CS261

SOFTWARE PROJECT

PLANNING & DESIGN

February 6, 2023

Group 24

Contents

1 Technical Description

Unsurprisingly, we find little precedent for out project specification. When restricting our list of features, we find some analogs in project management software (e.g. *Jira*, *monday.com*) as well as financial software solutions, which both generally allow for tracking (software) projects over time with some selection of metrics.

The novelty here largely stems from the *evaluative* functionality, which will this be the focus of our design discussions.

1.1 Model Metrics

Name	Description	Related Metrics	Source of Information
Total Budget	N/A.	Time Deadlines, Team-size, Scope Creep, Money Spent	User inputted value.
Team Size	N/A.	Total Budget	User inputted value.
Money Spent	Money spent so far in project development.	Project Progression, Time Deadlines, Total Budget	User inputted value.
Project Progression	Percentage of the to-do list of project tasks complete.	Time Deadlines, Money Spent, Git Commits, Git Bugs, Scope Creep	Calculated from data from the to-do list for given project.
Team Skill (1)	Average number of years of professional software engineering.	Communication, Team-size, Time Deadlines, Team Skill (2)	User inputted value.
Team Skill (2)	Experience each team member has with tools being used (scale 1-10).	Communication, Team-size, Time Deadlines, Team Skill (1)	User inputted value.
Scope Creep	User inputted rating of scope creep 1-10, 10 being double the initially projected scope, 1 being half.	Total Budget, Time Deadlines, Team Skill (2), Communication	User inputted value.
Communication	Overall evaluation of a teams communication calculated from a network flow graph.	Time Deadlines, Scope creep, Team Skill (1), Team Skill (2)	Network Flow evaluation described in section 1.4.
Git Commits	N/A.	Time Deadlines, Project Progression, Team Skill (2)	Automatically detected using GitHub API once the user links to the projects GitHub repository.
Git Bugs	N/A.	Time Deadlines, Team Skill (1), Team Skill (2), Project Progression	Automatically detected using GitHub API once the user links to the projects GitHub repository.
Time Deadlines	Cumulative total lateness/earliness of all tasks which are either late or have been completed early.	Git Bugs, Git Commits, Communication, Money spent	Automatically detected from a calculation done when the user inputs the deadlines for each individual task, weighted against earliness.
Parties Involved	Number of unique parties working on the project.	Communication, Team Skill (1), Team Skill (2)	User inputted value.

1.2 System Architecture & Design

1.2.1 Overview

In this section we cover the design choices made in terms of our software components and interfaces, discussing how the design will fulfill a range of important factors. We aim to provide a clear understanding of our key system architecture features, aiding us when we are in our development stage.

1.2.2 Extensibility

As the backend is written using lambdas, which are mutually exclusive, the functionality of the product can easily be added to. The UI will be able to updated easily as it's all cloud hosted and downtime for updates will be minimal due to the small size of the product.

1.2.3 Reliability

Given correct and accurate inputs the model should be able to output useful data as to metrics causing risk and overall risk of the project. This is because it's purely mathematical and will always produce a trustable output given correct inputs. The initial risk rating will likely be fairly inaccurate given the model relies heavily on metrics that are only inputted as the project progresses. This does however mean the model will be reliably once a few weeks of data are inputted into it, and will only be more accurate as time goes on.

1.2.4 Robustness & Fault-Tolerance

If an individual lambda fails the rest of the program will still operate such as the UI won't go down; lambdas are instance specific so even if it breaks for one user it will not impact other users on the product. If the UI breaks the entire database will be inaccessible for users as it's the only point of access for non-developers. The database will be hosted on AWS which is highly reliable but technically if it does go down the product will be inaccessible for that duration, backups are frequent and will enable viable versions to be restored with ease.

The program will be expecting reliable data, meaning if incorrect data is inputted it will largely skew the accuracy of the model but to combat this inputs can be changed at anytime, allowing errors to be corrected with ease.

1.2.5 Compatibility & Portability

The website will allow for devices of any standard screen size to view it. As it is a website it can be accessed by any device that can display HTML and should have a high uptime as it is run off of Netlify and AWS services, both being trusted cloud services.

In respect to portability this means it can be accessed accross multiple devices and even on a users phone, making it as accessible as possible in the current technical age.

