POLITECNICO
MILANO 1863

Software Engineering 2 Implementation & Test Deliever-able

Author(s): **Shreesh Kumar Jha - 11022306**
Samarth Bhatia - 11059097
Satvik Sharma - 11054680

Academic Year: 2024-2025
Version: 2.0
Release date: 29/01/2025

Contents

1 | Introduction

1.1. Purpose

The InternHub - Students & Companies (S&C) platform requires this Installation and Technical Document (ITD) in order to be set up and run. It has been painstakingly created to walk administrators, developers, and stakeholders through each stage of installation and upkeep. This paper guarantees a smooth setup process, reducing potential problems and promoting effective deployment by offering precise and comprehensive instructions. The ITD provides a comprehensive explanation of the platform's function, technical specifications, and architectural design in addition to installation. It highlights how crucial it is to manage dependencies and environment configurations appropriately in order to ensure seamless functioning across several systems. With this guide, users may deploy and configure the SC platform with confidence, knowing that they have a trustworthy resource to handle any issues that may come up.

Fundamentally, the goal of this paper is to make the platform configuration less complicated so that users with different degrees of technical expertise can utilize it. This ITD guarantees that you have the skills and resources required to succeed, whether you're a developer working on the backend, an administrator managing the deployment, or a stakeholder trying to comprehend the operation of the system. It lays the groundwork for a strong and effective internship management system that connects students, businesses, and educational institutions by promoting clarity and minimizing uncertainty.Fundamentally, the goal of this paper is to make the platform configuration less complicated so that users with different degrees of technical expertise can utilize it. This ITD guarantees that you have the skills and resources required to succeed, whether you're a developer working on the backend, an administrator managing the deployment, or a stakeholder trying to comprehend the operation of the system. It lays the groundwork for a strong and effective internship management system that connects students, businesses, and educational institutions by promoting clarity and minimizing uncertainty.

1.2. Scope

By connecting students, businesses, and academic institutions, **InternHub - Students & Companies (S&C)** is a platform that aims to improve and expedite the internship experience. It seeks to match students with appropriate internships, allowing universities to manage the full internship lifecycle and businesses to identify the best candidates.

Companies may post extensive internship openings, analyze applications, and oversee the selection process on the site, while students can create detailed profiles, apply for internships, and follow their progress. Universities serve as supervisors, guaranteeing adherence to academic standards and offering systems for observation and criticism.

The platform aims to streamline processes and promote openness and cooperation among stakeholders with features like intelligent recommendations, feedback and quality assurance tools, and strong communication channels. In order to guarantee a smooth, equitable, and effective experience for every user, it also incorporates analytics, training materials, and a grievance management system.

1.3. Definitions, Acronyms, Abbreviations

Below is a table that lists all the goals of the S&C platform:

Term/Acronym	Definition
S&C	Students & Companies Platform
RASD	Requirements Analysis & Specification Document
CV	Curriculum Vitae
UI	User Interface
API	Application Programming Interface
DBMS	Database Management System
SLA	Service Level Agreement
GDPR	General Data Protection Regulation
ITD	Installation and Technical Document

Table 1.1: Definitions, Acronyms, and Abbreviations

1.4. Revision History

Version	Date	Description	Authors
1.0	03 January 2025	Initial Release	Shreesh Kumar Jha, Samarth Bhatia
2.0	04 January 2025	LaTeX Format and Minor Fixes	Shreesh Kumar Jha, Samarth Bhatia

Table 1.2: Revision History

1.5. Reference Documents

- Reference to Previous Year Student Projects for Structuring the Document
- **Specification Document Assignment**

1.6. Document Structure

The document is divided into the following sections, each focusing on specific aspects of the installation and technical setup:

Introduction

- Establishes the purpose, scope, and structure of the document.
- Defines the key acronyms, abbreviations, and references used throughout.
- Provides historical revisions and context for updates.

Implemented Requirements

- Lists the core functional and technical requirements fulfilled by the platform.
- Includes features such as student and company registration, internship management, and data handling.

Design Choices

- Describes the architectural patterns, frameworks, and libraries employed in the platform's development.
- Covers key decisions related to scalability, reliability, and maintainability.

Source Code Structure

- Provides a breakdown of the backend, frontend, and database components.
- Includes directory hierarchies, file purposes, and dependencies.

Testing

- Outlines testing methodologies such as unit, integration, and end-to-end tests.
- Describes tools and strategies used to validate the platform's functionality and performance.

Installation Instructions

- Offers step-by-step guidance for setting up the backend, frontend, and database.
- Details prerequisites, installation commands, and troubleshooting tips.

Effort Spent

- Summarizes the contributions and hours worked by each team member on the document and platform.

References

- Lists all the sources and references used in the document.

2 | Implemented Requirements

2.1. Overview

This section describes the functionalities that have been implemented in relation to the requirements specified in the RASD paper. Every feature of the system is mentioned below, along with an explanation of how to handle the database, front end, and back end.

2.2. User Management

Implemented Requirements

- [**F1.1**] : The system allows students, companies, and university administrators to register with verified email addresses.
- [**F1.3**] : The system allows users to update their profile information, including contact details and preferences.
- [**F1.4**] : The system enforces role-based access control for students, companies, and administrators.
- [**F1.5**] : The system supports password reset functionality with email verification.
- [**F1.6**] : The system maintains audit logs of all user authentication activities.
- [**F1.7**] : The system allows users to manage notification preferences.
- [**F1.8**] : The system enforces strong password policies.

Non-Implemented Requirements

- [**F1.2**] : Secure authentication with optional two-factor verification has not been implemented due to time constraints and the additional complexity of integrating multi-factor authentication systems at this stage of development.

Database

Tables:

- `users`: Stores user details, including email, password, role, and status.
- `user_profiles`**(Student, Recruiter, Admin)**: Contains user-specific details such as contact information and preferences.
- `notification_preferences`: Stores user preferences for managing notifications.

Constraints:

- Emails are unique across all users and validated during registration.
- Passwords are hashed and salted using bcrypt before storage.
- Role-based constraints ensure that data integrity is maintained for each user type (student, company, administrator).

Back-end

Controllers:

- `userController.js`:
  - Handles user registration (`POST /register`), login (`POST /login`), and password reset functionality (`POST /reset-password`).
  - Verifies email addresses during registration by sending an email with a verification token.
- `studentController.js`, `recruiterController.js`, `adminController.js`:
  - Role-specific logic for profile updates and account management.

Functions:

- **Registration Validation:** Ensures that all required fields (e.g., email, password, role) are provided and correctly formatted.
- **Role Enforcement:** Restricts access to certain endpoints based on the role field stored in JWT tokens.

Front-end

Application and Website:

- **Registration:** Students, companies, and administrators register using forms with dynamic validation feedback.
- **Profile Updates:** Users can update personal details (e.g., contact information) through a dedicated profile section.
- **Password Reset:** Users initiate a password reset process, with password strength validation before submission.
- **Role-Based Dashboards:** Students, companies, and administrators are redirected to role-specific dashboards post-login.

