PROJECT DOCUMENT - GROUP C2

Team member name	**Detailed contributions per chapter**	**Percentage contributions overall**	**Signature**
2242777	Chapter 1: **1.1.1, 1.1.2**, **1.1.3, 1.1.4, 1.1.5, 1.2**	20%	A.Moore
2109066	Chapter 1: **1.2** Chapter 2 - ALL OF CHAPTER 2 Chapter 3 - ALL OF CHAPTER 3 Chapter 5 - ALL OF CHAPTER 5	45%	B.Sediqi
2204118	Chapter 4	7.5%	Dec
Enoch	Withdrawn (ECF)	0%	
Femi	Not Contributed At The Current Moment	0%	
2212102	Chapter 1: Problem specification **1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, 1.2**	12.5%	MBitalac
2199361	Chapter **1.3 (1.3.1, 1.3.2 and 1.3.3)** Chapter **4**	15%	*S. Baxter*

Chapter 1 - Problem Specification

1.1 Processes Used for Gathering Requirements

1.1.1 Use of a Combination of Adequate Requirements Gathering Methods

We decided that the best way to gather user requirements was to interview students themselves, those who were planning to take their optional placement year (and had begun applying) as well as potentially 3rd year students who'd already undergone their placement year to gain valuable insights from both perspectives. Our main goal was to find out how the application progress could've been made smoother & more manageable. We agreed on using two different methods to gather requirements for our users, multiple 1-1 interviews and a session with a medium-sized focus group. The advantages of 1-1 interviews were that we could really dig deep into the individuals needs and ask impromptu follow up questions based on the answers they provided. Being able to conversate with a single interviewee allows you to explore nuances you may have previously overlooked. In contrast to this, we believed it was necessary to complement these with a focus group session. The collaborative conversation allows others to build upon other's ideas and let a free-flowing discussion highlight different outlooks, confirm priorities & establish a consensus on collective necessities. These methods together allowed us to get a diverse amount of viewpoints to build a strong foundation for our app. Together we decided upon 10 questions to ask about aspects of our design then got to work finding suitable participants.

1.1.2 Clear and Complete Description of the Design of the Methods and Motivation for Using Them

Our group members first turned to other undergraduates from our course, asking the friends & acquaintances made throughout our time studying so far. As we are second-year students ourselves, this was the most logical course of action. They were happy to consent to our research. Subsequently, our members who lived in house-shares with other students were able to use their flatmates to be interviewees at a convenient period in their downtime. With these two sources of candidates, we obtained an adequate amount of data to start working on our user requirements by studying the trends in answers. Due to the specific nature of the application, we decided against standing outside a building on campus (such as the university library) and approaching students to answer our questions due to the ineffective use of our time and lack of incentive for someone to dedicate their time to us.

Considerations were made to conduct interviews ethically. All subjects were aged 18+, therefore able to provide their own informed consent. Every interview started with reading aloud a consent form in which was then signed by the interviewee, which they could re-read if they wished. Once complete, we began with the questionnaire. We made sure to keep the atmosphere light-hearted yet with professionalism. Participants were treated with respect, given an option to remain anonymous if they choose to and all goals of our research were made clear.

1.1.3 Brief Description of the Data Collected and Evidence of the Data - Interview Transcripts

The data collected from the transcripts confirmed that our app idea would be beneficial to potential users. Those we spoke to expressed interest in the idea of having an all-in-one way to manage their placements. Features such as a dashboard, page to handle your documents plus other helpful tools such as notifications and reminders proved to be popular.

1.1.4 In-Depth Discussion of the Patterns Identified on the Data Across Methods and the Process Used to Identify These Patterns

The next step was to identify patterns in answers received. We focused on finding commonalities between the participants since these were proven to be the most valuable needs for our users. Responses were examined one question at a time and summarised down to the main themes, like minimalism or specific terms like "to-do list". By doing this we were able to review answers quickly and concisely, getting a good grasp of the most desirable requirements when examining the frequency of the responses. To further narrow down the user's needs for our app, we made sure to take into account the context of the question, for example, a "to-do list" could also mean that other task management tools in general should be a focal point of our app design. This organised approach to our pattern identification gave us a good understanding of the priorities and also proved to us that some of our initial ideas about potential features were not actually as necessary as previously believed.

1.1.5 In-Depth Discussion of the Challenges Faced During the Process

One challenge we faced was how to deal with conflicting responses by our interviewees. To use an example, one participant said they would primarily use our application on their desktop computer while adding the fact they wouldn't care for phone/tablet integration, while another said they would use their phone daily to quickly check up on any updates made by employers.

