

Odin: Document Analysis Tool

Authors:

Vicente Barros 97787

Mariana Andrade 103823

Course: Technologies and Web Development

Year: 2024/25

Professors:

Carlos Santos

David Oliveira

Overview

In the digital age, individuals and organizations are often overwhelmed by vast amounts of information stored across numerous documents and files. While search engines and research platforms like Scopus offer keyword-based search and document organization, there is still a need for a personal, intuitive, and intelligent system that can analyze, organize, and make sense of user-uploaded documents.

The Odin Project proposes a solution to this challenge that allows the extraction of different types of documents, PDF, Word, and PowerPoint. It indexes the texts from the uploaded document and their visualisation and analysis, where it does not leave anything foreshadowed. The Odin Project, just like the Norse god, pursues actual knowledge.

Motivation

The motivation for this project came from a personal need from the authors of this project while searching papers and information for their state-of-the-art dissertation. When using the standard ways of searching for papers, such as Scopus, IEEE, and Google Scholar, we found it challenging to filter between the interesting papers and the ones to discard. Most of the search would be done by analyzing the title of the paper and then filtering the abstract, which can leave interesting knowledge behind. Even when using technologies such as ChatGPT and providing documents, it can leave behind insightful information.

With this problem in mind, we decided to create a proof-of-concept prototype of an application capable of fulfilling the following features:

- **File Upload:** Allow the upload of files working as a personal library
- **Document Visualisation:** The uploaded file will be scraped to allow better readability using markdown notation

- **Better Search:** Instead of only searching for the occurrences of a specific keyword in the title or abstract, the user can find the occurrences within the content of their library with proper highlighting and with the context where the keywords occurred.
- **Chat:** Allow the user to search for information using natural language in the form of a chat

Considering these features and the need for a straightforward application, we developed a proof-of-concept application. In the following sections, we will explain how it was designed.

Methodology

We adopted the Git workflow methodology to ensure efficient and collaborative development. This approach provided a structured framework for version control, facilitating seamless coordination among team members and maintaining high code quality throughout the project's lifecycle.

Branching Strategy

The Git Workflow for this project involved three main types of branches:

- **Main Branch:** Reserved for production-ready code, ensuring stability and reliability. Direct pushes to this branch were strictly prohibited.
- **Dev-Branch:** Used for integrating and testing features before merging them into the main branch. This branch served as the primary staging area for ongoing development.
- **Feature Branches:** Each new feature or enhancement was developed in its dedicated branch and branched off from the dev branch. This ensured the isolation of changes and prevented conflicts during development.

Code Quality and Peer Review

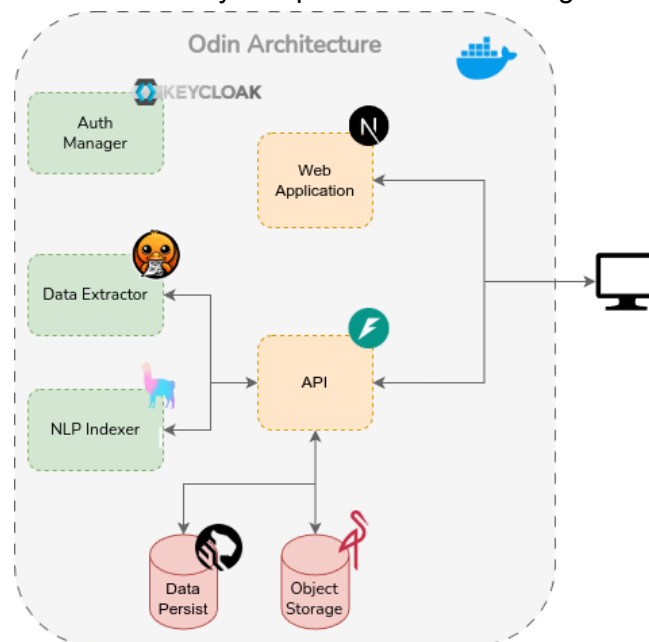
To maintain high standards of code quality, the following protocols were enforced:

- **Pull Requests (PRs):** Changes from feature branches were merged into the dev branch through pull requests. Direct pushes to both the primary and dev branches were disallowed.
- **Peer Reviews:** Each pull request underwent a rigorous peer review process, where team members evaluated the code for functionality, readability, and adherence to project guidelines.
- **SonarCloud Analysis:** The code was analyzed using SonarCloud as part of the pull request review process. This tool provided automated feedback on code quality,

highlighting potential issues such as bugs, security vulnerabilities, and code smells. Only pull requests that passed the SonarCloud checks were approved for merging.

Architecture

Although the assignment mainly concerns the application's visualization layer, the backend needed to work on this application is crucial. Thus, we will specify and explain how the architecture was designed. This project has used several large language models to address the different challenges that require a machine with proper specifications to handle them. In this section, we will delve into the necessary components for achieving the result.



