CVWO Final-Assignment Submission Emily Ong Hui Qi (A0240634M)

Embarking on this assignment has been very fulfilling and enriching. What I had originally thought was a simple task evolved to be something more complex than I had expected and filled with more intricacies that I would never have thought of if not for completing this assignment. In fact, completing the assignment seemed almost an overstatement at this point. At the start, I was very ambitious and had many plans to integrate complex functionalities and tools into the application, as what I had set out in the Mid-Assignment submission. I wanted to set up a complete continuous-integration and deployment pipeline, use Cron jobs, and go beyond the scope of the assignment to enable sharing and configuration of user roles, just like Google Drive. Although some of these tasks were completed, they were not quite as ideal as I would expect.

This brings me to one of the first learning points from this assignment: **Planning and managing expectations.** When I started out this project, I have a rough skeletal conception of how the application would look and behave. I wrote down the features that I wanted, designed the database diagrams and researched on possible tools and libraries that I would use. However, what was sorely missing was a timeline- how I should exactly be spending my time on. Software development is an incremental developmental process- what we should not be doing is to set a goal of spending 3 days to build feature XXX (like what I had initially done). Instead, we must be clear of what exactly we need to build feature XXX, and reframe our goal such that we know out of these 3 days, we might spend 1 day on part A of feature XX, and another day on part B of feature XXX etc. What was regrettable was that I only realized this and changed my strategy from trying to do everything in one-go, to incrementally building up each component of the application, later in the project. Finally, it was also vital to manage expectations- each project needs its own unique-selling point, instead of being jam-packed with various features (that no one might even know how to use!). It is important to prioritize core functionalities, which was why I decided to spend some good amount of days refactoring code, before moving on to solidify existing functionalities, making them better and better.

On another note, what was a great takeaway for me was learning more about **designing user experiences**. Often, we build projects from our own viewpoint: What do we want to see? How should the functionality behave? But what might seem intuitive to us might be totally foreign to another. This led me to understand the importance of collecting user stories, reframing perspectives, and building the project for *others* and not for *myself*. During the developmental phase, I constantly tried to adhere to the user specifications that I had set out in the mid-assignment submission and experimented with ways to ensure the application was user-friendly. One of the things that had bugged me the most was how error messages should be communicated from the backend to the frontend, and eventually to the user. For this application, I wanted user-facing error messages to be explicit and actionable- users should not have to spend too long thinking about what went wrong, and they should know how to rectify the error without spending too much time fiddling around. I categorized the various errors as follows: 1) Resource is not found, 2) Server error (i.e., an error occurred on the backend), 3) Unauthorized error, 4) Type mismatch of request payload and 5) Conflict error (e.g., creating a resource that has already existed). Now, server errors were logged and communicated to the user by prompting

them to refresh the browser, while a conflict error when the user is signing up for a new account would prompt the user to use a different email address, for instance. There were of course many improvements to be made to this system, and what I wanted out of this codification was for the frontend to be able to expect and infer the error, rather than receive a long chain of fuzzy error messages. Another aspect that I had considered in designing the user interface was to minimize the number of clicks and "hidden" features. I wanted the interface to be clean and explicit, so users do not need to go through the trouble clicking on various tool icons to figure out what they might be for, or to step through a sequence of steps when they could have done it in a single step. The idea of making it self-explanatory hopes to simplify user-onboarding, and this is aided by seeding sample data for the user. One inspiration that I took was from Trello: Newly signed up users see a series of boards and cards that pointed out to them what they were for and contained guides on how they can expand upon them. Lastly, my goal was on optimizing for small-screen devices. One of the primary design strategies was to make full use of CSS flexboxes and grids whenever possible. In the application, for instance, the boards that the user have access to are displayed in a tab component (on the top) on wide screens, while the same boards are displayed in a dropdown component (also on the top) on small screens.