Commentary on Non-Implemented Requirements

- [**F1.2**] : Multi-factor authentication was excluded due to the effort required for integrating and managing a third-party authentication service, such as Google Authenticator or Twilio for SMS. This feature is planned for future iterations to enhance security.

2.3. CV Management

Implemented Requirements

- [**F2.1**] : The system provides customizable CV templates for students.
- [**F2.2**] : Students can create and store multiple versions of their CVs.
- [**F2.3**] : Students can update their CVs at any time.
- [**F2.4**] : Students can control CV visibility to specific companies.
- [**F2.6**] : The system validates skill entries against a standardized skill database.

Non-Implemented Requirements

Database

Model: `Cv` **user:** References the User model to associate the CV with a specific student. **template:** Stores the CV template identifier. **data:** Contains

the detailed CV content, including sections like Education, Skills, and Experience. **visibility:** An array of company IDs defining which companies can view the CV.

Constraints:

- A student can have only one active CV at a time, with version updates stored dynamically.
- The `visibility` field ensures that only authorized companies can access the CV.

Back-End

Routes: Defined in `cvRoutes.js`

- `POST /`: Creates or updates a CV for the logged-in user.
- `GET /latest`: Fetches the latest CV of the logged-in user.
- `DELETE /`: Deletes the logged-in user's CV.
- `POST /update-visibility`: Updates the visibility settings for a CV.
- `GET /:studentId`: Retrieves a specific student's CV by their ID.

Controllers: Defined in `cvController.js`

- **createOrUpdateCV:** Handles both creation and updates to a CV. If a CV already exists for a student, it updates the template, data, and visibility fields.
- **getCV:** Retrieves the logged-in user's CV. If none exists, returns a 404 error.
- **deleteCV:** Deletes the logged-in user's CV.
- **updateVisibility:** Updates the list of companies that can view a specific CV.

Front-End

Application:

- **CV Builder:**

    * Students can select templates and customize sections like Education, Skills, and Experience.
    * Live previews of the CV are available during editing.
  - **Visibility Management:**
    * A toggle interface allows students to manage company permissions for viewing their CVs.

Website:

- The CV builder is integrated with the student dashboard.
- Visibility settings can be updated directly from the CV management panel.

Commentary on Non-Implemented Requirements

- [**F2.7**] Document upload functionality has not been implemented due to the complexity of file storage and retrieval. This feature is planned for future development.
- [**F2.8**] CV view statistics tracking is excluded as an analytics framework for monitoring company interactions has not yet been integrated.

2.4. Internship Management

Implemented Requirements

- [**F3.1**] : The system allows companies to create detailed internship postings.
- [**F3.2**] : Provides an application tracking system for companies.
- [**F3.3**] : Automatically notifies students of application status changes.
- [**F3.4**] : Provides advanced search and filtering capabilities.
- [**F3.6**] : Enables bulk application processing for companies.
- [**F3.7**] : Supports multiple rounds of application review.
- [**F3.8**] : Maintains a history of all internship postings.

Non-Implemented Requirements

– [**F3.5**] : Allowing companies to set application deadlines.

Database

Model: `Internship`

Fields:

– **Core Details:** `title`, `description`, `location`, `duration`, `stipend`.

– **Skills:** `requiredSkills` (mandatory), `preferredSkills` (optional).

– **Experience:** `experienceLevel` (e.g., beginner, intermediate, advanced).

– **Applicants:** Stores references to users who applied.

– **Interviews:** Includes details of scheduled interviews.

– **Status:** Tracks whether the internship is open, closed, or draft.

– **Timestamps:** `createdAt`, `updatedAt` fields auto-generated by Mongoose.

Back-End

Routes: Defined in `internshipRoutes.js`

– `POST /addinternship`: Adds a new internship (role restricted to recruiters).

– `GET /allinternships`: Retrieves all internship postings.

– `POST /:id/apply`: Allows students to apply for internships.

– `GET /recommended`: Provides personalized internship recommendations for students.

– `PATCH /:internshipId/interviews/:studentId/completed`: Marks interviews as completed.

Controllers:

– **addInternship:** Handles creation of internships, validates inputs, and ensures required fields are present.

– **getAllInternships:** Fetches all internships, sorted by the latest postings.

- **applyToInternship:** Handles application submissions, ensuring no duplicate applications.

Front-End

Application:

- **For Companies:** A guided form for creating internships with fields for `title`, `description`, `skills`, and experience requirements.
- **For Students:** Search and filter functionalities to find internships based on location, skills, and duration.
- **Dashboard Integration:** Displays a list of applied internships with their status.

Website:

- Applications dashboard for recruiters, showing real-time updates on applications received.

Commentary on Non-Implemented Requirements

- [**F3.5**] Application deadlines are not implemented in this version due to the additional logic required to enforce and validate deadlines during application submissions. This feature is planned for future iterations.

2.5. Interview Management

Implemented Requirements

- [**F4.1**] : The system provides a calendar interface for interview scheduling.
- [**F4.2**] : Tracks interview status and progress.
- [**F4.5**] : Supports virtual interview link generation.
- [**F4.6**] : Allows rescheduling with mutual agreement.
- [**F4.7**] : Maintains interview history.
- [**F4.8**] : Supports multiple interview rounds.

Non-Implemented Requirements

- [**F4.3**] : Structured feedback collection from both parties is not implemented.
- [**F4.4**] : Automated interview reminders are not supported.

Database

Model: `Internship`

Fields:

- **scheduledInterviews:** An array of interview objects, each containing:
  - **student:** References the User model to identify the applicant.
  - **dateTime:** Date and time of the interview.
  - **meetLink:** Google Meet or other virtual link for the interview.
  - **status:** Tracks the interview state (Scheduled or Completed).

Model: `Student`

Fields:

- **scheduledInterviews:** Tracks interviews scheduled for the student, referencing internship details and interview metadata.

Back-End

Routes: Defined in multiple controllers (`internshipRoutes.js`, `studentRoutes.js`, `recruiterRoutes.js`):

- `POST /:id/schedule`: Schedules an interview for a student (role restricted to recruiters).
- `GET /student/interviews`: Retrieves scheduled interviews for students.
- `GET /recruiter/interviews`: Fetches interviews scheduled by recruiters.
- `PATCH /:internshipId/interviews/:studentId/completed`: Marks an interview as completed.

- `GET /student/completed-interviews`: Retrieves a student's completed interviews.

Controllers:

- **scheduleInterview:** Generates a virtual link via Google Calendar API and saves interview details to both the internship and student profiles.
- **getRecruiterInterviews:** Fetches all interviews related to a recruiter's internships.
- **getStudentInterviews:** Retrieves all scheduled interviews for a student.
- **markInterviewAsCompleted:** Updates the interview status to Completed.