Frontend

The front end is the core part of the course. We developed it using the tools learned through the course. We chose the Next.js meta-framework of React because it offers many features outside the box, such as server-side render, file-based routing, and static site generation.

It uses two state managers: Tanstack Query to manage API data fetching and the Zustand library to manage local state management. Tanstack Query complements Next's server-side render feature, allowing data to be prefetched on the server and then hydrated on the client side. This makes the UX faster, as data is already displayed, and improves the application's SEO. Zustand was used to store the custom user settings for reading and previous chat responses.

For the UI, we used TailwindCSS in this project to style the components. Tailwind is a utility-first CSS framework that allows the developer to create custom designs without having to write custom CSS files. To complement TailwindCSS, we used the shadcn/ui component library. This

library takes a different approach from traditional components libs. It provides the complete component code, allowing customization instead of abstraction, which the developer cannot edit.

The frontend application was organized using the organization enforced from Next.js, in which the app folder defines the routing of the application where we have different folders. The first one is the API folder, where the authentication logic is stored these are the endpoints Auth.js will use to validate and authenticate the user. Then, the base folder stores all the routes of the applications, namely chat, dashboard, file, and search.

To organize the code, the logic associated with the pages was split between reusable components, which can be found in the components folder. The "UI" folder stores all the shading/ui components to avoid mixing the ones we developed.

Finally, there are three folders: hooks to store all the hooks used in the application, namely the ones used to fetch data from the API; stores which have all the code associated with the Zustand library; and finally, the lib folder is responsible for having utils functions with the highlight in the middlewares. We used middleware to block not logged-in people from accessing the application, having only the capability of accessing the landing page

In the section on the features, we will provide a component diagram of each frontend page and the associated components.



Welcome to Odin

Your intelligent document management and analysis platform. Extract insights, search effortlessly, and interact with your documents through AI.

[Get Started](#)[Watch Demo](#)

Key Features



File Upload & Analysis

Support for multiple document types with automatic text extraction and metadata identification.



Smart Search Engine

Powerful keyword-based search with intelligent highlighting and document relationships.



AI-Powered Chat

Interactive chat interface for querying documents and receiving AI-powered insights.

FastAPI








FastAPI is a modern, high-performance web framework for building APIs with Python. It uses Python's type hints for data validation and serialization using Pydantic, ensuring robust and efficient API development. Moreover, the implementation of automatically generated documentation, the high performance, and the fact that it was developed in Python made us choose this technology as the backbone of our backend.

The backend is divided according to good practices. The routers or controllers handle HTTP requests using the proper schemas for validation. The services house the business logic and allow users to view all the logic associated with the application's features. Moreover, the data persistence actions are in the crud folder, and the representation is presented in the models folder.

ODIN API 1.0.0 OAS 3.1
/api/v1/openapi.json
ODIN API

Authorize 

File Management File operations

POST	/api/upload/	Upload File	  ▼
POST	/api/upload/bulk/	Upload Multiple Files	 ▼
GET	/api/files/{file_id}/metadata	Get File Metadata	 ▼
GET	/api/files/	List Files	 ▼
GET	/api/files/{file_id}	Download File	 ▼
DELETE	/api/files/{file_id}	Delete File	 ▼

Authentication Operations related to authentication using Keycloak

POST	/api/auth/register	Register User	▼
POST	/api/auth/login	Login User	▼
POST	/api/auth/refresh	Refresh Token	▼

Docling

Docling is a recent technology that parses documents in a wide variety of formats, including PDF, DOCX, PPTX, XLSX, and Markdown. One of the key problems in this project was the parsing of documents, where a significant number of PDFs had issues in recognising page structures, reading order, and table configurations. With Docling, the problems are solved, leaving almost no misinterpreted document part, facilitating the implementation of the different features of the platform. The Docling, being a Python Library, was integrated directly into the backend of our application. We started by using Apache Tika for the document parsing, but due

to the simplicity of the parsing and some issues with the unique characters and line breaks, we decided to switch to Docling.

TimescaleDB

TimescaleDB is a database management system built on PostgreSQL. Its primary purpose is handling time series data. However, over time, it extended its features with a vector database, which was the core feature used for this project. In a nutshell, a vector database is a specialised system designed to store and manage high-dimensional vectors, which are numerical representations of data objects such as text. These vectors enable efficient similarity searches and semantic queries.

Keycloak

Keycloak is an open-source identity and access management solution designed to add authentication and secure services with minimal effort. It provides features such as user federation, strong authentication, user management, and fine-grained authorization. Keycloak was integrated to handle user authentication and authorization processes, creating a private workspace for each user. By utilizing Keycloak with the Auth.js library, we offloaded the complexities of managing user credentials and authentication flows, allowing our development team to focus on core functionalities.

