PROJECT TRACKING

UIT2739 FULL STACK DEVELOPMENT

PROJECT REPORT

Submitted by

Sreekar Kashyap C (3122225002133)
Suraj S (3122225002142)

Sri Sivasubramaniya Nadar College of Engineering

(An Autonomous Institution, Affiliated to Anna University)

NOVEMBER 2025

Sri Sivasubramaniya Nadar College of Engineering
(An Autonomous Institution, Affiliated to Anna University)

BONAFIDE CERTIFICATE

Certified that this Report titled "**PROJECT TRACKING**" is the bonafide work of **Sreekar Kashyap C (3122225002133)**, **Suraj S (3122225002142)** and is submitted for project review.

Dr. A. SHAHINA

Professor and Head of Department
Department of Information Technology
SSN College of Engineering
Kalavakkam – 603 110

SHANKAR SHANTHAMOORTHY

Associate Professor
Department of Information Technology
SSN College of Engineering
Kalavakkam – 603 110

Submitted for Project Review held on .....

EXAMINER

TABLE OF CONTENTS

PROJECT OVERVIEW

The proposed system aims to provide a project tracking interface enabling users to create projects, tasks, subtasks, collaborate with other users, approve tasks using role based access efficiently and thereby enable a niche planning space for iterative development(ideally) as well as managing the manifold requirements that are common to project spaces. The major objectives of the system include: Providing role based access - wherein only the project owner can approve the tasks created thereby enabling a verification-validation cycle for the proposed tasks within an iteration which aids the brainstorming sessions in the early phases of the software development life cycle; organizing the created tasks and subtasks within a task by priority levels and enabling assignment of work to respective collaborators who maybe added or removed to the project space- enabling analysis of importance for each of the tasks at hand for a given iteration or phase as well as productive split of work. This provides a productive space to brainstorm and aids the planning and requirements phase. The system thereby provisions an efficient system for brainstorming, managing, assigning, validating as well as prioritizing the tasks considered for a given iteration thereby emphasizing on negotiating tasks among several collaborators.

PROJECT REQUIREMENTS

2.1 FUNCTIONAL REQUIREMENTS

REQ. NO.	FUNCTIONAL REQUIREMENTS
FR1	USER AUTHENTICATION (OAuth)
	1.1: A user must be able to register and log in using a third-party OAuth provider (e.g., Google).
	1.2: On the first successful login, the system shall create a new User record in the database.
	1.3: The system shall issue a secure token (e.g., JWT) to the client upon login to authenticate all future API requests.
	1.4: A logged-in user must be able to log out, which invalidates their session/token.
FR2	PROJECT AND TEAM MANAGEMENT
	2.1: A logged-in user can create a new project. The user who creates it is designated the Project Creator.
	2.2: The Project model must have a ”Project Description” field, which only the Project Creator and Admin can set or update.

REQ. NO.	FUNCTIONAL REQUIREMENTS
	2.3: The Project Creator has full CRUD (Create, Read, Update, Delete) permissions on the project itself while the Admin has project creator controls for all projects on the application.
	2.4: The Project Creator can add other registered users to the project as ”Team Members.”
	2.5: Team Members can read (view) projects they are part of but cannot update or delete the project.
FR3	TASK AND SUBTASK MANAGEMENT
	3.1: The project board must have four columns: ”Pending Approval,” ”To Do,” ”In Progress,” and ”Done.”
	3.2 (Create): Any Team Member can create a new top-level task. The task is always added to the ”Pending Approval” column.
	3.3 (Approval): Only the Project Creator can move a task from ”Pending Approval” to ”To Do.”
	3.4 (Prioritization): A Task must have a priority field (e.g., ’Low’, ’Medium’, ’High’). Only the Project Creator can set or change this priority.

REQ. NO.	FUNCTIONAL REQUIREMENTS
	3.5 (Assignment): When any user moves a task from ”To Do” to ”In Progress,” the system must prompt them to select one or more assignees (Team Members) for that task.
	3.6 (Update): Any Team Member can move tasks between ”To Do,” ”In Progress,” and ”Done.”
	3.7 (Delete): Only the Project Creator can delete a task.
	3.8 (Subtasks): A Task can have child subtasks. Users can perform full CRUD on subtasks for a parent task. These subtasks are also Task objects, follow the same workflow (e.g., start in ”Pending Approval”), and are visibly nested under their parent in the UI.

2.2 NON-FUNCTIONAL REQUIREMENTS

REQ. NO.	NON-FUNCTIONAL REQUIREMENTS
NFR1	SECURITY
	All secret keys (OAuth, JWT) must be stored securely as backend environment variables.
NFR2	AUTHORIZATION

REQ. NO.	NON-FUNCTIONAL REQUIREMENTS
	The backend API must enforce role-based access control (RBAC) using authorization middleware.
NFR3	USABILITY
	The frontend UI must use conditional rendering to hide/disable admin-only buttons for non-admin users.
NFR4	VALIDATION
	All user input forms must have client-side JavaScript validation
NFR5	ARCHITECTURE
	The application must maintain a strict separation of concerns (Frontend, Backend, Database).
NFR6	DATA INTEGRITY
	The Task model must include a parentId field (self-referencing) to manage the subtask hierarchy.