1.2 User Requirements

1. Application should track important placement details: location, salary, role. Deadline, progress status, job type and job description. (*Source: Transcript 1, Q1; Transcript 2, Q2*)
2. Applications should have CVs and cover letters linked to them (*Source: Transcript 1, Q2*)
3. Placement information should be categorised by nearest deadlines and allow sorting by type, company or application status (*Source: Transcript 3, Q1*)
4. Users need to update statuses, set reminders, mark priorities, track deadlines and attach documents like CVs and cover letters. (*Source: Transcript 1, Q2; Transcript 3, Q2*)
5. Visual representation through colour coding and icons should be implemented: Green for accepted and red for rejected for example (*Source: Transcript 1, Q5; Transcript 2, Q5; Transcript 3, Q5*)
6. Placement details should display application dates and time since last updated (*Source: Transcript 3, Q5*)
7. A To-Do list should be implemented, keeping track of applications you wish to apply for (*Source: Transcript 1, Q7; Transcript 2, Q7; Transcript 3, Q7*)
8. The app should be easily accessible, with users interacting with the app from multiple times daily to 4 times a week on desktops, mobile devices and tablets (*Source: Transcript 1, Q3; Transcript 2, Q3, Q4; Transcript 3, Q3*)
9. Users need to be notified of upcoming deadlines through push notifications with the notifications being received through email and in-app (*Source: Transcript 1, Q6; Transcript 2, Q6; Transcript 3, Q6*)
10. Users want notifications to be modifiable and personalised to when they want to be notified (*Source: Transcript 1, Q6; Transcript 2, Q6*)
11. Minimalism should be prioritised over a feature-rich design, so it is easier for the user to navigate through the application (*Source: Transcript 1, Q8; Transcript 2, Q8; Transcript 3, Q8*)
12. Applications should be given priorities to identify placements with high value to the user (*Source: Transcript 1, Q5*)
13. Users should be able to apply filters to view certain applications, such as filtering by role to see all applications for a particular role (*Source: Transcript 1, Q5; Transcript 2, Q5; Transcript 3, Q9*)
14. Sorting should be allowed for user's to view their data by date applied, due soonest and priority (*Source: Transcript 1, Q5; Transcript 2, Q1*)
15. Application addition is repetitive and should be simplified for the user to quickly add applications (*Source: Transcript 2, Q4*)
16. A dashboard with a couple of deadlines for a simplified version of the whole list (*Source: Transcript 1, Q8*)
17. Users are mixed on keeping the roles applied for together or in separate profiles; Option for both can be implemented (*Source: Transcript 1, Q9; Transcript 2, Q9; Transcript 3, Q9*)

1.3 System Requirements

1.3.1 User Requirements Sorted into Functional and Non-Functional Requirements

Priority of functional requirements:

1. Dedicated interface for both PC & mobile devices
2. Push notifications with updates
3. Icons, instead of it being purely text based
4. Sort-by features (job type, A-Z, recently added, etc)
5. Ability to stay logged-in across sessions
6. Two-factor authentication
7. Relevant events for jobs
8. Language Support
9. Night mode

Priority of Non-Functional Requirements:

1. Security
2. Speed
3. Integrity

4. Usability
5. Reliability

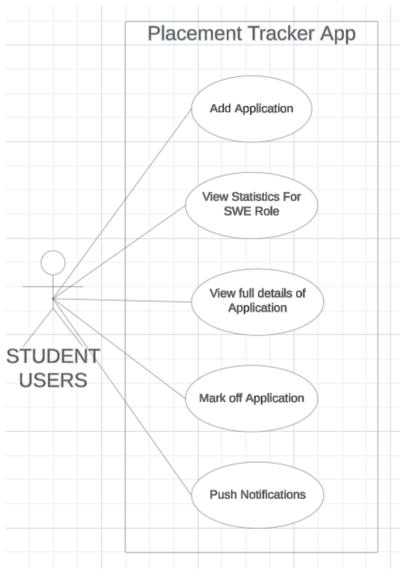
1.3.2 From Functional User Requirements to System Requirements
Placement Details Tracking (*Translated From User Requirement 1,2,4*)
Upon addition to the application, the system will allow inputs for: Location, Salary, Roles, Deadlines, Progress Status, Job Type, Job Description, priority to user as well as attaching CV and Cover Letters.
Validation: User Requirement 1 identifies the details wanted for each application. User Requirement 2 mentions the need for CVs and cover letters to be linked. User Requirement 4 states marking priority for applications.
Sorting and Filtering Applications (*Translated From User Requirement 3,13,14*)
- Displaying the list of applications will be modifiable by a sorting and filtering feature on the list. The option of sorting includes: sorting by soonest deadlines, job type, company or application status. The option of filtering includes: specific roles, statuses or priorities
Validation: User Requirement 3 specifies categorisation by nearest deadlines and sorting by type, company or status. User Requirement 13 identifies the need to filter by role or statuses. User requirement 14 highlights sorting applications by data, due soonest and priority.
Updating and Reminders (*Translated From User Requirement 4*)
- For the list of applications, for each application the system provides the option to update the status of the application from applied to first, second or third stage to accepted or rejected. The applications and their status will include optional deadlines from which the system will provide reminders through notifications.
Validation: User Requirement 4 identifies the need to modify the applications through status and setting reminders.
To-Do List (*Translated From User Requirement 7*)
- The system shall include a To-Do list page to track applications users wish to apply for.
Validation: User Requirement 7 specifies a To-Do List feature to keep track of applications users want to apply.
Dashboard (*Translated From User Requirement 16*)
- A home tab which hosts a dashboard will be implemented within the system. This dashboard will include a simplified overview of high priority applications as well as high priority To-Do applications.
Validation: User Requirement 16 mentions having a dashboard for a concise view of applications
Notifications (*Translated From User Requirement 9,10*)
- Notifications that come from upcoming deadlines will be delivered within the system itself as well as externally through email alerts. These notifications will be customisable to the user preferences through timing and frequency.
Validation: User Requirement 9 identifies the need to notify users of deadlines via push notifications through email and in app alerts. User Requirement 10 highlights personalising and modifying notifications based on preferences
Application Profiles (*Translated From User Requirement 17*)
- Feature to give users the option to either create profiles for a user's role such as "Software Engineer John Doe" and "Data Analyst John Doe" or group roles into a single general profile will be introduced into the system.
Validation: User Requirement 17 specifies the option of grouping roles into one profile or separating them through profiles.
Log-in Process (*Translated From User Requirement 8*)
- When logging in the system will host a remember me feature to ensure ease of access due to the repetitive action of user's logging in - "From multiple times daily to 4 times a week".
Validation: User Requirement 8 emphasises accessibility and frequent usage.
Application Addition (*Translated From User Requirement 15*)
- For low importance inputs such as job description, pre-defined information can be inputted in order to tackle the repetitiveness of adding applications.
Validation: User Requirement 15 highlights the need to simplify repetitive application addition process.

1.3.3 Non-functional System Requirements
Visual Representation (*Translated From User Requirement 5, 11*)
- The system needs to be minimalistic with information helped portrayed through colour coding - Red for rejected, green for accepted for example and icons to represent the stage of your application e.g. Phone icon for phone interview.
Validation: Users repeatedly highlighted the importance of minimalism over a clattered, feature-rich design stating

minimalistic design with icons to complement it would work best
Multi-Platform Access (*Translated From User Requirement 8*)
- The system should be available to view on desktop, tablet and mobile screens
Validation: Users stated they will be accessing the website not only multiple times a day but on different platforms too, the system should ensure this is doable.
Security
- Dealing with sensitive data, the system must implement data security on all levels from database to client through features such as hashing and role based data management.
Validation: The system is dealing with sensitive data such as names and locations. Privacy laws such as GDPR are put in place to protect users' data. Implementing features such as hashing ensures protection of data.
Speed
- Users will incorporate the system into their daily lives, for this reason the system must be fast and efficient in its activities in order to keep users satisfied.
Validation: Slow response times can lead to frustration and reduced productivity within the application process, making speed a key requirement for user satisfaction.
Reliability
- The system must be robust, handling system errors without crashing the whole site so users can continue to use the system.
Validation: Users rely on the system to track their applications and deadlines. It is of high importance that the system avoids crashing and loss of data through errors for user satisfaction to stay positive.

Chapter 2 - Design

The purpose of this section is to identify the main scenarios of our application and model the logic behind how the app would work.
2.1.2 Use Case Diagram LINK —-->
The actors would be students who want to track the placements they have applied for.
These users can:
1. Add an application
2. View Statistics for SWE Role
3. View full details of an application
4. Mark off an application (Accepted / Rejected)
5. Push Notifications