The screenshot displays the Keycloak Admin Console interface. On the left is a dark sidebar with navigation options: Manage, Clients, Client scopes, Realm roles, Users, Groups, Sessions, Events, Configure, Realm settings, Authentication, Identity providers, and User federation. The 'Clients' section is selected. The main content area shows the 'Client details' for 'tdw-client' (OpenID Connect). At the top right, there's a status 'Enabled' with a toggle switch and an 'Action' dropdown. Below this is a tabbed interface with 'Settings' as the active tab. The 'General settings' section includes fields for 'Client ID' (tdw-client), 'Name', and 'Description'. The 'Always display in UI' toggle is turned off. The 'Access settings' section includes fields for 'Root URL', 'Home URL', and 'Valid redirect URIs' (http://localhost:3000/*). A 'Jump to section' sidebar on the right lists: General settings, Access settings, Capability config, Login settings, and Logout settings. At the bottom, there are 'Save' and 'Revert' buttons.

MinIO

MinIO is a high-performance, open-source object storage system compatible with the Amazon S3 API. This enables seamless integration with existing S3 clients. MinIO can handle unstructured data such as photos, videos, log files, backups, and container images, and its maximum supported object size is 50TB. For this project, MinIO is responsible for storing all the files uploaded by the user in their original form.

The screenshot displays the MinIO Object Browser interface. On the left is a dark blue sidebar with navigation links for 'User', 'Object Browser', 'Access Keys', 'Documentation', 'Administrator', 'Buckets', 'Policies', 'Identity', 'Monitoring', 'Events', 'Configuration', and 'License'. The main area shows the 'fastapi-bucket' with details: 'Created on: Tue, Jan 14 2025 12:56:22 (GMT)', 'Access: PRIVATE', and '19.5 MiB - 8 Objects'. A search bar at the top right contains the text 'Start typing to filter objects in the bucket'. Below the bucket header is a table of objects:

Name	Last Modified	Size
12ef17bab9d44486b4f7b794fc558777	Sun, Jan 19 2025 12:33 (GMT)	3.7 MiB
14cf279a4924d4398598928667b31e9	Sat, Jan 18 2025 17:51 (GMT)	3.7 MiB
30979c2b86c649ffbc007c2cc0c5514f	Tue, Jan 14 2025 12:58 (GMT)	556.7 KiB
5c84a268d9284a47a862c86694be9cee	Wed, Jan 15 2025 18:34 (GMT)	1.4 MiB
64ef94f0a97f4b97853c2b2d7a01dcd4	Today, 04:17	3.7 MiB
a5448a9961384730a07fe4565f1cb998	Sat, Jan 18 2025 17:31 (GMT)	2.7 MiB
b7ac22a6d00c47d6bf8eaf2a33853766	Tue, Jan 14 2025 12:56 (GMT)	65.0 KiB
cfa8a232630443cf9027cd0a57ae747c	Sun, Jan 19 2025 21:41 (GMT)	3.7 MiB

On the right side of the interface, there are 'Actions' (Download, Share, Preview, Local Host, Download, Tags, Inspect, Display Object Versions) and 'Object Info' for the selected object '14cf279a4924d4398598928667b31e9'. The Object Info shows: Name: 14cf279a4924d4398598928667b31e9, Size: 3.7 MiB, Last Modified: 1 day ago, and ETAG: e07467ffb69828fa5754e2754ce6bc44.

Categorizer Model

Our text categorization system uses a pre-built language model called BART-large-MNLI. It works similarly to how a human would compare two pieces of text to determine if they're related. Instead of using traditional categorization methods that require extensive training data and predefined rules, this approach treats the problem as a text comparison task. The system takes your input text and compares it against descriptions of each possible category, determining which category best matches the content.

The BART-large-MNLI model comes pre-trained, meaning it already understands language patterns and relationships without needing additional training data from our specific use case. This is particularly useful when we need to add new categories or work with limited examples. The system works by taking each piece of text we want to categorize and comparing it against every possible category description, measuring how well they match. The 3 categories with the strongest match are selected as the final classification labels.

LlamaIndex & LlamaCPP

LlamaIndex is a data framework designed to facilitate the ingestion, structuring, and querying of private or domain-specific data using large language models (LLMs). It provides data connectors to ingest various data sources and formats, such as APIs, PDFs, and SQL databases, and structures this data into indices optimized for LLM consumption. This setup enables natural language querying and conversation with the data via query engines and chat interfaces.

LlamaCPP is a C++ implementation within the LlamaIndex framework designed to integrate LLMs with vector storage solutions efficiently. It supports multiple vector store options, allowing for flexible and efficient data querying and storage. Additionally, LlamaCPP offers observability tools to monitor the inputs and outputs of LLM applications, ensuring optimal performance and reliability.