2.3 USE CASES

USE CASE 1: PROJECT INITIALIZATION

Actor User (becomes Project Creator)

Goal: Create a workspace for a new initiative.

Flow:

1. User logs in via Google OAuth.
2. User navigates to the Dashboard.
3. User fills out the ”Create Project” form (Name: ”Website Redesign”, Desc: ”Q4 Marketing Site”).
4. User clicks ”Create.

Backend Logic: The server creates a Project document and automatically adds the User’s ID to the projectCreator field and the teamMembers array.

Outcome: A new project card appears on the dashboard.

USE CASE 2: TEAM ASSEMBLY

Actor Project Creator

Goal: Add collaborators to the project.

Precondition: Project exists

Flow:

1. Creator opens the Project Detail page.
2. Creator sees the ”Add Member” form (only visible to them).
3. Creator enters the email of a colleague.
4. Creator clicks ”Add.”

Backend Logic: API finds the User by email. If found, it pushes their ID into the project.teamMembers array.

Outcome: The collaborator can now see this project on their dashboard when they log in.

USE CASE 3: TASK CREATION

Actor Team Member (e.g., The collaborator that was added)

Goal: Propose a new task to be done.

Precondition: Project exists and collaborator is part of the project.

Flow:

1. Team Member opens the project board.
2. They click ”Create Task” and enter ”Fix Navbar CSS.”
3. They submit the form.
4. Creator clicks ”Add.”

Backend Logic: The task automatically goes to the ”Pending Approval” column. The Team Member cannot put it directly into ”To Do.”

Outcome: The task is visible but sits in the ”Pending” state, waiting for approval from the project creator.

USE CASE 4: TASK APPROVAL

Actor Team Member

Goal: Approve valid work for the team.

Precondition: Project exists and collaborator is part of the project.

Flow:

1. Creator sees ”Fix Navbar CSS” in the ”Pending Approval” column.
2. Creator approves the task to the ”To Do” column.

Backend Logic: The PendingApprovalState checks isProjectCreator. Since true, it allows the move.

Outcome: The task status updates to ”To Do.” The task is now deemed official.

USE CASE 5: MARK AS COMPLETE

Actor Assignee / Team Member

Goal: Start working on a task.

Precondition: Project exists and collaborator is part of the project.

Flow:

1. User clicks done.
2. User drags the task from ”In Progress” to ”Done”.

Backend Logic: state change causes the card to move from ”In Progress” to

”Done”.

Outcome: Task is marked complete.

USE CASE 6: PRIORITY MANAGEMENT

Actor Project Creator

Goal: Escalate an urgent issue.

Precondition: Project exists and collaborator is part of the project.

Flow:

1. Creator initializes a task to manage server downtime.
2. Creator edits the task and changes Priority to ”High.”

Backend Logic: If a normal Team Member tries to edit priority, the backend returns 403 Forbidden. Only the Creator controls priority.

Outcome: Team works based on the urgency of task.

USE CASE 7: SUBTASK CREATION

Actor Any Team Member

Goal: manage a complex task.

Precondition: Project exists and collaborator is part of the project.

Flow:

1. User opens a parent task card.
2. User sees a ”Subtasks” section.
3. User breaks down the overall task into more isolated functional subtasks.

Backend Logic: These are created as standard Task objects, but their parentId field is set to the ID of overall task

Outcome: The UI renders these nested inside the parent card.

DESIGN

3.1 SYSTEM ARCHITECTURE

The Project Tracking System is built upon a decoupled, service-oriented architecture that ensures strict separation of concerns between the user interface, business logic, and data persistence layers. This architecture is designed to support scalability, maintainability, and secure role-based access control.

3.1.1 High-Level Architecture (Client-Server-Database)

The system follows the standard Model-View-Controller (MVC) pattern adapted for modern web development (MERN Stack):

- **Client (Frontend):** Built with React.js, the client serves as the presentation layer. It manages the application state locally using React Hooks and communicates with the server via RESTful API calls. It is responsible for rendering the UI components (Dashboard, Kanban Board) and capturing user interactions.
- **Server (Backend):** Built with Node.js and Express.js, the server acts as the logic layer. It handles API requests, enforces authentication (OAuth) and authorization (RBAC) middleware, executes business rules (State Pattern for tasks), and interfaces with the database.
- **Database:** Built with MongoDB, the database serves as the persistence layer. It stores data in JSON-like documents, allowing for flexible schema definitions for Users, Projects, and hierarchical Tasks.

