messaging system. Once a connection is established, fiduciaries can send messages to their customers via the web application. Each message can optionally include a reference to a specific document previously uploaded by the customer. These messages are transmitted through the backend and delivered to the customer's mobile application. Customers can respond directly through the app, with replies again routed via the backend to the fiduciary's web interface.

## 2.2. Functional Requirements

With the above basic processes visualized, User Stories and Epics are use to document FRs.

### 2.2.1. Use Case Diagram

Figure 11 represents a visual overview of the requirements in the system, than can be explored further in Section 2.2.4.

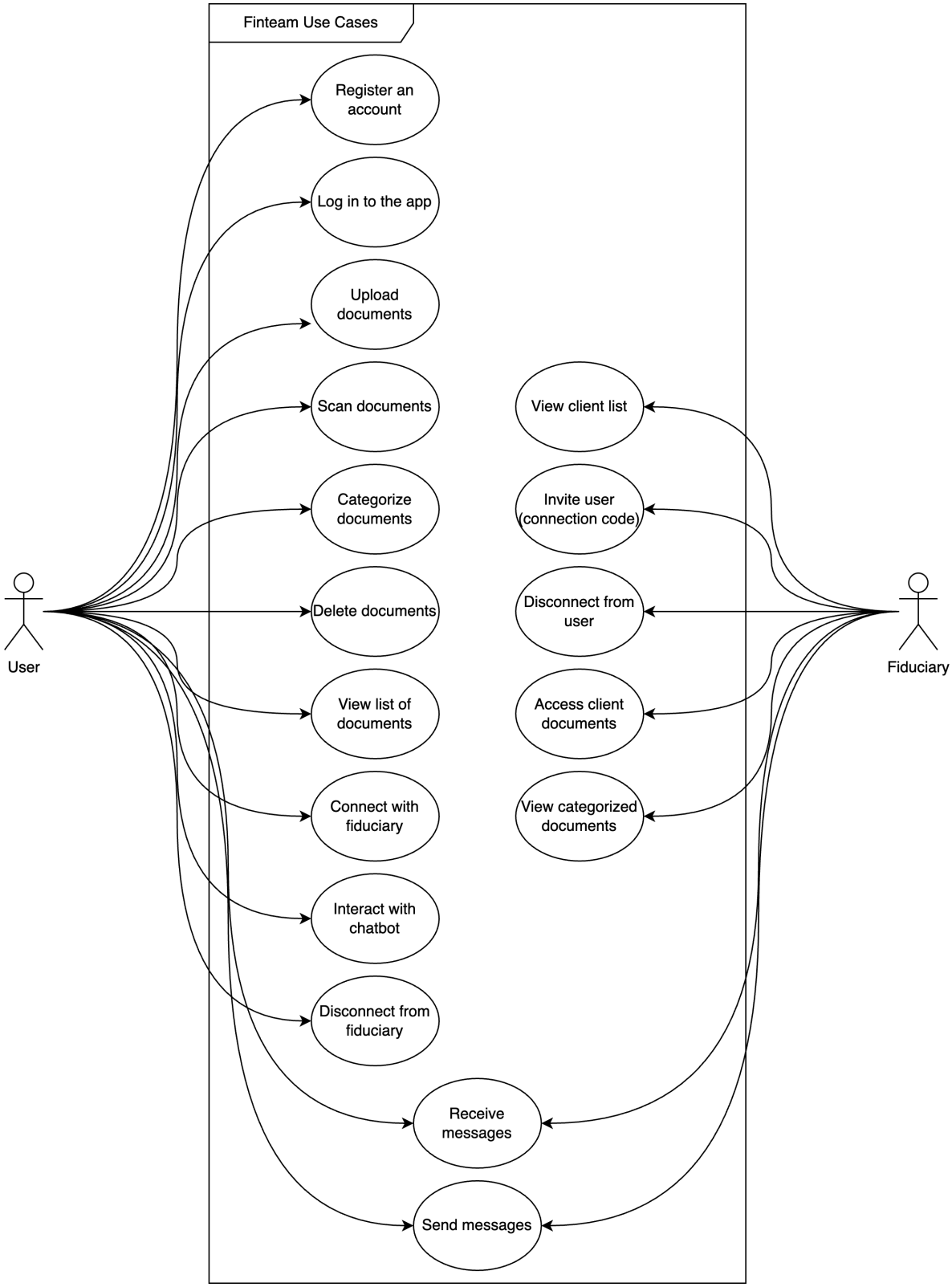



Figure 11: Use Case Diagram

### 2.2.2. Roles

There are **2 relevant roles** in the system:

- **User:** Someone filing their taxes. Typically, the customer of a fiduciary. This role can autonomously sign up for an account and start uploading their tax files. They can accept linking requests from a fiduciary to share their personal documents with said fiduciary.
- **Fiduciary:** The person or company responsible for advising and representing for filing their taxes. Can invite a user to connect, so that the user becomes their client and they can see all documents. Fiduciaries cannot sign up to the platform themselves. They must onboard manually.

There is no administrator role because the developed system has no administrator interface or other provision to manage it. All administrative tasks, such as approving a new fiduciary, will have to be done by an engineer with access to the production database. Of course, such a management system can be added at a later time without any architectural changes. But it is not in the scope of this thesis.

### 2.2.3. Epics

**Foundation:** Building up the fundamental system, initiating the codebases, writing the necessary boilerplate code, and deploying a first version of the backend and web application. This epic does not contain any user facing functionality, but it's a necessary base for further epics.

**User account management:** Allow users to create accounts, securely log in, manage their profile information, and related functionalities. This includes the mobile app for registration and login, the backend API for user authentication and authorization, and the data model for storing user profiles.

**Document management:** Allows users to scan, upload, and manage documents. This includes the necessary components in the mobile app and backend API endpoints, including a storage layer on the server.

**Fiduciary Client onboarding and Management:** User-invite system for fiduciaries and the tools to effectively manage their client relationships. This includes the one-way invite system for connecting fiduciaries and users, the initial setup and configuration of the fiduciary web app, and the core features for managing client lists and accessing basic client information.

**Enhanced fiduciary-user Interaction:** Enhance the interaction between fiduciaries and users by providing secure data access, document viewing, and a robust messaging system within the fiduciary web application. This includes access control measures for protecting user data, the document viewing capabilities of the fiduciary web app, and the messaging system for facilitating communication.

**Tax advisor chatbot:** Develop a chatbot utilizing RAG to provide users with accurate and relevant information based on the tax document. This includes the implementation of the RAG architecture for document retrieval and response generation, the ingestion and processing of the tax guidance content, and the user interface for interacting with the chatbot.

### 2.2.4. User and Developer Stories

User and developer stories describe the desired features and functionalities of the system from the perspective of its users and developers. Below is an example of a user story. The full set of user and developer stories can be found in Chapter 15. The testing protocol evaluating these stories can be found in Chapter 8.2.5.

The stories within the Foundation Epic are technical in nature and cannot be expressed meaningfully from a user's perspective. Therefore, Developer Stories are used instead.

**User Scans Documents with Camera**

| | |
|---|---|
| **ID** | FR-011 |
| **Name** | User Scans Documents with Camera |
| **User Story** | As a user, I want to be able to scan documents using my mobile device's camera so that I can easily upload paper documents. |
| **Epic** | Document Management |
| **Acceptance Criteria** | • Users can initiate the scanning process from within the mobile app.<br>• The app utilizes the device's camera to capture images of documents.<br>• Users can preview the scanned document before saving it. |

Table 1: FR-011

## 2.3. Non-Functional Requirements

The Non-functional requirements describe quality attributes of the system, such as performance, security, and usability. Below is an example of a Non-functional requirement. The full list can be found in Chapter 15.2. The testing protocol evaluating the requirements can be found in Chapter 8.2.6.

**Usability**

| **ID** | NFR-01 |
|---|---|
| **Description** | The application should be easy to use for new users. A new user can use the application effortlessly. |
| **Requirement** | Usability - Ease of Use |
| **Priority** | High |
| **Verification Process** | Conduct usability testing with two new users to assess their ability to complete key tasks without assistance. |
| **Measures** | • Task completion rate: >90% of new users can complete key tasks successfully without external guidance.<br>• Time on task: New users can complete key tasks in under 5 minutes. |

Table 2: NFR-01

# Chapter 3

# Architecture

This chapter describes the design decisions behind all aspects of the application in detail. Key design decisions are documented using Architectural Decision Record (ADR) and can be found in Section 15.3. The basic architecture is visualized using the C4 Notation (C4), and for the detailed architectural visualization of the mobile app, website, and API, layered diagrams are used.

## 3.1. C4 - Level 1 System Context Diagram

The System Context Diagram Figure 12 illustrates the external systems that the Finteam application interacts with. In this case, the application has a single external dependency: the OpenAI API [12]. This API is used to generate embeddings and handle chat completions, which are core to the functionality of the integrated chatbot.

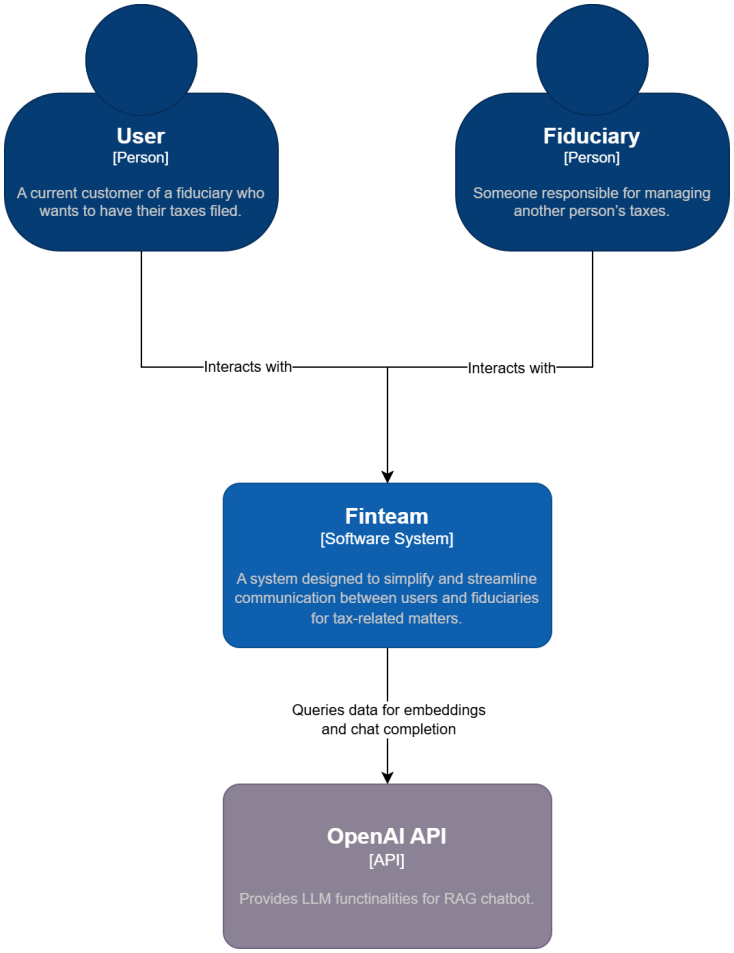



Figure 12: C4 system context diagram

## 3.2. C4 - Level 2 Container Diagram

The Container Diagram Figure 13 shows that two standalone applications are being developed: a mobile application for iOS and Android, intended for users, and a web application used exclusively by fiduciaries. Both applications interact with a shared Backend API, which serves as the central communication layer. This backend is responsible for handling client requests, enforcing business logic, and connecting to a PostgreSQL database to persist and retrieve data.

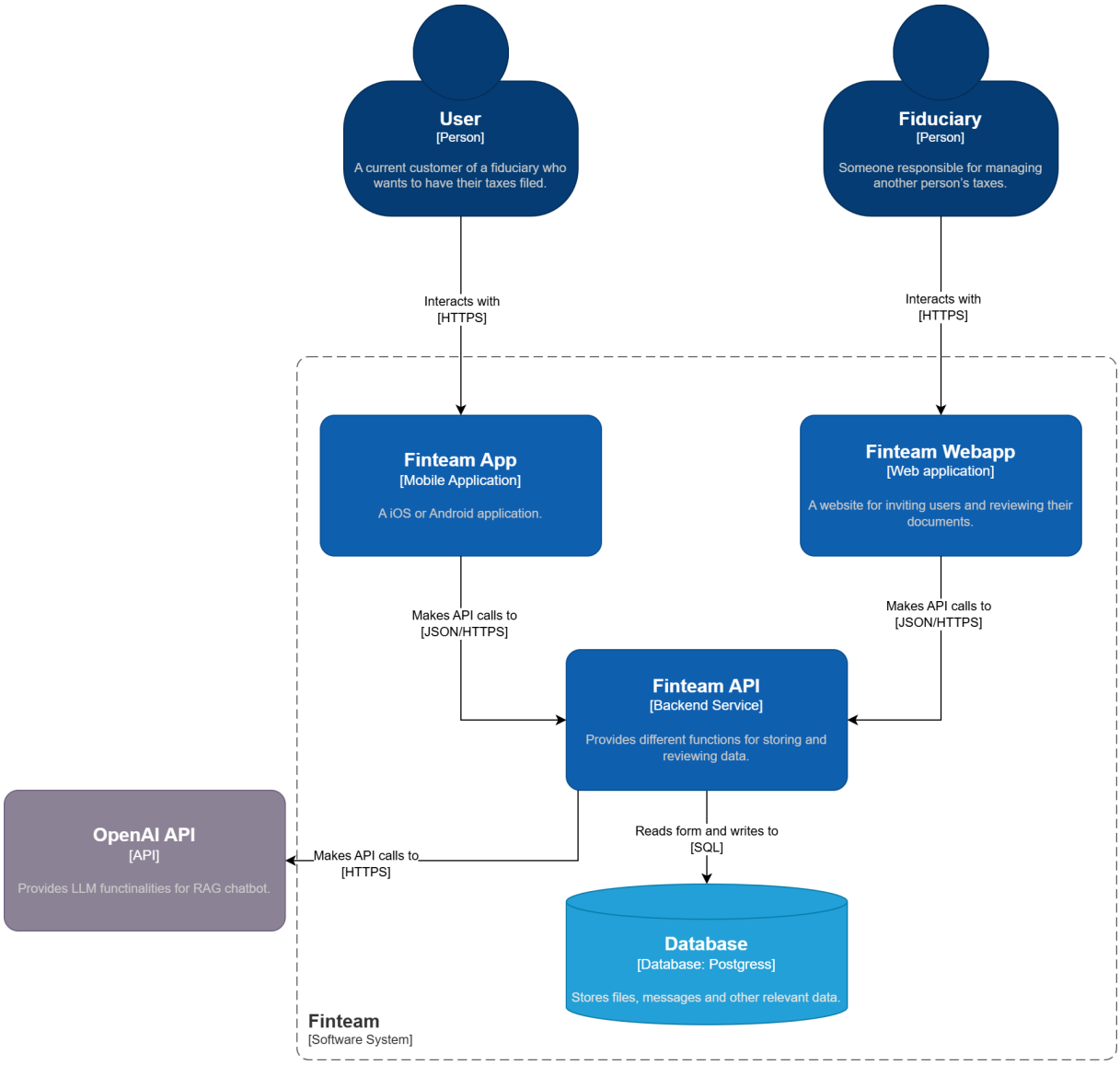



Figure 13: C4 container diagram

## 3.3. Domain Model

The domain model shown in Figure 14 provides a visual representation of the core entities within the application's domain and their relationships. This model serves as a fundamental blueprint for understanding the structure and functionality of the system.

The model includes both business-relevant attributes (such as `fullName` or `content`) and technical implementation details (including `id` primary keys, `createdAt`/`updatedAt` audit timestamps, and system-specific fields like `storageKey` and `hashedPassword`). This provides a comprehensive view of the domain entities as they will be implemented in the system. This serves as a bridge between the conceptual business domain and the actual database schema design.

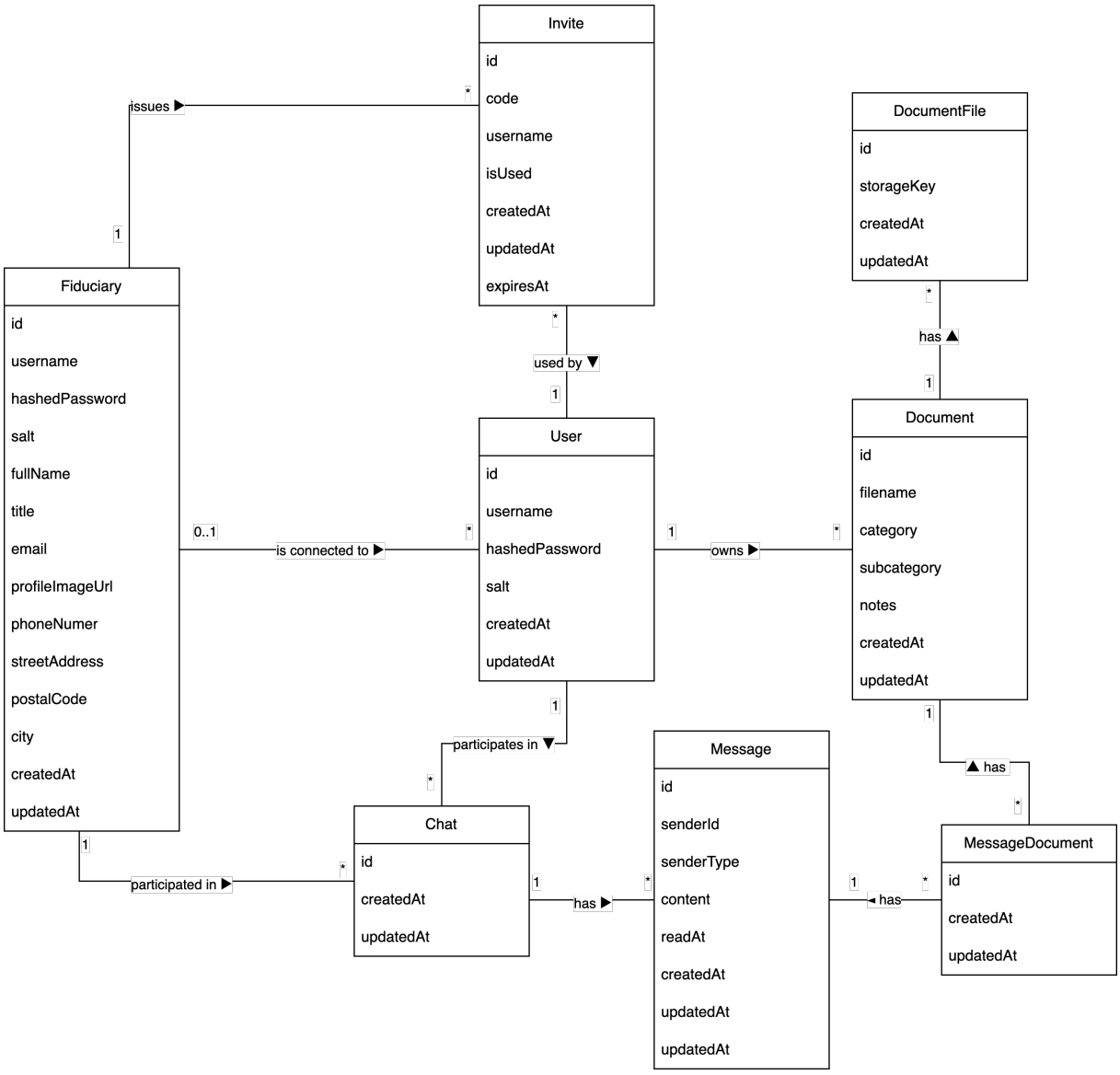



Figure 14: Domain model

The system involves several key components and their interactions as illustrated in the diagram:

- The **Fiduciary** represents a user with specific roles and permissions, interacting with the system. Fiduciaries can issue **Invites**.
- An **Invite** is a mechanism for granting access or inviting users. It has attributes like `code`, `username`, and `expiresAt`. An Invite is `used by` a **User**.
- A **User** is a central entity in the system. Users `own` **Documents**. Users also `participate in` **Chats**.
- A **Document** represents a piece of content within the system. Documents have attributes like `filename`, `category`, and `subcategory`. Documents can have associated **DocumentFiles**.
- A **DocumentFile** represents the actual file associated with a Document.
- A **Chat** facilitates communication between a user and a fiduciary. Users `participate in` Chats. Chats contain **Messages**.
- A **Message** is a unit of communication within a Chat. Messages have attributes like `senderId`, `senderType`, and `content`. Messages can have associated **MessageDocuments**.
- A **MessageDocument** links a Message to a Document, allowing for documents to be shared within messages.

## 3.4. Mobile Architecture

The mobile application's architecture is designed with a clear separation of concerns, organized into three primary layers: Presentation, Business Logic, and Data Layer. The Cross-Cutting Concerns Layer is an additional layer that spans all three core layers. It represents shared functionality that is used across the entire application hierarchy.

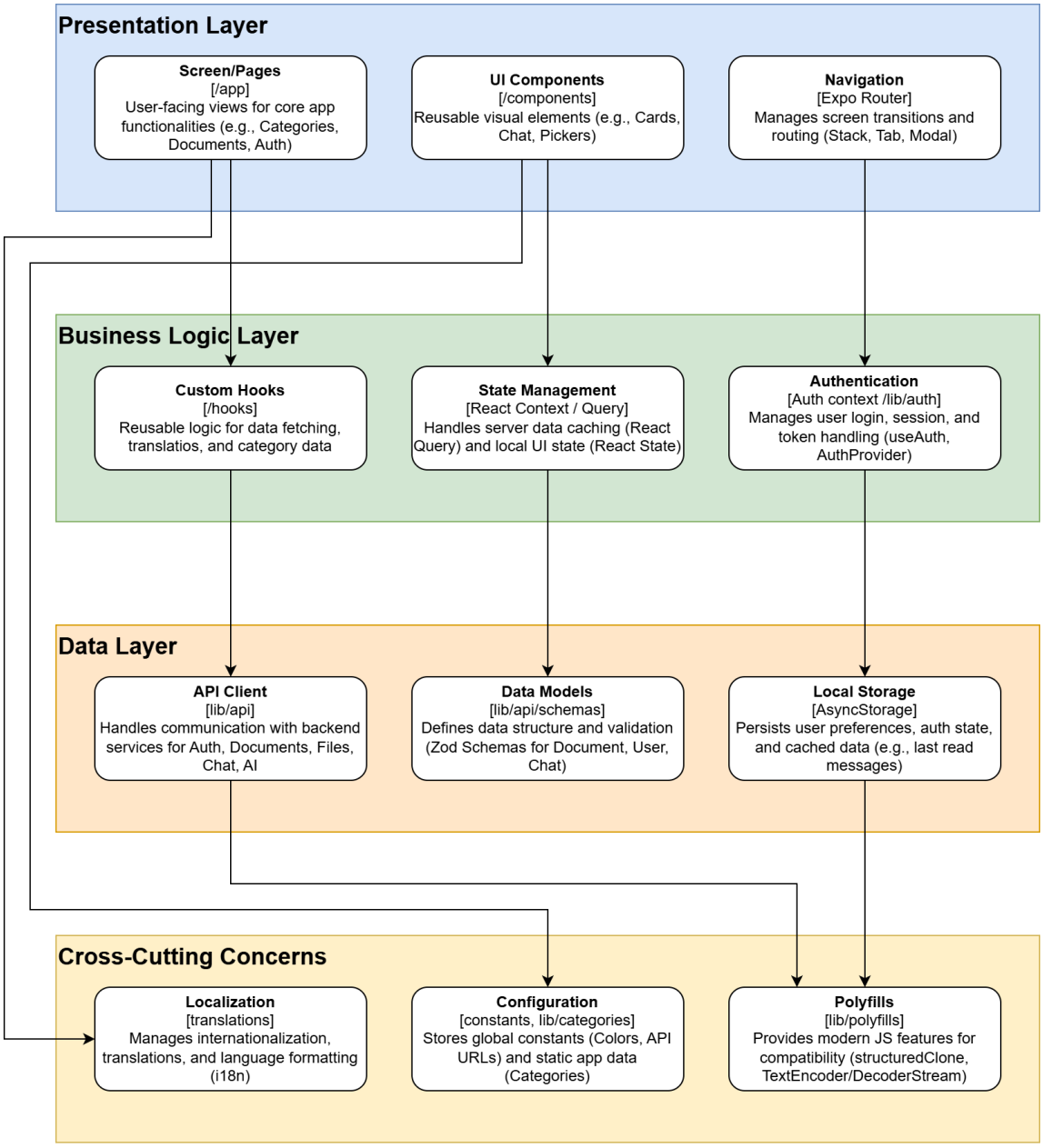



Figure 15: Mobile Architecture

### 3.4.1. Project Directory Structure Overview

The main directories at the root of the project include:

- **app:** Contains all files related to the Presentation Layer, specifically screens and navigation setup using Expo Router.
- **components:** Houses reusable UI components utilized across various screens.
- **hooks:** Stores custom React hooks that encapsulate reusable business logic and stateful operations.
- **lib:** A general-purpose directory containing core logic, including API clients, authentication services, data models/schemas, polyfills, and static data definitions.
- **translations:** Manages localization files and internationalization (i18n) setup.
- **constants:** Holds application-wide constant values, such as color palettes and configuration settings.

### 3.4.2. Presentation Layer

This layer is responsible for everything the user sees and interacts with. It is built using React Native components and Expo Router for navigation.

#### Screens/Pages

These are the individual views of the application, such as the main category overview, document display pages, user authentication forms (sign-in, sign-up), and settings panels. These are located within the `app` directory, where Expo Router parses routes based on file structure. For example:

- `index.tsx` represents the main (root) screen within a tab-based layout.
- `view.tsx` is a screen for viewing a specific document.