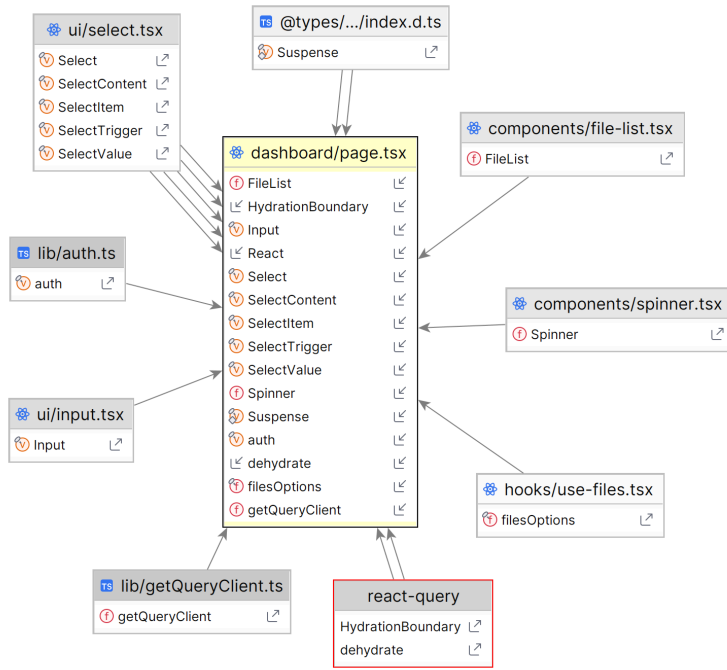
LlamaIndex and LlamaCPP were integrated to enhance the application's natural language processing capabilities. We ingested and structured various document formats into indices suitable for LLM consumption, utilising LlamaIndex's data connectors. LlamaCPP facilitated efficient interaction with these indices, enabling advanced querying and conversational capabilities within the application.

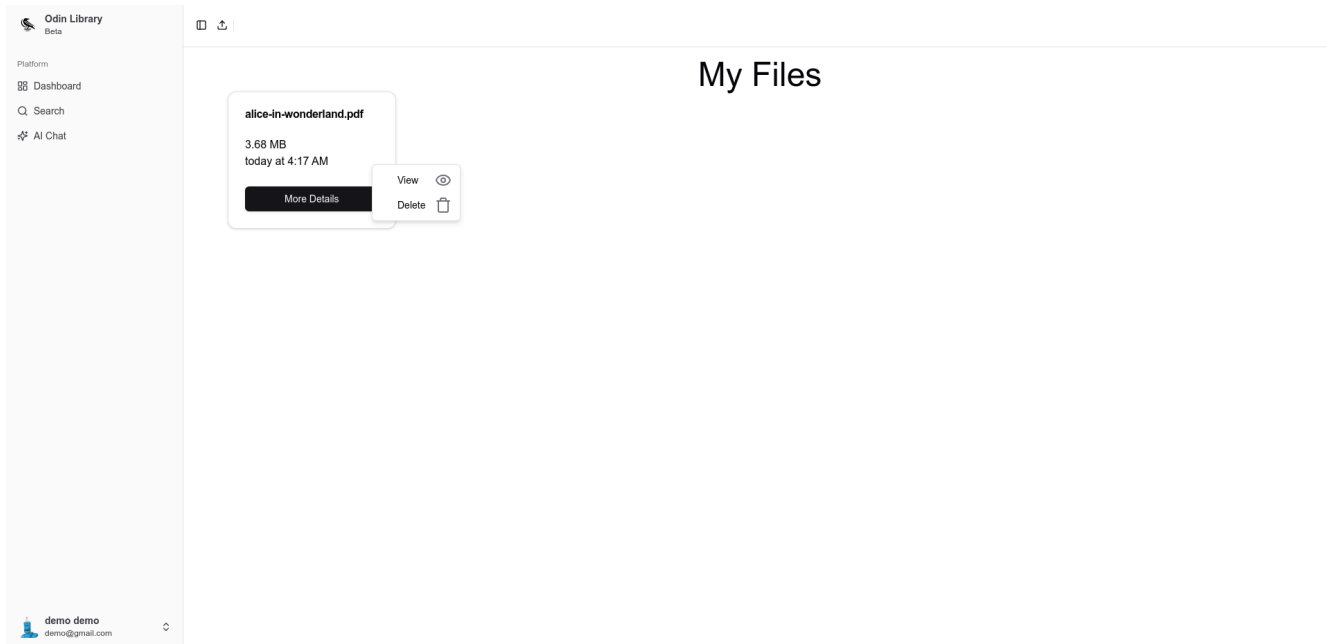
Features

In this section, we will specify the workflow of each feature presented earlier, what processes are associated with the backend, and the technologies previously explained for proper working.