Front-End

Application:

- **Calendar Integration:**
  - Students and recruiters can view interview schedules on an interactive calendar interface.
  - Recruiters can reschedule interviews using drag-and-drop functionality.
- **Interview History:**
  - Students can view completed interviews with details such as company name, date, and virtual link.

Website:

- Recruiters have a dashboard displaying all scheduled and completed interviews for their internships.
- Students can access a list of upcoming interviews along with relevant details.

Commentary on Non-Implemented Requirements

- [**F4.3**] : Feedback collection was deprioritized in this iteration due to the additional database design and UI components required for structured feedback forms.

- [**F4.4**] : Automated reminders for interviews were excluded to avoid the integration complexity of real-time notification systems. These will be considered in future versions.

2.6. Recommendation System

Implemented Requirements

- [**F5.1**] : Matches students with internships based on skills alignment.
- [**F5.2**] : Suggests qualified candidates to companies.
- [**F5.3**] : Learns from user interactions to improve recommendations.
- [**F5.4**] : Generates personalized internship suggestions.
- [**F5.5**] : Considers location preferences in matching.
- [**F5.6**] : Factors in past application success patterns.
- [**F5.7**] : Updates recommendations in real-time.
- [**F5.8**] : Explains recommendation reasoning to users.

Database

Model: `Internship`

Fields:

- **requiredSkills & preferredSkills:** Define skill requirements for internships.
- **location:** Captures location data for matching against student preferences.
- **applicants:** Tracks users who have applied to the internship.

Model: `Student`

Fields:

- **profile.skills:** Stores the student's skills for alignment with internship requirements.
- **profile.location:** Specifies location preferences.

- **appliedInternships:** Tracks internships the student has applied for.

Back-End

Routes: Defined in `internshipRoutes.js`

- `GET /recommended`: Fetches recommended internships for a student based on their skills, location, and profile data.

Controllers:

- **getRecommendedInternships:**
  - Fetches the student's CV and profile data.
  - Matches internships using weighted scoring:
    - **Skills Alignment (50%):** Compares student skills with required and preferred skills.
    - **Experience Level (30%):** Considers the total duration of relevant experiences.
    - **Location Match (20%):** Matches student's preferred location with internship location.
  - Excludes internships already applied for by the student.
  - Sorts and returns the top 10 recommendations with match scores and reasoning.
- **calculateSkillMatch:** Compares student and internship skills, returning a similarity score and list of matching skills.
- **calculateLocationPreference:** Scores based on the student's preferred and internship location match.

Front-End

Application:

- **Dashboard Integration:**
  - Displays personalized internship recommendations with match scores.

    * Shows reasons for each recommendation, such as skill match or location preference.

  – **Real-Time Updates:** Recommendations dynamically update based on profile or application changes.

Website:

  – Search results highlight recommended internships, ranked by match score.

  – Filters allow students to refine recommendations based on additional criteria.

2.7. Complaint Handling

Implemented Requirements

  – [**F6.1**] : Provides a structured complaint submission interface.

  – [**F6.2**] : Routes complaints to appropriate university administrators.

  – [**F6.3**] : Tracks resolution progress for each complaint.

  – [**F6.4**] : Supports an appeals process.

  – [**F6.5**] : Maintains a complete complaint history.

  – [**F6.6**] : Enables communication between involved parties.

  – [**F6.7**] : Generates complaint resolution reports.

  – [**F6.8**] : Enforces resolution timeframes.

Database

Model: `Complaint`

Fields:

  – **userId:** References the User model to identify the user who raised the complaint.

  – **role:** Specifies the role of the user (student or recruiter).

  – **title:** Title of the complaint for quick reference.

  – **description:** Detailed description of the issue.

- **status:** Tracks the state of the complaint (pending or resolved).
- **createdAt:** Timestamp for when the complaint was created.

Back-End

Routes: Defined in `complaintRoutes.js`

- `POST /create-complaint`: Allows students or recruiters to submit complaints.
- `GET /get-complaints`: Fetches all pending complaints (restricted to admins).
- `PATCH /:complaintId/resolve`: Marks a complaint as resolved (restricted to admins).
- `GET /my-complaints`: Retrieves complaints submitted by the logged-in user.

Controllers: Defined in `complaintController.js`

- **createComplaint:** Handles the creation of complaints, validates inputs, and stores them in the database.
- **getComplaints:** Retrieves all complaints with a pending status, populated with user details.
- **resolveComplaint:** Updates the status of a complaint to be resolved.
- **getMyComplaints:** Fetches complaints submitted by the logged-in user, allowing them to track their status.

Front-End

Application:

- **Complaint Submission Interface:**
  - Students and recruiters can submit complaints through a structured form with fields for the title and description.
- **Complaint History:**
  - Displays a list of all submitted complaints with their statuses.
  - Provides real-time updates on resolution progress.

Website:

- **Admin Dashboard:**
  - Administrators can view, filter, and resolve complaints.
  - Includes an interface for generating complaint resolution reports.

3 | Design Choices

3.1. Adopted Programming Languages: Advantages and Disadvantages

1. Frontend Language: JavaScript with React (and Vite)

React is a JavaScript library used for building interactive user interfaces. Coupled with Vite as a build tool, it provides a modern, efficient way to manage the frontend development of the application.

Advantages:

- **Component-Based Architecture:**
  - React uses reusable components, enabling developers to build a modular UI.
  - Components can be composed and maintained independently, increasing maintainability.
  - Example: A "Button" component can be reused across different parts of the application with varying styles or functionalities.
- **Virtual DOM:**
  - React's Virtual DOM improves performance by reducing direct interaction with the actual DOM.
  - Only the changes are updated in the DOM, resulting in faster rendering and better user experience.
  - Example: In an internship listing page, updating one entry does not require re-rendering the entire list.
- **Strong Ecosystem:**

    * Integration with tools like React Router for navigation and Redux for state management.
    * Large community support ensures abundant resources, libraries, and troubleshooting help.
    * Example: Material-UI provides pre-built, customizable UI components, accelerating development.
  - **Hot Module Replacement (HMR) via Vite:**
    * Vite enables instantaneous feedback during development by updating only the affected modules without a full reload.
    * Example: Editing the CSS of a single component updates the preview immediately without reloading the page.
  - **SEO and SSR Support:**
    * React supports Server-Side Rendering (SSR), which can improve SEO for pages needing indexing.
    * Example: Internship search pages rendered on the server can rank better on search engines.
  - **Cross-Platform Compatibility:**
    * React's flexibility allows the use of frameworks like React Native for mobile development, unifying frontend logic.

Disadvantages:

  - **Steep Learning Curve:**
    * Beginners may find it difficult to grasp advanced concepts like hooks, higher-order components (HOCs), and context API.
    * Example: The useEffect hook's dependency array often confuses developers, leading to unexpected behaviors.
  - **Boilerplate Code:**
    * Although modular, managing state and lifecycle events can introduce excessive boilerplate code, especially in larger applications.
    * Example: Redux requires setting up actions, reducers, and a store, which adds complexity.