1.2.6 Modularity and Reuse

The backend code will be written using lambdas which is innately modular as each lambda is completely independent from another. This allows for functionality to be removed, edited and added easily, without breaking other components and needing complete recompilation of the system. As the model will adapt according to the information gathered from past projects and only improve in its estimation of risk and success, the product will be only more useful and accurate in the future, implying high reusability.

1.2.7 Security

The database will have encrypted passwords and by using AWS the general program security is automatically up to current standards.

1.3 Nonlinear Optimisation

The variety of and interplay between metrics which contribute to project success/failure demands a model which is not only general in the *types* of metric in considers, but also able to incorporate their *relationships* in a versatile way.

A model based on nonlinear optimisation fits these requirements well, as it allows us to model an arbitrary number of relationships between arbitrary variables in a completely general manner. We define a set of constraints in terms of n metrics – in the form of variables over $\mathbb{R}$ – which in turn produce a feasible region in $\mathbb{R}^n$.

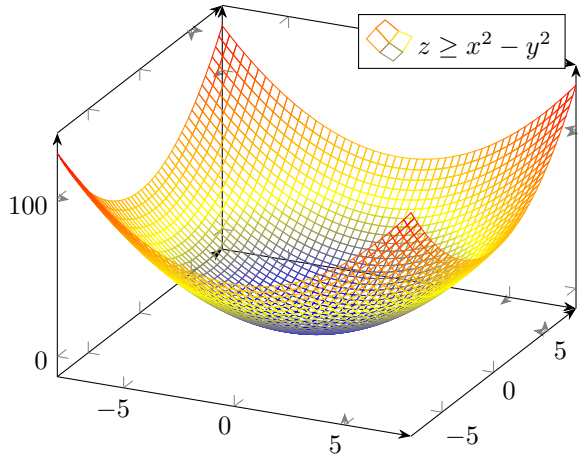

Figure 1: Example Optimisation Problem

Here the feasible represents the space of *feasible projects*. Using the metrics given by the user, we can plot the position of the user's project, determine whether this lies within the feasible region and – if not – calculate the shortest distance between the project point and the feasible region. This will allow the program to not only predict project success/failure, but also suggest amendments to make a currently unfeasible project more feasible.

Optimization Tools

Several python packages which support solving non-linear optimization problems exist, including `pyOpt`(1), `mystic`(2) and `scipy.optimize`(3). The latter appears the most widely used and well-maintained, and hence we plan to use `scipy.optimize` in our program.

1.3.1 Correctness

This way of modelling the data causes the accuracy to largely depend on our metrics relationships accuracy. These relationships are essentially regions of a graph, which is easily adaptable and able to be changed according to new data. This will allow the model to be adjusted over the production period and continually adapted after more project data reveals any flaws in the current state of the model. Over time the model will get more and more accurate until it is a good representation of the relationships between metrics and at that point it will still get more accurate over time. The initial values will be based on research and opinons of members of the development team, human intuition guiding the accuracy towards the right direction quickly.

1.4 Circulations

Not every proposed metric is best modelled through numerical constraints. The most prevalent example for our use case is *team structure*. Here we propose a model based on network flows – specifically, circulations – to evaluate the quality of a given team structure based on the degree of communication it facilitates.

We can model a given team structure as a circulation[1]. Here nodes which *supply* flow are those who primarily communicate information (e.g. developers in an SWE team), and nodes which need to *receive* or *demand* flow

[1] As described in Section 7.7 of *Algorithm Design*(4)

are those receive it (e.g. managers): feasible circulations correspond to an effective team structure.

Furthermore, the use of a graph structure leaves open the option of exploring further structural properties (e.g. degree distribution as a measure of connectedness, as in (5)).

2 Component Interaction

For this project, we use the microservices architecture pattern. As can be seen in Figure 2, both the Manager and Developer are part of the Client that can access the System through API. Blocks are the actions that client can do in the system, the arrow means after doing the action then the next step will be processed. The blocks in yellow can only be accessed by the manager, which means only the manager can assign jobs then set priorities for each job. Others can be accessed by either developer or manager.

