Architectural Requirements

Architectural Design Strategy

We used a hybrid strategy mainly making use of decomposition i.e. we broke down the system into components. We have a frontend component that handles the display and relay of information to a user. We have an API component that handles/dispatches services based on requests received from the JavaScript client running on the frontend. We have a backend component as well. Although API is commonly considered a large part backend we decided to label and interact with them separately because of the largely differing functions that would be carried out by our “backend” and the API.

Our design approach kept quality requirements in mind. We used some well-established open source technologies in our components where applicable instead of implementing our own as we believed this would improve both performance and reliability (the software would both be better tested and optimized than solutions we would attempt to design on our own).

This hybrid strategy was used as it allowed for separation of concerns which improved the overall structure and testability of the system. This made it much simpler to isolate parts of the system for review and allowed for parallel development across team members. The QR driven development aspect of our strategy was deployed simply because we felt making initial coding decisions within these system requirements from the ground up would be much simpler than fully designing a system then attempting to optimize to fit within said requirements. We also felt this would allow for more transparency with clients about what is and is-not feasible.

Architectural Styles & Patterns

Monolithic Architecture: This is a traditional software development model in which a singular codebase handles all functionality of the system. All components are integrated into a single unit.

We used this approach largely due to the simplicity of the architecture. Scalability of the system wasn’t the largest concern required for the system so attempting a microservice driven system would add unnecessary complexity to the production and maintenance of the system.

Quality Requirements

There are numerous quality metrics that our system aims to satisfy, covering a range of attributes such as performance, scalability, maintainability, usability, and security. However, for each category, we identified and prioritized one core quality requirement that was most critical to the success of our project. These selected requirements guided our architectural decisions and implementation strategies:

Performance Requirements:

- Dashboard Load Time: All dashboard pages must load within 2 seconds under normal network conditions. This was heavily prioritized as we felt that it was of the main functionality of the system and thus should always respond quickly to the user to provide relevant information.

- API Response: API endpoints must respond within 500ms for 95% of requests under typical load. We decided this was important as any user would ideally prefer a responsive system for time sensitive tasks

Scalability Requirements:

- Support up to 50 concurrent users with no degradation in performance. This was prioritized as the client stated the technology would often be employed for team use

Security Requirements:

- Access Control: Implement role-based access control (RBAC) for Managers and Team Members. This was seen as a top priority as sensitive business information is being handled thus only authorized users should be able to access it.

Reliability Requirements

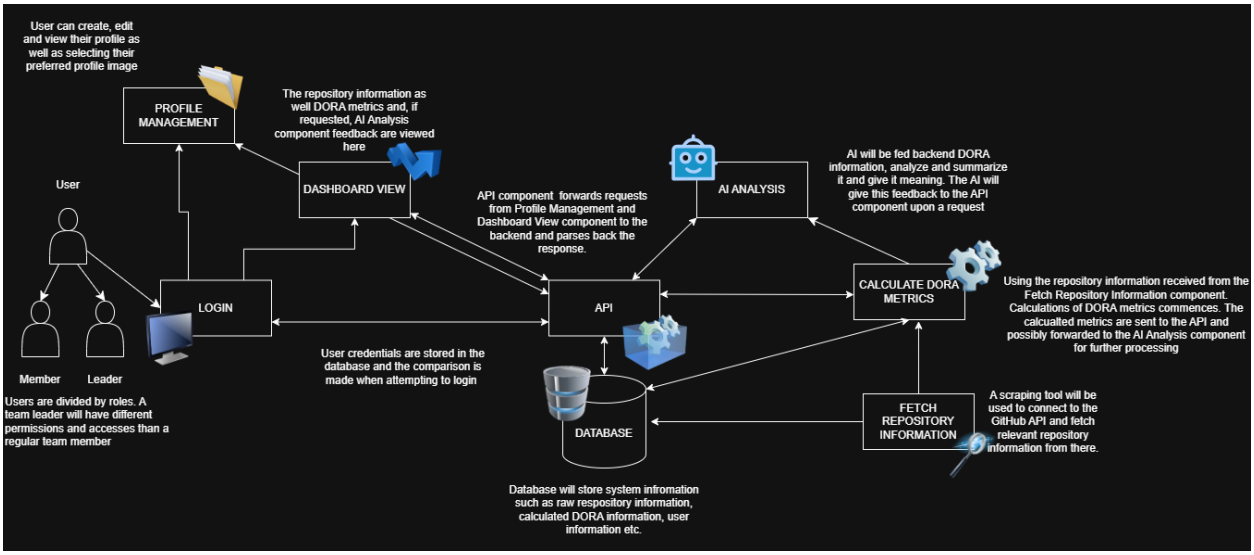
- Uptime: Achieve 99.5% annual uptime, excluding scheduled maintenance. This was important as it was emphasized a lot of work in the business is carried out through the use of Git-based workflows, it would thus be beneficial to have the system available to assist users and improve productivity be readily available always.