- **Heavy Dependency on External Libraries:**
  - React does not include built-in features like routing or HTTP requests, requiring additional libraries such as Axios or React Router.
  - This increases dependencies and potential compatibility issues.
- **Performance Bottlenecks with Large Applications:**
  - Excessive use of stateful components or improper state management can degrade performance.
  - Example: Rendering too many child components unnecessarily on state change can cause lag.

2. Backend Language: JavaScript with Node.js and Express

Node.js is a runtime environment for executing JavaScript on the server side, while Express.js is a lightweight framework for building web applications and APIs.

Advantages:

- **Asynchronous, Event-Driven Architecture:**
  - Node.js uses non-blocking I/O operations, making it highly efficient for real-time applications and heavy workloads.
  - Example: Simultaneous internship application submissions from multiple users can be processed without delay.
- **Single Programming Language:**
  - Using JavaScript for both the frontend and backend ensures seamless communication and reduces context switching for developers.
  - Example: Shared utility functions like data formatting can be reused across the application.
- **Vast Ecosystem:**
  - The npm ecosystem offers thousands of libraries and tools to extend functionality.
  - Example: Mongoose simplifies interactions with MongoDB, while jsonwebtoken enables secure authentication.

- **Lightweight and Scalable:**
  - Node.js is lightweight and scales efficiently with tools like clustering or load balancers.
  - Example: Horizontal scaling can be achieved to handle traffic spikes during peak application submission periods.
- **Microservices-Friendly:**
  - Node.js is suitable for building microservices with RESTful APIs.
  - Example: Separate APIs for internships, applications, and notifications allow independent scaling and maintenance.
- **Real-Time Communication:**
  - WebSockets in Node.js enable real-time features such as notifications or live chat.
  - Example: Students and recruiters can communicate directly via instant messaging.

Disadvantages:

- **Single-Threaded Nature:**
  - Node.js runs on a single-threaded event loop, which can cause performance issues with CPU-intensive tasks.
  - Example: Large computations, such as generating complex analytics for platform administrators, might block the thread.
- **Callback Hell:**
  - Complex logic with multiple asynchronous operations can lead to deeply nested callbacks.
  - Mitigation with Promises or async/await is essential but adds learning overhead.
- **Rapid API Changes:**
  - The fast-paced evolution of Node.js and its libraries can result in compatibility issues.

    * Example: Upgrading a major version of a library might require significant code refactoring.

  – **Security Challenges:**

    * The vast npm ecosystem includes packages with potential vulnerabilities.

    * Example: Using unverified or outdated third-party libraries can expose the application to risks like dependency injection attacks.

3. Database Language: MongoDB (Query Language)

MongoDB uses a document-oriented NoSQL query language for managing and manipulating JSON-like data.

Advantages:

  – **Schema Flexibility:**

    * MongoDB allows dynamic schema designs, which are beneficial for applications with evolving data models.

    * Example: New attributes can be added to internship or user profiles without altering the existing structure.

  – **High Scalability:**

    * Horizontal scaling with sharding enables MongoDB to handle large datasets efficiently.

    * Example: Storing thousands of CVs or application records across multiple servers.

  – **Integration with JavaScript:**

    * MongoDB's query syntax aligns with JavaScript, making it easier to use with Node.js.

    * Example: Queries can be written directly in JavaScript objects without additional abstraction layers.

  – **Performance with Write-Heavy Workloads:**

    * The database is optimized for high write speeds, suitable for logging and storing real-time data.

* Example: Real-time activity logs for internship applications or interactions.

Disadvantages:

- **Limited Query Capabilities:**
  * MongoDB lacks advanced relational query features like joins or transactions (in earlier versions).
  * Example: Complex relationships between students, companies, and internships require additional handling in the application logic.
- **Memory Usage:**
  * MongoDB uses significant memory due to its document-based design.
  * Example: Storing large datasets, such as resumes with attachments, can quickly consume resources.
- **Consistency Issues:**
  * In distributed systems, MongoDB prioritizes availability over consistency, which may lead to outdated data reads in certain scenarios.
- **Indexing Overhead:**
  * Proper indexing is critical for query performance but increases storage requirements and maintenance complexity.

3.2. Adopted Middleware: Advantages and Disadvantages

Middleware is a critical part of the software architecture that sits between the frontend, backend, and database. Middleware simplifies handling requests, responses, authentication, data validation, and more.

1. Express Middleware

Express.js provides built-in and third-party middleware to streamline the backend development process. It is central to handling HTTP requests, parsing data, and chaining operations in the backend.

Examples Used:

- Body Parsing: `express.json()` and `express.urlencoded()`.
- Routing Middleware: `express.Router()` for modular route handling.

Advantages:

- **Simplified Request Handling:**
  - Middleware functions provide modular, reusable code for handling HTTP requests and responses.
  - Example: Parsing JSON payloads with `express.json()` simplifies extracting data from API requests.
- **Modular Architecture:**
  - Enables dividing the application into smaller, manageable route handlers with `express.Router()`.
  - Example: Separate routes for user authentication, internships, and complaints.
- **Comprehensive Ecosystem:**
  - Integrates seamlessly with a wide range of third-party middleware for security, logging, and more.
  - Example: `helmet` middleware secures HTTP headers.
- **Chained Middleware Execution:**
  - Allows multiple middleware functions to process a request sequentially.
  - Example: Authentication -> Validation -> Business Logic -> Response.
- **Error Handling:**
  - Custom error-handling middleware improves debugging and ensures consistent error responses.

Disadvantages:

- **Manual Middleware Management:**
  - Developers must explicitly define middleware for each route or use-case.

    * Example: Forgetting to add a validation middleware for a route can cause unexpected errors.

- **Performance Overhead:**
    * Excessive middleware layers can increase request-processing time.
    * Example: Sequentially processing redundant middlewares can slow down responses for high-traffic endpoints.
- **Lack of Opinionated Defaults:**
    * While flexible, Express does not provide default configurations for security or performance, requiring extra effort to set up.

2. Mongoose Middleware (ODM for MongoDB)

Mongoose provides pre- and post-save hooks (middleware) for MongoDB models, enabling developers to handle logic at various stages of data interaction.

Examples Used:

- Pre-save Hooks: Validate data or hash passwords before saving.
- Post-remove Hooks: Clean up related data after a record deletion.

Advantages:

- **Built-in Lifecycle Hooks:**
    * Automates repetitive tasks like validation, password hashing, or cascading deletes.
    * Example: A pre-save hook to hash student passwords before storing them in the database.
- **Data Validation:**
    * Middleware ensures schema validation rules are applied consistently.
    * Example: Validating internship fields (e.g., stipend, duration) before saving to the database.
- **Centralized Logic:**

    * Reduces redundancy by allowing data-related logic to reside within the models.
    * Example: Automatically updating timestamps for records via pre-update hooks.
  - **Error Handling:**
    * Provides a robust mechanism for handling schema validation errors at the database level.
    * Example: Preventing invalid CV uploads by rejecting oversized files.

