# BitBoard: Bits of code, tasks, and chats

SRH University Heidelberg

Applied Computer Science (ACS)

**SUBJECT: Advanced Databases**

Group Members:

**Ayesha Ahmed Dhool - 11038668**
**Muhammad Bilal Hussain - 11038658**
**Revisha Shareen Vas - 11038663**
**Nayana Suresh – 11038475**
**Praharsha Sarraju – 11038716**

Submission Date: 14.06.2025

**Under the guidance of:** **Prof. Frank Hefter**

# Contents

# 1. Introduction

**BitBoard** is a real-time, collaborative Agile project management platform designed to enhance team productivity through an all-in-one interface. The system supports live code editing, Kanban-based task management, in-app chat, threaded comments, and interactive graph-based project visualizations. Built on a modern **MERN stack** architecture, BitBoard integrates **MongoDB** for task data, **Redis** for real-time communication, and **Neo4j** for visualizing relationships between tasks, team members, and project progress. These features empower users to plan, collaborate, and execute software development sprints more efficiently and transparently.

**Key Features:**

- Real-time Task and Board Creation (Kanban), drag/drop, subtasks
- Real-Time Collaborative Code Editing
- Code Execution within the Platform
- Real-Time Group Chat
- Threaded Comments
- Graph-Based Team and Task Visualizations

# 2. Organization

## 2.1 Project Plan

| | |
|---|---|
| **Approver** | Prof. Frank Hefter |
| **Contributors** | Muhammad Bilal, Ayesha Dhool, Revisha Vas, Nayana Suresh, Praharsha Sarraju |
| **Informed** | Prof. Frank Hefter |
| **Objective** | Build a real-time Agile collaboration tool with task boards, live coding, sprint planning, team graphs, and live communication. |
| **Due Date** | Jun 13, 2025 |
| **Key Outcomes** | • Collaborative code editing platform<br>• Interactive task boards & sprints<br>• Team structure & dependency graphs<br>• In-app chat, comments, and reporting tools |
| **Status** | In Progress |

## 2.2 Roles and Responsibilities

| Members | Feature | Description |
|---|---|---|
| Bilal Hussain | Real-Time Code Collaboration | • Real-time collaboration on code editing<br>• Comprehensive language support for versatile programming<br>• Code execution within the environment<br>• Unique room generation with room ID |
| Ayesha Dhool | Board & Task Management | • Create boards<br>• Create dynamic columns like To Do, In Progress, Done<br>• Create tasks & subtasks under columns<br>• Drag & Drop tasks between columns<br>• Labels, due dates, attachments<br>• Task Detail View |
| Revisha Vas | Graph-Based Project View | • Pre-filled Neo4j graph database for task-user-status relationships<br>• Cypher-based queries to retrieve task dependencies and team structure<br>• Integrated interactive graph visualization into dashboard UI |
| Nayana Suresh | Tasks Comments | Full threaded comments per task<br>• Typing Indicator<br>• Live Edit Updates<br>• Sub-comments appear in threads instantly<br>• Attachments<br>• Reactions and watching |

| | | |
|---|---|---|
| Praharsha Sarraju | Real-Time Chat System | • Direct messaging between two users<br>• Group discussion among team members in the same project |

## 2.3 Meetings, Milestones and deadlines

| Date | Topic | Outcome | Attendees |
|---|---|---|---|
| 18.05.2025 | Project Topic decision, alternatives | Contested ideas related to Use Cases, decided to work on Use Cases, initiated brainstorming session. | Revisha, Bilal, Nayana, Ayesha |
| 19.05.2025 | A new member has been added to the team | Gained a better understanding of the project concept and chose to begin initial research on the topic. | Revisha, Bilal, Nayana, Ayesha, Praharsha |
| 20.05.2025 | Confirmed Use Cases | Discussed and got approval on the use cases | Revisha, Bilal, Nayana, Ayesha, Praharsha |
| 22.05.2025 | Feature Allocation | Assigned individual modules and responsibilities to each team member | Revisha, Bilal, Nayana, Ayesha, Praharsha |
| 26.05.2025 | Database Architecture Discussion | Decided on using MongoDB, Redis, and Neo4j; finalized DB responsibilities | Revisha, Bilal, Nayana, Ayesha, Praharsha |