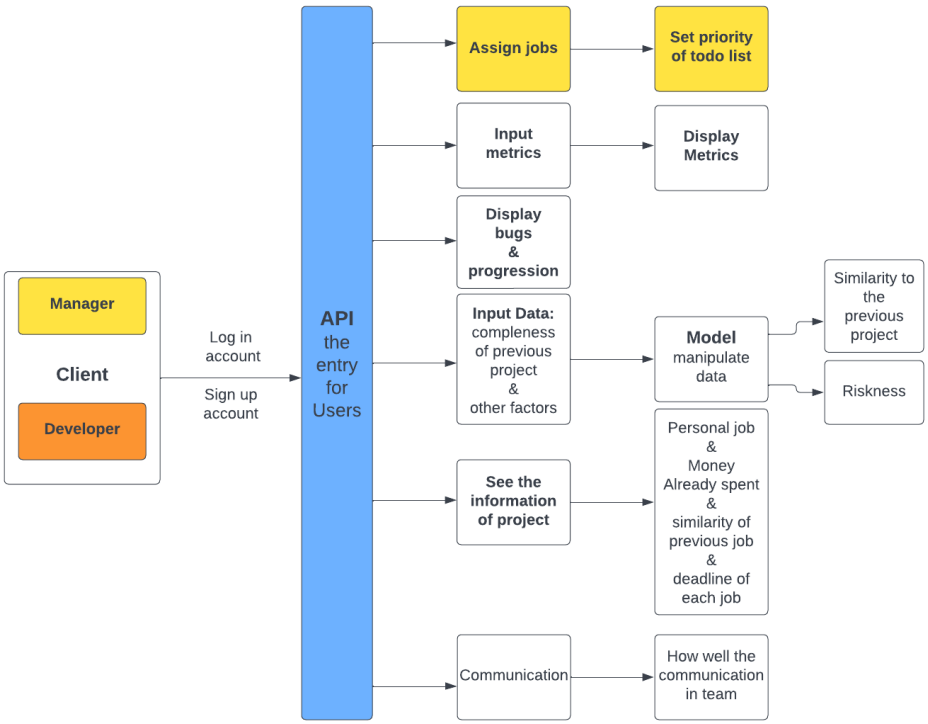

Figure 2: Overview of microservices architecture pattern

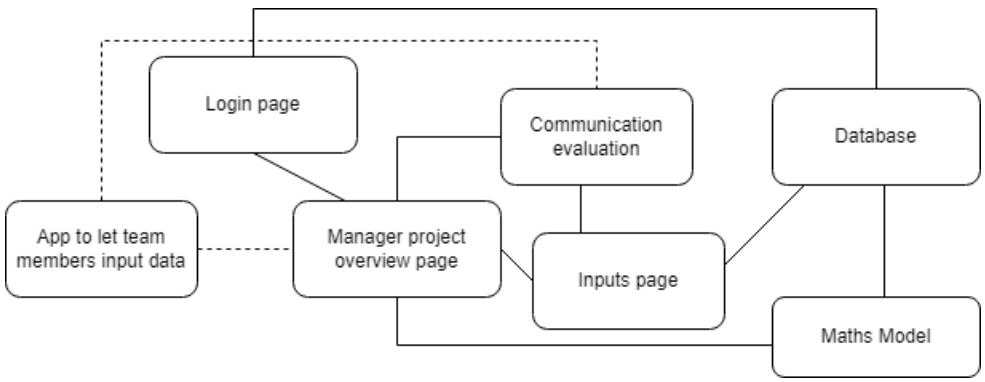

Figure 3: Context diagram of architecture pattern

We split the System into six subsystems shown (with the optional addition of a system to allow team members to

insert data relevant to themselves) in the figure above: Log-in System, overview system, Communication System, linear optimisation system, database system and the inputs system. The main system should be composed of the linear optimisation system which will contain the model of predicting success or failure of projects and the database. The overview system will monitor projects and show the information to the manager. The inputs system will allow the manager of the team to set tasks and deadlines for the project as well as input other needed data.

3 Interface Design

3.1 UI Prototype

For prototyping, we use *Figma*[2] for its collaborative features and ease-of-use.

We opt for a minimal, clean design which prioritises clarity and ease-of-use. Our design uses a two-tone color combination for background and foreground, with minor deviations to distinguish input field backgrounds and dis-activated buttons.

We apply the Getstalt principle of similarity by maintaining a consistent style for UI elements, and using icons to aid intuition[3]. The two fonts used in the design ("IBM Plex Sans" and "Inter") are both open-source and freely available at Google Fonts, which we will later make use of via the npm package `webfontloader`[4].