Dashboard

The dashboard is the application's simplest yet most important view. From here, the user manages their files like a file explorer, viewing or deleting them. This page is server-side rendered to optimize it and improve the user experience

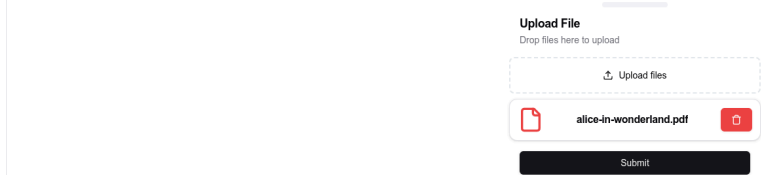
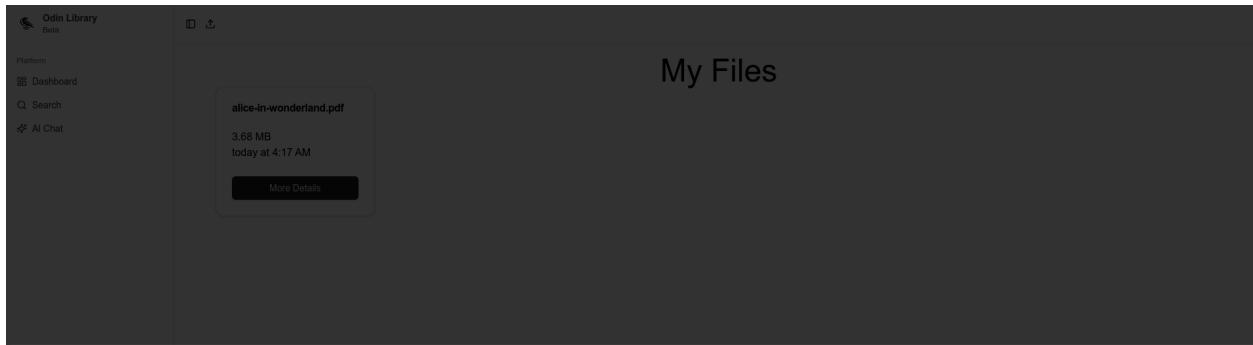




File Upload

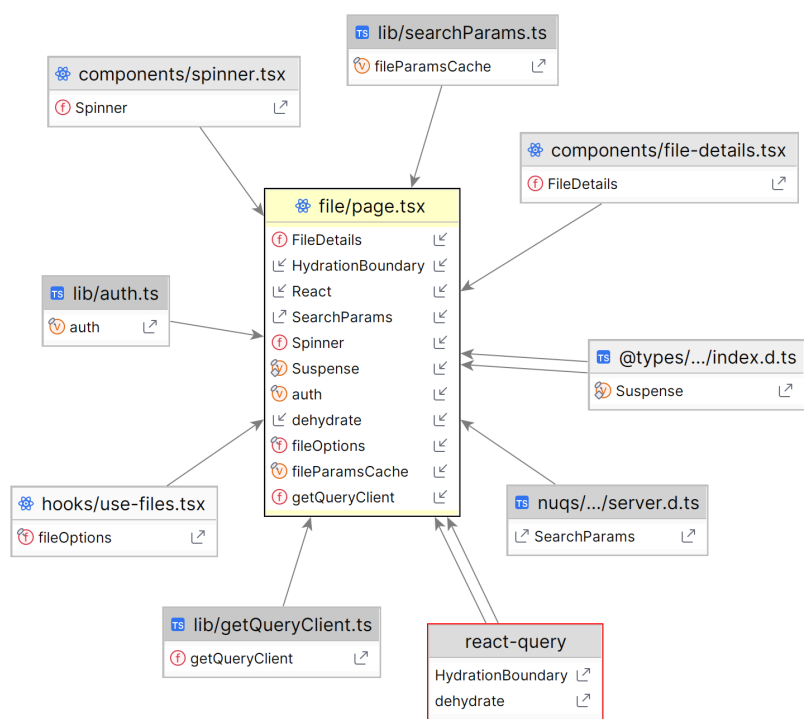
The first step of the application, after the user logs into the applications, is to upload files. Files are the cornerstone of the application. Thus, the upload was the file challenge we addressed. On the presentation layer, we tried to make a simple yet efficient solution where the user can upload a file on the different pages of the application. For this, the navigation bar on the top of the application has an upload icon where the user can drag and drop the desired files to be uploaded. On the backend part, when the document arrives in the backend, it is stored in the Minio bucket to be saved as an object. Then, the document is processed by the Docling library to transform into a markdown and stored in the vector database. When the processing is finished, a notification will be sent to the user, and the file will appear on their dashboard.