Maintainability Requirements:

- Code Quality**:** Maintain a codebase with at least 80% unit test coverage and enforce consistent linting rules via CI/CD pipeline. This was prioritized to ensure the code was well tested under all circumstances and to prevent any large scale system failures down the line.

Architectural Design & Diagrams

A detailed high level diagram showing the interaction of our components:



Architectural Constraints

- To ensure accessibility for all team members, we made a deliberate decision to use open-source tools and frameworks exclusively. This ensured no software licenses etc. were required for development, deployment or testing. This helped us maintain a consistent development environment across all contributors.
- Our team was geographically dispersed, with members residing far away from one another, this meant physical meetings were often not an option. This necessitated on a strong emphasis on remote development and the use of remote collaboration tools. We used platforms such as Discord and WhatsApp to coordinate tasks and decisions.
- The client provided us with a large degree of design autonomy. The main concern was having a system that met their expectations and use cases. This however placed a large responsibility on the team to set and meet appropriate and feasible architectural and technical standards.

Technology Choices

- Frontend – Login, View Dashboard, Profile Management

  We decided to use React for the design of our frontend as it was recommended by the client. Upon further research we discovered React offered a component-based architecture. It replaced the use of traditional DOM for handling state changes regarding our JavaScript.

  - This was advantageous as the declarative method of building UIs made development faster and easier to maintain. This aligned with our priority for maintainable code.
  - It was however disadvantageous due to have to learning a new technology under short notice.

  Another option considered was using traditional HTML and CSS for the frontend. It was the technology we had the most experience with as we had used it extensively for previous projects. Pages would be modelled using normal HTML and styled using CSS with the events/requests being listened to and handled by an accompanying JavaScript file for each webpage.

- The main appeal of this approach was our familiarity with HTML and CSS. Each member of the team would be able to collaborate and contribute in the development of pages as well as consult in the event of errors.
- A considerable drawback of this approach would be the manual DOM management as the pages grew increasingly complex. We planned for having pages that could dynamically change states upon receiving certain requests from a user and felt having to manually set up DOM and handling conflicts may prove challenging for a system of this scale.

A third considered technology was angular. This option was considered due to our experience with it (more than React but less than traditional HTML/CSS) and because its framework supported typescript which was a technology that was considered for the development of our API. We considered using this technology as the entire framework for our project building it from the ground up.

- This appeal of this specific approach was its All-in-one framework – we would completely integrated tools for testing, development, HTTP services, routing etc. all provided by one technology stack. This would improve reliability and align with our desire for maintainable code as we could use the in-house tools for all of our testing.
- We were however swayed away by the steep learning curve of this technology. With limited experience using Angular we felt that the time investment required to fully understand and implement the framework would reduce our development velocity and introduce unnecessary complexity during the early stages of the project.

- API

We decided to develop the API using JavaScript. We made use of many of the packages and libraries made available in the JavaScript framework. Requests from the frontend received would be handled using the appropriate service route. We used modules from the npm package such as jsonwebtoken and bycrpt.js to create JWTs and to hash passwords respectively to improve security.