The following state diagram demonstrates the flow of navigation:

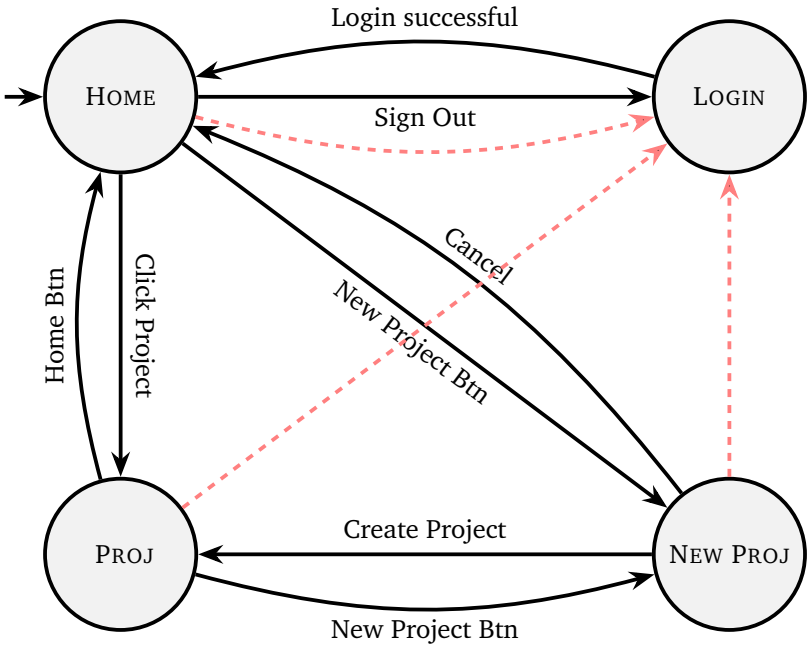

Figure 4: Navigation Flow (Dashed red arrows indicate automatic redirection where the user is unauthorised.)

[2]https://www.figma.com
[3]Specifically, we use feather icons, a tried and trusted source of icons for the web: `https://feathericons.com/`
[4]See `https://fonts.google.com/` and `https://www.npmjs.com/package/webfontloader`