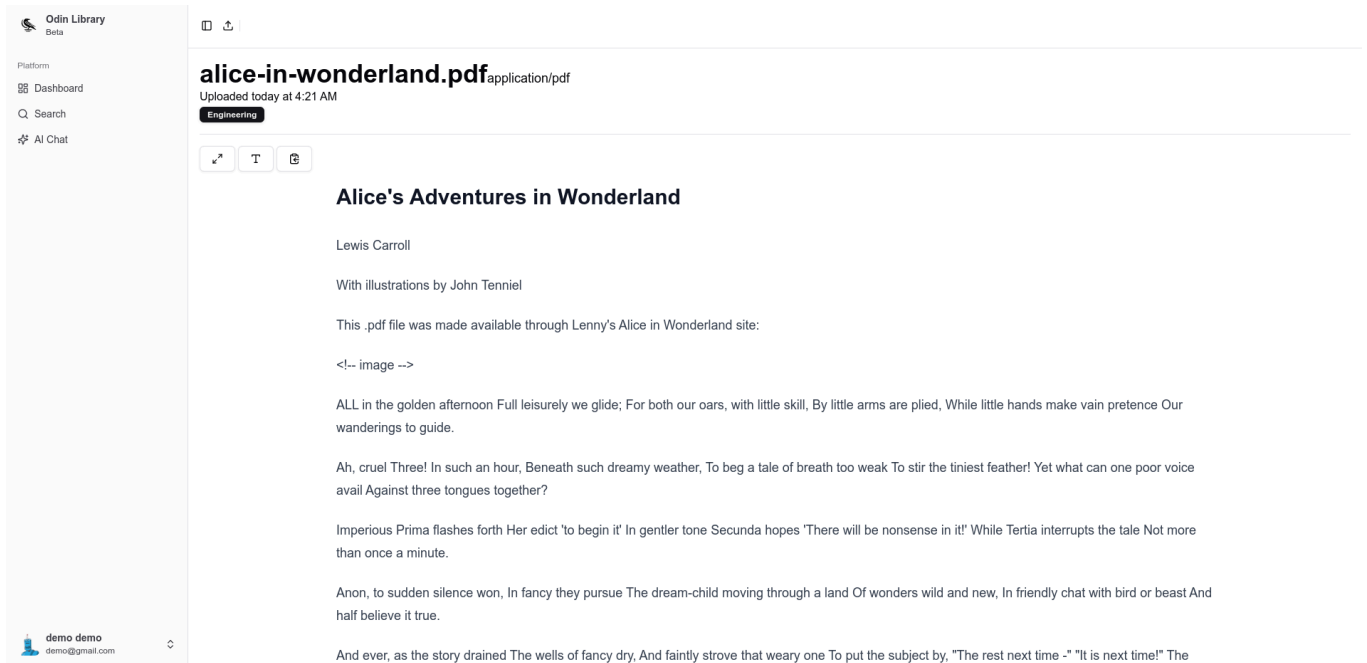




Document Visualisation

When the user selects the file, it will be presented with a view of the document in Markdown, which is the parsed version. In this view, the user can change the size of the text to help with the reading, select between the compact and expansive view of the document, and copy the content of the document to the clipboard. This configuration is stored in a Zustand store to make it persist between user sessions. This visualization might have issues if the Docling parsing misses something and does not have the best reading experience for the user.