3.1.2 Deployment Architecture

The application utilizes a distributed cloud deployment strategy to ensure reliability and accessibility:

- **Frontend Hosting (Vercel):** The React client is deployed on Vercel, utilizing its global CDN for fast static asset delivery. Client-side routing is handled via rewrite rules in `vercel.json`.
- **Backend Hosting (Render):** The Express server is deployed as a Web Service on Render. It is configured with environment variables for secure credential management and utilizes `connect-mongo` to persist user sessions across server restarts.
- **Database Hosting (MongoDB Atlas):** The data is hosted on a managed cloud cluster (Atlas), utilizing IP whitelisting to allow secure connections only from the Render backend server.

3.1.3 Directory Structure

The project follows a Monorepo structure, containing both client and server logic in a single repository to streamline version control while maintaining code separation:

- **/client:** Contains all frontend source code, assets, and React components.
- **/server:** Contains all backend logic, models, controllers, middleware, and configuration files.

3.2 DATA DESIGN

The data design utilizes a NoSQL document-oriented approach (MongoDB) to handle the hierarchical nature of tasks and subtasks efficiently. Mongoose schemas are used to enforce data integrity and structure at the application level.

3.2.1 Entity Relationships

The system comprises three core entities with specific relationships:

- **User-Project:** A One-to-Many relationship exists where a User (Creator) can own multiple Projects. A Many-to-Many relationship exists where Users can be Team Members of multiple Projects.
- **Project-Task:** A One-to-Many relationship exists where a Project contains multiple Tasks.
- **Task-Task (Recursive):** A One-to-Many recursive relationship exists where a Task can act as a parent to multiple Subtasks (children).

3.2.2 Schema Definitions

1. User Schema

Stores authentication and profile data retrieved from Google OAuth.

- `googleId` (String, Unique): The unique identifier from Google.
- `email` (String, Unique): User's email address.
- `name` (String): User's display name.
- `timestamps`: CreatedAt and UpdatedAt.
- `role`: admin and user.

2. Project Schema

Stores workspace information and team access lists.

- `name` (String): Title of the project.
- `description` (String): Detailed context of the project.
- `projectCreator` (ObjectId Ref User): Reference to the admin.
- `teamMembers` (Array of ObjectId Ref User): List of collaborators with read/write access to tasks.

3. Task Schema

Stores workflow items. It includes fields for the State Pattern and recursive hierarchy.

- `name` & `description`: Task details.
- `status` (Enum): 'Pending Approval', 'To Do', 'In Progress', 'Done'.

- `priority` (Enum): 'Low', 'Medium', 'High'.
- `project` (ObjectId Ref Project): Parent project link.
- `assignees` (Array of ObjectId Ref User): Users responsible for the task.
- `parentId` (ObjectId Ref Task, Default Null): Self-reference. If null, it is a main task; if populated, it is a subtask.

3.3 BACKEND DESIGN AND LOGIC

The backend logic serves as the central nervous system of the application, enforcing business rules, managing state transitions, and securing data access. It is designed around the RESTful MVC pattern but enhanced with behavioral design patterns to handle complex workflow constraints.

3.3.1 API Design Pattern (RESTful MVC)

The application exposes a RESTful API where resources (Projects, Tasks, Users) are manipulated using standard HTTP verbs (GET, POST, PUT, DELETE).

- **Controllers:** Act as the entry point for business logic. They receive the request, validate inputs, invoke the necessary services (like the State Context), and return formatted JSON responses.
- **Routes:** Map specific URL endpoints (e.g., `/api/tasks/:id`) to controller functions.
- **Statelessness:** The API is stateless; every request must contain all necessary authentication information (JWT), allowing the server to scale horizontally without maintaining session state in memory.

3.3.2 The State Design Pattern

To manage the complex business rules governing task movement (e.g., "Only a Creator can move a task from Pending to To Do"), the application implements

the **State Design Pattern**. This encapsulates state-specific behavior into separate classes, preventing massive conditional logic in controllers.

- **TaskContext:** The primary interface used by the controller. It holds a reference to the current state object and delegates all action requests (approve, assign, complete) to that state object.
- **TaskState (Abstract Base Class):** Defines the interface for all task actions. By default, it throws a ”Forbidden” error for all actions, enforcing a strict allow-list policy.
- **Concrete States:**
  - **PendingApprovalState:** Overrides the `approve()` method. Checks if the user is the `ProjectCreator`. If true, transitions state to 'To Do'; otherwise, throws an error.
  - **ToDoState:** Overrides `moveToInProgress()`. Enforces the rule that a task cannot proceed without an assignee.
  - **InProgressState** & **DoneState:** Define the rules for completion and regression of tasks.

3.3.3 Middleware Architecture

Security and validation are handled via a chain of middleware functions that act as ”gates” before a request reaches the controller logic.

1. Authentication Middleware (`protect`) This middleware verifies the JSON Web Token (JWT) sent in the HTTP Header. It decodes the token to identify the user and attaches the `req.user` object to the request. If the token is missing or invalid, it returns a 401 Unauthorized error immediately.

