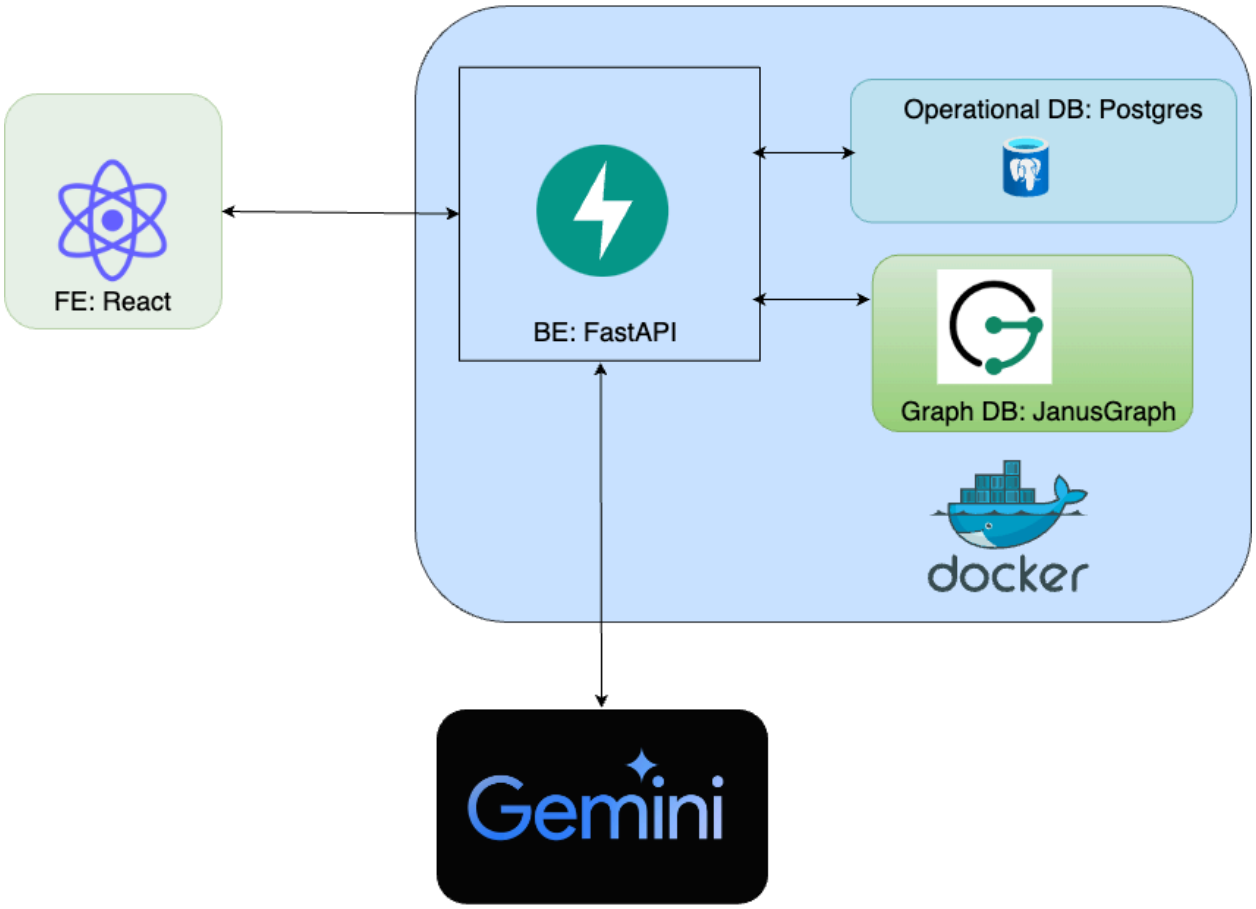
An overview diagram of runtime components



An overview diagram of code components

Front-End

Desktop		
Upload Module	**Network Service**	**Visualization Module**
Suggestion: Filepond	Suggestion: Native or Axios	Suggestion: D3 Cytoscape

Back-End Python 3.11

REST ENDPOINTS

Data Source Module

PDF Module
PyPDF2
Langchain
pandas

LLM Module
Gimini

Graph DB Module
gremlinpython

PostgreSQL

Docker image: latest

JanusGraph

Docker image: latest

OLLAMA with Mistral 7B

Docker image: latest

A summary of the underlying technology stack

The whole application is thought of as a web app. It will have its **FE** (UI) and a **monolithic BE** with all its services containerized with **Docker.**

Front-end:
We are going to use **React** with its libraries

- Upload module: We are using **Filepond**
- Networking/Communication: **Native** or **Axios** for sending **HTTP** requests
- Visualization module: We are going to use **D3** to show the user the knowledge graph

Back-end:
The backend part will be built around the **Python** ecosystem. The main app will use **FastAPI** as a web framework**.**
The supporting services will be the following:

Operational DB → **PostgreSQL**: Fast, reliable, scalable database.

GraphDB → **JansuGraph**: One of the most popular open source graph db which will help in saving the knowledge graph.

LLM → We employ **Gemini**, a large language model from Google, due to its large context window and fast processing speed, as our application handles a high volume of requests. Among many features, it exposes an endpoint that we can use to interact with a model.

We will use a pre-train LLM model, **Gemini,** to get a meaningful relation between the data. Using different prompts, select values for nodes and edges for meaningful graph

Packages which will help in working with PDFs are going to be:
PyPDF2: Read pdf byte data and transform it into a string.
LangChain: Is a framework designed to simplify the creation of applications using large language models. It also has other helper functionalities. Our use case is for text chunk creation to be fed to the LLM and in the future to make calls and build a Chatbot.

Pandas: Is a Python package for pre-processing and manipulating datasets.

Development and Deployment:
Docker: For consistent development environment and isolation
Git: Used for version control and efficient collaboration.

A textual explanation of the diagrams and choices

Based on the team's experience and initial requirements for the project, we decided on the below architectural & tech stack decisions to deliver within the deadline a final product.

Architectural Decisions:
Front-end: This layer handles the user interactions with the application. It makes it easy for the user to upload documents and interact with the graph.

Back-end: This is the logic layer, handling the data processing, for each document and creating the Graph. We agreed on a monolithic approach with supporting services since it is the fastest and easiest implementation to deliver an MVP.
Technology Choices:
The main technology used is reliable, open source, and offers great scalability.

React: Chosen for robust ecosystem, it is a component-based architecture. Effective UI and better user interaction.
D3: offers low-level control over every aspect of data visualization. can be integrated seamlessly with other JS libraries.
Python: The main language is Python since it was per the client's request. This led to the tech stack described above. It is easy to use and has extensive libraries, is well-suitable to develop backend logic, and works with the LLM model.
FastAPI: Essential for handling requests and interacting with the database has asynchronous support and is easy to learn.
Gemini: aligns well with our budgetary constraints due to its availability through Google's free credit. Additionally, it has a large context window and can be accessed quickly via the API.

Interaction Between Components:
The front-end communicates with the back-end through RESTful APIs. The back-end utilizes the LLM for processing the text chunks to create entities (nodes) & relations (edges) for the graph.
2 Databases, an operational one to save info. about the uploaded PDFs and a graph db to save the generated knowledge graph.

Scalability and Maintenance:
With a monolithic approach, it's easy to maintain and work as a team with different experiences.

Also, the app is very easy to scale horizontally or vertically. The only problem is if one of the services/modules needs to be scaled the whole app needs to be scaled. In the future having module separations helps to later substitute them with their service for a microservices architecture.