#### UI Components

To maintain consistency and reusability, common visual elements are encapsulated as React Native components. These include:

- Cards for displaying information (e.g., `CategoryCard`)
- Chat interfaces (`AiChat`, `Chat`)
- Input controls like image pickers (`ImagePickerButton`)

These components are stored in the `components` directory and imported into screens as needed.

#### Navigation

The application uses Expo Router to manage transitions between different screens. This includes:

- Stack navigation for hierarchical flows
- Tab navigation for primary sections (configured in files like `_layout.tsx`)

The routing system supports parameterized routes, which means inserting dynamic values such as an ID or user name directly into the URL. This allows screens to receive and use data passed via the navigation path, enabling more flexible and context-sensitive navigation.

### 3.4.3. Business Logic Layer

This layer acts as an intermediary between the Presentation Layer and the Data Layer, containing the application's core logic, state management, and data processing.

#### Custom Hooks

Reusable logic is extracted into custom React hooks in the `hooks` directory. Examples include:

- `useTranslation` for internationalization
- `useCategories` for category data
- `useAuthenticatedQuery` for authenticated API requests

#### State Management

Application state is managed using:

- React Query [13] for server state: fetching, caching, updating
- React's built-in `useState` and `useReducer` for local UI state

#### Authentication

User authentication and session management are handled via an auth context in the `lib` directory (e.g., `auth.tsx`), including:

- `AuthProvider` for global auth state
- `useAuth` for accessing user data and auth functions like login/logout
- Token management

### 3.4.4. Data Layer

Responsible for all data-related operations, including external API interaction and local persistence.

#### API Client

Located in the `api` directory, the API client includes modules for:

- Authentication
- Documents
- Files
- Chat and AI services

These modules handle HTTP requests and responses.

#### Data Models

Data structures are defined using schemas (Zod) for consistency and validation. Schemas like `Document`, `User`, `Chat` are located in files like `schemas.ts`.

#### Local Storage

Uses `AsyncStorage` to persist:
- Authentication tokens
- Cached data (e.g., recent documents or messages)

### 3.4.5. Cross-Cutting Concerns

Functionality is used across multiple layers of the application.

#### Localization

Internationalization (i18n) is implemented using:

- JSON translation files in `locales` (e.g., `de.json`)
- Initialization in files like `index.ts`
- Hooks/functions for accessing translations

#### Configuration

Constants and static data are stored in:

- `constants` directory (e.g., `colors.ts`)
- `lib` directory (e.g., `categories-data.tsx` for static categories or FAQs)

## 3.5. Web Architecture

The web application's architecture follows the same patterns as the mobile application. It is designed with a clear separation of concerns, organized into three primary layers: Presentation, Business Logic, and Data Layer. The Cross-Cutting Concerns Layer is again used over all layers.

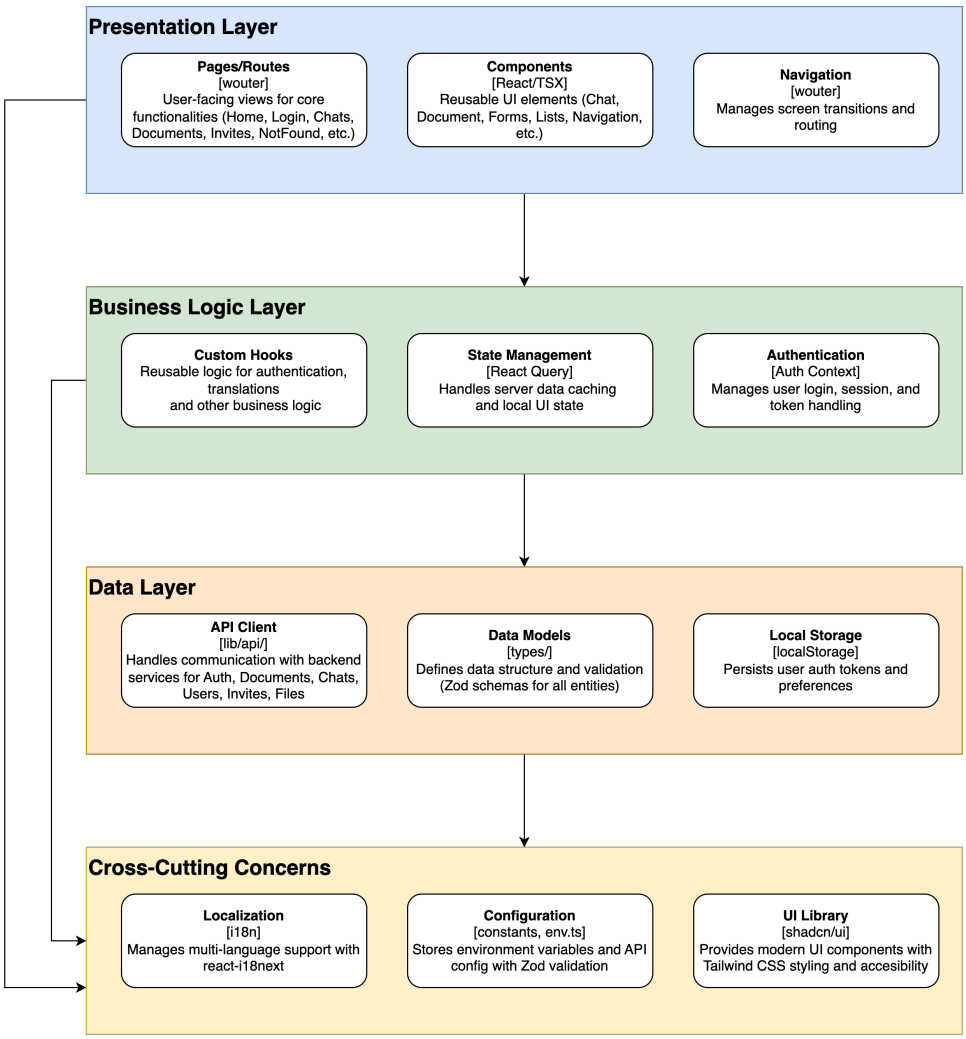



Figure 16: Web Architecture

### 3.5.1. Project Directory Structure Overview

The main directories at the root of the project include:

- **pages:** Contains all page components that represent different routes/screens in the application, including home, login, chats, documents, invites, and user management pages using wouter [14] for routing.
- **components:** Houses reusable UI components utilized across various pages, including chat interfaces, document viewers, navigation, forms, and authentication components.
- **hooks:** Stores custom React hooks that encapsulate reusable business logic, including authentication (use-auth.ts) and internationalization (use-translation.ts) functionality.

- **lib:** A general-purpose directory containing core logic, including API clients (api/), authentication services (auth.ts, auth-context.tsx), and utility functions (utils.ts).
- **i18n:** Manages internationalization setup with react-i18next, including translation configuration and locale files for multi-language support.
- **types:** Contains TypeScript type definitions and data models for different entities like chats, documents, users, invites, posts, and authentication.

### 3.5.2. Presentation Layer

This layer is responsible for everything the user sees and interacts with. It is primarily built using React components and wouter for navigation.

### 3.5.3. Screen/Pages

These are the individual views of the application, such as the home page, document display pages, user authentication forms (login), chat interfaces, and invite management panels. These are located within the `pages` directory, where wouter routing conventions define routes based on component imports and path configuration [14]. For example:

- `home-page.tsx` represents the main dashboard screen.
- `chats-detail-page.tsx` is a screen for viewing and interacting with specific user chats.
- `document-detail-page.tsx` handles document viewing and file management.

#### Components

To maintain consistency and reusability, common visual elements are encapsulated as React components. These include:

- Cards for displaying information (e.g., `ChatListCard`, `DocumentList`)
- Chat interfaces (`ChatInput`, `ChatList`, `ChatOverviewList`)
- Form controls and dialogs (`LoginForm`, `CreateInvite`, `DeleteUserDialog`)
- File management components (`FileContentViewer`, `FileList`)

These components are stored in the `components` directory and imported into pages as needed. A `ui` subdirectory contains base design system components built with Tailwind CSS.

#### Navigation

The application uses wouter to manage transitions between different screens. This includes:

- Route-based navigation for hierarchical flows (using `<Route>` and `<Switch>`)
- Responsive navigation bar with mobile sheet drawer (configured in `navigation.tsx`)
- Protected routes that require authentication
- Modal dialogs using sheet components

Dynamic routes allow data to be passed between screens during navigation, such as user IDs and document IDs extracted from URL parameters using the `useParams()` hook.

## 3.5.4. Business Logic Layer

This layer acts as an intermediary between the Presentation Layer and the Data Layer. It contains the application's core logic, state management, and data processing.

#### Custom Hooks

Reusable logic is extracted into custom React hooks in the `hooks` directory. Examples include:

- `useTranslation` for internationalization with react-i18next
- `useAuth` for authentication state and token management
- API-specific hooks like `useChatMessages`, `useSendChatMessage` for chat functionality

#### State Management

Application state is managed using:

- React Query [13] for server state: fetching, caching, updating, and synchronization
- React Context (AuthProvider) for global authentication state
- React's built-in `useState` and `useReducer` for local UI state

#### Authentication

User authentication and session management are handled via an auth context in the `lib` directory (`auth-context.tsx`), including:

- `AuthProvider` for global auth state management
- `useAuth` hook for accessing user data and auth functions like login/logout
- Token management using localStorage with automatic session expiry handling

## 3.5.5. Data Layer

Responsible for all data-related operations, including external API interaction and local persistence.

#### API Client

Located in the `lib/api` directory, the API client includes modules for all necessary API endpoints. Multiple individual endpoints are grouped, such as:

- Authentication (`auth-routes.ts`)
- Documents (`document-routes.ts`)
- Invite system (`invite-routes.ts`)

These modules handle HTTP requests and responses with built-in error handling and unauthorized response management.

#### Data Models

Data structures are defined using Zod schemas for runtime validation and type safety. Schemas like `ChatMessage`, `Document`, `User`, `Invite` are located in the `types` directory with files like `chat-types.ts`, `document-types.ts`, `user-types.ts`.

#### Local Storage

Uses browser `localStorage` to persist:

- Authentication tokens
- User session state
- No cached data persistence (relies on React Query for caching)

### 3.5.6. Cross-Cutting Concerns

Functionality used across multiple layers of the application.

#### Localization

Internationalization (i18n) is implemented using `react-i18next`:

- JSON translation files in `i18n/locales` (e.g., `de.json` for German)
- Configuration and schema validation in `i18n/i18n.ts` with Zod validation
- Custom `useTranslation` hook for type-safe access to translations

#### Configuration

Environment variables and configuration are managed using:

- `env.ts` file using `@t3-oss/env-core` with Zod validation for type safety
- Environment variables like `VITE_API_URL` for API endpoints

## 3.6. API Architecture

The API application's architecture also follows a layered approach. It is organized into four primary layers: Presentation, Business Logic, Service, and Data, with Cross-Cutting Concerns spanning across all layers.

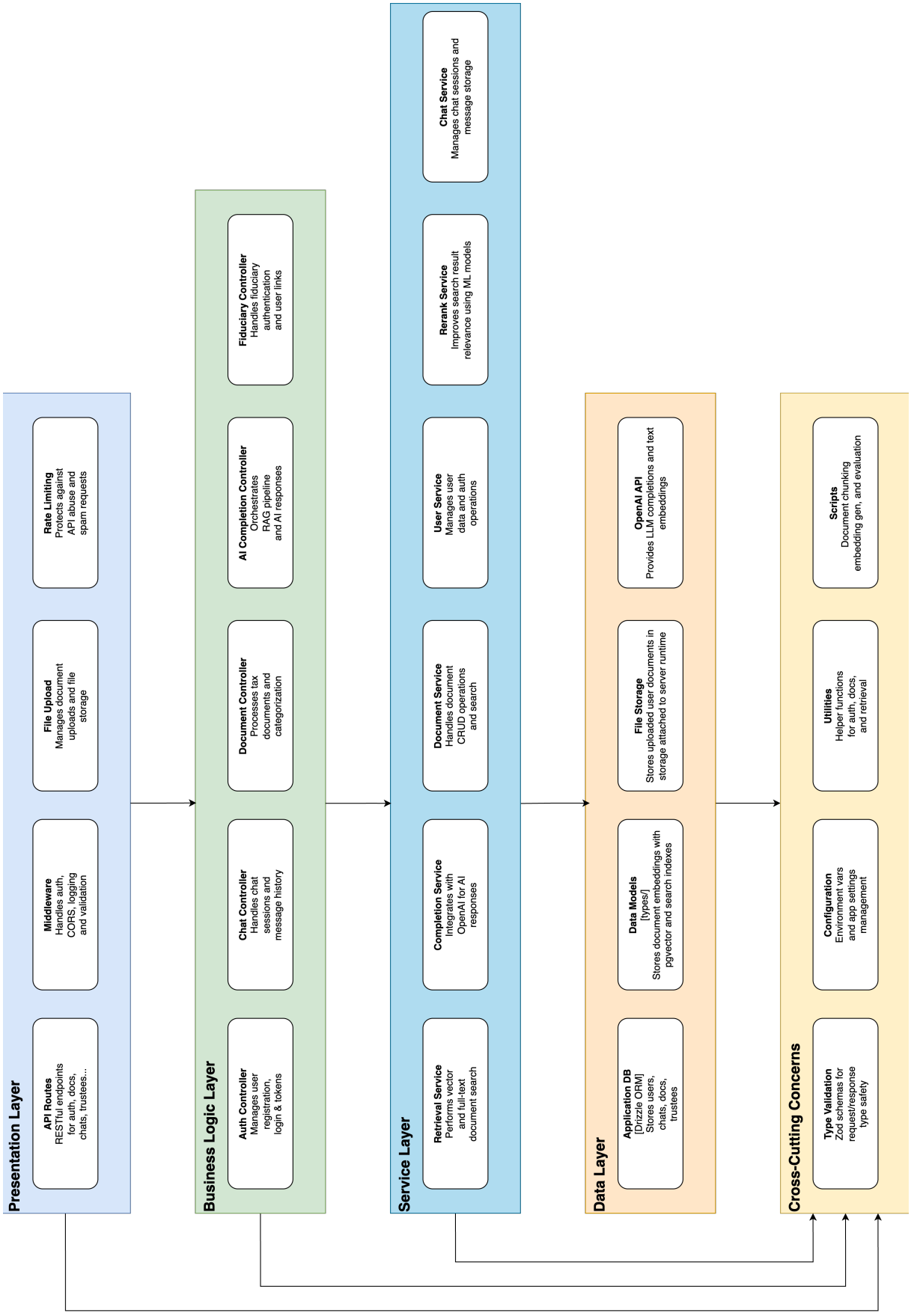



Figure 17: API Architecture

### 3.6.1. Project Directory Structure Overview

The main directories within the `src` folder include:

- **routes:** Contains all route definitions that map HTTP endpoints to controller functions, including authentication (`auth-routes.ts`), document management (`document-routes.ts`), chat functionality (`chat-routes.ts`), trustee operations (`trustee-routes.ts`), file handling (`file-routes.ts`), invite system (`invite-routes.ts`), and AI completion (`completion-routes.ts`).
- **controllers:** Houses request handlers that process HTTP requests, validate input using Zod schemas, and coordinate responses, including authentication logic, document processing, chat management, and AI completion orchestration.
- **services:** Stores business logic functions that handle core operations independently of HTTP concerns, including user management, document operations, AI retrieval, completion processing, and reranking algorithms.
- **middleware:** Contains Express middleware for cross-cutting concerns like JWT authentication (`auth-middleware.ts`), file uploads (`upload-middleware.ts`), and rate limiting (`ratelimit-middleware.ts`).
- **models:** Defines Drizzle ORM database schemas for all entities including users, documents, chats, trustees, and invites with proper relationships and constraints.
- **types:** Contains TypeScript type definitions and Zod validation schemas for different domains like authentication, documents, chats, and completion requests.
- **lib:** General-purpose utilities including authentication helpers (`auth.ts`), document processing utilities (`document-utils.ts`), LLM tools (`llm-tools.ts`), and retrieval algorithms (`retreival.ts`).

### 3.6.2. Presentation Layer

This layer handles all HTTP communication and serves as the entry point for client requests. It is built using Express.js with structured routing and middleware.

#### API Routes

RESTful endpoints organize the API into logical domains. These include:

- `/api/auth` handles user authentication with sign-in, sign-up, and session management.
- `/api/documents` manages tax document CRUD operations with categorization and search.
- `/api/invites` facilitates user invitation and connection workflows.

#### Middleware

Incoming requests are passed through Express middlewares:

- **Authentication middleware** (`auth-middleware.ts`) validates JWT tokens and distinguishes between user and trustee access levels.
- **File upload middleware** (`upload-middleware.ts`) processes multipart form data using Multer for document storage.

- **Rate limiting middleware** (`ratelimit-middleware.ts`) protects against API abuse using express-rate-limit.
- **CORS and logging** handled through `cors` and `morgan` packages for cross-origin support and request logging.

Authentication middleware includes specialized variants for user-only access (`userAuthVerification`), trustee-only access (`trusteeAuthVerification`), and combined access (`combinedAuthVerification`).

### 3.6.3. Business Logic Layer

Controllers act as the main entry point for business operations, handling request validation, service coordination, and response formatting.

#### Request Processing

Controllers in the `controllers` directory process all requests. These controllers include:

- `auth-controller.ts` handles user registration, authentication, and session management with JWT token generation.
- `document-controller.ts` processes tax document operations, including categorization, search, and metadata management.
- `invite-controller.ts` manages invitation workflows and connection establishment between users and trustees.

#### Input Validation

All controllers implement input validation using Zod schemas defined in the `types` directory. For example:

- `SignInSchema` and `SignUpSchema` in `auth.ts` validate authentication requests.
- `DocumentSchema` in `document.ts` ensures proper document metadata structure.
- `ChatMessageSchema` in `chat.ts` validates message content and formatting.

Controllers use `safeParse()` to validate request bodies and return appropriate error responses for invalid input, ensuring type safety throughout the request processing pipeline.

### 3.6.4. Service Layer

The service layer is closely tied to the business logic. It contains pure business logic functions that operate independently of HTTP concerns, which enables reusability across different contexts. Services follow functional programming patterns without class-based architecture.

#### Business Operations

Services in the `services` directory handle core application logic. These include:

- `document-service.ts` handles document persistence, categorization logic, and search operations.
- `retrieval-service.ts` performs vector similarity search and full-text search across document embeddings.
- `rerank-service.ts` improves search result relevance using machine learning models.

#### AI Integration

The completion service implements the RAG pipeline:

- **Document Assembly:** Combines vector search and full-text search results with configurable weights.
- **Reranking:** Applies ML-based relevance scoring to improve result quality.
- **Context Preparation:** Formats retrieved documents for LLM consumption with proper context management.
- **Streaming Responses:** Supports real-time AI completion streaming for improved user experience.

### 3.6.5. Data Layer

Responsible for all data persistence, retrieval, and external service integration.

#### Database Management

The application uses a dual-database approach:

- **PostgreSQL** with Drizzle ORM for relational data including users, documents, chats, trustees, and invites.
- **pgvector extension** for vector embedding storage with HNSW indexing for efficient similarity search.

Database schemas are defined in the `models` directory with proper relationships:
- `user-model.ts` defines user entities with trustee relationships and authentication data.
- `document-model.ts` handles tax documents with categorization and file associations.
- `chat-model.ts` manages conversation entities with participant tracking.

#### Vector Storage

Document embeddings are stored in a specialized schema (`vector-db/schema.ts`) optimized for similarity search:

- **Embeddings table** stores 1536-dimensional vectors with text content and metadata.
- **HNSW indexing** enables efficient cosine similarity search.
- **Full-text search** integration using PostgreSQL's built-in text search capabilities.

#### External Services

Integration with external APIs is abstracted through the data layer:

- **OpenAI API** for text completions using GPT-4.1-mini and embedding generation.
- **File system storage** for uploaded documents with configurable upload destinations.

### 3.6.6. Cross-Cutting Concerns

Functionality that spans multiple layers and provides foundational capabilities.

#### Type Safety and Validation

Comprehensive type safety is enforced throughout the application:

- **Zod schemas** in the `types` directory provide runtime validation and TypeScript type generation.
- **Environment validation** using `@t3-oss/env-core` ensures required configuration is present.
- **Database types** automatically generated from Drizzle schemas ensure type consistency.

#### Configuration Management

Centralized configuration through environment variables:

- **Database connections** for both PostgreSQL and vector database.
- **AI service configuration** including OpenAI API keys and model selection.
- **Application settings** for rate limiting, file upload paths, and JWT configuration.
- **RAG pipeline parameters** for retrieval counts, reranking settings, and search weights.

Configuration is validated at startup using Zod schemas in `env.ts` to prevent runtime errors.

#### Utilities and Tools

Supporting functionality organized in the `lib` directory:

- **Authentication utilities** (`auth.ts`) for JWT generation, password hashing, and verification.
- **Document utilities** (`document-utils.ts`) for file processing and metadata extraction.
- **LLM tools** (`llm-tools.ts`) for AI model integration and prompt management.
- **Retrieval algorithms** (`retrieval.ts`) for combining search results and relevance scoring.

#### Scripts and Processing

Specialized scripts in the `scripts` directory handle data processing:

- **Document chunking** (`chunking/chunker.ts`) splits large documents into processable segments.
- **Embedding generation** (`chunking/embed.ts`) creates vector representations of document content.
- **Evaluation frameworks** (`eval/`) for testing and improving AI response quality.

# Chapter 4

# Tools & Technologies

This chapter provides an overview of the technologies used in this thesis. All architecturally relevant decisions are elaborated on and justified in the Architecture chapter 3. General project management technologies and tools can be found in Section 9.5.

## 4.1. Mobile Application

The mobile application is developed using React Native [1] and the Expo framework [15]. React Native enables a single, shared codebase for both iOS and Android. The React Native documentation [1] recommends Expo due to its ease of use, especially for cross-platform development on both Windows and Mac. For example, it allows Windows users to deploy apps to iOS devices. Without Expo, the Mac-exclusive Xcode [16] would be required. Furthermore, Expo comes with built-in support for various dependencies, such as camera access, which simplifies dependency management. As is standard with all Expo setups, TypeScript [8] is used to enhance the standard feature set of JavaScript and improve maintainability through type safety. The development environment relies on Node.js [17] for its runtime and pnpm [18] for efficient package management. Navigation within the application is managed by Expo Router [19], which offers a file-system-based routing solution, simplifying the definition of screens and navigation paths.

### 4.1.1. Additional Libraries

- **Data Handling and Persistence:** The application does not use a local database for persistent storage. Instead, all relevant data is fetched and managed via Representational State Transfer Application Programming Interface (REST API). State management and caching are handled on the client side using React Query [13], which ensures efficient data retrieval, caching, and synchronization with the backend. This approach simplifies the architecture and reduces the complexity associated with maintaining a local database while still providing a responsive user experience through effective caching and background updates.
- **Remote State Management:** All application data is retrieved from and synchronized with a backend REST API. To manage asynchronous data fetching, caching, and background updates, the application uses React Query.
- **Data Validation:** To ensure type safety and data integrity, all responses from the backend API are validated using Zod [20]. Zod schemas define the expected structure and types of API responses, and every response is checked against these schemas before being processed in the frontend. This guarantees that only valid and expected data shapes are used throughout the application.

- **Internationalization:** To support multiple languages, the application utilizes i18n-js [21]. This setup allows for easy management of translations. Initially, the application does, however, only support German.

## 4.2. Backend API

The backend of the initial prototype was entirely powered by **Supabase** [22], a Platform as a Service (PaaS) powered by **PostgreSQL** [5]. The Supabase instance communicated directly with the React Native client. With this setup, there was no need for a dedicated backend as all server functionality is handled by the Supabase platform. For this thesis, it was decided to depart from this model and adopt a traditional client-server-database setup with a dedicated backend API. The reasons for this are:

- **Custom Postgres Hosting:** If Supabase is used in a client-to-database mode, it cannot be replaced without extensive code modifications. And while Supabase supports Switzerland as a hosting region [22], it might not be sufficient for future data protection requirements of the Finteam project. Using a regular Postgres database allows switching to any Postgres provider or self-hosted Postgres database without changing any application code.
- **Authorization Management**: The Row-Level Security (RLS) of Supabase is a powerful feature for enforcing access controls directly within the database, but it falls short compared to traditional server-side authorization. RLS policies are limited in flexibility, as they operate within SQL constraints and cannot easily accommodate complex business logic or external authorization sources. Moreover, debugging and maintaining RLS policies can be cumbersome, especially in large applications where access rules are dynamic or require conditional evaluation beyond simple SQL expressions. By handling authorization on the server side, the application retains full control over authentication, permission checks, and logging, ensuring a more scalable and auditable security model. This is particularly relevant for the Finteam domain model as there are both trustees and regular users with different access requirements.
- **Server-side Compute Workloads:** Having a dedicated backend enables offloading some computation tasks to the server. For example, it might become necessary to process uploaded documents with an external Large Language Model (LLM), which would be more cumbersome in a direct client-to-database architecture.

With this in mind, it was decided to instead use a custom backend with the technologies described in Section 4.2.1.

### 4.2.1. Libraries

Using a regular Postgres database instead of Supabase necessitates a dedicated backend, as the client can no longer access the database directly. This backend is powered by a Node.js [17] server with Express.js [3], written in TypeScript [8] for improved maintainability and type safety.

- **Database:** As already established, a Postgres database will be used. To communicate with this database, the Drizzle ORM [6] is utilized, just as it is on the client for reading the local SQLite database.