2. Authorization Middleware (RBAC) Once authenticated, the system checks permissions using context-aware middleware:

- `isTaskTeamMember`: Verifies if the authenticated user belongs to the project associated with the requested task.
- `isProjectCreator`: A stricter check ensuring the user is the owner of the project (required for deleting tasks or approving work).

3. Global Error Handling A centralized error handling middleware catches all exceptions thrown in the application (including State Pattern rejections). It formats the error into a standardized JSON structure (`message`, `stack`) and ensures the client receives the correct HTTP status code (e.g., 403 for Forbidden actions).

3.4 FRONTEND DESIGN

The client application is built using **React.js** with **Vite** and follows a hierarchical component-based architecture. This approach enhances code reusability, modularity, and maintainability.

3.4.1 Component Hierarchy (Pages vs. Components)

The structure is segregated into two primary types of components:

- **Pages (Containers):** High-level components (e.g., `DashboardPage`, `ProjectDetailPage`) responsible for fetching data, managing overall page state, and acting as containers for presentation components.
- **Components (Presentationals):** Low-level, reusable components (e.g., `TaskCard`, `AddMemberModal`) that receive data and callbacks via props and are responsible solely for rendering UI elements.

3.4.2 State Management (Hooks: `useState`, `useEffect`)

State management is handled primarily using standard React Hooks.

- **`useState`:** Used for local component state (e.g., form inputs, modal visibility).
- **`useEffect`:** Used to manage side effects, primarily data fetching from the backend API upon component mounting or when dependencies change.
- **Context API (Implicit):** Although no major global state library is used,

the application implicitly uses the context established by the **JWT** stored in `localStorage` to determine the user's role and display appropriate UI elements (**Conditional Rendering**).

3.4.3 API Service Layer

All communication with the backend is abstracted into a centralized service layer (`apiClient`). This client is configured to:

- Handle API base URL configuration (Render endpoint).
- Automatically include the JWT authentication token in the header of every request (**Bearer Token**).
- Centralize error handling for 401 (Unauthorized) and 403 (Forbidden) responses, prompting the user for re-authentication or displaying permission errors.

3.5. SECURITY ARCHITECTURE

Security is a primary concern, implemented across multiple layers including third-party identity providers, server-side session management, and strict role-based access control (RBAC) middleware.

3.5.1 Authentication Flow (Google OAuth 2.0 & Passport.js)

User identity is managed entirely by Google via OAuth 2.0.

- **Identity Provider:** Google handles user credential validation.
- **Passport.js:** Used as the authentication middleware framework in the Express backend. The `GoogleStrategy` is configured to handle the redirect flow and verify the user profile.
- **Token Generation:** Upon successful verification, the backend generates a secure JSON Web Token (JWT), which is redirected to the frontend URL

for storage and subsequent API authorization.

3.5.2 Session Management (`connect-mongo` & Cookie Security)

While the API utilizes JWTs for standard authorization, the initial Passport.js OAuth flow requires session management.

- **Session Store:** `connect-mongo` is used to store session data (the serialized user ID) directly in the MongoDB Atlas database. This avoids server memory saturation and allows for server restart persistence.
- **Cookie Security:** Session cookies are configured with security flags (`HttpOnly`, `Secure`, and `SameSite=None`) to mitigate cross-site scripting (XSS) attacks and ensure proper handling across the Vercel (client) and Render (server) origins.

3.5.3 Role-Based Access Control (RBAC) Logic

The system enforces access based on two primary roles, controlled by custom Express middleware:

- **Admin:** Granted special permissions for critical operations over the entire system landscape including:
  1. Granted project creator access for all projects created on the application.
- **Project Creator:** Granted special permissions for critical operations within the project that user creates, including:
  1. Approving tasks (Pending $\rightarrow$ To Do).
  2. Changing task priority.
  3. Deleting tasks or the entire project.
  4. Adding/Removing team members from the project.
- **Team Member:** Standard user privileges, limited to:
  1. Creating new tasks (default to Pending).
  2. Viewing task details.
  3. Moving tasks between 'To Do', 'In Progress', and 'Done'.

4. Creating subtasks.

This layered authorization ensures data integrity and prevents unauthorized modifications to the project scope and workflow.

3.6 DIAGRAMMATIC REPRESENTATIONS

To visually represent the system's scope, data flow, and behavioral logic, the following diagrams have been designed. These models adhere to standard structured analysis and design techniques.

3.6.1 Context Diagram (DFD Level 0)

The Context Diagram depicts the Project Management System as a single high-level process interacting with external entities. It establishes the system boundary.

- **System:** The Project Tracking System (Central Process).
- **External Entity 1 (User):** Provides login credentials, project details, and task updates. Receives dashboard views, alerts, and project status reports.
- **External Entity 2 (Google OAuth Provider):** Receives authentication requests. Provides authentication tokens and user profile data (email, name).