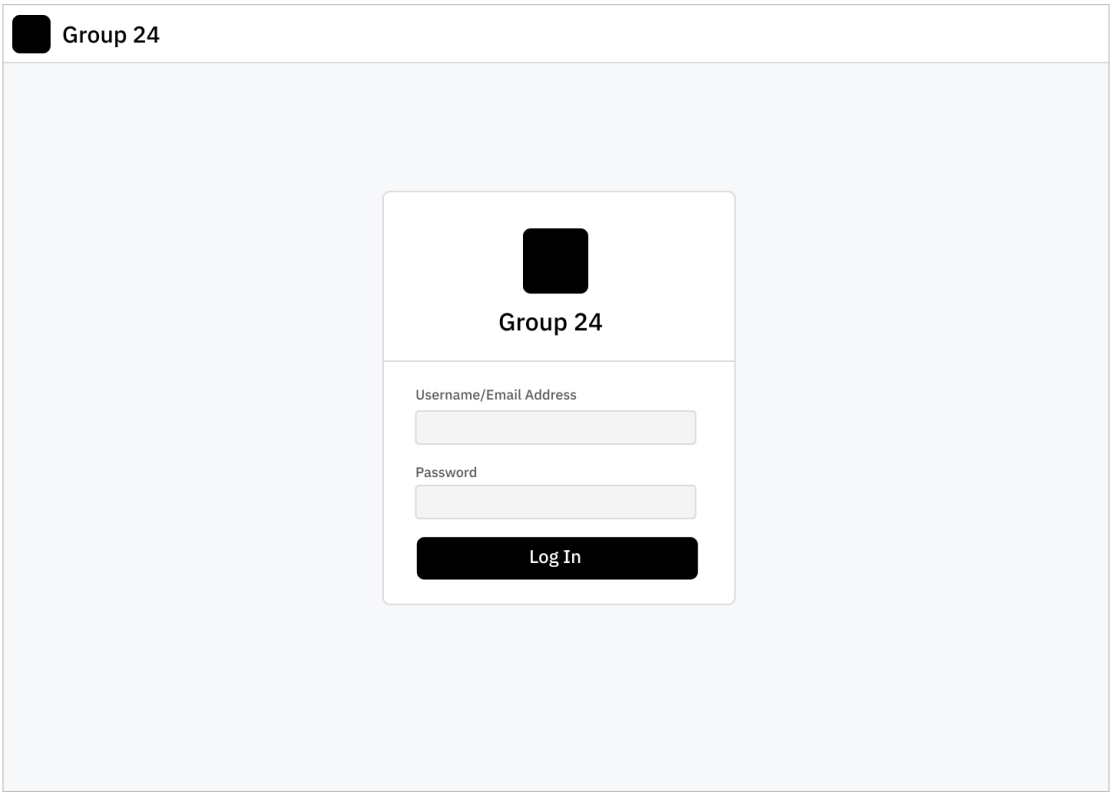

Figure 5: Login Page

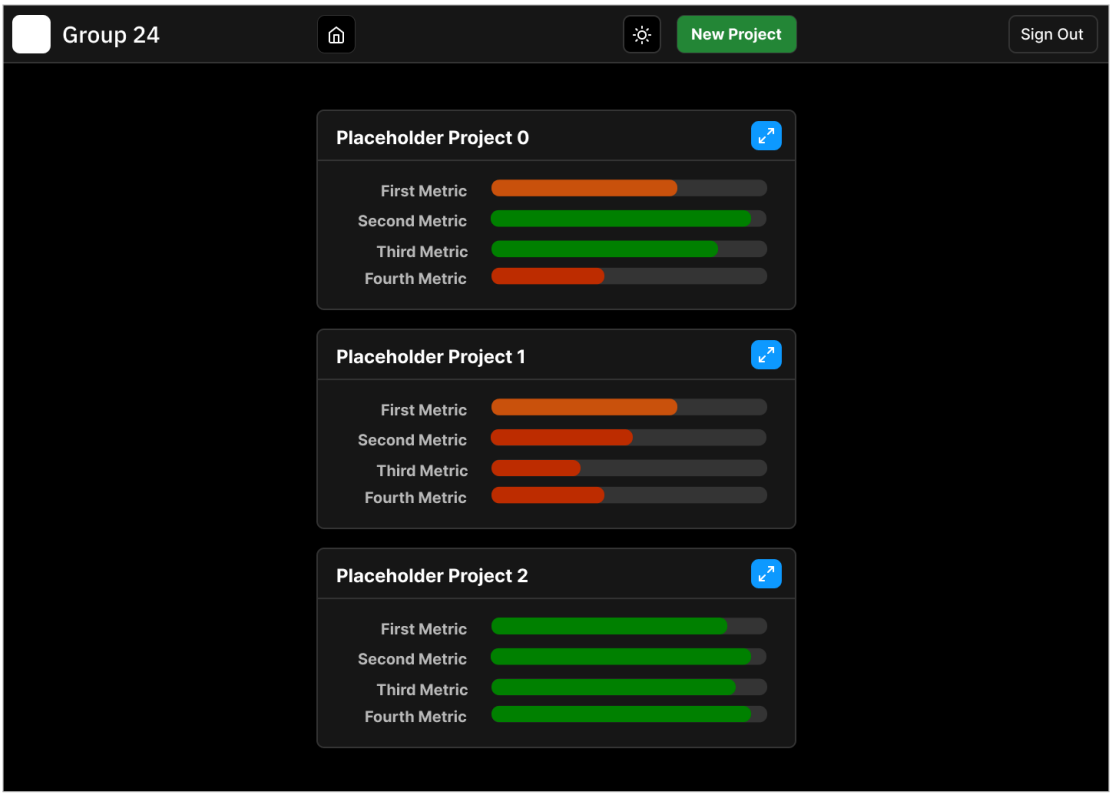

Figure 6: Home Page (Dark)