- **Authentication:** Authentication is handled directly on the dedicated API server using a username/password mechanism, with credentials securely stored in the Postgres database. This approach ensures full sovereignty over user management without the need for a hosted authentication solution. To securely provision authentication, Passport.js [23] and JWT [24] are used. Both the web and mobile clients can sign in and sign up through the backend API.
- **Bundler:** As the backend is written in TypeScript, it cannot be directly executed by Node.js. To create an executable production build, `tsc` along with `tsc-alias` [25] is used to transpile TypeScript with import aliases (such as `~/controllers/auth`) into valid JavaScript. For local development, `tsx` [26] is used to automatically refresh the development server on every code change.

## 4.3. Web Application

Trustees use the web application to manage their customers' filings. This is built as a conventional React Single Page Application (SPA) [7] without any server components. All server interaction is handled through the backend API. TypeScript [8] is used to ensure type safety within the code and therefore improve maintainability.

### 4.3.1. Additional Libraries

- **Bundling:** A React application needs to be bundled for both development and production. For this, Vite is used. Vite provides a Command Line Interface (CLI) that enables the initial setup of a React application. It also comes with TypeScript by default.
- **Routing:** As already described in the architecture chapter, the frontend is a single-page application SPA. To handle client-side routing, Wouter [14] is used. This is a minimalist routing library for React. It allows defining various routes, such as `/events` or `/login`, directly in the program code. Wouter was chosen because it is a simple and straightforward routing library. Other routing libraries come with a server component by default, which is not needed for this simple SPA.
- **State Management:** A React application is entirely based on states. To manage these globally within the application, React Query [13] is used. This enables a global state store to manage application state across multiple files.
- **Data Fetching:** To interact with the backend API, the browser's built-in Fetch API is utilized for HTTP calls. To efficiently manage asynchronous state within the application, React Query [13] is used. This state manager facilitates the synchronization of asynchronous states while also providing caching and revalidation helpers to optimize data fetching.
- **Validation:** The responses received from the backend must be validated and typed to ensure full type safety in the frontend. Zod [20] is used for this purpose. To validate user inputs in forms, React Hook Form is combined with Zod, ensuring a robust and type-safe validation process.

### 4.3.2. Hosting

Portability is the number one factor when it comes to hosting. All parts of this full-stack application can be hosted on various platforms or in a self-hosted Linux environment. For the duration of this project, both the website and the backend are hosted on Railway [27]. The platform makes it easy for developers to deploy code, which is built in an automated manner as soon as code is pushed to a GitHub repository.

Railway operates as a Platform-as-a-Service (PaaS) solution that abstracts away infrastructure complexity typically associated with application deployment. The platform provides automatic provisioning of resources, including compute instances, databases, and networking components, without requiring manual server configuration. The deployment pipeline integrates seamlessly with GitHub through webhook-based triggers, automatically initiating build processes that include dependency installation, compilation, and deployment to production environments when code changes are committed to the designated repository branch.

### 4.3.3. AI Chatbot

The chatbot relies on multiple LLM and an embedding model from OpenAI [12]. The Vercel AI SDK [28] was used to interact with the OpenAI models. The AI SDK Client library was used to implement client-side chat stream handling in the mobile app.

# Chapter 5

# Implementation

This chapter describes notable aspects of the implementation from the mobile app, web app, and API. While it is not intended to be a complete log for all aspects of the application, it does discuss all important components.

## 5.1. Mobile App UI Overview

The following figures illustrate the user interface of the mobile app to provide context for the implementation details discussed below.

The first figure on the left shows the app's start screen. It is divided into categories based on the official tax form from the Canton of Zurich and the project team's practical experience in filing taxes. To help users track their progress, tags under each category light up green when a document has been uploaded. Both categories and tags are currently developer-defined, but in the future, fiduciaries may be able to customize them on a per-user basis.

The middle figure displays a tooltip for a specific subcategory, designed to reduce confusion about which documents should be uploaded. In this example, the Swiss salary statement form is shown along with a brief description. This feature aims to reduce miscommunication between users and fiduciaries.

The last figure on the right demonstrates the document upload process from the user's perspective. Users can upload multi-page documents and attach notes to clarify specific details. These documents and notes then become visible to the connected fiduciary in their web application.

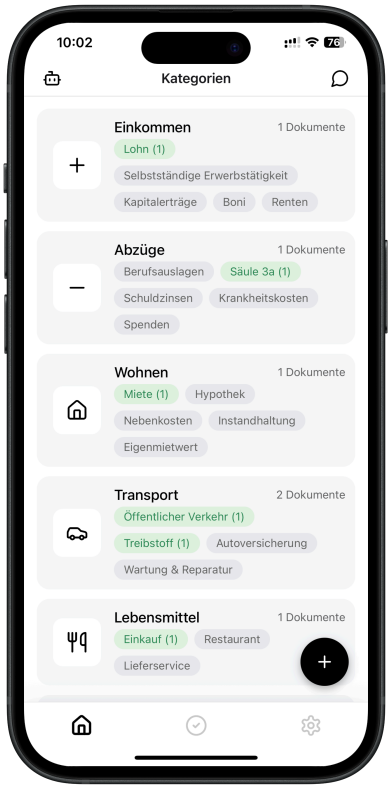



Figure 18: Mobile app start screen

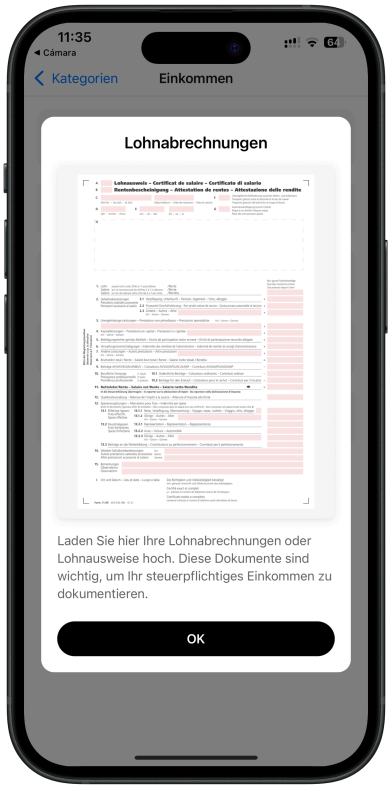



Figure 19: Mobile app tips

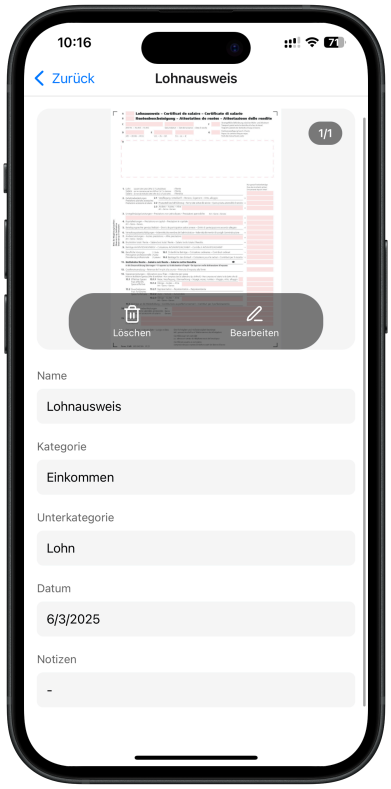



Figure 20: Mobile app document upload

## 5.2. Invite and Connection Management System

Finteam's user-fiduciary connection is invitation-based. The connection uses a mechanism that enables fiduciaries to establish secure relationships with users through randomly generated invite codes. This workflow makes sure that connections are established through explicit invitation and acceptance processes, so that no unintended pairings can occur.

### 5.2.1. Invite Flow

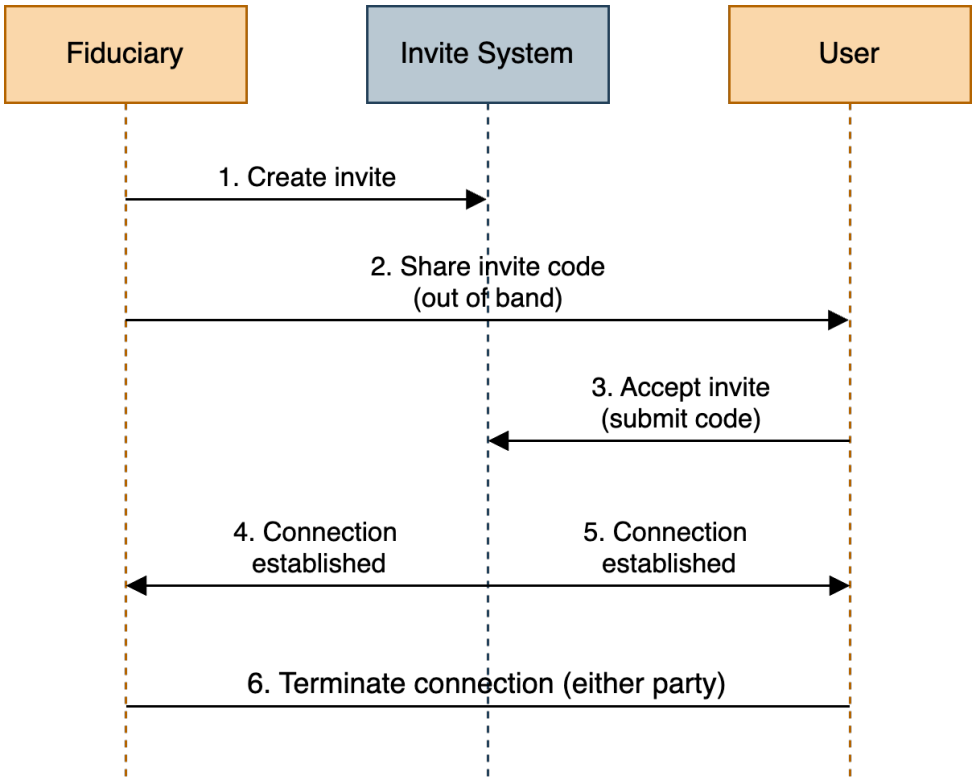



Figure 21: Invite Flow

The invitation system operates through a dedicated database table that manages the complete lifecycle of connection requests from creation through acceptance. Each invitation contains a unique invite code, expiration timestamp, and specific user targeting information.

- **code**: varchar(32)
- **trusteeId**: integer (references trusteesTable.id)
- **userName**: varchar(255), not binding but helps with identification
- **isUsed**: boolean (default false)
- **usedById**: integer (references usersTable.id)
- **expiresAt**: timestamp for expiration date

### 5.2.2. Cryptographic Code Generation

Invitation codes are generated using the Node.js cryptographic function `randomBytes(8).toString(hex)`. The system generates 16-character hexadecimal codes derived from 8 bytes of cryptographically secure random data. This is sufficient entropy to prevent brute force attacks while maintaining user-friendly code lengths.

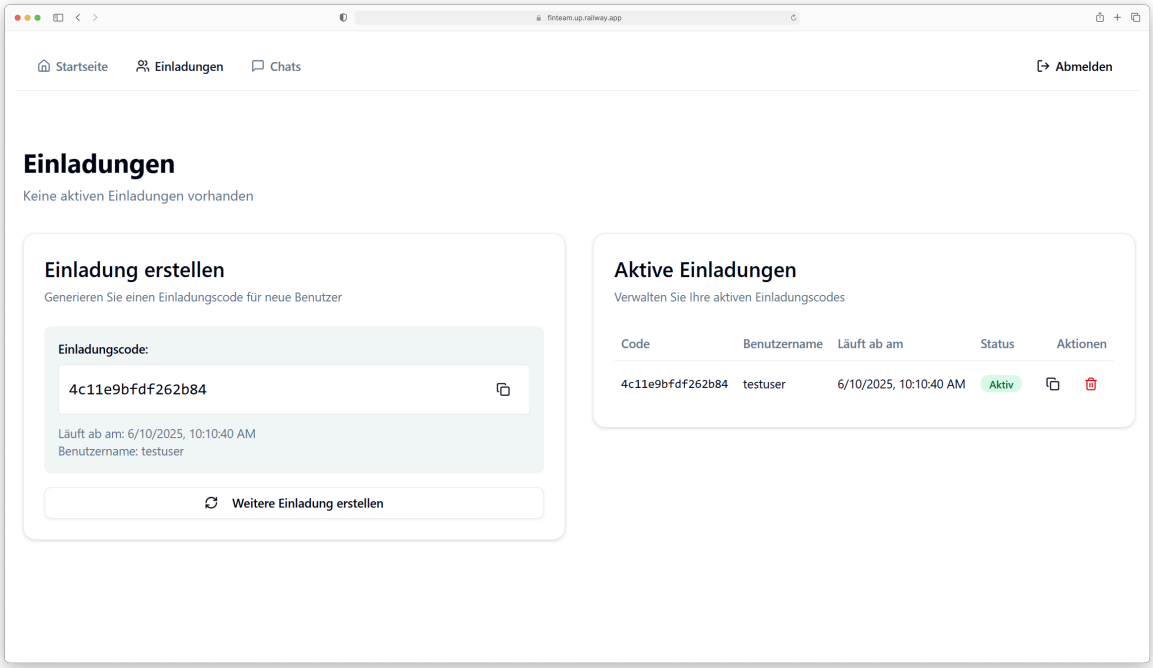



Figure 22: Web app invitation generation

To create a secure invitation code, the fiduciary presses the “Einladung erstellen” button in the web application. With that, the code is generated with the previously described method. As outlined in Figure 21, the generated code then needs to be transmitted to the user through an external communication channel of the fiduciary’s choice (e.g., email).

### 5.2.3. Connection Establishment Process

When users accept valid invitations, the system simultaneously marks the invitation as used and establishes the fiduciary relationship in a single database transaction. This transactional approach ensures data consistency. In addition to that, the system validates invitation status, expiration, and user eligibility before proceeding with the connection establishment.

In the app, this process is visualized as follows.

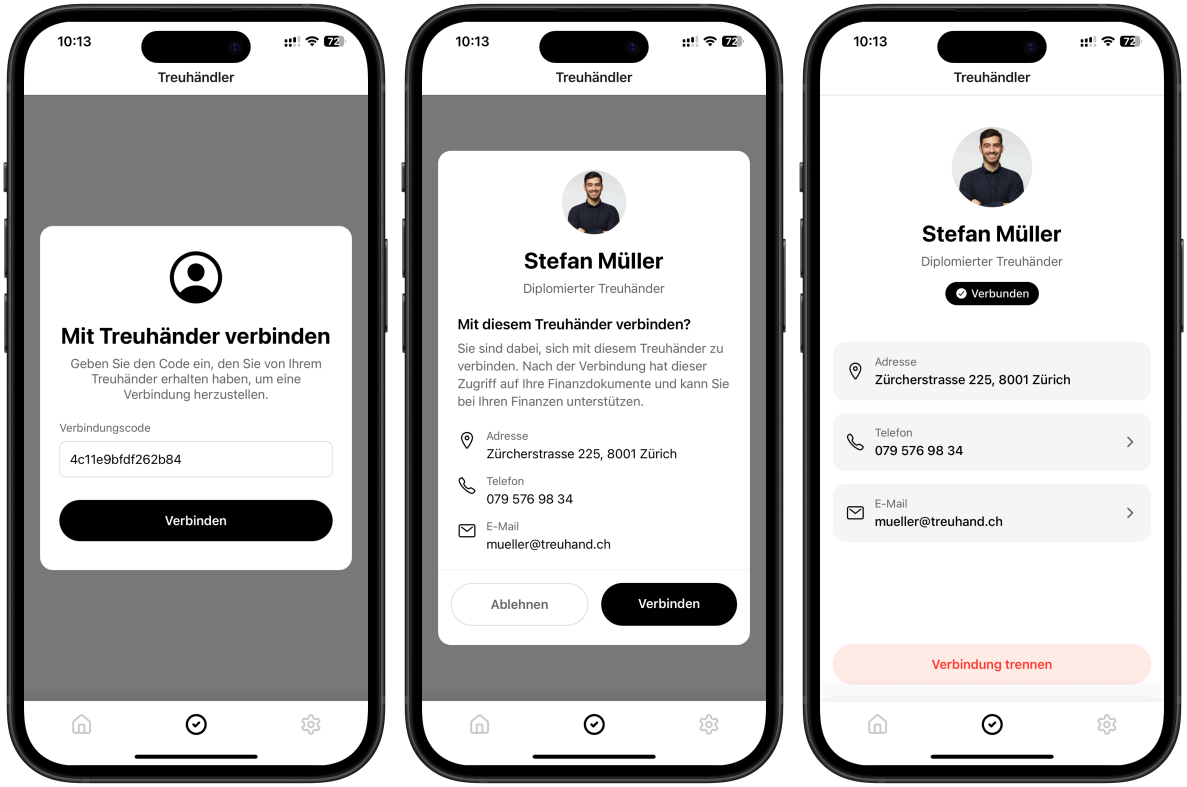



Figure 23: Mobile app fiduciary connect workflow

First, the user enters the code received from the fiduciary into the app. Then, the fiduciary's details are loaded so the user can review and verify that it is the correct fiduciary. Once accepted, the connection is established, granting the fiduciary access to the user's data.

### 5.2.4. Connection Termination and Management

The system allows both users and fiduciaries to terminate relationships through dedicated endpoints. Connection revocation immediately removes access permissions and updates all relevant database relationships. Such an endpoint can be seen in the last step of Figure 23.

Connection termination is immediate and irreversible. Once disconnected, it requires a new invitation process to re-establish relationships. This design ensures clear boundaries and prevents accidental or unauthorized relationship restoration.

## 5.3. Fiduciary Access Control System

The system implements an access control mechanism that enables fiduciaries to securely access and manage client documents while maintaining authorization boundaries. This occurs through a relationship-based permission model that makes sure fiduciaries can only access documents belonging to users who have explicitly granted them access through established connections.

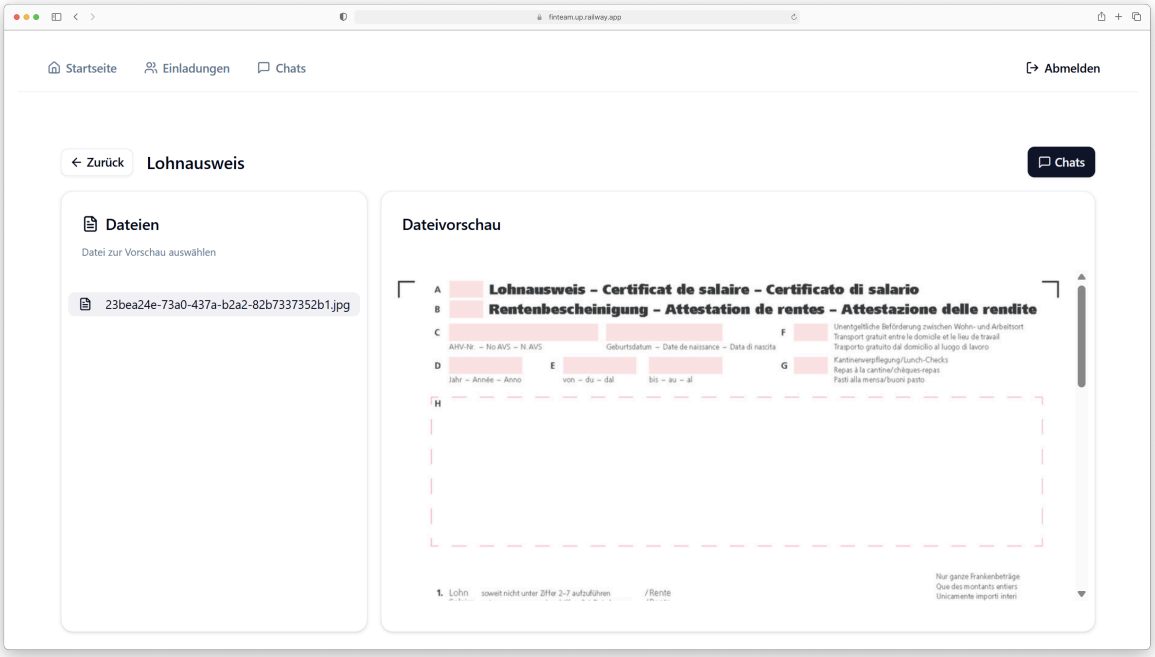



Figure 27: Web app client file access

This access system is built upon a foreign key relationship within the user model, where each user can optionally be associated with a single fiduciary through the `trusteeId` field. This one-to-many relationship allows fiduciaries to manage multiple clients. Users, however, can only be connected to a single fiduciary.

- **username**: varchar(255)
- **trusteeId**: integer (references trusteesTable.id)

### 5.3.1. Authentication and Authorization Framework

The backend API uses a dual-authentication approach that distinguishes between regular users and fiduciary professionals through JWT token claims. Fiduciary tokens include an `isTrustee` flag that enables the middleware to apply appropriate access controls and routing logic.

```
// Check if a user is connected to a trustee
export async function isUserConnectedToTrustee(userId: number, trusteeId: number) {
  const user = await db.query.usersTable.findFirst({
    where: eq(usersTable.id, userId),
    columns: {
      trusteeId: true,
    },
  });

  return user?.trusteeId === trusteeId;
}
```

### 5.3.2. Document Access Patterns

Every document access request from a fiduciary triggers a connection verification process that validates the relationship between the requesting fiduciary and the document owner. This fiduciary access pattern differs from standard user access patterns. When a fiduciary requests a document, the system first retrieves the document to identify its owner, then validates the fiduciary-user relationship before granting access.

The system performs this operation by querying the database to find the document's owner and then cross-referencing the fiduciary's permissions to ensure the connection is valid. If the document is not found or if the fiduciary does not have an established relationship with the user, access is denied.

This approach makes sure that fiduciaries cannot enumerate or access documents through direct ID manipulation, as each access attempt requires explicit relationship validation.

### 5.3.3. Security Considerations

The system enforces strict separation between user and fiduciary contexts through middleware-level authentication checks that prevent cross-usage of access permissions. Fiduciaries cannot access user-specific endpoints, and users cannot access fiduciary management functions, maintaining clear operational boundaries. Within controllers, there are additional authorization checks to make sure that users can only access data they own, and fiduciaries can only access data of connected users.

## 5.4. File Uploads

The backend implements file storage by combining local filesystem storage with PostgreSQL metadata management. This means that binary files are stored efficiently on disk while file metadata still benefits from the relational database patterns.

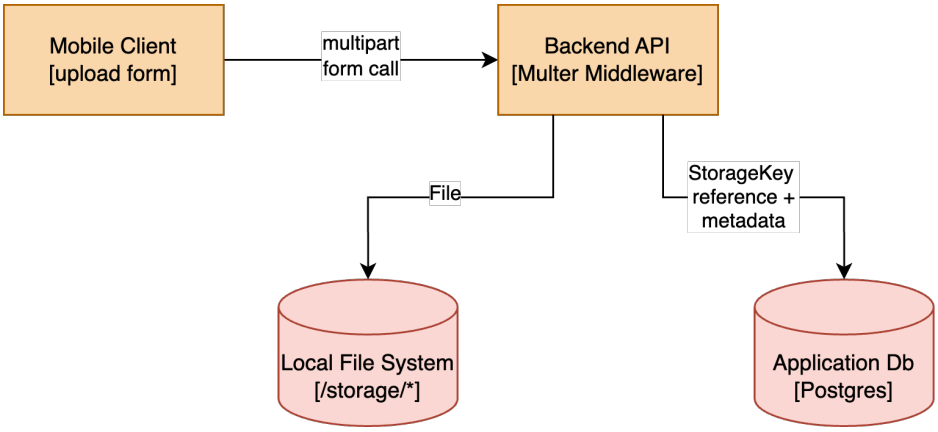



Figure 28: File Upload Workflow

### 5.4.1. Multer Middleware

File uploads are handled through a dedicated middleware layer built on Multer, a Node.js library specialized for handling multipart/form-data. This layer handles file naming through UUIDs to avoid collisions from user-provided filenames. Original filenames are preserved in the database.

```
const storage = multer.diskStorage({
  destination: (req: Request, file: Express.Multer.File, cb) => {
    cb(null, env.UPLOAD_DESTINATION);
  },
  filename: (req: Request, file: Express.Multer.File, cb) => {
    const extension = path.extname(file.originalname).toLowerCase();
    cb(null, `${randomUUID()}${extension}`);
  },
});
```

### 5.4.2. Type Validation and Constraints

As shown in Section 5.4.1, MIME type filtering restricts uploads to specific document formats. The current implementation supports PNG images, JPEG images, and PDF documents, which are all relevant formats for tax-related documents. There is a 5MB size limit per file.

### 5.4.3. Database Schema and Relationships

Document metadata is stored separately from binary files, directly in the application database. The `documentsTable` contains business logic attributes such as categories, subcategories, and user associations, while the `documentFilesTable` maintains the mapping between documents and their physical storage locations.

- **documentId**: integer (references documentsTable.id)
- **storageKey**: varchar(255) (UUID)

This structure supports multiple files per document. The storage key field contains the UUID-based filename that corresponds to the physical file location.

### 5.4.4. Access Control and Retrieval

File retrieval is secured through user authentication and ownership verification, with special provisions for fiduciary access. The system validates that users can only access files belonging to their documents or documents they are authorized to view through user-fiduciary relationships.

Served files include content headers to enable inline viewing of PDFs and images while serving original filenames instead of UUIDs in the user interface. This approach maintains the user experience while making sure that the underlying storage remains organized.

## 5.5. API Authentication System

The system implements authentication based on JSON Web Tokens (JWT) [24] and cryptographic password hashing, without the use of an external authentication provider. The authentication architecture supports role-based access patterns while maintaining a unified token-based approach for client integration.