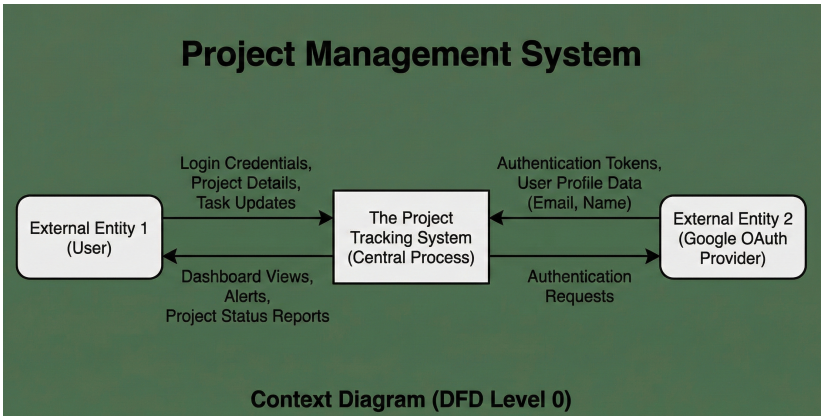

Figure 1: Context Diagram

3.6.2 Data Flow Diagram (DFD Level 1)

The Level 1 DFD expands the central process into three distinct subprocesses to illustrate the flow of information:

1. **Process 1.0 (Manage Authentication):**
    - *Input:* User Credentials via Google.
    - *Data Store:* User Database (D1).
    - *Output:* Session Token (JWT).
2. **Process 2.0 (Manage Projects):**
    - *Input:* Project Details, Team Member Emails.
    - *Data Store:* Project Database (D2).
    - *Output:* Updated Project List, Dashboard View.
3. **Process 3.0 (Manage Tasks & Workflow):**
    - *Input:* Task Creation, Status Change Requests.
    - *Data Store:* Task Database (D3).
    - *Logic:* Validates State Transitions (e.g., checks if user is Admin before approving).
    - *Output:* Kanban Board View.

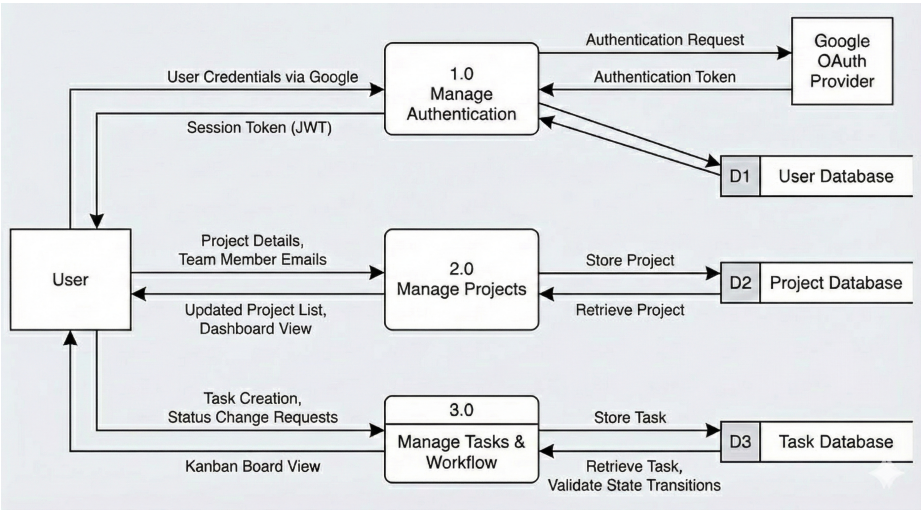

Figure 2: DFD Level 1 Diagram

3.6.3 State Transition Diagram

This diagram visualizes the State Design Pattern implemented in the backend, modeling the lifecycle of a single Task.

- **State 1: Pending Approval (Start)**
  - *Transition:* ”Approve” → Moves to ’To Do’.
  - *Guard Condition:* User must be Project Creator.
- **State 2: To Do**
  - *Transition:* ”Start Work” → Moves to ’In Progress’.
  - *Guard Condition:* Must assign a Team Member.
- **State 3: In Progress**
  - *Transition:* ”Complete” → Moves to ’Done’.
  - *Transition:* ”Block/Review” → Moves back to ’To Do’.
- **State 4: Done (End)**
  - *Transition:* ”Reopen” → Moves back to ’In Progress’.

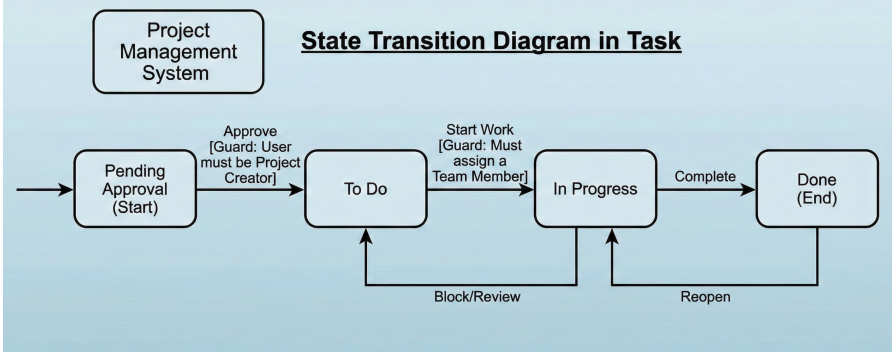

Figure 3: State Transition Diagram

TECHNICAL DETAILS

4.1 TECHNOLOGICAL STACK

The project is implemented using the MERN Stack (MongoDB, Express.js, React.js, Node.js), a robust JavaScript-based framework chosen for its scalability, JSON-native data handling, and unified language support across the full stack.