2.1.3 Description of Use Cases
Adding an application
User applied for placement and wants to track placement. Necessary information is needed.
Pre-Condition: Users must be registered and logged in but on the logic side.
Data Input: Placement Details: Company Name, Role, Salary, Location, Start Date, Website Link, Contact Information and current stage.
Data Output: Notification confirming application addition. Page refreshed showing application on the client interface.
Interaction Steps: 0: Entry point - The user clicks the "Add Application" button on the client interface.
1: The user enters all relevant information into the form and clicks the "Confirm" button.
2: Application Manager runs validateInputs(details) function to ensure the input data meets the required format.
3: If the data is valid, the system runs addApplication(details) from the Application Manager.
4: Application Manager sends POST request using the insertApplication(details) adding application to database.
5: Application Manager calls GET function retrieving all applications for the user using fetchAllApplications.
6: Application Manager runs displayAllApplications() function, client interface refreshes displaying updated list.
7: The system displays a notification pop-up confirming: "Application successfully added."
Error Case

1: If required fields are blank, the system highlights them and prompts the user to complete them.
2: If validateInputs(details) detects invalid formats, a notification displays: "Please Enter Valid Information."
4: If the POST request fails, a notification displays: "Failed to add the application. Please try again later."
5: If the GET request fails, a notification displays: "Application saved but not visible."

View Statistics for Software Engineering Role

User views statistics for SWE placement applications and statuses, displayed as a graph generated with MatPlotLib.

*Pre-Condition:*User must be registered,logged,having existing applications so data can be fetched in to make the graph.

*Data Input:*Role, used to get applications. Then generates graph which displays status of all applications for the role, all applications statuses would be an input too.

Data Output: A graph which visualises the number of placements as well as the statuses of all these placements.

Interaction Steps: Step 0: Entry point - The user clicks on the "Statistics" tab in the client interface.
Step 1: The user selects the option to view the statistics for a specific role (e.g., "Software Engineering") and confirms.
Step 2: The client interface calls generateGraph(role, filter) function from the statistics manager. Filter is initially set to None.
Step 3: Statistics manager calls getData(role, filter) to retrieve relevant data. Uses GET method to fetch application data.
Step 4: Statistics manager receives data and calls the makeGraph(data) function to generate the graph using MatPlotLib.
Step 5: The statistics manager sends the generated graph to the client interface using the showGraph(graph) function.
Step 6: The client interface displays the graph showing the statuses of applications for the selected role.

Error Case
Step 1**:** If no role is selected, the system will prompt the user to select a valid role.
Step 3**:** If no applications exist for selected role, system shows an error message: "No applications for the selected role."
Step 3**:** If fetch call fails, system will display an error message: "Unable to retrieve data for the selected role. Try again later."

View Full Details of an Application

User wants to see all details of a placement that they have previously added. While the dashboard shows brief details, on the application tab an application can be clicked to display all the details for said application.

Pre-Condition: Users must be registered, logged in and must have an existing application that they will then select..

*Data Input:*ApplicationId, when the user clicks on the application, ID is then retrieved in order to be used for data processing.

Data Output: The output produced would be a dialog box-like screen that shows all the details of the selected placement.

Interaction Steps: Step 0**:** Entry point - The user is on the "Applications" tab.
Step 1: The client interface runs the displayAllApplications() function.
Step 2: The Application Manager calls the fetchAllApplications() function using a GET request to retrieve all applications.
Step 3: The client interface displays the list of all applications.
Step 4: The user clicks on an application, triggering the showDetails(applicationId) function in the client interface.
Step 5: The Application Manager calls the fetchApplication(applicationId) using a GET request to retrieve the specific application's details from the database.
Step 6: The Application Manager passes the details to the client interface using showApplication(applicationId, details).
Step 7: The client interface displays the application details in a dialog box.

Error Case
Step 1**:** If there are no applications in the system, the client interface displays a message: "No applications available."

Step 4: If the application ID is invalid or not found, the system shows an error message: "Application details not found."
Step 5: If the fetch request fails or does not respond, the system displays a message: "Unable to retrieve application details. Please try again later."

Mark Off Application

User is rejected from placement. Upon rejection, feedback for improvement is provided which is stored in the application.
Pre-Condition: User must be registered, logged in and have an existing application that they will then update as rejected.
Data Input: ApplicationId, where when the user clicks on the application, its ID is then retrieved to be used for the website to update the application as well as feedback text provided by the company which can be copy and pasted into the Input.
Data Output: Notification - “We’re sorry to hear you got rejected” and the status of the application is updated as rejected as well as the client interface being refreshed so that the application is at the bottom of the list of applications.
Interaction Steps: Step 0: Entry point - The user navigates to the "Applications" tab.
Step 1: The user selects an application to view its full details and clicks the "Mark As Rejected" button.
Step 2: The client interface calls the rejectApplication(applicationId, feedback) function.
Step 3: Triggers Application Manager’s changeApplicationToRejected(applicationId, feedback) which updates database by: changing application’s status to "Rejected.", adding the feedback text to a new "Feedback" column in the application record.
Step 4: After successful update, Application Manager calls the displayAllApplications() function to refresh the client interface.
Step 5: Updated application status is displayed at bottom of the list, and notification appears: "Sorry to hear you got rejected”
Error Case
Step 1: If the user has selected an application already rejected, there will be no option to mark as rejected.
Step 2: If feedback is left blank, the system will prompt the user: "Is there any feedback?" to which the user can say yes and input the feedback or say no and the feedback parameter will be left as null. The backend would need to handle this.
Step 3: If the API call fails, the system will notify the user: “Failed to update the application. Please try again later.”
Step 5: If the client interface fails to refresh, the system will notify the user: "The application status has been updated, but the interface could not refresh. Please reload the page."