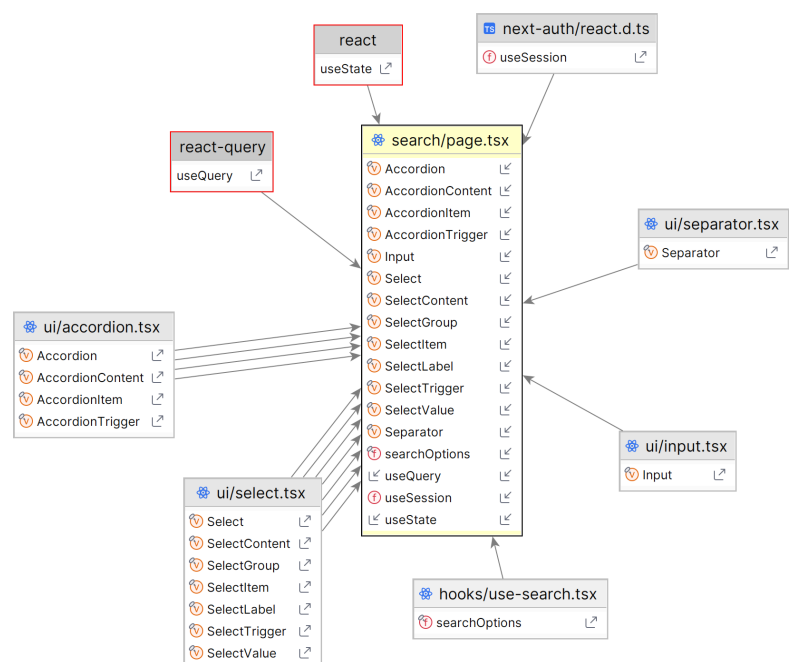


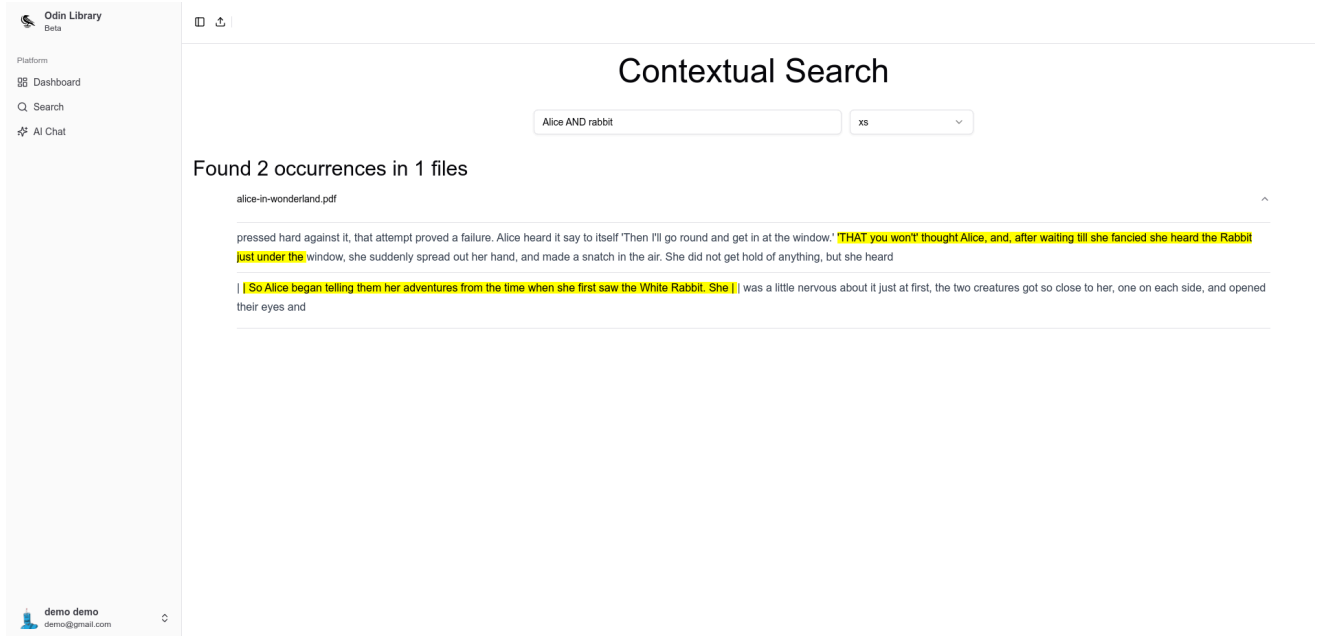


Better Search

One of the typical limitations when searching in the documents is to find the key occurrences of certain words. This feature was inspired by the common feature of text editors and IDEs, which see all the occurrences of a particular word or a group of words and display it where it is located to understand the context. For this feature, the user can use a query-like search with words such as AND, OR, and NOT to filter the desired keywords. The result will be a set of documents where the query appears and a list of occurrences associated with each document. The level of context can be filtered using different sizes to let the user decide the amount of desired context.

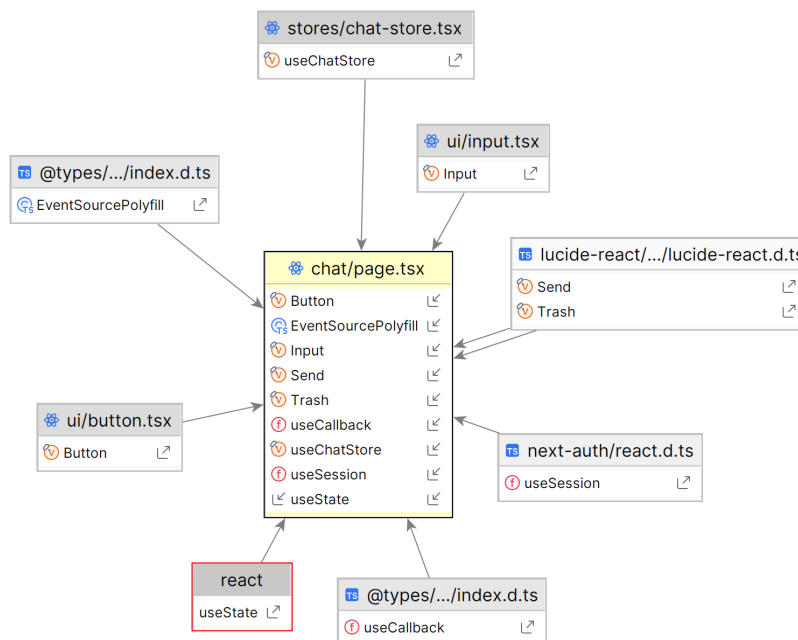
On the backend, this feature uses the capabilities of the vectorial database to search the occurrences of the query. However, since it is not a deterministic approach, some occurrences might fail and give some false positives.