1. Frontend: React.js (Client Side)

- **Framework:** React.js (v18) via Vite.
- **Justification:** React's component-based architecture allows for the creation of reusable UI elements (e.g., `TaskCard`, `ProjectCard`). The Virtual DOM ensures efficient updates for the drag-and-drop interface of the Kanban board.
- **Routing:** `react-router-dom` handles client-side navigation without page reloads.

2. Backend: Node.js & Express.js (Server Side)

- **Runtime:** Node.js provides a non-blocking, event-driven architecture suitable for handling multiple concurrent API requests.
- **Framework:** Express.js simplifies routing and middleware integration. It manages the middleware chain for authentication (`Passport.js`), authorization (RBAC), and global error handling.

3. Database: MongoDB (Persistence)

- **Type:** NoSQL Document Store.

- **Justification:** MongoDB stores data in BSON (Binary JSON) format, which maps directly to objects in the application code. This flexibility is crucial for the recursive nature of tasks (subtasks) and the polymorphic arrays used for team membership.
- **ODM:** Mongoose is used for schema validation and modeling relationships (refs) between Users, Projects, and Tasks.

4.2 ARCHITECTURE DIAGRAM

The system architecture follows a three-tier model deployed across distributed cloud services.

1. **Presentation Layer (Vercel):** The React application serves static assets and communicates with the backend via HTTPS.
2. **Application Layer (Render):** The Node.js server processes API requests, performs logic using the State Pattern, and manages sessions.
3. **Data Layer (MongoDB Atlas):** A managed cloud cluster hosting the database, secured via IP whitelisting.

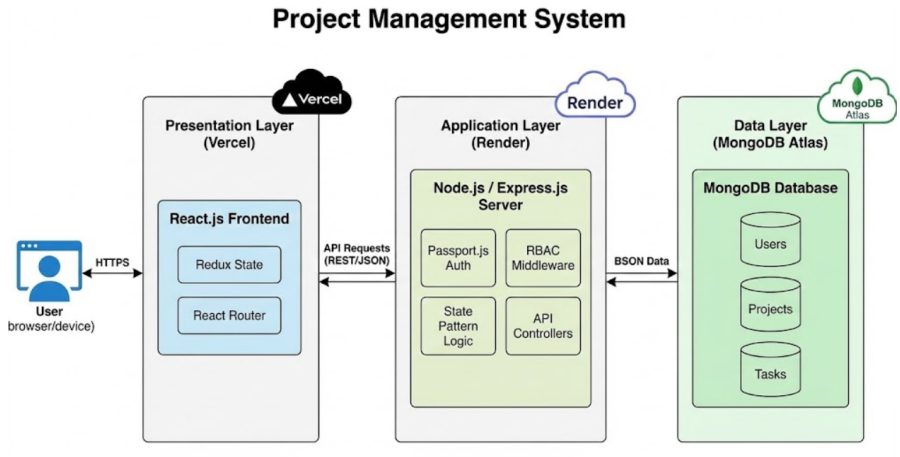

Figure 4: Architecture Diagram

4.3 DESIGN PATTERNS

The application incorporates several industry-standard software design patterns to ensure modularity, scalability, and clean code separation.

1. Model-View-Controller (MVC)

The backend structure is strictly divided:

- **Model:** Mongoose schemas (`User.js`, `Task.js`) define data structure and database rules.
- **View:** The React frontend consumes JSON data and renders the user interface.
- **Controller:** Functions in `/controllers` handle the business logic, bridging the gap between the Model and the View (API responses).

2. State Design Pattern (Behavioral)

Used to manage the complex lifecycle of a Task. Instead of using large conditional statements (`if/else`) to check permissions for every status change, the task delegates behavior to a specific State Object (e.g., `PendingApprovalState`, `ToDoState`). Each state object encapsulates the rules for that specific phase of the workflow.

3. Middleware Pattern (Structural)

Express.js relies on the Middleware pattern to process requests in a pipeline. The request passes through a chain of functions—`cors`, `json parser`, `session`, `passport`, `protect`, `rbac`—before finally reaching the controller. This allows for separation of cross-cutting concerns like security and logging from core

business logic.