### 5.5.1. Cryptographic Foundation

The authentication system uses cryptographic primitives for password security and token generation. Password storage uses PBKDF2 (Password-Based Key Derivation Function 2) with SHA-512 hashing. Each user password is combined with a unique 16-byte salt generated using Node.js crypto primitives. This approach ensures that identical passwords produce different hashes across users. This is robust protection against rainbow table and brute-force attacks [29].

### 5.5.2. JWT Token Architecture

The use of JWTs enables stateless session management without server-side session storage. The signed tokens contain essential user identification and role information, allowing middleware to make authorization decisions without database queries.

- **userId**: string
- **username**: string
- **isTrustee**: boolean

Tokens are signed using a configurable secret key and include automatic expiration handling with a default seven-day validity period. The optional `isTrustee` flag enables role-based routing and access control throughout the application without requiring additional database lookups.

### 5.5.3. Middleware Authentication

The authentication system provides three distinct middleware functions that handle different access patterns:

- user-only authentication
- fiduciary-only authentication
- combined authentication for shared resources

The middleware architecture enforces strict separation between user and fiduciary contexts, preventing cross-contamination of access permissions. Each middleware variant validates token authenticity, checks role appropriateness, and injects user context into the request object for downstream processing. Fiduciary tokens are explicitly rejected from user-only endpoints, while user tokens cannot access fiduciary-specific resources.

### 5.5.4. Registration and Sign-In

User registration is implemented such that each username may only exist once. Therefore, on every registration, this uniqueness is checked. For the password, the system enforces minimum security requirements including a four-character minimum for usernames and an eight-character minimum for passwords, with a 128-character maximum limit.

During sign-in, credentials are validated through constant-time comparison operations to prevent timing attacks. Successful authentication generates JWT tokens containing user identification and role information, enabling immediate access to protected resources without additional authentication steps.

### 5.5.5. Rate Limiting and Security Controls

The authentication endpoints implement rate limiting to prevent brute-force attacks and credential stuffing attempts. The system limits authentication attempts to five requests per minute per IP address. This allows a few tries for legitimate users but prevents a single actor from brute-forcing passwords.

### 5.5.6. Client Integration

The authentication system follows standard Bearer token patterns for HTTP authorization headers, ensuring compatibility with modern web and mobile development frameworks. Clients receive tokens upon successful authentication and include them in subsequent requests using the Authorization header format:

```
Authorization: Bearer YOUR_JWT_TOKEN
```

## 5.6. Web Client Authentication

The web client implements a React Context-based authentication system with localStorage persistence.

### 5.6.1. Token Storage and Persistence

The web client uses the browser's localStorage API for JWT token persistence. This allows users to maintain authenticated sessions across page reloads. This is then passed into React Context to provide global authentication state management.

To keep the session state up-to-date across tabs, the browser's storage event API is used. This ensures that authentication state changes in one browser tab are immediately reflected in all other open tabs.

```
// Listen for storage events (in case another tab changes auth state)
globalThis.addEventListener("storage", checkAuth);
```

### 5.6.2. Protected Route Implementation

The client implements route protection through a `ProtectedRoute` component that automatically redirects unauthenticated users to the login page. Protected routes monitor authentication state and redirect users when authentication is lost.

All API requests include JWT tokens through the standard `Authorization` header, using Bearer token format. API functions consistently check for token availability before making requests, ensuring fail-fast behavior when authentication is unavailable.

**Session Expiration Handling**

If the API responds with a `401 Unauthorized` error or the user manually signs out, the client automatically clears local authentication state and redirects to the sign-in route.

## 5.7. Mobile Client Authentication

The mobile client implements a React Native authentication system using AsyncStorage for token persistence and React Context for state management. This approach provides a seamless user experience across app sessions while maintaining security through JWT token management. The architecture leverages Expo Router for navigation and React Query for API state management.

### 5.7.1. Token Storage and Persistence

The mobile client utilizes React Native's AsyncStorage for JWT token persistence, enabling users to maintain authenticated sessions across app restarts. Tokens are stored using a consistent key, `AUTH_TOKEN_KEY`. AsyncStorage is a mobile API that offers secure storage, helping protect tokens against unauthorized access.

An `AuthProvider` component, using React Context, synchronizes authentication state between AsyncStorage and the UI. The context provides the token, loading status, user data, and authentication methods.

```
type AuthContextType = {
  token: string | null;
  isLoading: boolean;
  isSignedIn: boolean;
  user: any;
  signIn: (username: string, password: string) => Promise<{ success: boolean; error?:
string }>;
  signUp: (username: string, password: string) => Promise<{ success: boolean; error?:
string }>;
  signOut: () => Promise<void>;
};
```

The provider loads stored tokens on app initialization and fetches user data when a token is present, ensuring the authentication state is readily available.

### 5.7.2. Automatic User Data Fetching

User data is automatically fetched when the authentication token changes. This ensures that user context is always available for personalization and authorization without requiring manual API calls after login. This is typically handled within a `useEffect` hook in the `AuthProvider` that observes changes to the `token` state.

### 5.7.3. Navigation-Based Route Protection

Route protection is managed via Expo Router. The root layout component checks the authentication state and redirects unauthenticated users to the sign-in screen.

```
// Inside RootLayoutNav component
useEffect(() => {
  if (!isLoading && !isSignedIn) {
    router.replace("/sign-in"); // Redirect to sign-in if not authenticated
  }
}, [isSignedIn, isLoading]);
```

This ensures that protected sections of the app are not accessible without a valid session.

### 5.7.4. HTTP Request Authentication

Authenticated API requests include the JWT token in the `Authorization` header (Bearer token format). A centralized function, `createHeaders`, ensures tokens are consistently applied.

```
export function createHeaders(token?: string): Record<string, string> {
  const headers: Record<string, string> = { /* ... */ };
  if (token) {
    headers.Authorization = `Bearer ${token}`;
  }
  return headers;
}
```

### 5.7.5. Session Expiration Handling

Session expiration is handled within custom React Query hooks. A `handleUnauthorizedResponse` function detects 401 errors, triggers a sign-out, and clears the session. With this, users are gracefully logged out when their session is no longer valid.

## 5.8. AI Chatbot

The AI chatbot is built as a custom retrieval-augmented generation RAG system that uses a separate PostgreSQL database dedicated exclusively to vector embeddings and full-text search capabilities, as described in Section 3.6. The entire RAG pipeline is designed with modularity in mind, so that each component can be toggled on or off independently from the configuration. This enables detailed benchmarking of different retrieval strategies and evaluation of how individual improvements impact overall system performance.


Figure 29: Mobile app AI chat implementation

### 5.8.1. Configuration

The system combines multiple retrieval techniques in a hybrid approach. It utilizes both semantic vector search and PostgreSQL's native full-text search capabilities. Parameters such as retrieval count, reranking, query rewriting, and the weight distribution between vector and full-text search can all be configured through the configuration file `src/config.ts`.

```
export const INITIAL_RETRIEVAL_COUNT = 20; // k1
export const FINAL_CONTEXT_COUNT = 5; // k2
export const RERANKING_ENABLED = true;
export const FULL_TEXT_SEARCH_ENABLED = true;
export const QUERY_REWRITER_ENABLED = true;
```

### 5.8.2. Database

The vector database uses a dedicated PostgreSQL instance with the `pgvector` extension [30]. This is completely separate from the main application database. This maintains clean architectural boundaries between application data and chatbot data.

The database only contains the `documentEmbeddings` table, which stores document chunks with their corresponding vector embeddings, contextual information, and metadata:

- **text**: content of the chunk

- **context**: LLM-generated chunk summary, situating it within the entire document
- **embedding**: VECTOR(1536) (vector of 1536 dimensions)
- **identifier**: VARCHAR(255), allowing for segmentation

The table includes two indexes: a Hierarchical Navigable Small World Graph (HNSW) index on the embedding column using cosine distance for vector similarity search (`vector_cosine_ops`), and a Balanced Tree Index (B-tree) index on the identifier column for efficient filtering by document source.

Full-text search capabilities are implemented using PostgreSQL's native text search functionality, using the `to_tsvector` and `websearch_to_tsquery` functions. However, there is an important limitation: the full-text search is currently optimized specifically for German language content, as configured through the `FULL_TEXT_SEARCH_LANGUAGE` parameter set to "german". This language-specific optimization affects the Word Root Reduction (Stemming), Common Word Exclusion (Stop Word Filtering), and ranking algorithms.

The identifier field enables segmentation of documents within the database. It is currently set to "zh" for Zürich Wegleitung documents, and there are no other documents in the database. However, this design allows for future expansion to include tax guidance documents from other cantons. A user from Zurich, for example, would have no interest in receiving information based on Geneva tax guidance documents; therefore, segmentation is needed.

### 5.8.3. Ingestion Flow

The document ingestion process is implemented as a series of command-line scripts executed through `pnpm run parse` and `pnpm run embed`. This manual execution approach is intentional, as the current use case involves processing a single, stable document (the Zurich tax guidance document) that doesn't require automated ingestion workflows. Even if multiple other cantons were added, their respective guidance documents could be ingested manually without the need for fully automated processing.

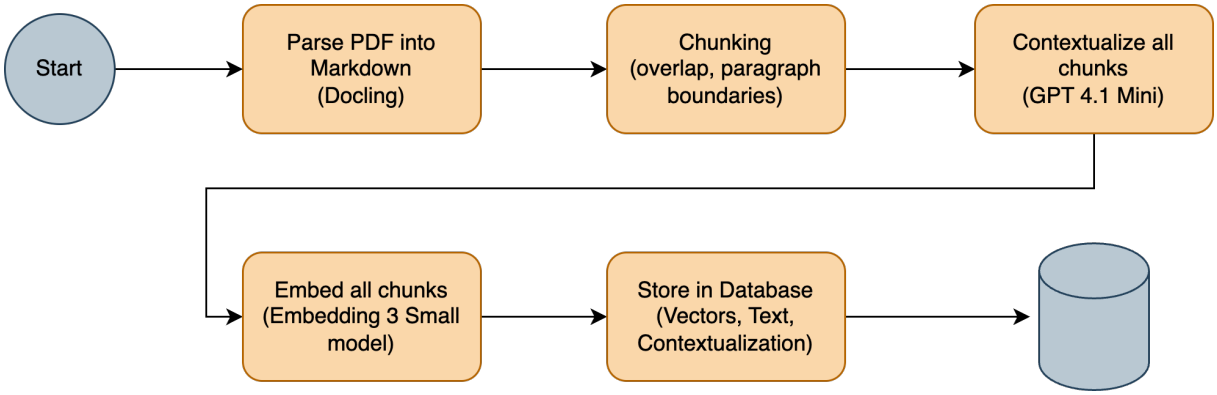



Figure 30: Document Ingestion Flow

#### 1. Parsing

The parsing phase transforms PDF documents into markdown format using Docling [31], an open-source document parsing library developed by IBM Research Zurich. Docling provides advanced PDF understanding capabilities including layout analysis, reading order detection,

and table structure recognition. Docling parsing is executed through a Docker container to ensure a consistent execution environment without the need for Python Virtual Environments.

A significant limitation of the current parsing setup is that Docling does not extract and annotate images from the PDFs. In the final markdown, they are represented by a placeholder comment. This limitation has minimal impact on the Zurich tax guidance document use case, as the images in these tax guidance documents don't contain critical textual information necessary for the AI chatbot's functionality. The vast majority of images just highlight the field of the physical tax declaration.

### 2. Chunking

The chunking implementation uses an overlapping strategy optimized specifically for large PDF documents. The chunking logic splits documents based on paragraph boundaries, which aligns well with the structured nature of tax guidance documents.

Key chunking parameters are configured through the centralized config system:

- Maximum chunk size: 1,800 characters (`MAX_CHUNK_SIZE_CHARS`)
- Minimum chunk size: 200 characters (`MIN_CHUNK_SIZE_CHARS`)
- Overlap size: 300 characters (`CHUNK_OVERLAP_CHARS`)

This chunking algorithm is inspired by research from Chroma, balancing between optimal performance for long-form, high-density PDFs and simplicity [32]. The chunking algorithm ensures that content at chunk boundaries is preserved through the overlap mechanism, preventing information loss stemming from arbitrary split points. This is in addition to the paragraph-based splitting strategy, which also respects the document's semantic structure, avoiding mid-sentence breaks that could compromise context.

### 3. Contextualization

Each chunk undergoes AI-powered contextualization using GPT-4.1 Mini with the full document content as context [33]. This is made possible by the 1M token context window capability of this model, which the entire tax guidance document fits into. This contextualization helps each chunk gain an understanding of the full document context. This approach has been studied by Anthropic and has been shown to significantly increase retrieval accuracy. This implementation follows the concept outlined by Anthropic [34]. How the contextualization implementation described here affects this system is discussed in Chapter 6.

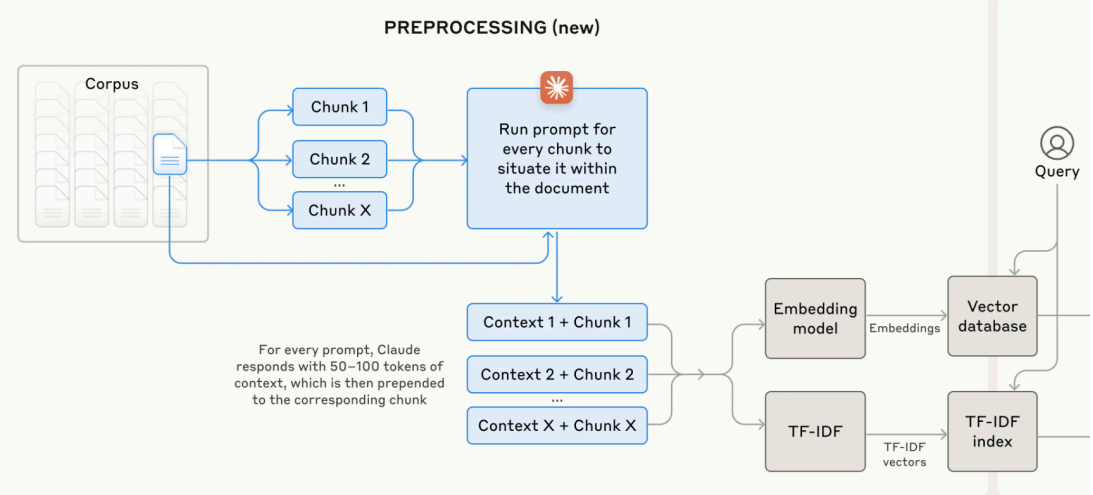



Figure 31: Contextualization pipeline. Source: Anthropic [35].

The contextualization prompt specifically instructs the model to:

1. Provide a concise summary of the chunk's content
2. Generate synonyms for key phrases to improve search retrieval
3. Respond in German to maintain language consistency (this would have to be changed to French or Italian to support other regions)

```
const systemPrompt = `You must give a short, succinct context to situate this chunk
within the overall document for the purposes of improving search retrieval of the chunk.
Your response should include:
1. A concise summary of the chunk's content
2. Synonyms for key phrases or concepts found in the chunk (to improve search retrieval).
Just give the synonyms.

- Answer only with the succinct context and nothing else.
- You MUST write your response in German.`;
```

The main benefits of this approach are: it improves semantic search accuracy by providing additional context clues, enhances keyword matching through synonym generation, and maintains document coherence by situating each chunk within the broader document structure. The processing is performed in parallel batches of 10 chunks to optimize API usage while maintaining reasonable processing speed. [35], [36]

### 4. Embedding

The final step generates vector embeddings for both the original chunk content and the AI-generated contextualization. The combined text (`${chunk.text} ${chunk.context}`) is embedded using OpenAI's `text-embedding-3-small` model. It outputs 1536-dimensional vectors.

The embeddings are stored in the PostgreSQL vector database alongside the original text, context, and metadata. This enables the hybrid retrieval system to perform both semantic vector search and traditional full-text search operations on the same content [30].

## 5.8.4. Retrieval

The retrieval system implements a multi-stage approach that begins with basic semantic search and progressively applies more advanced techniques, if enabled in the configuration. At its core, the system generates vector embeddings from user queries and performs similarity searches against the pre-computed document embeddings stored in the PostgreSQL vector database.

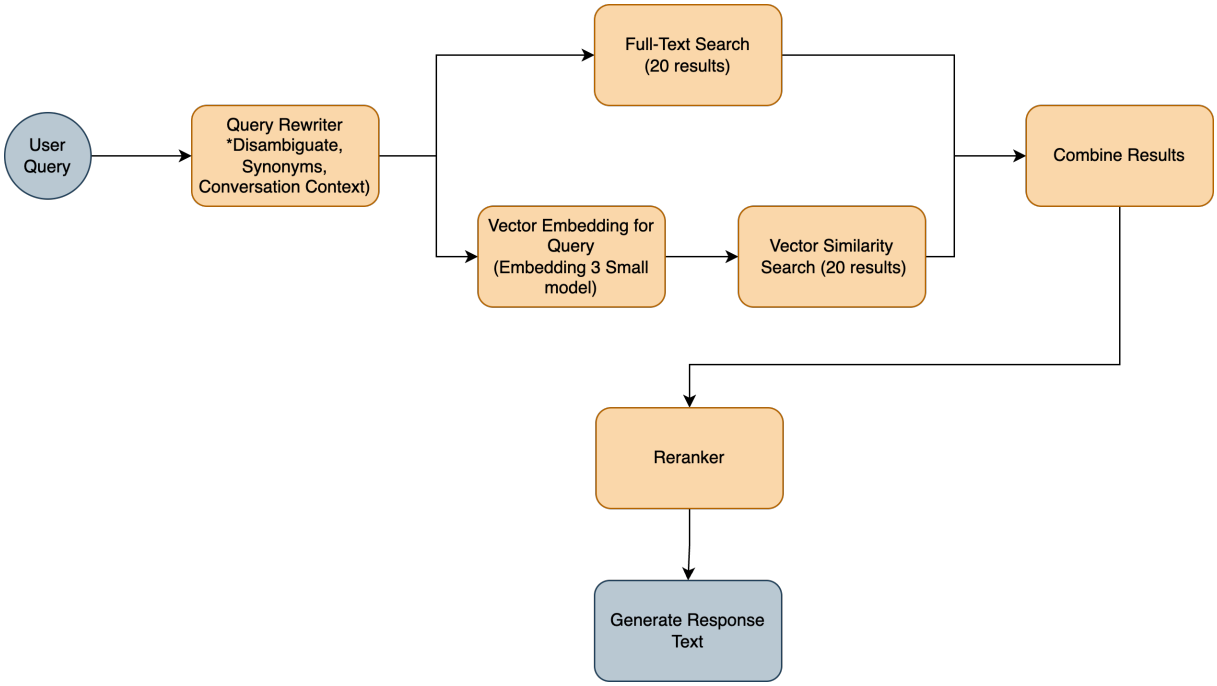



Figure 32: Complete Retrieval Flow

The basic system starts by embedding the user question using OpenAI's `text-embedding-3-small` model. This query embedding is then compared against all document embeddings in the database using cosine similarity:

```
const similarity = sql<number>`1 - (${cosineDistance(documentEmbeddings.embedding,
queryEmbedding)})`;
```

The system applies a configurable similarity threshold (default 0.3) to filter out irrelevant results and returns documents ordered by similarity score. The vector search uses the HNSW index for fast approximate nearest neighbor search.

### Hybrid Retrieval

The system uses an early fusion approach to combine vector and full-text search. This hybrid strategy utilizes the strengths of both retrieval methods: vector search excels at semantic similarity and conceptual matching, while full-text search provides precise keyword matching and handles exact terminology queries.

The retrieval configuration operates with a two-stage approach:

- **Initial Retrieval (k1)**: 20 results retrieved from each method (`INITIAL_RETRIEVAL_COUNT`)
- **Final Selection (k2)**: 5 results selected for LLM context (`FINAL_CONTEXT_COUNT`)

- **Hybrid Weighting**: 70% vector search, 30% full-text search (`VECTOR_SEARCH_WEIGHT = 0.7`)

When both search methods are enabled, the system executes parallel queries and combines results using score normalization. The `combineSearchResults` function performs min-max normalization on both similarity and full-text scores, then computes a weighted hybrid score:

```
const hybridScore = alpha * vectorScore + (1 - alpha) * fullTextScore;
```

The system handles document deduplication when the same document appears in both result sets, merging their scores and marking the retrieval source as "both." This early fusion approach ensures that documents benefiting from strong performance in either retrieval method are properly prioritized.

### Reranker

The reranking system performs intelligent relevance filtering using GPT 4.1 Nano, OpenAI's fastest and most cost-effective model. It assesses document relevance on a scale of 0-10 and applies a binary relevance filter to eliminate irrelevant documents before ranking. Each document is individually assessed in parallel, so that multiple candidates can be assessed simultaneously. This LLM-based reranking addresses the limitations of pure similarity scoring by understanding query context and document content.

The reranker processes each retrieved document individually, providing the model with:

- The original user query
- Document text and contextual information
- Retrieval source and confidence scores
- A binary relevance decision (relevant/not relevant)
- A numerical relevance score (0-10 scale)

```
const { object } = await generateObject({
  model: openai("gpt-4.1-nano"),
  system: "Determine how relevant the document is to the user query on a scale of 0-10,
where 10 is extremely relevant and 0 is not relevant at all.",
  schema: z.object({
    relevanceScore: z.number().min(0).max(10),
    isRelevant: z.boolean(),
  }),
});
```

Documents marked as irrelevant are filtered out. Relevant documents are ranked by their LLM-assigned relevance score. In cases of tied relevance scores, the system falls back to hybrid scores or original similarity scores for secondary ranking. The final selection respects the `FINAL_CONTEXT_COUNT` limit, so that the context fed into the LLM generation stage is limited.

The reranking feature can be toggled via `RERANKING_ENABLED`. When disabled, the system only selects the top K documents based on their hybrid or similarity scores. This enables consistent system behavior regardless of the reranking setting, so that the same final document is used no matter the current configuration.

#### Query Rewriter

The query rewriter tries to improve retrieval effectiveness through query reformulation using the entire conversation context. When enabled, the LLM uses the `ragSearchTool` to analyze user input and conversation history and generate an optimized search query.

The query rewriter handles several retrieval challenges:

- **Conversational Context**: Resolves pronouns and implicit references using conversation history
- **Ambiguity Resolution**: Converts vague queries into specific, keyword-rich search terms
- **Synonym Generation**: Adds alternative terminology and synonyms
- **Technical Precision**: Maintains domain-specific terminology while improving searchability

In detail, the rewriting tool calls are structured as follows:

1. **Step 0**: Forced tool execution to generate an optimized query and perform the search with it
2. **Step 1**: Regular completion with retrieved context

```
experimental_prepareStep: async ({ stepNumber }: { stepNumber: number }) => {
  if (stepNumber === 0) {
    return {
      toolChoice: { type: "tool" as const, toolName: "ragSearch" as const },
      experimental_activeTools: ["ragSearch"] as ["ragSearch"],
    };
  }
  if (stepNumber === 1) {
    return {
      experimental_activeTools: [],
    };
  }
}
```

### 5.8.5. Chat

The chat system consists of the retrieval pipeline and a GPT 4.1 Mini model instructed to act as a Swiss tax assistant. This model generates the final response for the user. The system prompt defines this persona:

```
system: `You are a knowledgeable Swiss tax assistant that helps answer questions about
filing taxes (Steuererklärung) in the canton of Zurich, Switzerland.
Answer concisely and accurately based ONLY on the context provided. Do not give out
information which does not exist in the context.
Do NOT give a disclaimer about not being a tax professional since you just repeat
information from the context which is an official document.
Never reference the document which you got the information from (Bad: "siehe Dokument
5").
If the context doesn't contain relevant information to answer the tax-related question,
admit that you don't know and suggest consulting with a qualified tax professional or
the local tax authority.
Do not make up information or use knowledge outside of the provided context. Answer in
the language of the question, which might differ from the language of the context.
```

```
Important: Make it clear that you provide general information only and not personalized
tax advice.`
```

The prompt instructs the model to provide definitive answers based on official documentation. The completion service formats retrieved documents for optimal LLM comprehension using the `formatDocumentsToStringForLLMContext` function:

```
function assembleMessagesWithContext(messages: Message[], documents:
RetrievedDocument[]) {
  const context = formatDocumentsToStringForLLMContext(documents);
  return [
    {
      role: "system" as const,
      content: `Context from documents:\n\n${context}`,
    },
    ...messages,
  ];
}
```

Documents are passed in a structured format with clear delineation:

```
[DOCUMENT 1]
Text: [document content]
Context: [AI-generated chunk context]

[DOCUMENT 2]
Text: [document content]
Context: [AI-generated chunk context]
```

The system handles edge cases, including scenarios where no relevant documents are found. In that case, it provides a fallback system message instructing the model to acknowledge knowledge limitations.

### Streaming

The completion service implements chat features through two distinct approaches: traditional REST API responses and HTTP streaming responses. In production environments, only the streaming endpoint is utilized, while the non-streaming variant serves primarily as a development and testing tool for simpler debugging scenarios.

The streaming implementation uses the Vercel AI SDK's `streamText` function and `pipeDataStreamToResponse` method to transfer data from the server to clients [28]. When a user submits a query through the `/chat-stream` endpoint, the system processes the request identically to the non-streaming variant but returns responses as incremental chunks rather than waiting for complete generation.

```
// Generates a streaming completion response and pipes it to the Express response
export async function generateCompletionStream(
  messages: Message[],
  documentIdentifier: string | undefined,
  response: Response
) {
  const completionData = await prepareCompletionData(messages, documentIdentifier);
```

```
  const result = streamText(completionData);
  return result.pipeDataStreamToResponse(response);
}
```

The `pipeDataStreamToResponse` method handles the complex HTTP streaming protocol, setting appropriate headers (`Content-Type: text/plain; charset=utf-8`) and managing chunk boundaries. This creates a standardized data stream format that includes not only text deltas but also metadata such as token usage, tool calls, and completion status.