Push Notifications

User wants to be notified of imminent deadlines. Will need to interact with the client interface to turn on push notifications.
*Pre-Condition:*User must be logged in and have an existing application that has deadlines due to receive notifications for.
Data Input: User’s decision to turn push notifications on or off, set though a toggle on or off button through the client interface. Also a day amount, declaring how many days before your deadline you want to be emailed and your email too.
*Data Output:*Confirmation of notification status and email reminders.
Interaction Steps: Step 0: Entry Point: The user navigates to profile manager section of the client interface – Settings
Step 1: The user locaties the “Push Notifications” option and toggles it on
Step 2: When toggled on, the client interface triggers the pushNotifications(userId) function.
Step 3: The profile manager then uses it’s API function to update the the user’s notification preference
Step 4: A confirmation message is displayed saying notifications turned on
Step 5: When a deadline is upcoming in 3 days, an email notification is sent to the user’s email
Error Case Step 1: If toggle button unresponsive, system displays: "Unable to update notification preferences. Try again."
Step 4: If the API call fails while updating preferences the system displays: "Failed to save notification settings. Try

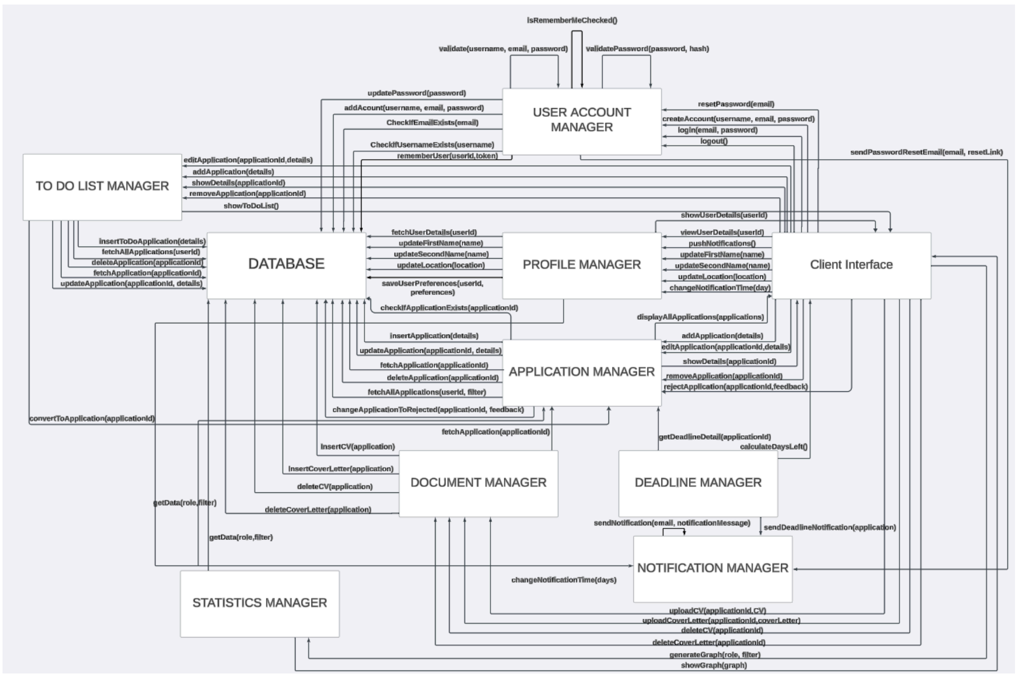
again."
Step 5: If notifications cannot be delivered: System notifies the user: "Email notifications could not be sent. Try again."

Challenges Faced in the Design Process
There was difficulty in identifying use cases that were sophisticated enough to demonstrate the flow of the application. Additionally, some use cases were extended off of already existing use cases. For example, one use case we had chosen originally was adding a document to the application after the application had already been added. However, we realised that this scenario would have to be explicitly mentioned in the Use Case Diagram through an overly long Use Case Name which may not be appropriate or visually clear in the diagram. As a result, we settled for an alternative use case that was just as sophisticated but separate in its functionality so it is easily represented and comprehended within the use case diagram.