Nevertheless, there were many aspects of the application and coding standards that I am proud of. I learnt much about **design patterns, coding standards and code quality.**

One of my takeaways was the idea of atomic web design, where components are built on top of each other much alike the atoms, molecules and organisms in biological systems. This made very intuitive sense for me, as I had commonly seen React projects structured "component-wise" where components used to make component XXX are parked under the same folder as component XXX, or other similar strategies, which seemed too convoluted. Throughout the project, I tried to adhere to the idea of atomic web design, building components that are reusable and abstractable (by prop designs), ensuring that components at the lowest level are purely presentational.

Another takeaway was the learning of Golang and React, both I was previously unfamiliar with, and it turned out to be an awesome experience despite a tumultuous start! Initially, I relied on blog articles and documentations to set things up, such as configuring JSON Web Tokens for user authentication, or how to use Redux to manage state, but as I slowly became more comfortable with development, I was better able to accomplish my goals and things began to pick up pace!

On the aspect of backend development, I learnt a lot on database design and how to model relationships with primary and foreign keys. One of the pain points during development was that the SQL queries were quite slow. As I used Gorm as an ORM to interact with the database, it was harder to investigate the cause as I had to add debug statements to print out what the resultant queries were. Notably, it happened after I switched from file-based SQLite to PostgreSQL. And thus, the next milestone of this project is to fully investigate slow queries, since this affects user experience greatly. Some ideas to approach this that I can think of is to connect a performance monitoring dashboard first. This allows us to know which queries are the slowest, and to start fixing from that query. To understand the problem better, I would try writing raw SQL statements and compare that with using Gorm prepared statements,

and to also look further into Gorm's documentation on other speedups or optimization tricks.

Furthermore, I am glad to subject my project under strong coding quality standards right from the start: I set up linters, used Typescript and tried to organize my code and comments in a structured manner. This proved to have pay off later as the project scales. It became easier in a way to build the project as many components and functionalities are reusable, and it was also easier to debug any problems with the application. It was also a fun challenge for me to set up Makefile and scripts to handle common shell commands, such as running or deploying the application.

One of the things that I am satisfied with was also being able to deploy my application and configuring a domain (https://tuskmanager.rocks/) and subdomain (https://app.tuskmanager.rocks/). In deploying the application, I chose to deploy the backend on Heroku (https://tusk-manager-backend.herokuapp.com/), as Heroku provides many out-of-the-box addons, such as connection to a Postgres service, and strong command-line interface support. It was also easy to use the dashboard on Heroku to configure the application or to view the logs in case any errors happen. However, the backend folder was originally in a monorepo and it was not possible to set up automatic deployment on Heroku whenever a commit is pushed to the main branch on Github. Thus, I picked up Docker and set up Github Actions to build a Docker image and push to Heroku directly whenever a commit is pushed to the main branch. For the frontend, I used Vercel for the ease of setting up. It was very easy to connect a Github repository to Vercel, and Vercel offers many integrations and configurations with Github. Finally, I made use of the Github education PRO account to obtain a free domain name for the application. I learnt a little bit more on how to configure the domain name system to point the domain to the Vercel site, which was quite a frustrating but worthwhile experience! On the next step, I want to improve the pipeline and set up tests for both the frontend and backend applications, which I did not manage to accomplish during this iteration due to the lack of time. More so, I would like to learn how to write good unit tests and what it means to improve code coverage, as well as setting up a "Storybook" for the React frontend, which I believe is vital as the project scales, and more components are added to the application. Finally, I would like to explore the idea of setting up development, staging and production environments. Currently, the Postgres instance is connected similarly to both development and production environments, which makes data messy and harder to analyze in later stages of the project. Moreover, changes are pushed directly from development to production, which falls short of the aspect of quality assurance. However, such features are likely to have minimal impact as of now, since the project is maintained by one person and used on a small scale.

In conclusion, I am glad to have taken up this assignment. There were many aspects that I was prouder of, some less, but it was an overall awesome learning experience, that I never expected myself to have invested much time and effort into. It is probably the largest project that I have ever done, exploring both backend and frontend development, and being able to make decisions about the project direction and specifications provided me with a strong sense of ownership. I hope to continue this spirit on a larger scale and be able to learn how to better work with others on real-world projects that are no longer prototypical, but rather, can make a genuine impact on someone else's life- and that, I believe is the value of software development.