The streaming format follows the AI SDK's data stream protocol, which is designed to be compatible with client-side libraries like `@ai-sdk/react` with the `useChat` hook. This standardization means that mobile applications using the corresponding React Native or Expo libraries can consume these streams without custom parsing logic. The `useChat` hook handles stream parsing, message reconstruction, and state management.

The end result is that users see responses flow in in real time and won't have to wait until the entire response has been generated. However, there is still an initial delay until the RAG pipeline has completed, since the response won't start generating before completion of the retrieval.

## 5.9. AI Chat Evaluation

In order to assess the efficacy of the different retrieval optimizations (full-text search, query rewriter, reranker), an automated evaluation pipeline had to be developed. This evaluation framework makes it possible to benchmark different RAG configurations.

The evaluation system combines multiple assessment approaches: LLM-as-a-Judge evaluation, semantic similarity scoring, and performance latency measurement. This multi-faceted approach ensures that improvements are measured not only in terms of answer quality but also system responsiveness.

### 5.9.1. Synthetic Test Data Generation

The evaluation framework uses a synthetically generated dataset. This is because it would have taken a lot of time to manually generate a large question-answer dataset. Therefore, the generation process uses an LLM to create the question-answer evaluation corpus. It processes the full Zurich tax guidance document through a pipeline that chunks the content into 4,000-character segments with 400-character overlaps. For each document chunk, GPT 4.1 Mini generates between 2-5 realistic question-answer pairs based only on the content within that chunk. The generation prompt specifically instructs the model to create questions that typical Swiss taxpayers would ask. It is tasked to avoid overly technical or obscure queries that don't reflect real-world usage patterns:

```
const prompt = `You are an expert question generator for RAG evaluation.
The topic is the tax code in the Swiss Canton of Zurich. Users are citizens
looking for information when filing their tax return. Your question-answer
pairs should match what such a user might ask.

IMPORTANT GUIDELINES:
1. Focus on questions a real user would ask
```

```
2. Questions must be answerable using ONLY the information in the provided text
3. Answers must be direct quotes or closely paraphrased from the text
4. Avoid trivial or simple yes/no questions
5. Each answer should be concise but complete`;
```

After generating the questions, a deduplication mechanism using embedding-based similarity detection is run. Each generated question is embedded using OpenAI's `text-embedding-3-small` model, and cosine similarity calculations identify near-duplicate questions above a 0.9 threshold. This ensures a diverse dataset and prevents evaluation bias from repetitive content, which could occur due to the overlapping chunks.

After running this pipeline, the final synthetic dataset contains 157 unique question-answer pairs. This scale enables robust evaluation of retrieval optimizations.

## 5.9.2. Evaluation Metrics

The evaluation framework uses a dual-judge approach that combines objective semantic similarity measurements with subjective quality assessments from an LLM judge. The setup is based on information found in [37], [38].

### LLM Judge Evaluation

The primary evaluation mechanism utilizes OpenAI's o4 Mini model as a judge to assess answer quality across four dimensions. The LLM judge evaluates each model response against the ground truth answer using a 1-5 scale:

- **Correctness (1-5)**: Factual accuracy of the information compared to the ground truth.
- **Helpfulness (1-5)**: How directly and comprehensively the answer addresses the user's question.
- **Clarity (1-5)**: Answer clarity, phrasing quality, and appropriate conciseness.
- **Overall Quality (1-5)**: Holistic assessment considering all factors above.

The evaluation prompt instructs the model to provide objective assessments while including explanatory reasoning for each score.

```
const SYSTEM_PROMPT = `You are an objective evaluator for question-answering systems.
Evaluate the model's answer on the following criteria on a scale from 1-5:
- Correctness: Is the information factually correct according to the ground truth?
- Helpfulness: Does the answer directly and comprehensively address the user's question?
- Clarity: Is the answer clear, well-phrased, and appropriately concise?
- Overall Quality: Overall quality score considering all above factors.

Provide a brief explanation justifying your scores.`;
```

### Semantic Similarity Scoring

In addition to the LLM judge, the system implements embedding-based semantic similarity evaluation using cosine similarity between ground truth and model answer embeddings. Both texts are embedded using OpenAI's `text-embedding-3-small` model, which produces 1536-dimensional vectors.

The cosine similarity score ranges from 0 to 1, with values closer to 1 indicating higher semantic alignment between the model answer and ground truth. This metric is more objective and reproducible than the subjective LLM evaluation, so they complement each other well.

#### Performance Metrics

The evaluation framework tracks response latency for each question. It measures the complete pipeline from query input to final answer generation. This measurement covers all stages, including network time from the laptop to the server, database access, retrieval, reranking overhead, and LLM generation latency. This comes with the caveat that the network communication overhead from the benchmark client to the backend server is also included. However, this is considered negligible compared to the time required for LLM generation, which dominates the overall response time.

Latency measurements enable evaluation of the practical implications of each optimization: while reranking may improve answer quality, it introduces additional processing time that affects user experience. By measuring both quality and performance, the evaluation can track these tradeoffs.

### 5.9.3. Evaluation Runner

During evaluation, each configuration undergoes 5 complete evaluation runs. Each with all 157 synthetic quesitons, in random order. This makes it possible to provide statistically significant results even with the non-deterministic nature of LLM responses.

For reference, these are the 8 options evaluated:

1. **Base**: Vector search only (baseline)
2. **Full Text**: Vector + full-text search hybrid
3. **Rerank**: Vector search + LLM reranking
4. **Rewriter**: Vector search + query rewriting
5. **Full Text + Rerank**: Hybrid search + reranking
6. **Full Text + Rewriter**: Hybrid search + query rewriting
7. **Rerank + Rewriter**: Vector search + reranking + query rewriting
8. **Full Text + Rerank + Rewriter**: All optimizations enabled

#### Concurrency and Rate Limiting

The evaluation pipeline uses concurrency to balance execution speed with API rate limits. Questions are processed in batches of 15 concurrent evaluations with 250ms throttling between requests. This aims to prevent being rate limited by OpenAI while maintaining reasonable evaluation throughput.

The system processes both model answer generation and LLM judge evaluation in parallel. Error handling ensure that individual question failures don't compromise entire evaluation runs.

### Result Summarization

For each metric, the system calculates mean scores and standard deviations across the 5 evaluation runs. The system calculates a composite "Total Combined Score" that normalizes and weights all evaluation metrics:

```
const normalizedSimilarity = 1 + avgRunScores.similarity * 4;
const totalScore =
  (avgRunScores.correctness +
    avgRunScores.helpfulness +
    avgRunScores.clarity +
    avgRunScores.overallQuality +
    normalizedSimilarity) /
  5;
```

This metric serves as a single performance indicator that balances semantic similarity with LLM judge assessments. This way, the different configurations can be compared against each other through a single value.

## 5.10. Deployment

The entire application stack is hosted on the PaaS cloud provider Railway [27]. The server location is `eu/amsterdam` as they do not offer Switzerland as a location. This location is acceptable for the MVP phase, as no real user data is involved. For a production setup, a server location in Switzerland would have to be considered. As discussed in Chapter 3, the entire application is built in a provider-agnostic way so that moving it across hosting solutions is straightforward.

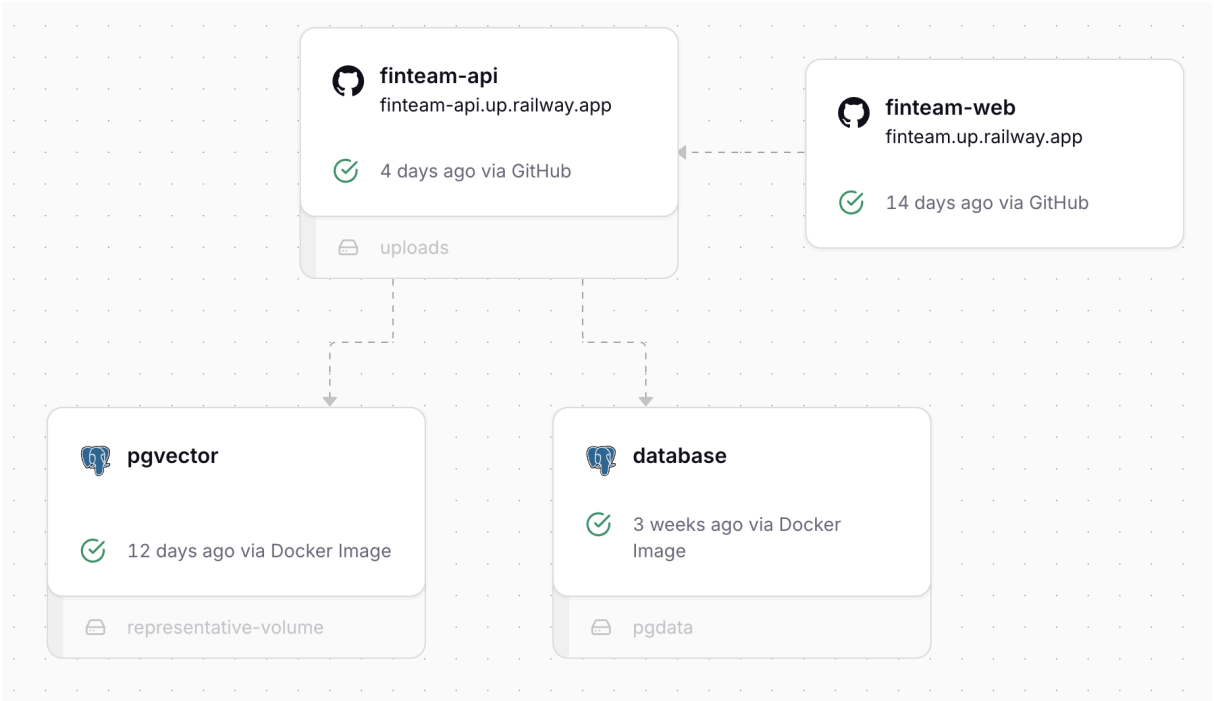



Figure 33: Railway Screenshot: Hosting Setup

### 5.10.1. Containers

As shown in Figure 33, the hosting consists of 4 components, each running as its own Docker container [39].

- **finteam-api**: Hosts the Express.js API backend. To store the tax documents that users upload, a 5GB volume is mounted to the server as `/app/data`. The application can read and write to this volume to access documents. The average idle memory utilization for the backend is 89 MB.
- **finteam-web**: Hosts the static React Vite frontend through a Caddy server. There is no compute involved other than serving the static assets. The average idle memory utilization is 21 MB.
- **database**: PostgreSQL database that hosts all user data, other than the binary files uploaded by users. The average idle memory utilization is 25 MB.
- **pgvector**: PostgreSQL database with the `pgvector` extension enabled. It solely serves the chunked document for retrieval. The average idle memory utilization is 26 MB.

### 5.10.2. Build system

The build system uses Nixpacks, a tool designed to simplify deployment and environment consistency by generating Docker build environments by automatically detecting the tech stack used by the codebase. In most cases, this results in developers not having to write any configuration to get the project hosted [40].

The deployment process is fully automated via GitHub integrations: whenever a developer pushes changes to the main branch of either the frontend or backend repository, a GitHub Action is triggered. This action initiates the build process using Nixpacks, compiles the code, runs tests, and upon successful completion, deploys the updated application to the hosting platform. This workflow enables continuous delivery with minimal manual intervention and maintains up-to-date production environments with every commit.

### 5.10.3. Environment Variables and Secrets

The backend API has multiple environment variables provided through Railway's secure secret management system:

- `DATABASE_URL` (application DB)
- `VECTOR_DATABASE_URL` (vector DB)
- `JWT_SECRET`
- `OPENAI_API_KEY`
- `UPLOAD_DESTINATION` (path to attached volume)

# Chapter 6

# Results

## 6.1. AI Chat Evaluation Results

As described in Chapter 5.9, the evaluation involved a comprehensive assessment of 157 questions, each tested across 5 separate runs and 8 unique system configurations. This structure enabled a robust comparison of various retrieval and optimization methods—including vector search, full-text search, reranking, and query rewriting—under consistent conditions. The primary aim was to determine which configurations provide the best balance between answer quality, semantic similarity, and response latency, and to assess the variability across multiple test iterations. Below are the results from this evaluation:

| Configuration | LLM Judge Avg | Semantic Sim | Latency (ms) | Combined Score |
|---|---|---|---|---|
| **Vector Only** | 3.96 ± 0.016 | 0.847 ± 0.003 | 4605 ± 239 | 4.04 ± 0.021 |
| **Full Text** | 3.97 ± 0.015 | 0.849 ± 0.003 | 3933 ± 20 | 4.06 ± 0.023 |
| **Rerank** | 4.04 ± 0.032 | 0.846 ± 0.003 | 6348 ± 1108 | 4.11 ± 0.049 |
| **Rewriter** | 3.35 ± 0.020 | 0.796 ± 0.003 | 5489 ± 164 | 3.52 ± 0.030 |
| **Full Text + Rerank** | 4.03 ± 0.019 | 0.850 ± 0.003 | 6997 ± 1769 | 4.11 ± 0.029 |
| **Full Text + Rewriter** | 3.42 ± 0.015 | 0.797 ± 0.003 | 5038 ± 139 | 3.58 ± 0.021 |
| **Rerank + Rewriter** | 3.53 ± 0.018 | 0.803 ± 0.003 | 6630 ± 396 | 3.67 ± 0.025 |
| **Full Text + Rerank + Rewriter** | 3.52 ± 0.036 | 0.802 ± 0.004 | 5893 ± 158 | 3.66 ± 0.055 |

Table 3: AI chat evaluation results

Overall, the results demonstrate that combining multiple retrieval enhancements generally improves answer quality, albeit with increasing latency trade-offs. For example, the **Full Text + Rerank** configuration achieved the highest combined score, but with variability similar to that of **Rerank**, suggesting both configurations perform comparably when considering their spread. Pure vector search yielded the fastest response times but lower overall quality, while adding full-text search produced a modest improvement in semantic similarity and judge scores with only a small latency increase. Reranking introduced a larger latency penalty but significantly boosted clarity and overall quality, while query rewriting alone provided minimal gains except when paired with reranking. These findings, along with the variability seen in standard deviations,

will be explored further in the following sections, where each configuration's contributions and trade-offs will be analyzed in detail.

### 6.1.1. Charts

**LLM Judge vs Sematic Similarity**

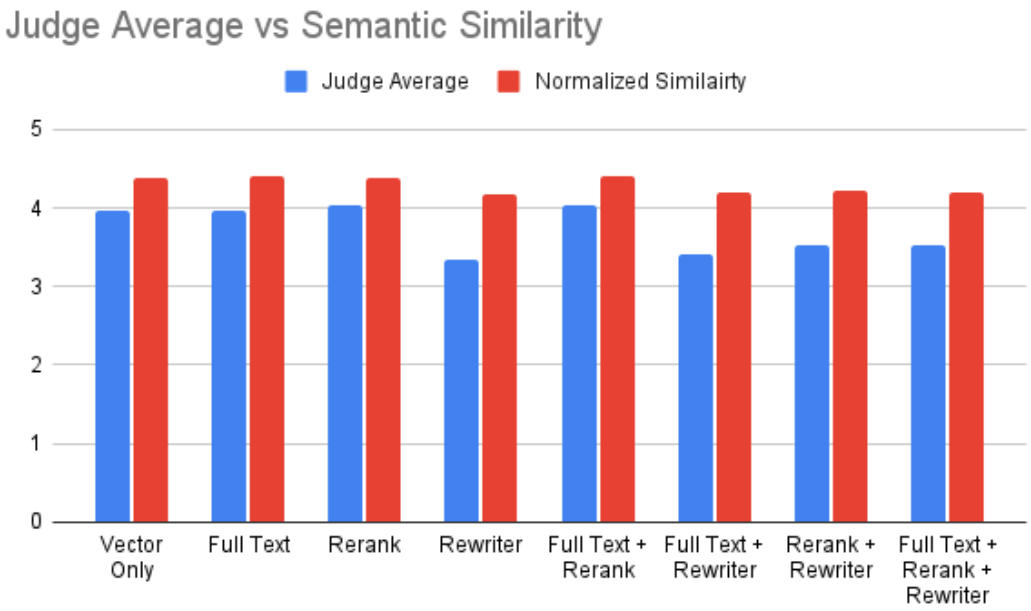



Figure 34: LLM Judge vs Semantic Similarity Normalized

Figure 34 reveals a pattern where **semantic similarity** consistently remains high across configurations, but **judge scores** exhibit greater variability, particularly for `Rewriter` and `Rerank + Rewriter`. This suggests that while automated metrics align closely across methods, human evaluators discern more nuanced differences in content quality.

**Latency Bar Chart**

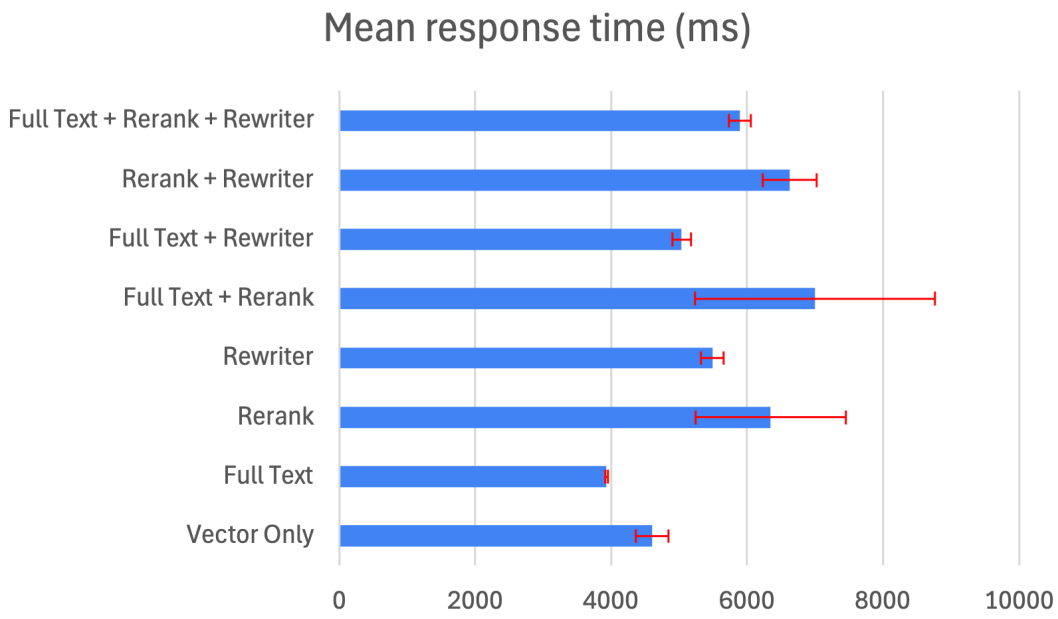



Figure 35: Response time analysis

Figure 35 illustrates that **response times** increase predictably with added retrieval features, but the variability across runs (as shown by blue error bars) varies dramatically. For instance, **Rerank** and **Full Text + Rerank** exhibit not only the highest mean latencies (6348 ms and 6997 ms, respectively) but also the widest variability (SD ≈ 1108 ms and 1769 ms) shown in red, indicating performance inconsistency under load. In contrast, configurations such as **Full Text** and **Vector Only** display both lower mean latencies and much tighter standard deviations (e.g., **Full Text** with 3933 ms ± 20 ms), suggesting predictable, reliable response times. This pattern emphasizes a trade-off: while adding features like reranking can improve quality, they introduce greater variability and latency costs, potentially challenging for real-time systems.

**Latency vs Score**

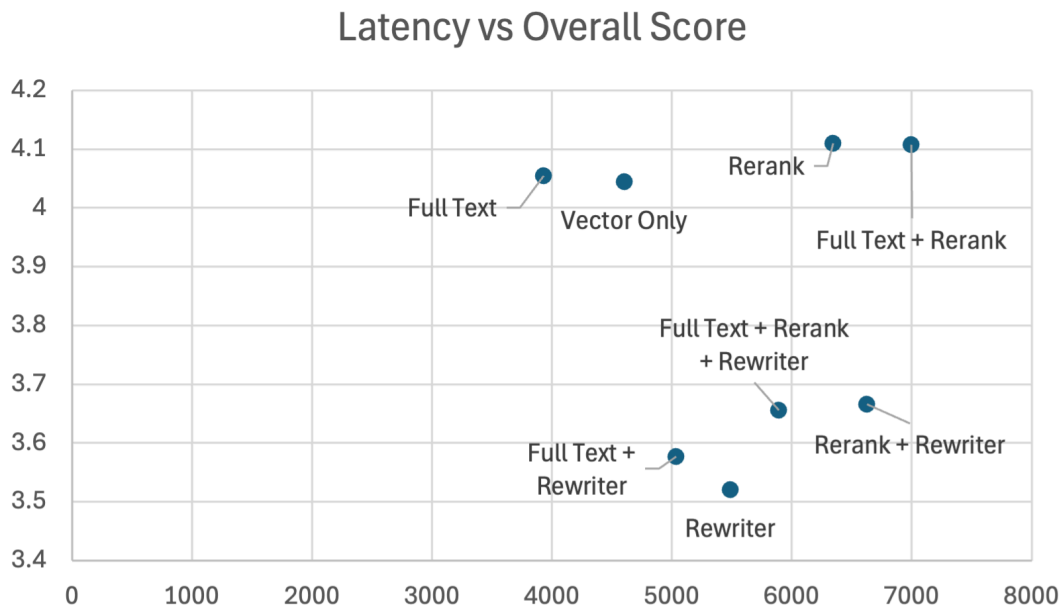



Figure 36: Latency vs Overall Score

Figure 36 offers valuable insights into the trade-offs between system performance and quality as perceived by the LLM judge. The scatter plot reveals that configurations like **Vector Only** and **Full Text** achieve both lower latencies (4605 ms and 3933 ms, respectively) and high combined scores (4.04 and 4.05). Conversely, **Full Text + Rerank** and **Rerank** show higher latencies (6997 ms and 6348 ms) but slightly improved scores (4.11 and 4.10), reflecting the additional processing required for reranking.

Interestingly, **Rewriter** and **Full Text + Rewriter** exhibit a noticeable drop in scores (3.52 and 3.58) despite moderate latency values (5489 ms and 5038 ms). This suggests that while rewriting may increase processing time, it does not guarantee higher perceived quality. The variability in **Rerank + Rewriter** (latency 6630 ms, score 3.67) further supports this observation.

Overall, the plot illustrates that incorporating advanced features such as **Reranking** can lead to higher overall scores, indicating improved semantic relevance. However, configurations that combine multiple enhancements (e.g., **Rerank + Rewriter** or **Full Text + Rerank + Rewriter**) often result in increased latency without a proportional gain in quality. This suggests that beyond a certain point, additional complexity may not yield meaningful improvements. When choosing a configuration for production, it's important to consider both response time and retrieval effectiveness to ensure a balanced and efficient user experience.

### 6.1.2. Recommendation

Based on the quantitative evaluation, the **Rerank** configuration achieves the highest **Total Combined Score** of **4.1098**. Its detailed judge metrics are:

- **Correctness**: 4.04

- **Helpfulness**: 4.04
- **Clarity**: 4.04
- **Overall Quality**: 4.04
- **Semantic Similarity**: 0.8468

In contrast, the baseline **Vector Only** configuration shows a combined score of **4.0446** with a mean latency of **4605 ms**. The **Rerank** setup incurs a mean latency of **6348 ms**—an increase of **1743 ms** (≈38% longer)—but delivers a **+0.0652** improvement in combined score.

While the **Rerank** configuration introduces a noticeable latency penalty, this extra processing time may be acceptable in scenarios where answer quality is paramount. For use cases such as tax-related inquiries or situations where users expect the most accurate, contextually relevant results, the improved correctness and clarity justify the increased response time. However, if the application must support strict real-time requirements (e.g., live chat with sub-second feedback), one might consider the **Full Text** configuration instead, since it already outperforms the baseline in both speed (3933 ms) and combined score (4.0546), offering a lightweight trade-off when minimal latency is crucial.

In summary, for applications prioritizing **maximum answer quality**, adopt the **Rerank** configuration. If **predictable low latency** is equally critical, choose **Full Text** as a balanced alternative.

# Chapter 7

# Conclusion and Further Work

## 7.1. New Achievements

This project successfully delivered a fully functioning MVP of Finteam, a cross-platform application that bridges the gap between fiduciaries and their clients. Among the key accomplishments:

- **Mobile app**: Developed using React Native and Expo, the mobile application enables clients to effortlessly upload tax documents, communicate directly with their trustee via chat, and manage their fiduciary relationship. This streamlined experience helps users maintain an organized overview of their submissions while minimizing uncertainty about which documents are required.
- **Web application for fiduciaries**: Built with React, Tailwind CSS, and React Query, the web application offers fiduciaries a clear overview of submitted documents, efficient client management, and integrated communication tools. By reducing client confusion and the need for clarifying calls or emails, fiduciaries can spend less time on administrative overhead and more time focusing on their core responsibilities.
- **Backend API**: Engineered with Express.js and TypeScript, the backend features a layered, modular architecture designed for scalability, maintainability, and ease of testing. Type safety and rigorous input validation are ensured through Zod schemas, enabling robust and predictable API contracts.
- **RAG-based AI Chatbot**: A sophisticated Retrieval-Augmented Generation (RAG) pipeline was designed and integrated to provide precise, context-aware responses to tax-related queries. Multiple retrieval strategies—including hybrid vector and keyword search, reranking, and query rewriting—were evaluated to optimize answer accuracy and relevance based on real-world Swiss tax guidance documents.

These milestones enabled the creation of a solid foundation that can be used with minimal changes in production.