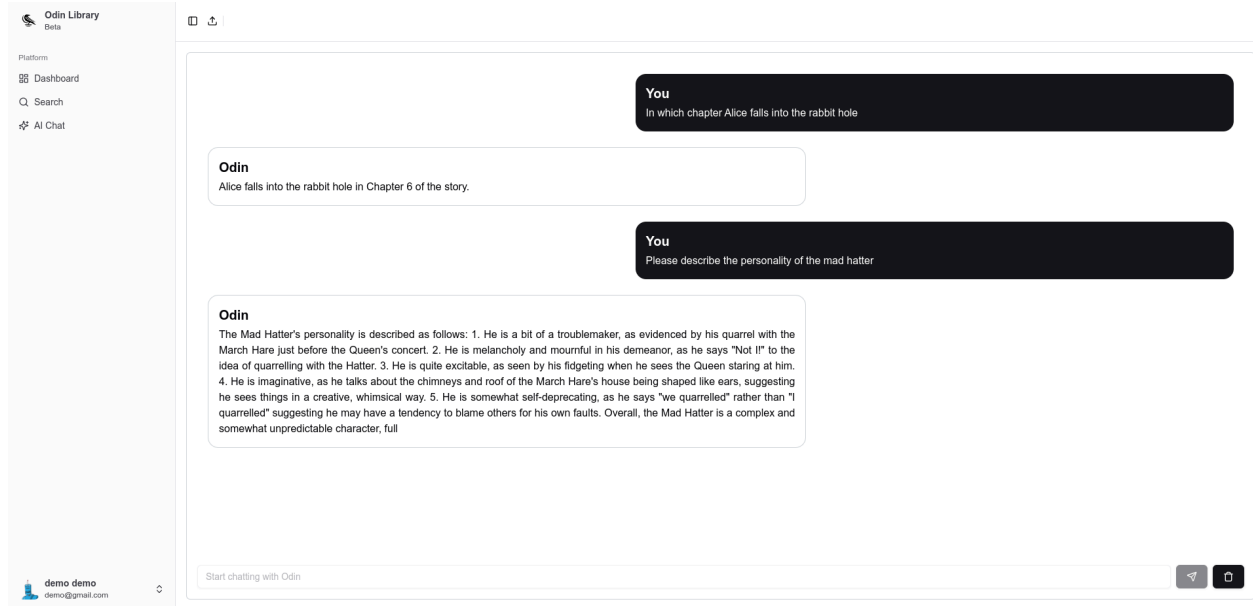




Chat

Lastly, the chat feature enhances the workspace by providing a search engine using natural language. When using this feature, the user can search for knowledge using a chat-like system where they ask some questions, and the model will answer them. For this feature in the backed part of the project, the LlamaIndex and LlamaCPP with the Hermes-3-Llama-3.2-3B-GGUF model, which allows a templating questioning. We did some prompt engineering to achieve the desired goal where the model can answer the requested questions in a perceptible way. On the front end, to give some realism to the live stream, the EventSource was used, which allows direct communication with the backend, giving the sense that the chat is really occurring in real-time. Moreover, the current conversation is stored, making it possible for the user to start from where it ended, having the possibility of clearing the chat to start a new discussion.





Limitations

Some limitations need to be addressed. Due to the lack of time and the size of the team, some features need optimizations, such as better search, which requires improvements to the search algorithm to filter out false positives. Since this project uses large language models and other heavy technologies, the machine specifications will be the bottleneck, making the file processing, chat, and search slower than when used on a better machine.

Another limitation was the choice of architecture. Due to a lack of time and people, the application was developed by grouping all the processing in the API, which made it slower. The optimizations will be addressed in the next section.

Conclusions and Future Work

This project challenged us not only because of the scope of the course, which was the application's presentation layer but also because of its backend part. With this project, we developed our skills in delivering a pleasant and functional application to the user with applications in real-world workflows. Nowadays, machine learning and large language models are state-of-the-art ways to make task-solving easier, and by developing Odin, we have delivered a small yet capable application that helps people analyze documents and research.

If we continue the development of this project, the first problem to address is the coupling in the API layer. To handle it, the architecture will change by having a new block called process workers, which will communicate with the API through asynchronous calls of a message broker such as Apache Kafka or Rabbitmq. This worker would deal with the processing of the docling and categorization, freeing the API just to serve HTTP requests. Then, the development of a better and more robust model to enhance the quality of results. In the presentation layer, better

features such as collection are developed to allow the user to organize their files in a custom way.