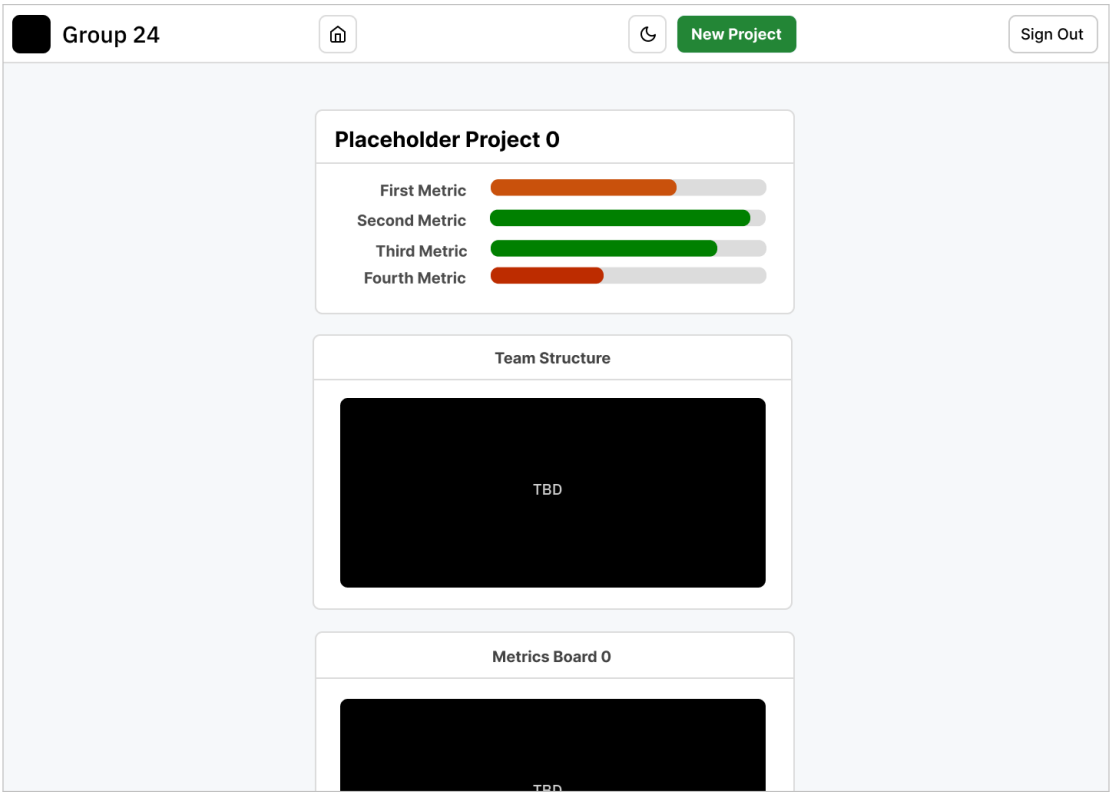

Figure 7: Project Page (Widgets labelled "TBD" will be fleshed out during our coming sprints.)

The "New Project" page and Light Home page are omitted due to its size. Our skeleton design can be viewed in full at `https://www.figma.com/community/file/1203795846694624972`

3.1.1 UI architecture

The figure below (figure 9) illustrates the functions of the system and how people interact with the system. In our system, clients can input data, list tasks and deadlines and get summative and risk related information for their projects. Users can also see how well the project doing through in the form of a risk evaluation.