## 7.2. Challenges and Unresolved Issues

The implementation and evaluation phases of the RAG system were quite time-consuming and challenging. Ultimately, the true value of the AI chat feature will only become apparent once it is deployed in a real production environment, as the responses generated by large language models can sometimes be unpredictable.

Regarding the fiduciary onboarding process, it remains manual at this stage due to security concerns. This manual approach ensures that every fiduciary can be thoroughly checked and

reviewed. Developing a scalable and secure onboarding interface is still pending and will be necessary for future growth.

Finally, there were limitations on the mobile platform side. The iOS deployment and review process could not be validated because of the lack of access to the Apple Developer Program, which restricted testing and certification on Apple devices.

## 7.3. Future Directions

One important step is the development of a secure, self-service onboarding process for fiduciaries. By implementing a document-verified onboarding flow, fiduciaries can register and be validated without requiring manual intervention from administrators. This will significantly reduce operational overhead and enable the platform to scale more efficiently.

Additionally, an administrative interface is needed to provide oversight and control. This dashboard would allow admins to manage fiduciary accounts, monitor platform activity, and resolve exceptional cases as they arise. It would also support auditing and compliance tasks by offering visibility into system interactions and user behavior.

Another major improvement involves the automatic pre-filling of tax forms. Since many tax-related documents follow standard formats or contain recurring structures, it is feasible to extract relevant data from uploaded files and populate tax forms directly. This feature would streamline the process further, reducing manual effort for both clients and fiduciaries and minimizing the risk of data entry errors.

Another improvement that could be implemented is end-to-end encryption for all user documents. By encrypting documents both in transit and at rest and ensuring only authorized parties can decrypt and access them, the platform would significantly enhance data privacy and security. This measure is especially critical given the sensitive nature of tax and financial documents, and it would help build trust with users by ensuring their information is protected at all times.

## 7.4. Final Reflection

The Finteam project demonstrated the feasibility and value of optimizing tax-related communication through a well-designed digital solution. It combined best practices in modern full-stack development, AI integration, and mobile-first design. Project members successfully collaborated using agile principles, version control workflows, and CI/CD pipelines while exploring cutting-edge AI technologies.

Although some compromises were necessary to meet the given timeframe and constraints, the resulting MVP forms a solid technical and conceptual foundation for a real product. This work not only provided practical experience across the entire software stack, but also deepened the team's understanding of secure system design, AI evaluation, and the challenges of building scalable platforms.

# Part II

# Project Management

# Chapter 8

# Quality Measures

## 8.1. Git Workflow

In this section, the Git workflow across all repositories of this project is described.

### 8.1.1. Polyrepo

The project is structured as a polyrepo setup, with separate repositories for the mobile application, web application, and API. This separation facilitates clear code organization, autonomy between modules, and simplified access control. The reasons for this configuration are explained in more detail under Section 15.3.8.

### 8.1.2. Workflow

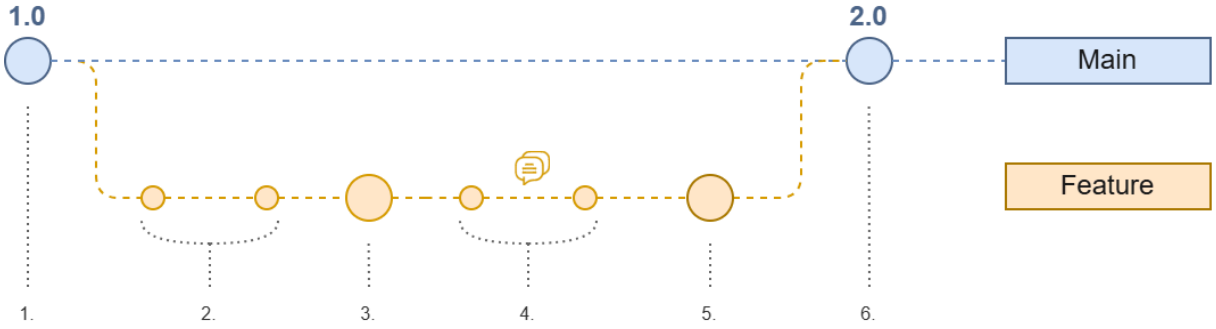



Figure 37: Git Workflow

1. Create a new branch using the pre-defined branch name of the Linear task.
2. Develop the feature as described in the Linear task and commit new changes.
3. Open a pull request on GitHub from the feature branch to the `main` branch.
4. Assign the pull request to the other team member for review and commit new changes on request.
5. Push final changes.
6. Merge the `feature` branch into the `main` branch.

## 8.2. Code Quality

### 8.2.1. Linter and Prettier

To ensure a consistent coding style and reduce formatting errors across all three repositories (web, mobile and API), ESLint and Prettier were integrated into the development workflow. ESLint is configured with modern settings and extends recommended rulesets from `@eslint/js` and `@typescript-eslint`. This is complemented by selected custom rules to enforce best practices while allowing necessary flexibility. Prettier handles automatic code formatting based on clearly defined style guidelines, including specific import ordering.

### 8.2.2. Static Analysis

The project uses the TypeScript compiler (tsc) [8] for static analysis across all codebases. It enforces strict type safety by enabling settings such as strict mode. These settings help identify potential type inconsistencies and runtime errors during development. By integrating TypeScript's static typing system, the project maintains high code quality and robustness in all components of the application.

#### 8.2.2.1. GitHub Actions

A comprehensive Continuous Integration (CI) pipeline was implemented across all three repositories using GitHub Actions, ensuring code quality and consistency throughout the development process. The workflow automation provides multiple layers of validation that execute automatically on every push and pull request, creating a robust quality gate before code integration.

The CI pipeline consists of several critical stages that run in parallel to optimize execution time:

- **TypeScript Type Checking**: The `tsc` compiler performs static type analysis without emitting JavaScript files, catching type errors and ensuring TypeScript compliance across the entire codebase. This step is particularly crucial given the project's emphasis on type safety and prevents runtime errors caused by type mismatches.
- **Build Process Validation**: Each repository undergoes a complete build process to verify that all dependencies resolve correctly and the application can be successfully compiled. This step catches integration issues, missing dependencies, and build configuration problems before they reach production environments.
- **ESLint Static Analysis**: Comprehensive linting rules enforce code style consistency and identify potential bugs, security vulnerabilities, and maintainability issues. The ESLint configuration includes TypeScript-specific rules, React hooks dependencies validation, and custom rules aligned with the project's coding standards.

The workflow configuration ensures that all checks must pass before code can be merged into the main branch, maintaining high code quality standards and preventing the introduction of breaking changes. Failed CI runs provide detailed feedback to developers, including specific line numbers and error descriptions, facilitating rapid issue resolution.

The standardized CI pipeline across all three repositories (backend API, frontend web application, and mobile application) ensures consistent quality standards regardless of the technology stack, while the parallel execution of checks minimizes the time developers spend waiting for validation results.

#### 8.2.2.2. Railway Auto Deploy

The backend API and static website employ automated deployment pipelines through Railway's integrated deployment platform, enabling continuous delivery from the development environment to production. This automation eliminates manual deployment steps and reduces the potential for human error during the release process.

**Backend API Deployment**: The Node.js Express server automatically deploys to Railway's cloud infrastructure following successful completion of the GitHub Actions CI pipeline. The deployment process includes:

- Automatic dependency installation using pnpm
- Environment variable configuration for production settings
- Database migration execution for schema updates
- Health check validation to ensure successful deployment
- Zero-downtime deployment through Railway's blue-green deployment strategy

**Static Website Deployment**: The frontend React application has an automated build and deployment process, with Railway handling the static file serving and CDN distribution. The deployment pipeline includes build optimization, asset minification, and cache invalidation to ensure users receive the latest application version.

The Railway integration monitors the main branch of both repositories, triggering deployments automatically when new commits are pushed after passing CI validation. This approach ensures that the production environment always reflects the latest stable code while maintaining deployment consistency and reliability.

**Mobile Application Deployment Strategy**: Unlike the web-based components, the mobile application requires manual submission processes to Apple App Store and Google Play Store, which involve platform-specific review processes and compliance checks. For this reason, no automated deployment workflow was implemented for the mobile repository.

### 8.2.3. Unit Tests

Vitest [41] is used as the testing framework for the Node.js Express API due to its modern architecture and TypeScript-first approach.

#### 8.2.3.1. Test Objects

Unit tests focus specifically on crucial business logic within the API, particularly complex algorithms and data processing functions that require validation of their correctness. The testing strategy deliberately excludes frontend components, as the application architecture separates

business logic from presentation concerns, with the frontend primarily handling user interface interactions rather than complex computational tasks.

A prime example is the retrieval system's `combineSearchResults` function, which implements a sophisticated hybrid search algorithm combining vector similarity and full-text search results. This function requires comprehensive testing due to its critical role in the application's search functionality and the complexity of its scoring algorithm. The tests validate multiple scenarios including:

- **Result combination logic**: Ensuring proper merging of vector and full-text search results
- **Hybrid scoring algorithm**: Verifying the weighted combination of similarity scores
- **Edge case handling**: Testing behavior with empty inputs, single-source results, and various alpha parameter values
- **Sorting correctness**: Confirming results are properly ordered by computed hybrid scores

The test suite demonstrates thorough coverage of the function's behavior across different input combinations and parameter configurations, ensuring the hybrid search algorithm performs correctly under all expected conditions.

#### 8.2.3.2. Test Implementation and Type Safety

All tests are implemented with strict TypeScript compliance, ensuring type safety throughout the testing process. During development, TypeScript compiler errors in test files are addressed immediately to maintain code quality standards. For instance, the retrieval tests required fixes for potential undefined object access and adherence to linting rules regarding number formatting.

The testing approach emphasizes readable assertions and comprehensive scenario coverage. Each test case includes descriptive names and detailed comments explaining the expected behavior, making the test suite serve as both validation and documentation of the system's functionality.

#### 8.2.3.3. Test Coverage and Strategy

While the current test coverage is selective rather than comprehensive, this approach reflects a pragmatic testing strategy focused on high-value, high-risk components. The emphasis on testing crucial business logic rather than achieving arbitrary coverage metrics ensures that testing efforts are concentrated where they provide the most value in terms of preventing critical failures.

The testing strategy recognizes that not all code requires the same level of testing investment. Simple CRUD operations, basic routing logic, and straightforward data transformations are considered lower risk and are primarily validated through integration testing and manual testing during development. This approach allows for more thorough testing of complex algorithms and business logic that would be difficult to debug if errors occurred in production.

### 8.2.4. End-to-End Tests

End-to-end tests validate the complete system workflow from the user interface through to the backend data services, ensuring that all components interact correctly under realistic conditions. A **manual test flow** was executed to simulate real user actions across both web and mobile clients.

#### 8.2.4.1. Test Flow

**User Onboarding**
- Launch mobile app and register
- Create an account
- Login

**Document Upload and Processing**
- Access mobile app; initiate camera scan of a sample document
- Preview captured image; submit for upload
- Verify document appears in web application with correct metadata

**Fiduciary Connection**
- In web application, generate connection code for a user
- In mobile app, enter code and confirm link establishment
- Validate that both parties see shared document list

**Data Retrieval and Display**
- Navigate through document categories in web client
- Open individual document entries; confirm correct rendering

**Session Termination and Logout**
- Perform logout on web client.
- Repeat logout on mobile client.

### 8.2.5. Protocol for Functional Requirements

Here, the functional requirements listed in Section 15.1 are validated to see if they have been met.

| ID | Name | Status | Explanation |
|---|---|---|---|
| FR-001 | Setup Backend | Fulfilled | |
| FR-002 | Configure Database | Fulfilled | |
| FR-003 | Configure File System | Fulfilled | |
| FR-004 | Setup Mobile App Framework | Fulfilled | |
| FR-005 | Setup Web App | Fulfilled | |
| FR-006 | Setup Lint and Format Rules | Fulfilled | |
| FR-007 | Create User Account | Fulfilled | |
| FR-008 | User Account Login | Fulfilled | |
| FR-009 | Update Profile Information | Not fulfilled | Not necessary since fiduciary holds data, could be added later on. |
| FR-010 | Account Security | Fulfilled | |
| FR-011 | User Scans Documents with Camera | Fulfilled | |

Table 4: Protocol Functional Requirements (Part 1)

| ID | Name | Status | Explanation |
|---|---|---|---|
| FR-012 | User Deletes Document | Fulfilled | |
| FR-013 | User Categorizes Documents | Fulfilled | |
| FR-014 | Fiduciary Invites User to Connect | Partially fulfilled | To avoid usage of third party email services the fiduciary has to send the code to the user. |
| FR-015 | Fiduciary Views Connected Users List | Fulfilled | |
| FR-016 | Fiduciary Revokes User Access | Fulfilled | |
| FR-017 | User Revokes Fiduciary Access | Fulfilled | |
| FR-018 | Fiduciary Accesses User Data | Fulfilled | |
| FR-019 | User Notifies Fiduciary Documents Ready | Fulfilled | |
| FR-020 | Fiduciary Requests More Information | Fulfilled | |
| FR-021 | User Asks Fiduciary Questions | Fulfilled | |
| FR-022 | User Asks Fiduciary Questions | Fulfilled | |

Table 5: Protocol Functional Requirements (Part 2)

### 8.2.6. Protocol for Non-functional Requirements

| ID | Name | Status |
|---|---|---|
| NFR-01 | Usability - Ease of Use | Fulfilled |
| NFR-02 | Maintainability & Extensibility - Clean Architecture | Fulfilled |
| NFR-03 | Performance Efficiency - Responsiveness | Fulfilled |
| NFR-04 | Performance Efficiency - Capacity | Fulfilled |
| NFR-05 | Compatibility - Browser Compatibility | Fulfilled |
| NFR-06 | Security - Data Integrity and Confidentiality | Fulfilled |
| NFR-07 | Cost Optimization | Fulfilled |
| NFR-08 | Portability - Vendor Independence | Fulfilled |

Table 6: Protocol Non-functional Requirements

**NFR-1: Usability**

A usability test was conducted for both the mobile and web apps, with 2 end users each. Each participant had to complete the same series of events.

**User tasks:**

In the mobile app, tested on an iPhone 12 Pro through the Expo Go simulator.

- Register
- Connect to fiduciary with code
- Scan and upload a new document
- Find 2 specific user documents
- Send chat message to fiducary

**Fiduciary tasks:**

In the web app, tested on a MacBook Pro 14 inch.

- Sign in
- Issue user connection invite code
- Find 2 specific user documents
- Send chat message referencing a document

#### Outcome

All sessions were completed in under 5 minutes with more than 90% of the tasks successful. Participants most struggled with finding specific documents in the mobile application. **Note:** This test was conducted to gain a quick overview of whether the application is logically structured and to obtain new ideas for possible changes to the design. To conduct a genuine usability study, more people would need to be surveyed.

### NFR-2: Maintainability & Extensibility

The architecture of all three codebases are documented in detail in Chapter 3. The meeting minutes show that all features were developed and completed within the allocated timeframe.

### NFR-3: Performance

This benchmark was done through the web app, but it relies on the backend API. Therefore it is relevant to both the web and mobile app. All benchmarks were conducted using Google Chrome Browser Dev Tools. Each measurement was taken 5 times. The numbers shown here are the average. The page loading time was measured at 1.2 seconds, which includes the duration until all necessary API calls to fetch and display event data were completed. Mutations (POST or DELETE calls to the server) averaged 780 milliseconds. It is important to note that these benchmarks do not account for variabilities such as the latency between a user's home internet connection and the server, which can vary significantly for different users. All results fulfill the requirements.

### NFR-4: Capacity

This benchmark was done directly against the backend API. Therefore it is relevant to both the web and mobile app. This test was conducted using the wrk HTTP benchmarking tool [42]. It was run on the GET documents endpoint with the JWT token of an authenticated user. At the target concurrency of 50 parallel requests, the system achieved a throughput of 105.9 requests per second, exceeding the required minimum. The average latency recorded at this load was 330 ms, within the acceptable limit of 1 second.

### NFR-5: Browser Compatibility

As the development browser for the web application, Google Chrome was used. Additionally, every member of the team works with Google Chrome, which is why our web application has already been extensively tested. Same goes for the screen size of 12 to 15 inches, as both team members use screens in that size range. This is relevant only to the web app.

### NFR-6: Security

Data in transit is encrypted using Transport Layer Security (TLS) protocols across all network communication. Access to user data within the application is controlled through a role-based access control (RBAC) [43] system. User authentication is enforced via the use of JSON Web

Tokens (JWTs). Upon successful login, a JWT is issued to the user, which is then used to authenticate subsequent requests to protected resources.

The system was developed with a focus on resisting common web vulnerabilities as outlined by OWASP [44]. Input validation and sanitization is applied to user-provided data to mitigate risks such as SQL injection and cross-site scripting (XSS). Tthe sign-in endpoints are rate-limited to prevent brute-force attacks. Regular code reviews and adherence to secure coding practices were employed throughout the development lifecycle to identify and address potential security flaws proactively.

#### NFR-7: Cost

The infrastructure on Railway is able to automatically scale vertically in order to minimize resource usage. During this project with no production usage the hosting cost amounted to approximately 2 US-Dollars per month. This makes the project inexpensive to host, fulfilling the requirement.

#### NFR-8: Portability

The system is designed to facilitate deployment across various environments. The application is running in Docker containers on Railway, which allows for consistent deployment on different hosting providers or in a self-hosted environment with minimal configuration changes. No vendor-specific code was written. Data is stored in standard PostgreSQL databases, and local file storage is utilized for document management. This avoids the reliance on vendor-specific solutions and makes data sovereignty possible.

# Chapter 9

# General Project Management

## 9.1. Roles and Team Organization

Noah Fleischmann and Dominik Rüegg are equal collaborators in the project. There are no specialized roles. Both are responsible for all aspects of the project.

Furthermore, there is the following stakeholder:

- **OST Advisor**: Marco Lehmann

## 9.2. Method

The project will follow an agile methodology, with the work divided into 2-week cycles. At the end of each cycle, the team reviews the progress made and plans the tasks for the upcoming cycle.

Due to the agile and exploratory nature of the project, it is not feasible to define all tasks from the outset. Instead, a set of clear milestones is outlined in the next section. This approach ensures the project stays on track while allowing for flexibility and iterative development.

The team operates entirely remotely, with members located in Switzerland, Mexico, and Taiwan. Collaboration across three different time zones requires effective asynchronous communication and careful coordination.

## 9.3. Project Plan

The project runs dynamically and without predefined phases. Key milestones are set in advance at meetings to ensure progress stays on track. These milestones are integrated into the linear task management system, with each task being assigned a specific milestone for completion.

## 9.4. Collaboration Workflow

### 9.4.1. Review Meetings

The review meetings with the advisor are scheduled every other Wednesday from 15:30 to 16:30 (UTC +1). Before the meeting, the project team shares the current version of the project documentation, Linear task tracker, and code base.

The purpose of the meeting is to present the current progress, address potential issues, and receive valuable feedback.

### 9.4.2. Team Meetings

Following the review meeting on Wednesday, an internal team meeting takes place.

The meeting agenda is:

- Revisiting advisor feedback and turning it into actionable tasks
- Reviewing the last cycle
- Planning the next cycle
- Addressing potential problems

Progress throughout the cycle will be shared through chat. Additional meetings for collaboration will take place whenever deemed necessary.

## 9.5. General Project Technologies

### 9.5.1. Linear

Linear [45] task management is utilized to organize project workflows. Tasks are categorized, prioritized, and assigned specific milestones to ensure timely completion and alignment with project goals.

### 9.5.2. Microsoft Teams

The project communication and file sharing have been conducted through Microsoft Teams.

### 9.5.3. Typst

The documentation is written in Typst [46]. It is a typesetting system similar to LaTeX, using a markup-based syntax to create structured and formatted documents. Like LaTeX, it allows for precise control over the layout and formatting of text, but with a first-party web-based editor that enables collaborative real-time editing.

### 9.5.4. Time tracking

Each team member used their preferred time tracking tool. At the end of each week, the recorded hours were consolidated and manually entered into a table in timetracking to maintain a collective overview of the time spent.

The team logged a total of 701 hours, compared to the approximately 720 hours planned for the bachelor thesis. Completing the project slightly under the planned time is considered a success. The workload was distributed very evenly among the team members.

**Noah Fleischmann** contributed **352 hours**.

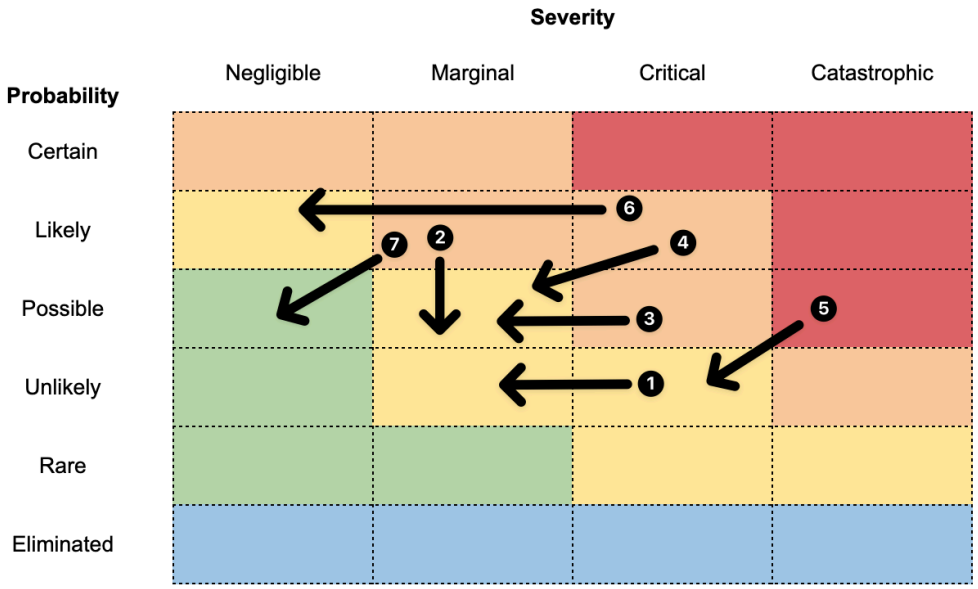
**Dominik Rüegg** contributed **349 hours**.

### 9.5.5. AI Systems

LLM from providers such as OpenAI [12], Anthropic [35], and Google [47] were utilized as coding assistants through Integrated Development Environment (IDE) integration. The AI Chatbot exclusively uses OpenAI models. Additionally, these tools were employed to refine and improve documentation. Their use significantly reduced the time required for both development and documentation tasks.

## 9.6. Risk Management

The following risks were identified for the Finteam project.




**Risiken**

1. Cloud Provider Downtime
2. Apple Review prevents app release
3. Team member gets sick
4. Tax Regulation Complexities
5. Data Security Breach
6. Progress on API implementation blocking other developers work
7. Progress on API implementation blocking Client work
8. Lack of expertise regarding Mobile development

Figure 38: Risk matrix

### 9.6.1. R1 - Cloud Provider Downtime

There is a possibility that the cloud provider Railway will become unavailable due to downtime, removal of essential features, or other incidents that prevent their use. Their hosted services are

used during development. Such as accessing the API from the Finteam mobile app. Or using the hosted database during local development of the API.

**Mitigation:** The software is built in a cloud provider agnostic way. Meaning, everything could be deployed on a different provider or self-hosted without changing the source code. For a temporary downtime incident during development, both developers could provision a local PostgreSQL instance and run everything locally.

**Assessment:** Before mitigation: Critical, Unlikely. After mitigation: Marginal, Unlikely.

**Result:** The risk did not materialize.

### 9.6.2. R2 - Apple Review prevents app release

For the Finteam iOS app to be published in the App Store it must be manually approved by Apple. During this process, Apple checks that each app meets the App Review Guidelines [48]. This process applies to both regular App Store and TestFlight releases, although review for TestFlight is less stringent. Without this, the App cannot be tested on any iOS device other than those of the project members.

**Mitigation:** The project team will carefully review Apple's App Store Guidelines before and during development to ensure compliance. Further, app releases will be submitted with ample time for the review process to go through, including potential revisions, before any potential deadline.

**Assessment:** Before mitigation: Marginal, Likely. After mitigation: Marginal, Possible.

**Result:** Due to the Apple Developer account not being provided on time, the project team was unable to proceed with the App Store deployment. This risk is therefore now considered out of scope.

### 9.6.3. R3 - Team member gets sick

A team member may fall ill, briefly or for an extended period. This could result in tasks assigned to that team member not being completed on time.

**Mitigation:** For brief sickness, tasks can be rescheduled for later. Since the project spans many weeks and there are no hard deadlines before the final hand-in, missed work can be performed later. In the event of prolonged illness (more than two weeks), the team will consult the advisor and study administration to determine how to proceed. Precedents exist within the university for such cases.

**Assessment:** Before mitigation: Critical, Possible. After mitigation: Marginal, Possible.

**Result:** The risk did not materialize.

### 9.6.4. R4 - Tax Regulation Complexities

None of the team members are experts in Swiss tax law. Therefore it is challenging to model all intricacies of Swiss tax law in the app. There is also the threat of showing incorrect information in the app.

**Mitigation:** The goal of this project is not to encode the entirety of the Swiss tax law. Instead, the focus will be on building a simple version focusing on things that are constant between all Swiss cantons. If anything needs to be canton specific, Zurich will be used.

**Assessment:** Before mitigation: Critical, Likely. After mitigation: Marginal, Possible.

**Result:** The risk did not materialize.

### 9.6.5. R5 - Data Security Breach

A security vulnerability in the Finteam mobile application or API server could lead to a data breach, exposing sensitive user information. This is a particularly severe risk on this project, as Finteam is dealing with sensitive financial documents of users.