4. Factory Pattern (Creational)

The application utilizes the Factory Method pattern to manage the instantiation of Task State objects. The `taskStateFactory` module encapsulates the object creation logic, dynamically determining which concrete state class (e.g., `PendingApprovalState`, `ToDoState`) to instantiate based on the task's current status string. This decouples the client code (`TaskContext`) from the specific state implementations, ensuring that adding new workflow states does not require modifying the core logic.

4.4 IMPLEMENTATION

Authentication Module

The implementation of the authentication module utilizes Google OAuth 2.0. The following screenshots demonstrate the secure login flow and the subsequent redirection to the user dashboard.

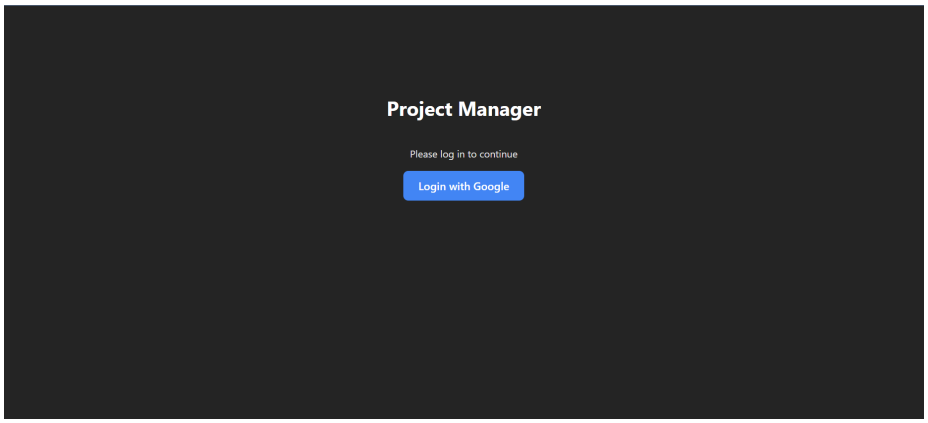

Figure 5: User Login Interface with Google OAuth Integration

Project Dashboard

Upon successful authentication, the user is presented with the Project Dashboard. This interface allows for the creation of new projects and provides a summary view of all projects where the user is either a creator or a team member.

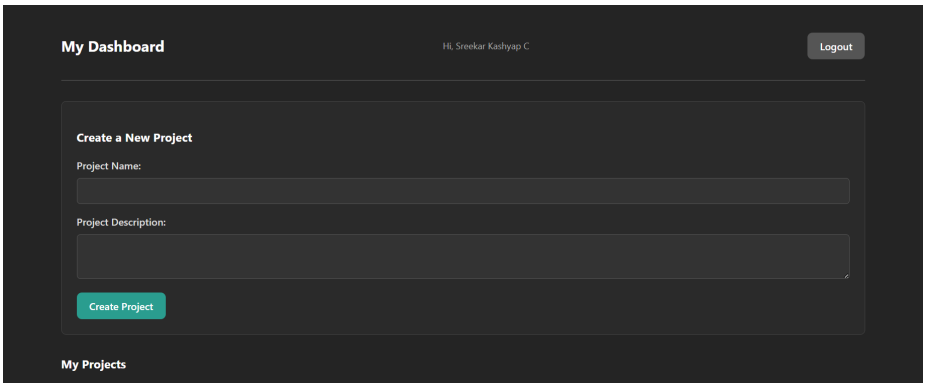

Figure 6: Project Dashboard displaying creation form

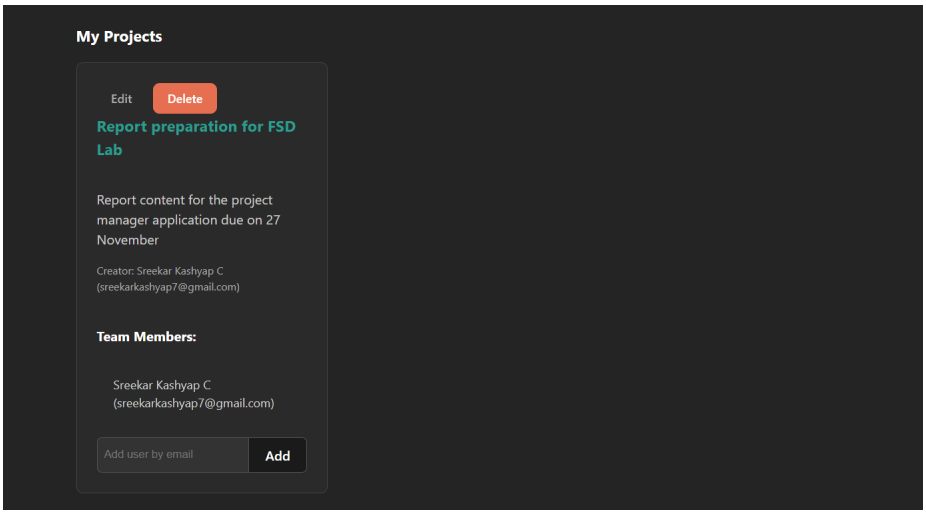

Figure 7: Project Dashboard displaying active projects

Kanban Workflow Interface

The core feature of the system is the Kanban-style task management board. This interface implements the State Design Pattern, enforcing role-based restrictions

on task movement between ’Pending Approval’, ’To Do’, ’In Progress’, and ’Done’.