Disadvantages:

  - **Performance Impact:**
    * Hooks can introduce latency, especially if they involve complex logic or external service calls.
    * Example: A pre-save hook that verifies unique email addresses via an API may slow down user registration.
  - **Debugging Complexity:**
    * Errors in middleware logic are harder to trace because they occur outside of the main request-response flow.
    * Example: Misconfigured validation middleware causing silent failures.
  - **Limited Customization:**
    * Mongoose middleware can be rigid, making it challenging to accommodate advanced use cases without workarounds.

3. Authentication Middleware: JSON Web Tokens (JWT)

JWT middleware provides secure authentication and authorization for APIs by issuing signed tokens that clients include in subsequent requests.

Examples Used:

  - `jsonwebtoken` for signing and verifying tokens.
  - Middleware to decode and validate JWT tokens on protected routes.

Advantages:

- **Stateless Authentication:**
  - No need for server-side session storage; the client stores tokens, reducing backend load.
  - Example: Students and recruiters authenticate using tokens stored in their browsers or apps.
- **Scalability:**
  - Works efficiently in distributed systems since tokens do not rely on centralized storage.
  - Example: APIs can verify user identity without querying a database for each request.
- **Flexibility:**
  - Tokens can include custom claims for roles, permissions, or session data.
  - Example: A JWT can contain a role field (e.g., "student" or "recruiter") for role-based access control.
- **Security Features:**
  - Tokens are signed, ensuring data integrity and preventing tampering.
  - Example: If the signature does not match, the token is invalid.

Disadvantages:

- **Token Storage Risks:**
  - Storing tokens insecurely (e.g., in local storage) can expose the application to attacks like XSS.
  - Mitigation: Use HTTP-only cookies for storing tokens.
- **No Built-in Expiration Handling:**
  - JWT does not include token revocation by default, requiring additional logic to handle blacklisting.
  - Example: Invalidating tokens when users log out requires a server-side token blacklist.

- **Payload Size:**
  - Large payloads can increase request sizes, affecting performance in some cases.
  - Example: Including unnecessary claims in the token may bloat the payload.

4. Logging Middleware: Morgan

Morgan is a logging middleware for Express applications, primarily used to track HTTP requests and responses.

Advantages:

- **Simplifies Debugging:**
  - Logs critical request details, such as HTTP method, status code, and response time.
  - Example: Logs can identify slow endpoints causing performance bottlenecks.
- **Customizable Formats:**
  - Supports pre-defined and custom log formats for varying use cases (e.g., combined, dev).
  - Example: Using the "combined" format for production and "dev" for debugging.
- **Integrates with External Services:**
  - Compatible with log management tools like Winston or external logging services (e.g., Elasticsearch).
  - Example: Piping logs to a centralized dashboard for analytics.

Disadvantages:

- **Performance Overhead:**
  - Generating and storing logs for every request can slightly impact performance.

    * Example: High-traffic APIs may experience latency due to excessive logging.

  - **Security Concerns:**
    * Improperly configured logs may expose sensitive data, such as authorization headers or tokens.

3.3. APIs Not Included in the Design Document (DD)

This section outlines the APIs that were not explicitly mentioned in the Design Document (DD) but could significantly enhance the functionality, usability, and scalability of the InternHub - Students & Companies (S&C) platform.

1. File Upload API

Purpose: Manage user file uploads, including CVs, certificates, and other required documents.

Potential Use Cases:

- Allow students to upload resumes and certificates directly to their profiles.
- Enable recruiters to upload job descriptions or supporting files for internships.

Advantages:

- Secure and efficient storage for user documents.
- Built-in versioning to track changes and updates.
- Integration with cloud storage services like AWS S3 or Cloudinary for scalability.

Disadvantages:

- Requires additional infrastructure for large-scale file handling.
- Increased bandwidth costs for large files.

2. Notification API

Purpose: Extend the notification system to support multi-channel delivery, including SMS, email, and push notifications.

Potential Use Cases:

- Notify students about internship application updates.
- Send interview reminders via SMS or email.

Advantages:

- Improved user engagement with timely notifications.
- Multi-channel support ensures critical updates reach users even if they are offline.

Disadvantages:

- Requires integration with external providers (e.g., Twilio, SendGrid), adding to costs.
- Dependency on third-party services introduces potential downtime risks.

3. Analytics API

Purpose: Provide insights into system usage, application trends, and user engagement.

Potential Use Cases:

- Track the number of applications submitted per internship.
- Generate reports on platform activity for administrators.

Advantages:

- Helps stakeholders make data-driven decisions.
- Real-time analytics can identify bottlenecks or underperforming features.

Disadvantages:

- Complex queries and real-time processing may increase backend load.
- Requires additional storage for aggregated data.

4 | Source Code Stucture

This section provides a detailed explanation of the backend and frontend source code, covering the purpose of every file and folder in your project structure. The description is focused on providing a high-level understanding of what each part of the project contains.

4.1. Backend Project Structure

The backend is implemented using Node.js and Express.js, organized into modular directories. Below is an explanation of each directory and its files.

4.1.1. Root Directory

- `server.js`: The main entry point of the backend application. It initializes the Express server, connects to the database, and loads routes and middleware.
- `app.js`: Configures middleware and error handling for the application.
- `.env`: Environment variables, such as database connection strings, API keys, and port numbers.
- `package.json` and `package-lock.json`: Define project dependencies, scripts, and metadata.
- `babel.config.js`: Configures Babel for JavaScript transpilation.

4.1.2. controllers/

The controllers directory contains the logic for handling incoming HTTP requests and returning responses. Each controller corresponds to a specific feature or domain.

- `adminController.js`: Handles administrative functions, such as managing users and settings.

- `authController.js`: Manages user authentication, including login, registration, and password resets.
- `complaintController.js`: Processes complaints submitted by users.
- `configurationController.js`: Handles system configuration settings.
- `cvController.js`: Manages CV uploads, updates, and retrievals.
- `internshipController.js`: Handles CRUD operations for internships.
- `messagingController.js`: Enables communication between users.
- `recommendationController.js`: Generates personalized recommendations for internships or candidates.
- `recruiterController.js`: Manages recruiter-specific functionalities, such as posting internships.
- `reportController.js`: Generates reports for admins and recruiters.
- `studentController.js`: Handles student-specific functionalities, such as profile management and applications.
- `userController.js`: General user operations, such as profile updates and retrieving user information.

4.1.3. models/

This directory contains the Mongoose schemas for MongoDB collections.

- `Complaint.js`: Schema for complaints, including fields like userId, description, and status.
- `Configuration.js`: Stores system configurations, such as feature toggles.
- `Cv.js`: Represents a student's CV, including file paths and metadata.
- `Internship.js`: Schema for internships, with fields like title, company, description, and applications.
- `Message.js`: Represents messages between users, including sender, receiver, and timestamps.
- `Recruiter.js`: Schema for recruiters, including company information.
- `Student.js`: Stores student details, such as name, email, and skills.