**Mitigation:** Follow security best practices, such as encrypting sensitive data, implementing proper authentication and authorization, rate limiting and staying updated on security patches.

**Assessment:** Before mitigation: Catastrophic, Possible. After mitigation: Critical, Unlikely.

**Result:** The project team followed best practices, and no security vulnerabilities were known at the time of handover to the stakeholder. The system was thoroughly tested, and therefore no risks materialized during the project period. However, encryption of data at rest was deemed out of scope. But data in transit is always encrypted.

### 9.6.6. R6 - Progress on API implementation blocking other developers work

The developers rely heavily on functions from the backend API. However, development must happen in parallel. It must not happen that client development has to be paused because certain backend functionalities are still missing.

**Mitigation:** API mocking allows the backend functionality to be imitated - even if it has not yet been implemented. This allows development to proceed independently of each other. In the end, only the integration via the API needs to be checked.

**Assessment:** Before mitigation: Critical, Likely. After mitigation: Negligible, Likely.

**Result:** The risk did not materialize. Development was able to proceed.

### 9.6.7. R7 - Lack of expertise regarding Mobile development

The project team is experienced in writing backend APIs and Web applications. However, the team members only have minimal experience developing Mobile Applications. This could lead to slowdowns in development speed and failing to reach the defined milestones in time.

**Mitigation:** The team has extensive experience developing React Web applications. By using React Native to build the mobile apps, this experience translates well. There is minimal native-specific knowledge needed, all of which can be easily acquired on demand.

**Assessment:** Before mitigation: Marginal, Likely. After mitigation: Negligible, Possible.

**Result:** The risk did not materialize. This is largely thanks to React Native with Expo, which leverages web technologies that the project team was already familiar with. Expo abstracts mobile APIs through easy-to-use interfaces.

# Part III

# Appendix

# Chapter 10

# Glossary

***ADR* – Architectural Decision Record**: “A document that captures an important architectural decision made along with its context and consequences. ADRs help teams record, communicate, and justify decisions over time.” 13

***B-tree* – Balanced Tree Index**: “A standard indexing structure in relational databases for efficient lookups and sorting.”, “Ensures low-latency filtering even as the number of documents grows.” 49

***C4* – C4 Notation**: “A hierarchical modeling technique for visualizing the static structure of software systems. C4 stands for Context, Container, Component, and Code, representing different levels of abstraction in software architecture diagrams.” 13

***CLI* – Command Line Interface**: “A text-based interface that allows users to interact with a computer or software by entering commands, often used for automation, scripting, and administrative tasks.” 34

***FRs* – Functional Requirements**: “A specification that defines the expected behavior, features, and functions of a system, describing what the system should do.” 4, 8

***HNSW* – Hierarchical Navigable Small World Graph**: “An efficient indexing structure for high-dimensional vector similarity search.”, “Optimized for fast and scalable retrieval of semantically similar document chunks.” 49

***i18n* – Internationalization**: “The process of designing software applications so that they can be adapted to various languages and regions without engineering changes.” 21

***IDE* – Integrated Development Environment**: “A software application that provides comprehensive facilities to computer programmers for software development, typically including a source code editor, build automation tools, and a debugger.” 81

***LLM* – Large Language Model**: “The technology used to parse extracted text into structured JSON output.” 33, 35, 81

***MVP* – Minimal Viable Product**: “The smallest functional version of a product that can be released to early users for feedback and validation.” II

***NFRs* – Non-Functional Requirements**: “A specification that defines the quality attributes, system constraints, or operational characteristics of a system.” 4

***PaaS* – Platform as a Service**: “A cloud computing model that provides a complete development and deployment environment, including infrastructure, runtime, and tools, allowing developers to build, test, and deploy applications without managing underlying hardware or software.” 33, 59

***RAG* – Retrieval-Augmented Generation**: “A technique that combines information retrieval with text generation, where a model retrieves relevant documents from a knowledge base and uses them as context to generate more accurate and informed responses.” II, 2, 10, 30, 48, 56

***REST API* – Representational State Transfer Application Programming Interface**: “Principles allowing clients to interact with resources using standard HTTP methods like GET, POST, PUT, and DELETE.” 32

***RLS* – Row-Level Security**: “A data access control mechanism that restricts users to only view or manipulate specific rows in a database based on their roles, permissions, or attributes, enhancing security and compliance.” 33

***SPA* – Single Page Application**: “A web application that dynamically updates content on a single page without requiring full page reloads, providing a smoother and faster user experience.” 34

***Stemming* – Word Root Reduction**: “Reduces words to their root form to improve search matching across grammatical variations.” 49

***Stop Word Filtering* – Common Word Exclusion**: “Ignores frequently used words (like ‘und’, ‘der’) to focus search on meaningful terms.” 49

Chapter 11

# List of Figures

Chapter 12

# List of Tables

# Chapter 13

# Directory of Resources

| Task Area | Tools |
|---|---|
| Literature Research and Management | Google, Perplexity, ChatGPT, Anthropic Claude |
| Data Analysis and Visualization | Google Sheets |
| Text Generation and Translation | ChatGPT, DeepL, Google Translate, Typst |
| Coding | GitHub, Visual Studio Code, GitHub Copilot, Cursor |
| Design | Figma, Draw.io |
| Project Management and Collaboration | Linear, Teams, Miro |
| Presentation Creation | PowerPoint |
| DevOps | Railway |

Table 7: Directory of Resources

# Chapter 14

# Bibliography


[1] Meta Platforms, Inc., "React Native." Accessed: May 17, 2025. [Online]. Available: https://reactnative.dev/

[2] Figma, Inc., "Figma." Accessed: May 17, 2025. [Online]. Available: https://www.figma.com/

[3] T. Team, "Typescript documentation." Accessed: Feb. 12, 2025. [Online]. Available: https://www.typescriptlang.org/docs

[4] "Express.js." Accessed: Feb. 17, 2025. [Online]. Available: https://expressjs.com/

[5] P. G. D. Group, "PostgreSQL Documentation." Accessed: Feb. 12, 2025. [Online]. Available: https://www.postgresql.org/docs/

[6] D. Team, "Drizzle ORM Documentation." Accessed: Feb. 12, 2025. [Online]. Available: https://orm.drizzle.team/docs

[7] I. Meta Platforms, "React Documentation." Accessed: Feb. 12, 2025. [Online]. Available: https://react.dev/

[8] Microsoft, "TypeScript Documentation." Accessed: Feb. 12, 2025. [Online]. Available: https://www.typescriptlang.org/docs/

[9] "Tailwind CSS." Accessed: Feb. 17, 2025. [Online]. Available: https://tailwindcss.com/

[10] "Finteam OST · GitHub." Accessed: Jun. 04, 2025. [Online]. Available: https://github.com/orgs/finteam-ost/repositories

[11] Miro, Inc., "Miro." Accessed: May 17, 2025. [Online]. Available: https://miro.com/

[12] "OpenAI." Accessed: May 19, 2025. [Online]. Available: https://openai.com/

[13] "React Query." Accessed: Feb. 17, 2025. [Online]. Available: https://tanstack.com/query/latest/docs/framework/react/overview

[14] Molefrog, "Wouter: A minimalist-friendly ~2.1KB routing for React and Preact." Accessed: Feb. 12, 2025. [Online]. Available: https://github.com/molefrog/wouter

[15] "React Native Expo." Accessed: Feb. 17, 2025. [Online]. Available: https://expo.dev/docs

[16] Apple Inc., "Xcode." Accessed: May 17, 2025. [Online]. Available: https://developer.apple.com/xcode/

[17] O. Foundation, "Node.js Documentation." Accessed: Feb. 12, 2025. [Online]. Available: https://nodejs.org/en/docs

[18] pnpm Contributors, “pnpm.” Accessed: May 19, 2025. [Online]. Available: https://pnpm.io/

[19] Expo Team, “Expo Router.” Accessed: May 19, 2025. [Online]. Available: https://expo.dev/router

[20] “Zod Validator.” Accessed: Feb. 17, 2025. [Online]. Available: https://zod.dev/

[21] fnando, “i18n-js.” Accessed: May 19, 2025. [Online]. Available: https://www.npmjs.com/package/i18n-js

[22] Supabase, “Supabase Documentation.” Accessed: Feb. 12, 2025. [Online]. Available: https://supabase.com/docs

[23] “Passport.js Documentation.” Accessed: Feb. 17, 2025. [Online]. Available: https://www.passportjs.org/docs/

[24] “JSON Web Tokens (JWT).” Accessed: Feb. 17, 2025. [Online]. Available: https://jwt.io/

[25] “TSC Alias.” Accessed: Feb. 17, 2025. [Online]. Available: https://www.npmjs.com/package/tsc-alias

[26] “TSX.” Accessed: Feb. 17, 2025. [Online]. Available: https://github.com/privatenumber/tsx

[27] “Railway.” Accessed: Feb. 17, 2025. [Online]. Available: https://railway.app/

[28] “Vercel AI SDK.” Accessed: Feb. 17, 2025. [Online]. Available: https://ai-sdk.dev/docs/introduction

[29] “PBKDF2: Password Based Key Derivation.” Accessed: Feb. 17, 2025. [Online]. Available: https://www.ssltrust.com.au/blog/pbkdf2-password-key-derivation

[30] “pgvector.” Accessed: Feb. 17, 2025. [Online]. Available: https://github.com/pgvector/pgvector

[31] “Docling.” Accessed: Feb. 17, 2025. [Online]. Available: https://docling-project.github.io/docling/

[32] “Evaluating Chunking Strategies for Retrieval.” Accessed: Feb. 17, 2025. [Online]. Available: https://research.trychroma.com/evaluating-chunking

[33] “Introducing GPT 4.1 Generation.” Accessed: Feb. 17, 2025. [Online]. Available: https://openai.com/index/gpt-4-1/

[34] “Introducing Contextual Retrieval.” Accessed: Feb. 17, 2025. [Online]. Available: https://www.anthropic.com/news/contextual-retrieval

[35] “Anthropic.” Accessed: May 19, 2025. [Online]. Available: https://www.anthropic.com/

[36] D. Rau, S. Wang, H. Déjean, and S. Clinchant, “Context embeddings for efficient answer generation in rag,” *arXiv preprint arXiv:2407.09252*, 2024.

[37] C.-Y. Lin and F. Och, “Looking for a few good metrics: ROUGE and its evaluation,” in *Ntcir workshop*, 2004, pp. 1–8.

[38] S. Liu *et al.*, "Judge as A Judge: Improving the Evaluation of Retrieval-Augmented Generation through the Judge-Consistency of Large Language Models." [Online]. Available: https://arxiv.org/abs/2502.18817

[39] "Docker." Accessed: Feb. 17, 2025. [Online]. Available: https://www.docker.com/r

[40] "Nixpacks." Accessed: Feb. 17, 2025. [Online]. Available: https://nixpacks.com/docs/getting-started

[41] "Vitest." Accessed: Feb. 17, 2025. [Online]. Available: https://vitest.dev/

[42] "wrk - a HTTP benchmarking tool." Accessed: Feb. 17, 2025. [Online]. Available: https://github.com/wg/wrk

[43] "What is role-based access control (RBAC)?." Accessed: Feb. 17, 2025. [Online]. Available: https://www.ibm.com/think/topics/rbac

[44] "OWASP Top Ten." Accessed: Feb. 17, 2025. [Online]. Available: https://owasp.org/www-project-top-ten/

[45] L. Team, "Linear." Accessed: Jan. 24, 2025. [Online]. Available: https://linear.app/

[46] T. Team, "Typst." Accessed: Jan. 24, 2025. [Online]. Available: https://typst.app/

[47] "Google." Accessed: May 19, 2025. [Online]. Available: https://www.google.com/

[48] Apple, "App Review Guidelines." Accessed: Mar. 04, 2025. [Online]. Available: https://developer.apple.com/app-store/review/guidelines/

[49] "OpenAPI." Accessed: Feb. 17, 2025. [Online]. Available: https://www.openapis.org/

[50] "Swagger." Accessed: Feb. 17, 2025. [Online]. Available: https://swagger.io/

[51] "tRPC." Accessed: Feb. 17, 2025. [Online]. Available: https://trpc.io/

[52] "Vite." Accessed: Feb. 17, 2025. [Online]. Available: https://vite.dev/guide/

[53] "Next.js." Accessed: Feb. 17, 2025. [Online]. Available: https://nextjs.org/

[54] "Angular." Accessed: Feb. 17, 2025. [Online]. Available: https://angular.dev/

[55] "Java Spring Boot." Accessed: Feb. 17, 2025. [Online]. Available: https://spring.io/projects/spring-boot/

[56] "Flutter." Accessed: Feb. 17, 2025. [Online]. Available: https://flutter.dev/

[57] "Swift Language." Accessed: Feb. 17, 2025. [Online]. Available: https://www.swift.org/

[58] "Kotlin Language." Accessed: Feb. 17, 2025. [Online]. Available: https://kotlinlang.org/

[59] "Session Based Authentication." Accessed: Feb. 17, 2025. [Online]. Available: https://supertokens.com/blog/session-based-authentication

[60] "Clerk Authentication." Accessed: Feb. 17, 2025. [Online]. Available: https://clerk.com/

[61] "Amazon AWS S3." Accessed: Feb. 17, 2025. [Online]. Available: https://aws.amazon.com/s3/

[62] S. Consortium, “SQLite Documentation.” Accessed: Feb. 12, 2025. [Online]. Available: https://www.sqlite.org/docs.html

[63] GitHub, Inc., “GitHub.” Accessed: May 17, 2025. [Online]. Available: https://github.com/

# Chapter 15

# Requirements and decisions

This chapter contains the complete list of user, developer stories and architectural decisions. The stories within the Foundation Epic are technical in nature and cannot be expressed meaningfully from a user's perspective. Therefore, developer stories are used instead.

## 15.1. Functional Requirements

**Setup Backend**

| | |
|---|---|
| **ID** | FR-001 |
| **Name** | Setup Backend |
| **Description** | Initialize the backend application with the chosen framework and tools. |
| **Epic** | Foundation |
| **Acceptance Criteria** | • The backend codebase is initialized using Express.js.<br>• The codebase is written in TypeScript.<br>• Data validation is implemented using Zod.<br>• The backend application is successfully deployed to Railway.<br>• Basic API endpoints are defined and functional.<br>• Logging and error handling are implemented. |

**Configure Database**

| | |
|---|---|
| **ID** | FR-002 |
| **Name** | Configure Database |

| | |
|---|---|
| **Description** | Set up the database connection and ORM for data persistence. |
| **Epic** | Foundation |
| **Acceptance Criteria** | • A PostgreSQL database is provisioned on Railway.<br>• Drizzle ORM is configured to interact with the PostgreSQL database.<br>• The application can successfully connect to and perform basic CRUD (Create, Read, Update, Delete) operations on the database.<br>• Database schema changes can be applied using Drizzle Push. |

**Configure File System**

| | |
|---|---|
| **ID** | FR-003 |
| **Name** | Configure File System |
| **Description** | Set up file storage mechanisms for the application. |
| **Epic** | Foundation |
| **Acceptance Criteria** | • A Railway Volume is configured for persistent file storage.<br>• The application can successfully connect to and interact with the Railway Volume.<br>• The system can store and retrieve files of various types (e.g., images, PDFs, documents).<br>• The system implements appropriate file size limits. |

**Setup Mobile App Framework**

| | |
|---|---|
| **ID** | FR-004 |

| | |
|---|---|
| **Name** | Setup Mobile App Framework |
| **Description** | Initialize the mobile application codebase with React Native Expo. |
| **Epic** | Foundation |
| **Acceptance Criteria** | • The mobile app codebase is initialized using React Native.<br>• The project is configured to use Expo for development and deployment.<br>• The codebase is written in TypeScript.<br>• The app can be successfully built and run on both iOS and Android emulators/ simulators.<br>• Basic routing navigation is implemented. |

**Setup Web App**

| | |
|---|---|
| **ID** | FR-005 |
| **Name** | Setup Web App |
| **Description** | Initialize the web application codebase with the chosen technologies. |
| **Epic** | Foundation |
| **Acceptance Criteria** | • The web app codebase is initialized using React and Vite.<br>• Routing is configured using Wouter.<br>• Data validation is implemented using Zod.<br>• The codebase is written in TypeScript.<br>• Data fetching and caching are implemented using React Query.<br>• The web app can be successfully built and deployed to a staging environment.<br>• Deployed on Railway as a Static Site through Caddy. |

**Setup Lint and Format Rules**

| | |
|---|---|
| **ID** | FR-006 |
| **Name** | Setup Lint and Format Rules |
| **Description** | Establish consistent code style and quality standards across the project. |
| **Epic** | Foundation |
| **Acceptance Criteria** | • A linter ESLint) is configured for TypeScript<br>• Formatting rules are defined and enforced using a code formatter (Prettier).<br>• The linting and formatting rules are integrated into the development workflow (CI/CD pipeline).<br>• The linting and formatting rules are documented and communicated to the development team.<br>• The linting and formatting rules are regularly reviewed and updated as needed. |

**Create User Account**

| | |
|---|---|
| **ID** | FR-007 |
| **Name** | Create User Account |
| **User Story** | As a user, I want to be able to create an account with my username and password so that I can access the application. |
| **Epic** | User Account Management |
| **Acceptance Criteria** | • Users can access a registration page to create a new account.<br>• Users are required to provide a unique username and a strong password.<br>• The system validates the username and password according to predefined rules.<br>• The system securely stores the user's credentials (password hashing with salt). |

| | |
|---|---|
| | • Users can access the application after successfully creating and verifying their account. |

**User Account Login**

| | |
|---|---|
| **ID** | FR-008 |
| **Name** | User Account Login |
| **User Story** | As a user, I want to be able to log in to my account with my username and password so that I can access the application. |
| **Epic** | User Account Management |
| **Acceptance Criteria** | • Users can enter their registered username and password to log in.<br>• The system validates the username and password against the stored credentials.<br>• The system authenticates the user and grants access to their account upon successful login.<br>• The system displays an error message if the login fails due to incorrect credentials.<br>• The system implements security measures to protect against brute-force attacks (rate limiting). |

**Update Profile Information**

| | |
|---|---|
| **ID** | FR-009 |
| **Name** | Update Profile Information |
| **User Story** | As a user, I want to be able to update my profile information (e.g., name, address, phone number) so that my information is always current. |
| **Epic** | User Account Management |

| | |
|---|---|
| **Acceptance Criteria** | • Users can access a profile editing page from their account settings.<br>• Users can modify their name, address, phone number, and other relevant profile fields.<br>• The system validates the input data to ensure it is in the correct format.<br>• The system saves the updated profile information to the database.<br>• Users receive a confirmation message after successfully updating their profile.<br>• The updated profile information is immediately reflected throughout the application. |

**Account Security**

| | |
|---|---|
| **ID** | FR-010 |
| **Name** | Account Security |
| **User Story** | As a developer, I want to ensure account security to protect user data and prevent unauthorized access. |
| **Epic** | User Account Management |
| **Acceptance Criteria** | • Passwords are securely hashed using a strong hashing algorithm (e.g., bcrypt).<br>• Rate limiting is implemented on login attempts to prevent brute-force attacks.<br>• Access control mechanisms are in place to restrict access to sensitive data based on user roles and permissions.<br>• JWT (JSON Web Tokens) are used for secure authentication and authorization.<br>• Regular security audits are conducted to identify and address potential vulnerabilities. |

**User Scans Documents with Camera**

| | |
|---|---|
| **ID** | FR-011 |
| **Name** | User Scans Documents with Camera |
| **User Story** | As a user, I want to be able to scan documents using my mobile device's camera so that I can easily upload paper documents. |
| **Epic** | Document Management |
| **Acceptance Criteria** | • Users can initiate the scanning process from within the mobile app.<br>• The app utilizes the device's camera to capture images of documents.<br>• Users can preview the scanned document before saving it. |

**User Deletes Document**

| | |
|---|---|
| **ID** | FR-012 |
| **Name** | User Deletes Document |
| **User Story** | As a user, I want to be able to delete a document so that I can remove it from the application. |
| **Epic** | Document Management |
| **Acceptance Criteria** | • Users can easily delete a document from the document list or preview screen.<br>• The system displays a confirmation prompt before permanently deleting the document.<br>• The system provides a visual indication that the document has been deleted.<br>• The deletion process is secure and prevents unauthorized access to the deleted document. |

**User Categorizes Documents**

| | |
|---|---|
| **ID** | FR-013 |
| **Name** | User Categorizes Documents |
| **User Story** | As a user, I want to categorize my documents into tax related areas |
| **Epic** | Document Management |
| **Acceptance Criteria** | • Users can assign predefined categories to their uploaded documents.<br>• The system provides a list of relevant tax-related categories (e.g., Income, Expenses, Deductions, Credits).<br>• Users can easily change the category of a document.<br>• Users can filter and sort their documents by category. |

**Fiduciary Invites User to Connect**

| | |
|---|---|
| **ID** | FR-014 |
| **Name** | Fiduciary Invites User to Connect |
| **User Story** | As a fiduciary, I want to be able to invite a user to connect with me through the app so that I can manage their tax information. |
| **Epic** | Fiduciary Client Onboarding and Management |
| **Acceptance Criteria** | • Fiduciaries can easily initiate an invitation to a user through the fiduciary web app.<br>• The system tracks the status of the invitation (open, used).<br>• Fiduciaries can resend invitations if necessary. |

**Fiduciary Views Connected Users List**

| | |
|---|---|
| **ID** | FR-015 |
| **Name** | Fiduciary Views Connected Users List |
| **User Story** | As a fiduciary, I want to be able to view a list of my connected users so that I can easily see all of my clients. |
| **Epic** | Fiduciary Client Onboarding and Management |
| **Acceptance Criteria** | • Fiduciaries can easily access a list of their connected users from the fiduciary web app.<br>• The list displays key information about each user (e.g., name, connection status).<br>• The list is paginated to handle a large number of connected users efficiently. |

**Fiduciary Revokes User Access**

| | |
|---|---|
| **ID** | FR-016 |
| **Name** | Fiduciary Revokes User Access |
| **User Story** | As a fiduciary, I want to be able to revoke a user's access to my account so that I can terminate the relationship. |
| **Epic** | Fiduciary Client Onboarding and Management |
| **Acceptance Criteria** | • Fiduciaries can easily remove a user through a clear and intuitive interface.<br>• The system provides a confirmation message to the fiduciary after revoking access.<br>• The user is notified that their access has been revoked. |

**User Revokes fiduciary Access**

| | |
|---|---|
| **ID** | FR-017 |
| **Name** | User Revokes fiduciary Access |
| **User Story** | As a user, I want to be able to revoke a fiduciary's access to my account so that I can terminate the relationship. |
| **Epic** | Fiduciary Client Onboarding and Management |
| **Acceptance Criteria** | • Users can easily revoke a fiduciary's access to their account through a clear and intuitive interface.<br>• Revoking access immediately prevents the fiduciary from accessing the user's data.<br>• The system provides a confirmation message to the user after revoking access.<br>• The fiduciary is notified that their access has been revoked. |

**Fiduciary Accesses User Data**

| | |
|---|---|
| **ID** | FR-018 |
| **Name** | Fiduciary Accesses User Data |
| **User Story** | As a fiduciary, I want to be able to access a user's documents and tax data so that I can provide them with tax advice. |
| **Epic** | Enhanced fiduciary Interaction |
| **Acceptance Criteria** | • Fiduciaries can easily navigate to a user's profile and access their documents and tax data.<br>• The system displays the user's documents and tax data in a clear and organized manner.<br>• Access to user data is controlled by appropriate access control measures to ensure data security and privacy. |

| | |
|---|---|
| | • Fiduciaries can view different categories of tax data (e.g., income, expenses, deductions). |

**User Notifies Fiduciaries Documents Ready**

| | |
|---|---|
| **ID** | FR-019 |
| **Name** | User Notifies fiduciary Documents Ready |
| **User Story** | As a user, I want to be able to notify my fiduciary that my documents are ready for review so that they know when to take action. |
| **Epic** | Enhanced fiduciary-user Interaction |
| **Acceptance Criteria** | • Users can easily indicate that their documents are ready for review.<br>• Fiduciaries receive a clear and timely notification when a user marks their documents as ready.<br>• Users can optionally add a message to the fiduciary when notifying them. |

**Fiduciary Requests More Information**

| | |
|---|---|
| **ID** | FR-020 |
| **Name** | Fiduciary Requests More Information |
| **User Story** | As a fiduciary, I want to be able to request more documents or information from a user |
| **Epic** | Enhanced fiduciary-user Interaction |
| **Acceptance Criteria** | • Fiduciaries can send a request for specific documents or information to a user.<br>• Fiduciaries can provide a clear explanation of why the additional information is needed. |

| | |
|---|---|
| | • Users receive a notification when a fiduciary requests more information.<br>• Users can respond to the request by uploading the requested documents or providing the requested information. |

**User Asks fiduciary Questions**

| | |
|---|---|
| **ID** | FR-021 |
| **Name** | User Asks fiduciary Questions |
| **User Story** | As a user, I want to be able to ask a fiduciary questions about my tax situation so that I can get help with my taxes. |
| **Epic** | Enhanced fiduciary-user Interaction |
| **Acceptance Criteria** | • Users can send text-based questions to their assigned fiduciary.<br>• Users can view a history of their questions and the fiduciary's responses.<br>• The system provides a clear and intuitive interface for asking questions.<br>• Fiduciaries receive timely notifications when a user asks a question. |

**Chatbot Answer Question**

| | |
|---|---|
| **ID** | FR-022 |
| **Name** | Chatbot Answer Question |
| **User Story** | As a user, I want to be able to ask the chatbot questions about the tax guidance document so that I can get quick answers to my tax-related inquiries. |
| **Epic** | Steuerwegleitung RAG Chatbot |
| **Acceptance Criteria** | • Users can type questions into a chat interface to interact with the chatbot. |

| | |
|---|---|
| | • The chatbot utilizes the tax guidance document as its exclusive knowledge source.<br>• The chatbot provides relevant and accurate answers to user questions based on the document's content.<br>• The chatbot handles questions that are outside the scope of the tax guidance document by indicating that it does not have that information. |

## 15.2. Non-functional Requirements

Below the complete list of Non-functional requirements.