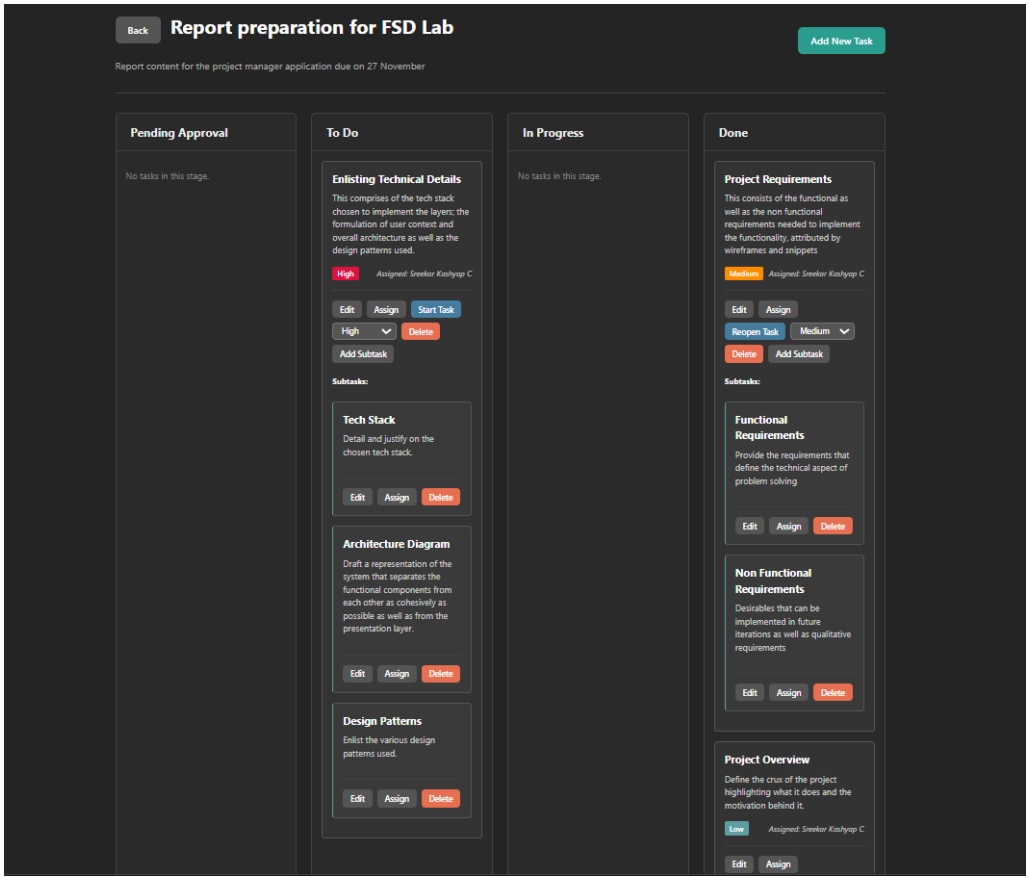

Figure 8: Kanban Board showing task workflow states

Team Member Management

This enables the addition of team members to collaborate on the project planning and brainstorming as well as efficient work distribution.

Figure 9: Team Management Interface for adding collaborators

CONCLUSION

The ”Project Tracking System” successfully demonstrates the effective application of the MERN stack in creating a scalable and secure collaborative environment. By integrating robust architectural patterns such as the State Design Pattern and Role-Based Access Control (RBAC), the system moves beyond simple task logging to provide a structured workflow engine that enforces business rules and ensures data integrity. The implementation of ”Backend-First” development ensures that the API is secure and testable, while the React-based frontend provides a responsive and intuitive user experience. The successful deployment on a distributed cloud architecture (Vercel, Render, and MongoDB Atlas) further validates the system's readiness for real-world usage scenarios. This project serves as a comprehensive proof-of-concept for modern, full-stack web application development.

FUTURE ENHANCEMENTS

To elevate the system to an industry-grade standard and compete with established market solutions, the following enhancements are proposed for future iterations:

1. **Real-Time Collaboration:** Integration of WebSockets (via Socket.io) to enable live updates on the Kanban board. This would allow team members to see task movements and edits instantly without refreshing the page.
2. **Automated Task Breakdown:** Future modules will introduce intelligent processing to analyze project descriptions and automatically suggest a breakdown of required subtasks, reducing the planning overhead for project creators.

3. **Mobile Application:** Leveraging the existing RESTful API to develop a native mobile application using React Native, ensuring accessibility for users on the go.
4. **Advanced Analytics:** Implementation of a reporting dashboard to visualize team velocity, task completion rates, and individual contributions over time.