- `UsageLog.js`: Tracks API usage and system activity for auditing.
- `User.js`: Base schema for all users, storing common fields like email, password, and role.

4.1.4. routes/

Defines all API endpoints and maps them to the respective controller methods.

- `adminRoutes.js`: Routes for admin-specific operations (e.g., `/admin/manage-users`).
- `authRoutes.js`: Authentication endpoints (e.g., `/auth/login`, `/auth/register`).
- `complaintRoutes.js`: Endpoints for submitting and resolving complaints.
- `configurationRoutes.js`: Routes for updating system configurations.
- `cvRoutes.js`: Routes for uploading and managing CVs.
- `internshipRoutes.js`: CRUD operations for internships (e.g., `/internships/create`).
- `messagingRoutes.js`: Endpoints for sending and retrieving messages.
- `recommendationRoutes.js`: Routes for generating recommendations.
- `reportRoutes.js`: Endpoints for generating reports.
- `userRoutes.js`: General user-related routes (e.g., `/user/profile`).

4.1.5. middleware/

Contains reusable middleware for processing requests and responses.

- `authMiddleware.js`: Verifies user authentication and role-based access control.
- `errorHandler.js`: Centralized error handling for the application.
- `logUsage.js`: Logs API usage for monitoring and debugging.

4.1.6. utils/

Holds utility functions for common operations.

- `hashPassword.js`: Functions for hashing and validating passwords.
- `generateToken.js`: Creates JSON Web Tokens (JWTs) for authentication.

4.1.7. config/

Configuration files for setting up the application environment.

- `env.js`: Loads and exports environment variables.

4.1.8. tests/

Contains unit and integration tests for backend functionality.

- Examples:
  - Tests for controllers (`authController.test.js`).
  - Route integration tests (`authRoutes.test.js`).

4.2. Frontend Project Structure

The frontend uses React.js with Vite, organized into a modular directory structure for scalability and reusability.

4.2.1. Root Directory

- `main.jsx`: Entry point for rendering the React application.
- `App.jsx`: Root component containing global routes and layouts.
- `index.html`: HTML template for the application.
- `package.json`: Lists dependencies like React, Vite, and testing libraries.

4.2.2. components/

Contains reusable React components for UI elements.

- `ProtectedRoute.jsx`: Handles route protection based on user authentication.
- `UserMenuDropdown.jsx`: Dropdown menu for user navigation and settings.

4.2.3. pages/

Contains React components representing individual pages.

- `AdminDashboard.jsx`: Admin-specific dashboard showing user and system

statistics.

- `StudentDashboard.jsx`: Displays internship applications and updates for students.
- `CompanyDashboard.jsx`: Allows recruiters to post and manage internships.
- `CVBuilder.jsx`: Interface for students to create or upload CVs.
- `InternshipApplication.jsx`: Page for applying to internships.

4.2.4. context/

Implements global state management using React Context API.

- `authContext.jsx`: Provides authentication state across the application.
- `userContext.jsx`: Manages user data and roles globally.

4.2.5. services/

Handles API interactions with the backend.

- `api.js`: Functions for making HTTP requests (e.g., `getUser()`, `createInternship()`).

4.2.6. assets/

Stores static files like images, icons, and fonts.

4.2.7. styles/

Contains global and component-specific styles.

- `App.css`: Global styles for the application.

4.2.8. tests/

Front-end test cases for React components and services.

array booktabs

4.2.9. API Endpoints Used

Route	Method	Middleware	Controller	Description
/api/auth	Various	None	authRoutes	Handles authentication-related routes for students, recruiters, and administrators
/api/internships	Various	authMiddleware	internshipRoutes	Manages internship-related operations (add, list, apply, schedule interviews)
/api/users	Various	authMiddleware	userRoutes	Handles user-related operations (profiles, roles, recruiters)
/api/cv	Various	authMiddleware	cvRoutes	Manages CV-related operations (create, update, fetch, delete)
/api/recommendations	GET	authMiddleware	recommendationRoutes	Handles internship recommendations based on CV and profile data
/api/messaging	Various	authMiddleware	messagingRoutes	Manages messaging and chat features between users
/api/admin	Various	authMiddleware, roleMiddleware	adminRoutes	Admin operations such as managing users and roles
/api/complaints	Various	authMiddleware	complaintRoutes	Handles creation, resolution, and retrieval of complaints
/api/reports	Various	authMiddleware, roleMiddleware	reportRoutes	Provides usage statistics, user analytics, and downloadable reports
/api/configurations	GET, PUT	authMiddleware	configurationRoutes	Manages application configurations and settings
/api	ALL	logUsage	logUsage Middleware	Logs API usage data for analytics and tracking
/health	GET	None	Inline Response	Health check for server status

Table 4.1: API Endpoints Used

Table 4.2: API Endpoints Overview

Endpoint	Type	Params	Response Codes	Description
`/users`	GET	**Query:** {search, role}	200: Success, 500: Server Error	Fetch all users with optional filters (search and role).
`/users`	POST	**Body:** {email, password, role, firstName, lastName}	201: Created, 400: Validation Error	Add a new user to the system.
`/users/:id`	PUT	**Body:** {firstName, lastName, role, profile}	200: Success, 404: User Not Found	Update an existing user's details.
`/users/:id`	DELETE	**Path:** {id}	200: Deleted, 404: User Not Found	Delete a user by their ID.
`/create-complaint`	POST	**Body:** {title, description}	201: Created, 500: Server Error	Create a new complaint for the current user.
`/get-complaints`	GET	None	200: Success, 500: Server Error	Fetch all complaints (admin only).
`/:complaintId/resolve`	PATCH	**Path:** {complaintId}	200: Resolved, 404: Complaint Not Found	Resolve a complaint by its ID (admin only).
`/my-complaints`	GET	None	200: Success, 500: Server Error	Fetch complaints submitted by the current user.
`/addinternship`	POST	**Body:** {title, description, location, duration, stipend, requiredSkills, preferredSkills, experienceLevel, applicationDeadline}	201: Created, 500: Server Error	Create a new internship (recruiter only).
`/allinternships`	GET	None	200: Success, 500: Server Error	Fetch all available internships.
`/:id/apply`	POST	**Path:** {id}	200: Applied, 404: Internship Not Found, 400: Already Applied	Apply for an internship (student only).
`/recommended-internships`	GET	None	200: Success, 404: No CV Found	Fetch recommended internships based on the user's CV and profile.
`/send`	POST	**Body:** {conversationId, content}	201: Created, 400: Validation Error	Send a new message within a conversation.
`/start`	POST	**Body:** {receiverId}	201: Conversation Created, 400: Validation Error	Start a new conversation with another user.
`/profile`	GET	None	200: Success, 500: Server Error	Fetch the current user's profile.
`/profile`	PUT	**Body:** {updates}	200: Updated, 500: Server Error	Update the current user's profile.
`/student/register`	POST	**Body:** {email, password, firstName, lastName, profile}	201: Registered, 400: Validation Error	Register a new student account.
`/student/login`	POST	**Body:** {email, password}	200: Success, 401: Invalid Credentials	Login to a student account.
`/company/register`	POST	**Body:** {email, password, firstName, lastName, profile}	201: Registered, 400: Validation Error	Register a new recruiter account.
`/company/login`	POST	**Body:** {email, password}	200: Success, 401: Invalid Credentials	Login to a recruiter account.