**Usability**

| | |
|---|---|
| **ID** | NFR-01 |
| **Description** | The application should be easy to use for new users. A new user can use the application effortlessly. |
| **Requirement** | Usability - Ease of Use |
| **Priority** | High |
| **Verification Process** | Conduct usability testing with two new users to assess their ability to complete key tasks without assistance. |
| **Measures** | • Task completion rate: >90% of new users can complete key tasks successfully without external guidance.<br>• Time on task: New users can complete key tasks in under 5 minutes. |

**Maintainability & Extensibility**

| | |
|---|---|
| **ID** | NFR-02 |

| | |
|---|---|
| **Description** | The codebase should follow clean architecture principles to allow easy modifications, future extensions, and long-term maintainability by HL developers after the completion of this project. |
| **Requirement** | Maintainability & Extensibility - Clean Architecture |
| **Priority** | Medium |
| **Verification Process** | Maintain regular communication with all developers to discuss technology choices and architectural decisions. Conduct code reviews to ensure adherence to clean architecture principles and assess the ease of making modifications and adding new features. Additionally, assess documentation completeness. |
| **Measures** | • Codebase adheres to defined architectural principles (e.g., separation of concerns, dependency inversion).<br>• Time to implement a new feature: A new feature can be implemented within a reasonable timeframe (under 1 week).<br>• Effort to modify existing code: Modifications to existing code require minimal effort and have minimal impact on other parts of the system.<br>• Comprehensive documentation of the development process, including design choices, architectural decisions, and codebase organization. |

**Performance**

| | |
|---|---|
| **ID** | NFR-03 |
| **Description** | The system must deliver fast response times, with key actions occurring within a reasonable timeframe. The user experiences |

| | |
|---|---|
| | virtually no waiting times while navigating the application. |
| **Requirement** | Performance Efficiency - Responsiveness |
| **Priority** | High |
| **Verification Process** | Measure the average loading time for individual pages during normal business hours. |
| **Measures** | • Average page load time: The average loading time for individual pages must not exceed 3 seconds during normal operating hours. |

**Capacity**

| | |
|---|---|
| **ID** | NFR-04 |
| **Description** | The application must function properly even under high load. The system should be able to handle at least 50 parallel requests per second. |
| **Requirement** | Performance Efficiency - Capacity |
| **Priority** | Low (for prototype) |
| **Verification Process** | Conduct load testing to simulate a high volume of concurrent users and measure the system's throughput and response times. |
| **Measures** | • Throughput: The system can handle at least 50 parallel requests per second without significant performance degradation.<br>• Average response time: The average response time remains within acceptable limits (sub 1 second) even under high load. |

**Browser Compatibility**

| | |
|---|---|
| **ID** | NFR-05 |
| **Description** | Ensure browser compatibility and optimization for Google Chrome as the primary client. All functions and elements of the application must function fully and correctly in the Google Chrome browser without performance or display issues. Assumes normal usage on an average monitor between 12 and 15 inches. |
| **Requirement** | Compatibility - Browser Compatibility |
| **Priority** | High |
| **Verification Process** | Perform cross-browser testing in Google Chrome on monitors between 12 and 15 inches to verify functionality, performance, and display accuracy. |
| **Measures** | • All core functionalities are fully functional in Google Chrome.<br>• No display issues (e.g., broken layouts, overlapping elements) are present in Google Chrome.<br>• The application renders correctly on monitors between 12 and 15 inches in Google Chrome. |

**Security**

| | |
|---|---|
| **ID** | NFR-06 |
| **Description** | The system must ensure data confidentiality, integrity, and availability. User authentication and authorization should be implemented to restrict access. |
| **Requirement** | Security - Data Integrity and Confidentiality |
| **Priority** | High |

| | |
|---|---|
| **Verification Process** | Conduct regular security audits and penetration testing to identify and address potential vulnerabilities. Implement and monitor access control mechanisms. |
| **Measures** | • Data is encrypted in transit.<br>• Access to sensitive data is restricted based on user roles and permissions.<br>• The system is resistant to common web vulnerabilities as defined by OWASP (e.g., SQL injection, cross-site scripting). |

**Cost**

| | |
|---|---|
| **ID** | NFR-07 |
| **Description** | The system should optimize resource usage to minimize Railway costs. |
| **Requirement** | Cost Optimization |
| **Priority** | Medium |
| **Verification Process** | Check that chosen solutions were selected with cost in mind. Verify that deployments are properly cleaned up and resources are released when no longer needed. |
| **Measures** | • Utilize Railway's cost management tools to identify the costs of resource configurations.<br>• Implement scaling mechanisms to provision and de-provision resources based on demand, optimizing resource utilization and reducing costs. |

**Portability**

| | |
|---|---|
| **ID** | NFR-08 |
| **Description** | The system should be designed for portability to minimize vendor lock-in and facilitate |

| | |
|---|---|
| | deployment on various providers or self-hosting, enabling data sovereignty (ideally hosted within Switzerland later). |
| **Requirement** | Portability - Vendor Independence |
| **Priority** | High |
| **Verification Process** | Verify that the system can be deployed and run on different hosting providers (e.g., AWS, Google Cloud, self-hosted) with minimal configuration changes. Assess the reliance on vendor-specific services and technologies. |
| **Measures** | • The system supports using Docker for easy deployment across different environments.<br>• Data is stored in a standard PostgreSQL database, avoiding vendor-specific database solutions.<br>• Local file storage is used for document management, avoiding reliance on vendor-specific storage services.<br>• The application can be deployed and run in a self-hosted environment. |

## 15.3. Architectural Decision Records

This chapter documents the essential architectural decisions that have shaped the design of this project. Following the ADR (Architectural Decision Record) template, each decision is documented to provide reasons and context for the choices made. Each ADR contains the following categories: Title, Context, Decision Drivers, Options, Decision, and Consequences. Status and Date for the decision have been omitted since they are not widely applicable to a project of this relatively narrow scope.

### 15.3.1. ADR 01 – Database

**Context**

The initial prototype of Finteam used Supabase in a *client-direct-to-database* setup, because it offered a hosted PostgreSQL instance together with authentication, storage and row-level-security (RLS) rules.

**Decision Drivers**

- **Portability & vendor independence:** The project must remain movable to any cloud or an on-prem installation in Switzerland without code architecture changes.
- **Fine-grained authorisation and future business logic:** RLS alone is not flexible enough for complex fiduciary / client rules; a server layer allows arbitrary logic, external identity providers and audit logging.
- **Server-side compute workloads:** Features such as document OCR, LLM–based enrichment and RAG require CPU/GPU time that should not run on the client.

**Options**

- Continue with hosted Supabase [22]
- Self-host Supabase
- Dedicated PostgreSQL instance [5]

**Decision**

It was decided to use a managed PostgreSQL instance on Railway, accessed exclusively through an **Express/TypeScript backend** that exposes a REST API. Database access in the API is implemented with **Drizzle ORM**; clients never talk to the database directly.

**Consequences**

- **Positive**
  - Data can be moved to another cloud or to an on-prem server without touching client code.
  - Security model is clearer: all requests pass one gateway; complex fiduciary/user rules are easy to enforce.
  - Enables future features (AI document analysis, background jobs) on the server side.
  - Database schema and migrations are fully under version control.
- **Negative**
- Additional backend to design, implement and operate.

### 15.3.2. ADR 02 – Data Transmission Standard

**Context**

The API handles communication between clients (both mobile and web) and the backend. Consistency and reliability of data exchange are crucial, especially in a system dealing with sensitive tax documents and fiduciary communications. The API must enforce type safety and ensure that the data exchanged matches the expected structures.

**Decision Drivers**

- **Type safety and validation:** The system must enforce strict data validation and type safety for all communications between client and server.
- **Developer experience:** The approach should integrate smoothly with the development process and provide clear guidance for future developers.
- **Maintainability:** The chosen method should minimize overhead and complexity while allowing future extensions.
- **Portability:** The solution should be portable to different environments and not rely heavily on specific tools or configurations.

**Options**

- OpenAPI/Swagger [49], [50]
- tRPC [51]
- Zod [20]

**Decision**

It was decided to use **Zod** for validating and typing API responses. Zod is simple to implement, provides runtime validation, and integrates directly with TypeScript types for a seamless development experience. This decision ensures that the system enforces consistent data structures and avoids the overhead of OpenAPI or tRPC while retaining strong type safety.

**Consequences**

- **Positive**
  - Enforces consistent and correct data structures across client and server.
  - Simplifies development by integrating type safety and validation without external tooling.
  - Enhances maintainability and reduces the risk of runtime errors.
- **Negative**
  - Requires manual definition of schemas and validation in both client and server.
  - Does not provide automatic documentation or client generation like OpenAPI/Swagger.

### 15.3.3. ADR 03 – Platform for Fiduciaries

**Context**

The application required a platform for fiduciaries to access and manage client tax documents, communicate with clients, and oversee submissions. The decision focused on whether this platform should be implemented as a mobile application or a web application.

**Decision Drivers**

- **User environment:** The platform must be accessible from environments where fiduciaries typically perform their work.
- **Usability and productivity:** The interface should support complex tasks with multiple data points.
- **Device compatibility:** Consideration of the range of devices and screen sizes used by fiduciaries in professional settings.
- **Development efficiency:** The platform should be maintainable and extensible with minimal overhead.

**Options**

- Mobile app
- Web app

**Decision**

It was decided to implement a **web application** for fiduciaries, as they typically perform their work in office settings using desktops or laptops.

**Consequences**

- **Positive**
  - Offers a larger workspace suitable for document review, communication, and data management.
  - Simplifies deployment and access through common browsers.
  - Reduces complexity by focusing development efforts on a single web platform.
- **Negative**
  - Limited mobile access for fiduciaries, though not deemed critical.

### 15.3.4. ADR 04 – Frontend Framework

**Context**

The frontend framework was a key decision to support the development of the client-facing interfaces of the system, both for the mobile and web applications.

**Decision Drivers**

- **Performance:** The framework should provide fast load times and efficient rendering.
- **Developer experience:** It should be easy to learn, use, and integrate with existing tooling.
- **Maintainability and extensibility:** The solution should support a modular codebase that can grow with the project.
- **Ecosystem and community support:** A mature and widely adopted framework ensures long-term viability.

**Options**

- Vite React [7], [52]
- React Next.js [53]

- Angular [54]

**Decision**

It was decided to use **Vite React** as the frontend framework for both mobile and web applications.

**Consequences**

- **Positive**
  - Provides fast build times and efficient development workflows.
  - Leverages React's component-based structure and mature ecosystem.
  - Simplifies setup with minimal configuration.
- **Negative**
  - Lacks some features of Next.js, such as built-in SSR.
  - Not a full framework like Angular, but sufficient for the project's scope.

### 15.3.5. ADR 05 – Backend Framework

**Context**

The system architecture required a backend to handle API requests, business logic, authentication, and data management. This decision considered the trade-offs between building a custom backend or using Supabase exclusively.

**Decision Drivers**

- **Control and flexibility:** The backend should allow for implementing custom business logic and integrations.
- **Scalability and maintainability:** The solution should be able to scale with increased usage and remain manageable.
- **Developer productivity:** The framework should integrate well with the existing tech stack and be easy for the team to work with.
- **Portability:** The backend should not depend on a specific vendor or platform.

**Options**

- Next.js (API routes) [53]
- Express.js with TypeScript [4], [8]
- Supabase only [22]
- Java Spring Boot [55]

**Decision**

It was decided to use **Express.js with TypeScript** for the backend.

**Consequences**

- **Positive**
  - Provides a flexible and scalable architecture for handling complex business logic.
  - Integrates seamlessly with the chosen PostgreSQL database and authentication mechanisms.

- Maintains full control over the backend with clear separation from the frontend.

- **Negative**
  - Requires more development effort compared to using Supabase directly.
  - Does not include built-in tools like Supabase's authentication and storage features.

## 15.3.6. ADR 06 – Mobile Framework

**Context**

The mobile framework selection focused on the client-facing application for users to upload and manage tax documents, communicate with fiduciaries, and interact with the system.

**Decision Drivers**

- **Cross-platform support:** The framework should enable development for both iOS and Android.
- **Developer productivity:** Should allow rapid development and debugging with minimal setup.
- **Ecosystem and community support:** Must have strong adoption and long-term viability.
- **Integration with existing tooling:** Should integrate well with the chosen backend and frontend technologies.

**Options**

- Expo React Native [1], [15]
- Pure React Native
- Flutter [56]
- Native development with Swift and Kotlin [57], [58]

**Decision**

It was decided to use **Expo React Native** for the mobile application.

**Consequences**

- **Positive**
  - Simplifies cross-platform development with shared codebase for iOS and Android.
  - Provides built-in modules and managed workflow for rapid development.
  - Reduces setup complexity compared to pure React Native or native development.

- **Negative**
  - Slightly limited flexibility compared to pure React Native or native development.
  - Performance may not match fully native applications in complex use cases.

## 15.3.7. ADR 07 – Authentication Strategy

**Context**

The system required an authentication mechanism to manage user access and protect sensitive tax data. The decision centered around how to implement authentication while balancing security, privacy, and ease of use.

**Decision Drivers**

- **Security:** The system must protect against unauthorized access and ensure data confidentiality.
- **Privacy and self-hosting:** The solution should not introduce external dependencies or data privacy concerns.
- **Scalability:** Should be able to handle increased user load without significant changes.
- **Developer experience:** Should integrate well with the existing backend framework and technology stack.

**Options**

- Stateless JWT authentication [24]
- Session-based authentication [59]
- 3rd-party authentication with Clerk [60]

**Decision**

It was decided to implement **stateless JWT authentication** for the system.

**Consequences**

- **Positive**
  - Simplifies scaling as authentication is stateless and does not rely on server-side session management.
  - Enhances privacy and control by avoiding reliance on third-party authentication providers.
  - Easily integrates with the chosen Express.js backend.
- **Negative**
  - Requires careful management of token lifetimes and refresh strategies.
  - Stateless nature can make revoking tokens more challenging compared to session-based approaches.

### 15.3.8. ADR 08 – Repository Structure

**Context**

The system's codebase includes the backend API, web application, and mobile application. The repository structure needed to support collaborative development while balancing simplicity and maintainability.

**Decision Drivers**

- **Simplicity:** Should allow straightforward project management without introducing complex tooling or dependencies.
- **Separation of concerns:** Each application component should be manageable and deployable independently.
- **Scalability:** The structure should accommodate project growth and new components.

- **Ease of collaboration:** Supports clear boundaries between components for multiple developers.

**Options**

- Monorepo
- Polyrepo

**Decision**

It was decided to use a **polyrepo** structure for the system.

**Consequences**

- **Positive**
  - Each component (backend, web, mobile) is isolated and can be managed independently.
  - Simpler initial setup and fewer dependencies compared to a monorepo.
- **Negative**
  - Harder to share code and types between repositories.
  - Requires more effort to maintain consistency across multiple codebases.

### 15.3.9. ADR 09 – File Upload Strategy

**Context**

The system needed a mechanism for uploading and storing files such as tax documents. This decision focused on balancing complexity, cost, and control.

**Decision Drivers**

- **Simplicity:** The solution should be easy to integrate and maintain without introducing unnecessary dependencies.
- **Self-hosting and control:** The system should avoid reliance on third-party services for privacy and data sovereignty reasons.
- **Performance and scalability:** Should provide acceptable upload and retrieval performance for expected usage levels.

**Options**

- Direct file storage with attached volume
- Amazon S3 or equivalent third-party storage provider [61]

**Decision**

It was decided to use **direct file storage with an attached volume** on the backend server.

**Consequences**

- **Positive**
  - Simple setup with minimal dependencies.
  - Full control over file storage and easy portability to self-hosted environments.
- **Negative**
  - No built-in scalability or redundancy compared to third-party storage providers like S3.

- Potentially higher operational overhead if scaling beyond a single server.

## 15.3.10. ADR 10 – Deployment Platform

**Context**

The system needed a deployment strategy for the backend, web application, and supporting services. This decision considered trade-offs between managing self-hosted infrastructure and using a managed Platform-as-a-Service (PaaS).

**Decision Drivers**

- **Ease of deployment and management:** Minimize operational overhead and streamline CI/CD workflows.
- **Scalability and reliability:** The platform must support scaling with project growth and ensure high availability.
- **Cost-effectiveness:** Consider the total cost of ownership, including setup, maintenance, and resource utilization.
- **Portability:** Maintain flexibility to move to different platforms if needed in the future.

**Options**

- Self-hosting on Linux
- Railway (PaaS) [27]

**Decision**

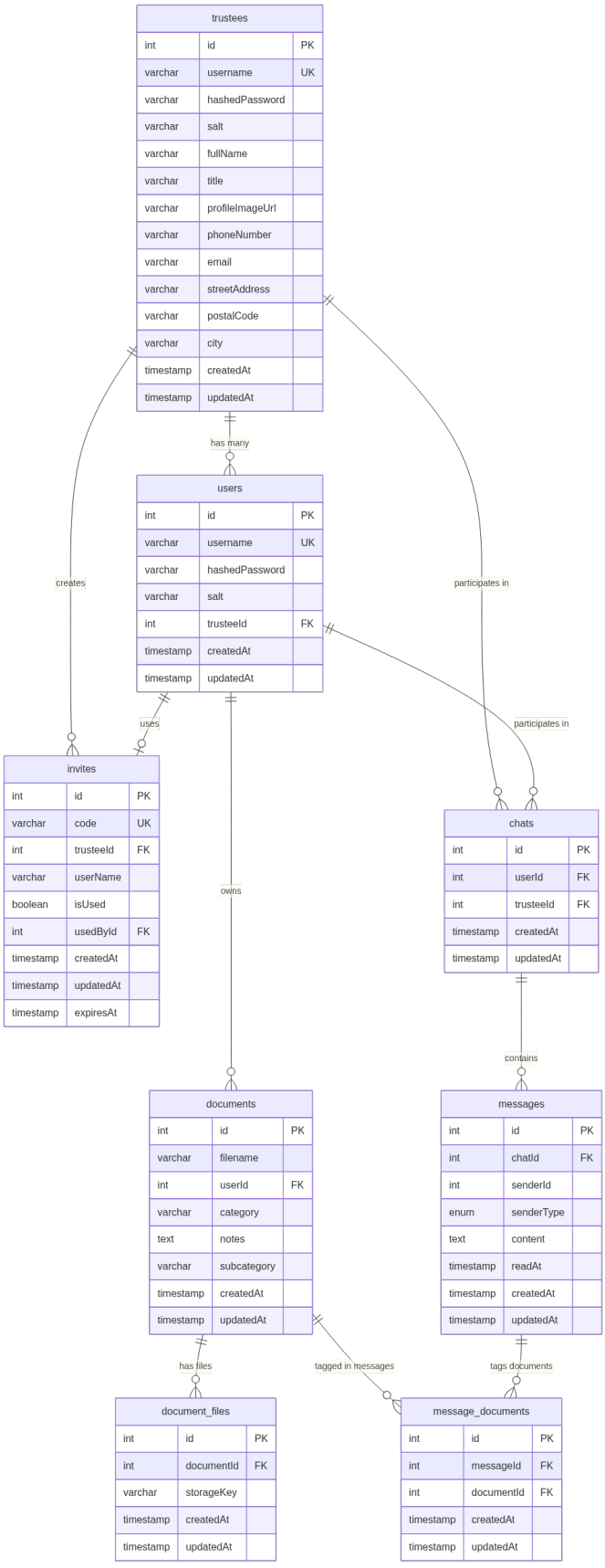
It was decided to deploy on **Railway** for hosting the backend, web application, and database.

**Consequences**

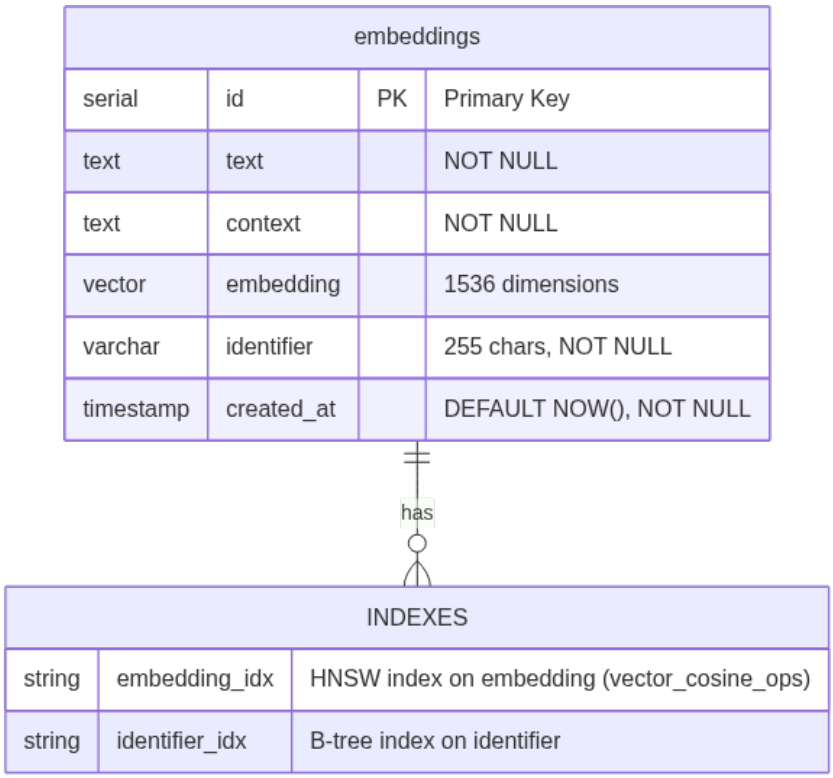
- **Positive**
  - Provides automatic resource provisioning, CI/CD integration, and monitoring tools.
  - Reduces deployment complexity and operational burden compared to managing Linux servers.
  - Streamlines deployment with GitHub integration and auto-deploys on code push.
  - While tied to Railway for hosting, the code can also be deployed on a self-hosted Linux environment without any code changes.
- **Negative**
  - Ties the system to a third-party PaaS provider for hosting.
  - Less granular control over infrastructure compared to self-hosting.

# Entity Relationship Diagrams

**Application DB**


trustees
int id PK
varchar username UK
varchar hashedPassword
varchar salt
varchar fullName
varchar title
varchar profileImageUrl
varchar phoneNumber
varchar email
varchar streetAddress
varchar postalCode
varchar city
timestamp createdAt
timestamp updatedAt
has many
users
int id PK
varchar username UK
varchar hashedPassword
varchar salt
int trusteeId FK
timestamp createdAt
timestamp updatedAt
creates
participates in
uses
participates in
invites
int id PK
varchar code UK
int trusteeId FK
varchar userName
boolean isUsed
int usedById FK
timestamp createdAt
timestamp updatedAt
timestamp expiresAt
owns
chats
int id PK
int userId FK
int trusteeId FK
timestamp createdAt
timestamp updatedAt
contains
documents
int id PK
varchar filename
int userId FK
varchar category
text notes
varchar subcategory
timestamp createdAt
timestamp updatedAt
messages
int id PK
int chatId FK
int senderId
enum senderType
text content
timestamp readAt
timestamp createdAt
timestamp updatedAt
has files
tagged in messages
tags documents
document_files
int id PK
int documentId FK
varchar storageKey
timestamp createdAt
timestamp updatedAt
message_documents
int id PK
int messageId FK
int documentId FK
timestamp createdAt
timestamp updatedAt

### Vector DB


embeddings
serial
id
PK
Primary Key
text
text
NOT NULL
text
context
NOT NULL
vector
embedding
1536 dimensions
varchar
identifier
255 chars, NOT NULL
timestamp
created_at
DEFAULT NOW(), NOT NULL
has
INDEXES
string
embedding_idx
HNSW index on embedding (vector_cosine_ops)
string
identifier_idx
B-tree index on identifier

# Initial Project proposal

SmartFin ist eine App, welche die Anwender unterstützt, Dokumente rund um die Finanzen und die Steuererklärung einfach und strukturiert zu sammeln.

Die App ist eine digitale Weiterentwicklung von Papier-“Checklisten” wie sie von Treuhändern an ihre Kunden abgegeben werden um die Steuererklärung vorzubereiten. Dieser Prozess, sowie die Kommunikation zwischen Treuhänder und Kunde soll mit einer App effizienter gemacht werden. Ausserdem lassen sich Dokumente mit ML klassifizieren und auswerten.

Im Rahmen dieser BA soll ein existierender Prototyp analysiert und verbessert werden. Insbesondere stehen folgende Aspekte im Fokus

- Software Architektur: eine zweckmässige, erweiterbare Architektur soll erarbeitet werden. Dabei müssen Anforderungen an die Smartphone App, an ein Service-Backend, sowie eine Web-App analysiert werden und in die Architektur einfliessen.
- Entwicklung ausgewählter Funktionen auf dem Smartphone (React-Native) inklusive der dazugehörigen Server Endpoints und Persistence Layer.
- optional/Idee: integration von einfachen ML Modellen zur Prüfung von Eingaben und Dokumenten (z.B. ein Photo als “Lohnausweis” klassifizieren)

Der genaue Inhalt und Schwerpunkt der Entwicklung kann im Rahmen der Arbeit diskutiert werden und orientiert sich auch an den Interessen der Studierenden.