2.2 Architectural Design

The purpose of system architecture is to lay the structural groundworks that translates system requirements into a functional design. This is done by an architectural model made on LucidChart.

2.2.2 Architectural Model


isRememberMeChecked()
validate(username, email, password)
validatePassword(password, hash)
USER ACCOUNT MANAGER
updatePassword(password)
addAcount(username, email, password)
CheckIfEmailExists(email)
CheckIfUsernameExists(username)
rememberUser(userId,token)
resetPassword(email)
createAccount(username, email, password)
login(email, password)
logout()
sendPasswordResetEmail(email, resetLink)
TO DO LIST MANAGER
editApplication(applicationId,details)
addApplication(details)
showDetails(applicationId)
removeApplication(applicationId)
showToDoList()
insertToDoApplication(details)
fetchAllApplications(userId)
deleteApplication(applicationId)
fetchApplication(applicationId)
updateApplication(applicationId, details)
DATABASE
fetchUserDetails(userId)
updateFirstName(name)
updateSecondName(name)
updateLocation(location)
saveUserPreferences(userId, preferences)
checkIfApplicationExists(applicationId)
PROFILE MANAGER
showUserDetails(userId)
viewUserDetails(userId)
pushNotifications()
updateFirstName(name)
updateSecondName(name)
updateLocation(location)
changeNotificationTime(day)
Client Interface
displayAllApplications(applications)
insertApplication(details)
updateApplication(applicationId, details)
fetchApplication(applicationId)
deleteApplication(applicationId)
fetchAllApplications(userId, filter)
changeApplicationToRejected(applicationId, feedback)
APPLICATION MANAGER
addApplication(details)
editApplication(applicationId,details)
showDetails(applicationId)
removeApplication(applicationId)
rejectApplication(applicationId,feedback)
convertToApplication(applicationId)
fetchApplication(applicationId)
getDeadlineDetail(applicationId)
calculateDaysLeft()
InsertCV(application)
InsertCoverLetter(application)
deleteCV(application)
deleteCoverLetter(application)
DOCUMENT MANAGER
DEADLINE MANAGER
getData(role,filter)
getData(role,filter)
sendNotification(email, notificationMessage)
sendDeadlineNotification(application)
NOTIFICATION MANAGER
changeNotificationTime(days)
STATISTICS MANAGER
uploadCV(applicationId,CV)
uploadCoverLetter(applicationId,coverLetter)
deleteCV(applicationId)
deleteCoverLetter(applicationId)
generateGraph(role, filter)
showGraph(graph)

[Link to architectural model]

Client Interface: What the user sees, used to interact with the backend. Made up of several pages such as login page. Interacts with all the components except for the database, being the user's gateway to data manipulation.
Database: Where user's data will be stored. Application's managers will perform functions through their API's for data manipulation and as a result are linked with all components except for the client interface.
User Account Manager: Manages user login, registration and account functions like rememberUser(userId) (taken in consideration from Interview on 25th) and addAccount(username,email,password), interacting with the database through the integrated API. Meets system requirements through functions such as rememberUser(userId).
Profile Manager: Manages user profile data (e.g., fetchUserDetails(userId), updateFirstName(name)) and handles user preferences, including notifications, interacting with the Notification Manager. Uses HTTP methods to call on the database and manipulate data to do with the actual user.
Application Manager: Handles application data (add, update, delete) through functions like displayAllApplications(userId) and fetchAllApplications(userId, Filter). It interacts with the database using HTTP methods and other components such as the document and deadline managers, to display and update application details, one of the main system requirements.
Notification Manager: Dedicated component handling sending out notifications. Through interviews we understood users wanted to be notified and by email, this component ensures that happens. Interacts with profile manager, using the user's preference for when to be notified as an input in the changeNotificationTime(days) as well as taking input from deadline manager using sendDeadlineNotification(application) to identify which application to send notifications for. The notification

manager then uses sendNotification(email, notificationMessage) to send the notification to the user's email with the message being constructed within the notification manager, using the input's it received.
Deadline Manger: Component dedicated to application's deadlines using the application manager and sending details to notification manager in order to send notification for deadlines.
Document Manager: In an instance where a document isn't uploaded upon initial addition of the application, the document manager handles this scenario. Mainly it uses a function to trigger Application Manager's fetchApplication(applicationId) in order to then add and delete CV's and Cover Letters from the document. This would be done as a result of client interface interaction.
To-Do List Manager: Manages the To-List applications, handles actions such as adding,editing,deleting through HTTP requests. If an application is applied for, it's moved to the applied list using the convertToApplication(applicationId) function meaning there is an interaction with the application manager necessary too. Handles the To-Do list system requirement.
Statistics Manager: For our application we wanted a statistics section to help visualise the application process. This manager handles that. It interacts with the database directly and with the application manager with the getData(role,filter) functions in order to fetch the necessary data to generate statistical graphs through MatPlotLib. Once generated, the Statistics Manager then interacts with the Client Interface to display the graph.