Continued on next page.

API Endpoints Overview (continued)

Endpoint	Type	Params	Response Codes	Description
`/admin/register`	POST	**Body:** {email, password, firstName, lastName}	201: Registered, 400: Validation Error	Register a new admin account.
`/admin/login`	POST	**Body:** {email, password}	200: Success, 401: Invalid Credentials	Login to an admin account.
Configurations Endpoints				
`/`	GET	None	200: Success, 500: Server Error	Fetch all configurations.
`/:type`	PUT	**Path:** {type}, **Body:** {value}	200: Updated, 500: Server Error	Update a specific configuration by type.
CV Endpoints				
`/`	POST	**Body:** {template, data, visibility}	201: Created/Updated, 500: Server Error	Create or update a CV.
`/latest`	GET	None	200: Success, 404: CV Not Found	Fetch the latest CV of the logged-in user.
`/`	GET	None	200: Success, 404: CV Not Found	Fetch the logged-in user's CV.
`/`	DELETE	None	200: Deleted, 404: CV Not Found	Delete the logged-in user's CV.
`/update-visibility`	POST	**Body:** {cvId, companyIds}	200: Updated, 500: Server Error	Update the visibility of a CV for specific companies.
`/:studentId`	GET	**Path:** {studentId}	200: Success, 404: CV Not Found	Fetch a specific student's CV by ID.
Reports/Analytics Endpoints				
`/usage-statistics`	GET	None	200: Success, 500: Server Error	Fetch system usage statistics (admin only).
`/user-analytics`	GET	None	200: Success, 500: Server Error	Fetch user analytics (admin only).
`/download`	GET	**Query:** {type}	200: File Downloaded, 400: Missing Params, 500: Server Error	Download a specific report as a PDF.

5 | Testing

5.1. Backend Testing

Admin Controller Tests			
Functionality	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`registerAdmin`	Register a new admin	Valid admin details	Admin registered successfully, status: 201
	Validate password length	Password length < 8	Returns 400, message: "Password should be at least 8 characters long"
	Admin with existing email	Existing admin email	Returns 400, message: "Admin already exists"
`loginAdmin`	Login admin with correct credentials	Valid email and password	Returns 200, token present
	Login admin with incorrect credentials	Invalid password	Returns 401, message: "Invalid credentials"

Table 5.1: Admin Controller Tests

Auth Controller Tests			
Functionality	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`register`	Register new user	Valid user details	User registered successfully, status: 201
	Handle registration errors	Hashing fails	Returns 500, message: "Hashing failed"
`login`	Login user successfully	Valid email and password	Returns 200, token present
	Invalid credentials	Incorrect password	Returns 401, message: "Invalid credentials"
	Handle login errors	Database error	Returns 500, message: "Database error"

Table 5.2: Auth Controller Tests

Complaint Controller Tests			
Functionality	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`createComplaint`	Create a complaint	Valid complaint details	Returns 201, complaint created successfully
	Handle creation errors	Database error during complaint creation	Returns 500, message: "Creation error"
`getComplaints`	Fetch all pending complaints	-	Returns 200, complaints fetched successfully
	Handle fetching errors	Database error	Returns 500, message: "Fetch error"
`resolveComplaint`	Resolve a complaint	Valid complaint ID	Returns 200, complaint resolved
	Resolve non-existent complaint	Invalid complaint ID	Returns 404, message: "Complaint not found"

Table 5.3: Complaint Controller Tests

Configuration Controller Tests			
Functionality	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`getConfigurations`	Fetch all configurations	-	Returns 200, configurations fetched successfully
	Handle fetching errors	Database error	Returns 500, message: "Failed to fetch configurations."
`updateConfiguration`	Update existing configuration	Valid configuration data	Returns 200, configuration updated successfully
	Create new configuration	New configuration data	Returns 200, configuration created successfully
	Handle update errors	Database error	Returns 500, message: "Failed to update configuration."
	Resolve non-existent complaint	Invalid complaint ID	Returns 404, message: "Complaint not found"

Table 5.4: Configuration Controller Tests (Updated)

CV Controller Tests			
Functionality	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`createOrUpdateCV`	Create a new CV	Valid CV details	Returns 201, CV created successfully
	Update existing CV	Existing CV with updated details	Returns 200, CV updated successfully
`getCV`	Fetch user CV	Valid user ID	Returns 200, CV fetched successfully
	Fetch non-existent CV	Invalid user ID	Returns 404, message: "CV not found"
`deleteCV`	Delete user CV	Valid CV ID	Returns 200, CV deleted successfully
	Delete non-existent CV	Invalid CV ID	Returns 404, message: "CV not found"

Table 5.5: CV Controller Tests

Internship Controller Tests			
Functionality	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`addInternship`	Add a new internship	Valid internship details	Returns 201, internship added successfully
`applyToInternship`	Apply to an internship	Internship not found	Returns 404, message: "Internship not found"

Table 5.6: Internship Controller Tests

Messaging Controller Tests			
Functionality	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`getAvailableUsers`	Fetch available users	-	Returns 200, users fetched successfully
`getRecentChats`	Fetch recent chats	-	Returns 200, chats fetched successfully
`getMessages`	Fetch messages	Valid conversation ID	Returns 200, messages fetched successfully
`startConversation`	Start a new conversation	Valid receiver ID	Returns 201, conversation started successfully
`sendMessage`	Send a message	Valid message content	Returns 201, message sent successfully

Table 5.7: Messaging Controller Tests

.5 Recommendation Controller Tests			
Functionality	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`getRecommendedInternships`	Fetch recommended internships	Database error	Returns 500, message: "Error fetching recommendations"
	Fetch with no CV	User has no CV	Returns 404, message: "No CV found. Please create a CV first."