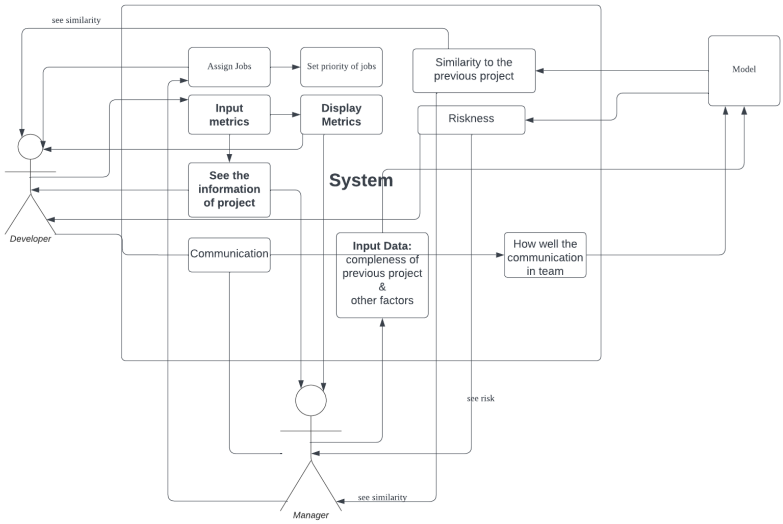

Figure 8: UML Use case diagram

4 Testing

4.1 Overall Testing plan

Throughout development we will carry out static testing in the form of reviewing each other's code at the end of each sprint as well as carrying out pair programming wherever possible. We can then act upon each other's feedback before continuing with our next sprint. Our group structure facilitates this as we have 2 backend developers who can carry out pair programming on the backend as well as a designer and frontend developer who can carry out pair programming on the frontend. Moreover, our 'largely informal structured approach' means that each team member can contribute to the codebase, therefore making it easier to organise pair programming sessions. Conducting static testing before carrying out dynamic testing will allow us to more easily discover errors hiding behind other errors, test our code before it is complete, and also consider the quality of our code with respect to improving maintainability, removing redundant code, improving code efficiency and meeting style standards (as we are using python for the backend we will adhere to the PEP 8 style guide, not including snake casing).

After carrying out static testing we will then carry out dynamic testing in order to detect any errors we may have missed. We will derive our structural tests from control-flow diagrams for our system using a path coverage scheme - this will likely make up most of our unit tests. Our functional tests will be derived from identifying all the tasks our system is expected to perform - this will likely make up most of our requirement verification tests. Additionally, frontend developers will develop the dynamic tests for the backend and backend developers will develop the dynamic tests for the front end. This will reduce unconscious bias and therefore improve the robustness of our tests. Additionally, writing tests requires a clear understanding of what the code is meant to do. Therefore, the frontend developers will develop a good knowledge of how the backend works and backend developers will develop a good knowledge of how the front end works. This will make integrating the frontend and backend much easier as everyone will have a wider knowledge base to draw from.

4.2 Requirement Tracking

For each requirement we have a test we can run to objectively verify if it has been met. If during development we would like to update our list of requirements we will also update our list of tests. Additionally, we will have weekly communication with the customer where we will ask the customer to test/provide feedback on the requirements we have implemented in a given sprint - this will ensure that we know we are meeting our customer's needs. This will work in tandem with our development approach in which each sprint is centred around implementing a particular subset of requirements. The priority of our requirements will also impact how we move forward with development. For example, we will only move on to user acceptance testing if our 'must' requirements pass their respective tests. However, if some 'could' requirements fail their tests, we will still move forward if we are tight for time.

4.2.1 Requirements Tests Sample

Requirement Code	Test Description	Test Success Definition
FM1	Ask 3 users to make an account the first 2 users should make valid accounts and then the third user should attempt to make an account with the same email as the first user. All users should then attempt to login to their accounts.	The first 2 users successfully create accounts. The third user is presented with an 'email already in use' error message. Then only the first 2 users should be able to login to their accounts.
FM9	Ask a user to input a priority ordered todo list.	A priority ordered todo list is now visible on the GUI, the database will contain each task entered and it's priority.

4.3 Unit and Integration Testing

We will design our unit tests with the rule that the result of each test must either confirm a behaviour or reveal a problem. We will use Typescript based tests for each React component we build as part of the frontend. We

will use python based unit tests to test each function built in our backend in order to test for logic errors. Each unit will be tested using valid, invalid and boundary data in order to ensure that our system can correctly handle all possible inputs. Additionally, for each object we will set and get all attributes, run all events that can cause a state to change as well as test each object in many possible states. We will also take into account all possible object permutations that can happen as a result of inheritance.

As we pass all unit tests for specific 'groups' of functions, we will integrate them into bigger sub-components , which can then be tested, before being further integrated into the full system. For example, once we pass all unit tests for the functions that will make up our non-linear optimisation model, we will bring those functions together and then test the model as a whole. This will allow us to identify errors that can occur when different units are interacting. For example, the data passed from 'unit1' into 'unit2' may not be of the datatype unit2 is expecting.

4.3.1 Unit Tests Sample

Unit being Tested	Test Type	Input Data	Expected Outcome
Budget Input Box	Invalid Data	-1	Error message stating 'a negative budget cannot be input'
Deadline Input Box	Invalid Data	01/01/1900	Error message stating 'the deadline cannot be before the current date'.

4.4 User Acceptance Testing

The final stage of our testing process will be user acceptance testing. The minimum prerequisites to carrying out user acceptance testing will be: all unit tests passed, all integration testing passed.

Before our final user acceptance tests . We take a Beta testing approach to user testing and ask people who are not members of our team to use our system alongside a form asking them to perform specific tasks and asking users to rate different aspects of our system such as how intuitive it is to use, how difficult each task was to complete on a scale from 1 to 5 etc. Beta testing involves a large number of users and this will prevent any bias that could arise from always asking the same person to test our system.