2.2.3 Trade-offs and Rationale

Performance vs Scalability: We prioritised a scalable architecture, handling future user growth and a high amount of applications, at a cost of performance however. We did this by splitting certain processes to their own managers. This was achieved by rather than having all application related behaviour handled by the Application Manager, it is split into To Do List Manager, Application Manager and Document Manager for the whole process of a user interacting with an application. This separation ensures modular scalability but introduces overhead from inter-manager communication which impacts performance.
Maintainability vs. Complexity: We opted for a modular design, where each part of the system operates as an independent block. This means that the code is much more maintainable and understandable while introducing some complexity due to the need for coordination between these modular components.
Security vs Usability: Based on User feedback, Users prioritise usability as well as security. In order to handle both, the rememberMe(userId) logic was added to the User Account Manager. This maintains security through the requirement of a login, but also allows ease of access through the remember me function. Components were also introduced to handle all situations within the app such that the app is as usable as possible.

2.2.4 Architectural Pattern

Our component-based architectural pattern ensures for a modular and reusable system that has separate blocks of codes for separate functionality as well as object oriented code. By dividing the system into these separate components, we can develop, test and maintain each module separately, promoting scalability and flexibility. This aligns with the functional requirements we have such as managing different application processes being broken down into their respective components: To-Do List applications handled by To-Do List Manager, Documents handled by Document Manager and Applications handled by Application Manager. This architectural pattern also aligns with the non-functional requirements such as maintainability and scalability due to the functionality of our application being in separate classes. If we were to add new features, new components could be added with minimal disruption to the existing structure, ensuring seamless scalability. In addition to this, changes to one part of the system can be made without affecting others, promoting maintainability. Some of the difficulties we encountered were actually identifying the components. For example, we added user email as a component as it was a part that the system would interact with. However we soon realised that it was more so an object oriented approach we were going for so rather than implementing a user email, its interaction would be visualised within the User Account Management component itself. A similar process was taken when we added an API component. While we believed that the components would be used to interact with the API, we realised this would introduce overhead through inter-manager communication as well as unnecessary complexity and it would be much simpler to implement the API's functionality within the existing Manager Components themselves. The component-based approach was chosen over a Layered Architecture because it avoids the hierarchical constraints of layers. Our application requires frequent and direct interactions between components. This is handled naturally with the component based approach. In contrast, a layered approach requires components to communicate through an intermediate layer, introducing unnecessary complexity and overhead in implementation. In a layered architecture, there is a predefined flow, where layers communicate with the layer directly above or below in a sequential manner. This rigid structure doesn't suit our application as the flow of the app is dynamic. For example, components such as the Application, To-Do and Document Managers all work with the same data to perform separate functions, this can be effectively done using a component based pattern where each component interacts directly with each other instead of the layered approach where data needs to go through unnecessary layers.

Chapter 3 - Implementation

Version Control: GitHub https://github.com/GROUP-2C-UOP/.github.git

Languages: HTML, CSS, JavaScript and Python

It was universally agreed upon that Python should be the primary language, however after some discussion, we decided to use the traditional front-end languages while dedicating Python to the backend. Although Python libraries such as PyQt could be used to make the front-end, we felt that learning JavaScript and using HTML and CSS would provide more flexibility and result in a more visually appealing and polished application compared to using PyQt and TKinter.

Front-End: React

We chose React due to its strong community support, being the most popular framework for the past 5 years. It also has a component based structure which promotes reusability enabling us to scale up our project. Lastly, React uses a virtual DOM, meaning optimised updates by re-rendering only changed parts of the page, leading to a faster UI and smoother experience. However, there is a steep learning curve with key concepts like JSX, state and props. A trial on CodeSandBox showed React's simpler setup and structure compared to Angular which is also written in TypeScript, making React the better choice.