Table 5.8: Recommendation Controller Tests

Recruiter Controller Tests			
Functionality	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`registerRecruiter`	Register a new recruiter	Valid recruiter details	Returns 201, recruiter registered successfully
	Validate password length	Password length < 8	Returns 400, message: "Password should be at least 8 characters long"
`loginRecruiter`	Login recruiter with correct credentials	Valid email and password	Returns 200, token present

Table 5.9: Recruiter Controller Tests

Report Controller Tests			
Functionality	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`getUsageStatistics`	Fetch usage statistics	-	Returns 200, statistics fetched successfully
`getUserAnalytics`	Fetch user analytics	-	Returns 200, analytics fetched successfully
`generateReportPDF`	Generate a report PDF	Valid report type	Returns generated PDF as a buffer

Table 5.10: Report Controller Tests

Student Controller Tests			
Functionality	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`registerStudent`	Register a new student	Valid student details	Returns 201, student registered successfully
`loginStudent`	Login student with correct credentials	Valid email and password	Returns 200, token present

Table 5.11: Student Controller Tests

User Controller Tests			
Functionality	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`getUserProfile`	Fetch user profile	Valid user ID	Returns 200, user profile fetched successfully
`updateUserProfile`	Update user profile	Valid user details	Returns 200, profile updated successfully

Table 5.12: User Controller Tests

Health Check			
Functionality	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`Health Check`	Check server health	-	Returns 200, message: "Server is healthy"

Table 5.13: Health Check

5.2. Frontend Testing

UI Component Tests			
Component	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`ProtectedRoute`	Renders allowed content	Valid user role and route (Role: student, Route: `/student`)	Displays `"Student Dashboard"`
	Redirects unauthenticated	No user authentication (Route: `/student`)	Redirects to `"Login Page"`
	Redirects unauthorized	Unauthorized role for route (Role: student, Route: `/admin`)	Displays `"Student Dashboard"`
	Renders alternate route	Valid user role and alternate route (Role: student, Route: `/profile`)	Displays `"Profile Page"`
`UserMenuDropdown`	Renders dropdown button	`-` (Initials: JD)	Displays `"JD"`
	Shows dropdown menu	Click on dropdown button (Button clicked)	Displays `"Logout"`
	Background color by role	Admin role (Role: admin)	Button has class `bg-blue-500`
	Logout functionality	Click on logout (Clicked logout)	Clears token, redirects to login

Table 5.14: UI/Front-End Component Tests

AddInternship UI Tests			
Component	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`AddInternship`	Renders form	Loads component	Displays all form fields and buttons
	Handles input changes	User inputs valid data (Title, description, location)	Updates input fields accordingly
	Validates duration	User inputs invalid duration (Duration > 24)	Displays `"Duration cannot exceed 24 months"`
	Validates stipend	User inputs negative stipend (Stipend < 0)	Displays `"Stipend must be a non-negative number"`
	Adds and removes skills	Add and remove skills (Skill added/removed)	Updates skill list accordingly
	Handles form submission	Submits valid data (Valid form fields)	Sends API request, displays success message
	Handles submission error	API error on submission (Server error during submission)	Displays error alert
	Disables submit on error	Validation errors (Invalid inputs)	Submit button is disabled
	Cancel button click	User clicks cancel (Cancel clicked)	Redirects to company dashboard

Table 5.15: AddInternship Component Tests

Admin Dashboard UI Tests			
Component	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`AdminDashboard`	Renders main sections	Loads component	Displays `"User Management"`, `"Generate Reports"`
	Handles user interactions	Button clicks	Renders appropriate sections

Table 5.16: Admin Dashboard UI Tests

Authentication Screen UI Tests			
Component	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`Authentication Screen`	Renders login form	Loads component	Displays email and password input fields
	Handles form submission	Valid email and password	Calls login function

Table 5.17: Authentication Screen UI Tests

Company Dashboard UI Tests			
Component	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`CompanyDashboard`	Renders dashboard	Loads component	Displays dashboard header, buttons, and stats sections
	Handles navigation	User clicks on action buttons (Click events)	Navigates to respective pages
	Opens messaging system	User clicks "Open Messaging" (Button clicked)	Displays messaging system

Table 5.18: Company Dashboard UI Tests

CVBuilder UI Tests			
Component	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`CVBuilder`	Renders main sections	Component is open (Open state)	Displays `"Personal Information"` and `"Skills"` sections
	Adds personal info	User inputs valid details (Name, email, phone, location)	Updates form fields accordingly
	Adds and removes skills	Add and remove skills (Skill added and removed)	Updates skill list accordingly
	Template switching	User switches CV template (Template selection)	Updates selected template
	Submits CV	User submits valid data (Filled form)	Sends API request, closes form

Table 5.19: CVBuilder UI Tests

InternshipDetails UI Tests			
Component	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`InternshipDetails`	Renders internship details	Loads component	Displays internship details, requirements, and benefits
	Handles back	User clicks "Back to Search" (Click event)	Navigates to search page

Table 5.20: InternshipDetails UI Tests

StudentDashboard UI Tests			
Component	**Test Description**	**Input/Conditions**	**Assertions/Expected Outcome**
`StudentDashboard`	Renders dashboard	Loads component	Displays header, stats, action buttons, and recent matches

Table 5.21: StudentDashboard UI Tests

6 | Installation Instructions

6.1. Setup Guide

6.1.1. Clone the Repository

Clone the repository using the following command:

```
git clone https://github.com/JhaBhatiaSharma/JhaBhatiaSharma.git
```

Navigate to the project directory:

```
cd JhaBhatiaSharma
```

6.1.2. Environment Configuration

Create a `.env` file in the root directory of the project. Add the following variables to the file:

```
PORT=5001
JWT_SECRET=your_jwt_secret_here
MONGO_URI=your_mongo_uri_here
```

6.1.3. Backend Setup

Requirements

- Internet connection
- At least 50MB free disk space for Node.js binary.
- At least 100MB free disk space for dependencies.

Setup

Navigate to the backend folder:

```
cd backend
```

Install dependencies:

```
npm install
```

Run

Start the backend server:

```
npx nodemon server.js
```

6.1.4. Frontend Setup

Requirements

– Internet connection

Setup

Navigate to the frontend folder:

```
cd frontend
```

Install dependencies:

```
npm install
```

Run

Start the frontend development server:

```
npm run dev
```

6.1.5. Easier Installation (Using Docker)

For an easier installation, you can use **Docker** to set up both the frontend and backend automatically.

1. Go to the project home directory:

   ```
   cd JhaBhatiaSharma
   ```

2. Run the following command to build and start the application:

```
docker-compose up --build
```

3. Once the containers are running, access the application at:

   `http://localhost:5173/`

6.1.6. Access the Backend

Open your browser and navigate to the URL:

`http://localhost:5001`

7 | Effort Spent

Team Member	Task	Hours Spent
Shreesh Kumar Jha	1. Introduction 2. Architectural Design 3. User Interface Design 4. Requirements Traceability 5. Implementation, Integration, and Test Plan	35
Samarth Bhatia	1. Introduction 2. Architectural Design 3. User Interface Design 4. Requirements Traceability 5. Implementation, Integration, and Test Plan	37
Satvik Sharma	1. Requirements Analysis 2. UML Diagrams 3. Backend Setup 4. Backend Implementation 5. API Refactor	32

Table 7.1: Effort spent by each member of the group.

8 References

8.1. References

– Software Engineering 2 Course Materials, A.Y. 2024-2025.

– Daniel Jackson, *Software Abstractions: Logic, Language, and Analysis*.

– Assignment RDD AY 2024-2025.pdf.

– MongoDB, Inc. (n.d.). MongoDB Manual.

– Node.js Foundation. (n.d.). Node.js Documentation.

List of Tables