- JavaScript was advantage ous as it's a very widely used language. It has good performance and is lightweight. It was also appealing because it is generally an easy language to pick up and further learn considering the team was already familiar with it.
- A drawback of this approach that we faced was handling of conflicts between npm modules. It was important that we only used the necessary modules and that including any didn't cause clashes

A technology we briefly considered was Typescript. It was a superset of JavaScript with the addition of strong static typing for variables.

- This addition would allow for better code quality and improved maintainability of code.
- We were however swayed away from type script as we felt the added tooling overhead was unnecessary for the current scope and size of our project.

The third considered technology was Python. It would've mainly used packages like Flask or FastAPI for development. From that point routes/services would be handled similarly as it would be in related JavaScript and Typescript programs.

- An API designed in Python would boast great simplicity and readability. The use of well-established Flask and FastAPI packages would also ensure reliability in the development of our API
- The main con of this was managing cross-language communication. It seemed unnecessary to make an API in Python and attempt to communicate with the underlying JavaScript from the frontend component.

- Backend – Fetch Repository Information, Calculate DORA metrics

  The function of our backend component was to scrape data (that would be used in the calculation of DORA metrics) from GitHub repositories.

  The technology we decided on was Octokit. This was the official GitHub SDK for JavaScript. Octokit provides a well-documented, actively maintained, and strongly-typed interface for accessing repository data.

  - This was beneficial as it simplified making API requests, handling response and managing authentication. It eliminated the need for generating JSON web tokens manually. There is also always the added benefit of improved reliability when using official software
  - There wasn't any major drawback to using Octokit over other technologies except for minor compatibility issues involving maintaining the system if there ever is an update to the GitHub API.

  We also briefly considered the use of Python to interact with the GitHub API. It would make use of the requests package in the Python framework.

  - As previously mentioned working with Python would provide improved simplicity and readability of code.
  - The drawback would be similar to that previously mentioned. It was deemed unnecessary to try and manage the cross-language communication when there are alternatives that eliminate this issue.

  The last technology we looked into would be using manual curl requests. We would need to generate an authenticated GitHub token before interaction can commence however.

  - The advantage of curl would be its ability to function with any language. This would make it suitable for any environment.
  - A large drawback of this approach would be the security risks. It was considered unsafe to directly pass tokens in command strings, shell scripts and other curl requests to the GitHub API.

- AI Analysis

For our AI component we decided to use Ollama. This technology allowed us to run open-source language models locally and interact with them using our API.

- Ollama is generally a fast, quick responding LLM. The use of this technology would help us meet our performance quality requirement for a responsive system. Another appeal of this technology was its model management. It could easily download different models where they would be needed by different agents. This allowed us to tailor each model to the task at hand—for example, using smaller, faster models for lightweight summarization tasks, and larger, more capable models for deeper analysis,
- A drawback of this technology was the significant memory and compute requirements. Running large models efficiently often requires a high-performance GPU, and support for older or lower-end hardware is limited.

An alternative considered was LM studio. M Studio is a desktop application that allows users to run and interact with open-source language models locally. It is built on the same underlying technology as Ollama—llama.cpp

- One of its key advantages is its ease of setup. It provides a user-friendly graphical interface that allows models to be downloaded and tested with minimal configuration
- It is however less suited for backend integration. For our system where AI functionality needed to be imbedded into the full-stack application and accessed via API calls, LM Studio's GUI based nature made it an impractical choice.

A second alternative considered was the OpenAI API. This is a cloud based service and provide access to language models such as GPT-4 and GPT-3.5

- One of its main advantages would be the ease of integration and setup. This approach requires no local setup or hardware investment, and would provide highly reliable and consistent AI responses.
- However, we ultimately decided not to use OpenAI due to privacy concerns as we did not want to send sensitive user data to a third party cloud provider. This would violate our quality requirement regarding designing a secure system. It also conflicted with our design decision to only use open source software as OpenAI API services require a subscription.