Back-End: Django

We chose Django for our backend as it offers robust features such as user authentication, security tools and an ORM for easy Database Manipulation, making it a stronger choice for other python-based frameworks such as Flask. Similar to React, Django has a large active community with extensive documentation and plenty of tutorials but also has a steep learning curve. Django also encourages design patterns that might not fit every project's needs, leading to less flexibility.

Database: PostgreSQL

We chose Postgres for its scalability, robust security and our familiarity with it. PSQL is highly scalable and has security features such as role-based authentication and we have been learning PSQL for a year now. However, while we are familiar and comfortable with PSQL it has more advanced features such as JSON handling, triggers and stored procedures are not part of our current skill set and needs dedicated time in learning for implementation. While we were confident with our choice in PSQL, to validate our decision, we shared a few resources within the team to ensure we understood the advantages of PSQL over alternatives: PostgreSQL vs MySQL

https://dbconvert.com/blog/mysql-vs-postgres-in-2024/ PSQL - DB of the Year

Data Visualisation: MatPlotLib

We chose MatPlotLib for our statistics page of our web application due to its versatility and compatibility with Python. It supports various plot types, integrates well with GUIs and offers extensive customisation. Despite requiring more setup for advanced visuals, its strong community support and documentation make it a reliable choice.

Chapter 4 - Testing

Req. No.	Requirement Desc.	Functional Requirement	Non-Functional Requirement	Test Case	Status
1	Users receive notifications about application status	Push notifications	Usability	Verify notifications trigger correctly	Pass/Fail
2	Users authenticate using two-factor authentication	Two-factor authentication	Security	Test login with and without 2FA	Pass/Fail
3	The application must work on both PC and mobile devices	Dedicated interface for both PC & mobile devices	Usability	Test functionality on both PC and mobile	Pass/Fail
4	Users should remain logged in across sessions	Ability to stay logged-in across sessions	Usability	Check session remains online after inactivity	Pass/Fail
5	Users should be able to sort placements easily	Sort-by features (job type, A-Z, recently added, etc)	Usability	Test sorting functionality for accuracy	Pass/Fail
6	Users should receive information about relevant job events	Relevant events for jobs	Usability	Test accuracy and timeliness of event updates	Pass/Fail
7	The system should verify user identity	Is the user who they say they are?	Security	Conduct security tests to verify identity	Pass/Fail

8	The system should block unauthorized users	Block bad actors from getting employed by companies	Security	Test system for vulnerabilities and breaches	Pass/Fail

Chapter 5 - Critical Analysis

Leadership

Issues Faced:

1. Lack of clarity in role assignments during the project's initial phase led to the group doing the wrong task for chapter 2 (e.g. creating a physical design rather than a logical design).
2. A lack of a leader led to the group's development of work to be slow at first.

How Issues were Solved:

1. Once the issue was identified, 2109066 immediately made the architectural design and use case graph in order for the development of chapter 2 to progress smoothly without the issue dragging on.
2. Transferred leader responsibility week to week so someone always has leadership and gives people tasks.

Lesson Learned:

1. Follow closely through the guidance provided by Claudia in terms of the mark scheme as it guides us on how to develop our work.
2. A leader should always be in place for team members to report back on. This implements discipline, makes clear what members need to do and holds people accountable for their work.

Progress Monitoring

Issues Faced:

1. We presumed that all group members would be present for the work. This was not the case, delaying project development.
2. Inconsistent updates from members made it challenging to track overall progress effectively and re-distribute work, if not done.

How Issues were Solved:

1. Work was redistributed based on participation levels. Core chapters 1 and 2 were assigned to active members while less critical tasks were assigned to non-contributing members.
2. Rather than waiting for updates, members agreed to give updates during meetings and if updates were needed it was okay to message each other personally in order to get these updates

Lesson Learned:

1. Adapting to different members capabilities is critical in order to identify where work may not be delivered and how to make sure you an adapt to these changes
2. A software for task management ensures better visibility, communication and accountability across the team, much better than manually gathering updates.

Conflict Resolution

Issues Faced:

1. Members had different interpretations of how detailed certain sections of the chapters should be with some members prioritising high detail with others prioritising readability.
2. While drafting the Design Chapter, we made the physical design through Figma rather than Logical design. This could have led to frustration as time and effort were wasted on the wrong task.

How Issues were Solved:

1. Members came to an agreement to follow what the mark scheme states and prioritise fitting all the content into 10 pages.
2. UP2109066 dedicated time to quickly create the Architectural Design and Use Case Diagram, resolving the issue without any delay.

Lesson Learned:

1. Clarifying the structure of tasks should be prioritised before starting work to allow for the work to ensure alignment with the marking scheme
2. Double checking expectations and deliverables before starting any task can prevent unnecessary rework and frustration.