Our final user acceptance tests will take the form of asking our customer to carry out some tasks which will correlate to checking if all the must and should requirements are met.

4.4.1 Beta Testing Form Sample

Task	Difficulty 1 = easy and 5 = impossible	Time taken to complete task
Create an account	1	1 minutes 39 seconds
Input project metrics	3	2 minutes 01 seconds

5 Processes and Evaluation

5.1 Business Analysis

With the industry of software development being valued in the hundreds of billions and constantly increasing, the management of these project becomes a prevalent issue. With the CHAOS survey of 2020 reporting that only a third of projects are completed successfully, software dedicated to preventing failed projects immediately becomes valuable to any competitor within this market. The consequences of these failures are clear, decreased productivity, investment, profits, reputation, employee morale and an increase in costs.

We intend for the software to allow for the proper planning, budgeting, effective risk management, adequate resource allocation and clear communication to take place within project teams to increase the rate that software

development projects are successful and to mitigate the costs of unsuccessful projects as well as the undertaking of such projects in the first place.

5.2 Group Structure

We have decided to take a fairly standard approach to roles with the slight variation of integrating testing into the system architect and the designer for the front end and back end respectively. This is because with such a small team testing can't practically be fully separated from the members working on the product.

The roles were outlined in the requirement analysis document. The given roles should allow us to prevent many common risks; being a small team lets us adapt quickly to changes, having a project manager allows us to keep a level of accountability in the group and having everyone take part to a degree in the coding of the product, means all members are kept in the loop on changes as soon as they happen removing islands of knowledge.

5.3 Methodology

The timeline outlines the tasks to be completed during the implementation of the system with each weekly sprint. The tasks are designated to the allocated team members as described before with major design considerations collaboratively discussed among the team. The final report is also produced throughout the entire development period, with each respective team member working on the section of the document relevant to them and the project manager overviewing the entire document. As a scrum methodology has been adopted, the sprint tasks will likely change to keep in line with the project deadline, with the final report being continuously added to as to not leave the final report to the last second.

5.4 Planning Tools & Timeline

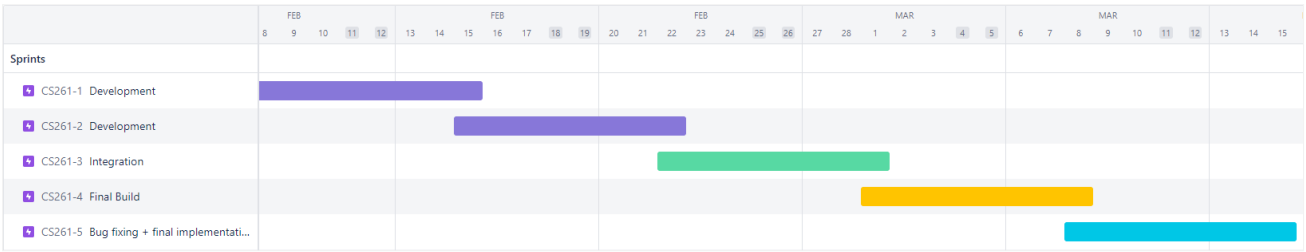

Figure 9: Planned sprints

Sprint	Estimated Time	Tasks
1	1 week	Build Tests and begin API and UI implementation
2	1 week	API and UI implementation
3	1 week	Begin Integration
4	1 week	Complete integration, Final testing and
5	1 week	Bug fixing, additional feature implementation, final report, script planning and final delivery of product

References

[1] “Pyopt documentation.” [Online]. Available: http://www.pyopt.org/index.html

[2] “Mystic package documentation.” [Online]. Available: https://mystic.readthedocs.io/en/latest/

[3] “Optimization (scipy.optimize).” [Online]. Available: https://docs.scipy.org/doc/scipy/tutorial/optimize.html#defining-nonlinear-constraints

[4] J. Kleinberg and Tardos, *Algorithm design*. Harlow, Essex: Pearson, 2014.

[5] C. Christensen and R. Albert, “Using graph concepts to understand the organization of complex systems,” *International Journal of Bifurcation and Chaos*, vol. 17, no. 07, pp. 2201–2214, 2007.