| | | | |
|---|---|---|---|
| 28.05.2025 | Mid-Development Sync-up | Reviewed individual progress on feature development. | Revisha, Bilal, Nayana, Ayesha, Praharsha |
| 30.05.2025 | Mid-Development Sync-up | Identified and discussed roadblocks in integrating integration. | Revisha, Bilal, Nayana, Ayesha, Praharsha |
| 02.06.2025 | Mid-Development Sync-up | Reviewed implementation progress; resolved technical dependencies. | Revisha, Bilal, Nayana, Ayesha, Praharsha |
| 04.06.2025 | Mid-Development Sync-up | Finalized integration touchpoints; discussed testing structure. | Revisha, Bilal, Nayana, Ayesha, Praharsha |
| 06.06.2025 | Resolved GitHub merge conflicts | Resolved GitHub merge conflicts across features. | Revisha, Bilal, Nayana, Ayesha, Praharsha |
| 10.06.2025 | Pre-final Review | Conducted code and functionality review of all features. | Revisha, Bilal, Nayana, Ayesha, Praharsha |
| 12.06.2025 | Final Documentation Session | Finalized documentation and formatted report for submission. | Revisha, Bilal, Nayana, Ayesha, Praharsha |

| Milestone | Owner | Deadline | Status |
|---|---|---|---|
| Finalize feature assignments | Team | May 18, 2025 | Completed |
| Document feature descriptions | All Members | May 21, 2025 | Completed |
| Implement Board & Task Mgmt | Ayesha Dhool | June 5, 2025 | Completed |
| Implement Real-Time Code Collab | Bilal Hussain | June 8, 2025 | Completed |
| Implement Notifications & Graphs | Revisha Vas | June 5, 2025 | Completed |
| Implement Task Comments System | Nayana Suresh | June 5, 2025 | Completed |
| Implement Chat System | Praharsha Sarraju | June 7, 2025 | Completed |
| Integration & Testing | Team | June 9, 2025 | Completed |
| Final Review & Documentation | Team | June 10, 2025 | Completed |

## 2.4 Tools Used

| Areas | Tools |
|---|---|
| Development | VS Code, Node.js, React |
| Database | MongoDB, Redis, Neo4j |
| Testing | Postman |
| Source Control | GitHub |
| Documentation | MS Word |
| Communication | Microsoft Teams, WhatsApp |

## 3. User Stories

Each user story is written from the perspective of a specific user type, describing a feature's purpose and expected benefit.

| **User Story 1** | |
|---|---|
| **As a** | team member |
| **I want** | to create, update, and move tasks between columns (To Do, Doing, Done, etc) |
| **To** | manage project tasks in an organized, visual workflow |
| **Use Cases:** | |
| Add new tasks to a Kanban board, drag and drop between columns, assign users, set deadlines | |

| **User Story 2** | |
|---|---|
| **As a** | developer |
| **I want** | to collaborate on code in real-time using a shared editor |
| **To** | enable seamless pair programming and code review |
| **Use Cases:** | |
| Live code editing, syntax highlighting, code execution, room-based collaboration | |

| **User Story 3** | |
|---|---|
| **As a** | Project Member |
| **I want** | to view an interactive graph of all tasks, team members, and statuses |
| **To** | understand task distribution, ownership, and dependencies across the project |
| **Use Cases:** | |
| Visualize task statuses and assignments, identify task dependencies and blockers, Explore overall project flow through an interactive graph | |

| **User Story 4** | |
|---|---|
| **As a** | team member |
| **I want** | to chat with groups about a task in real-time |
| **To** | facilitate quick discussions and decision-making |
| **Use Cases:** | |
| Group and direct messaging, reactions, real-time message delivery and presence tracking | |

| **User Story 5** | |
|---|---|
| **As a** | user |
| **I want** | to leave comments on tasks or code snippets |
| **To** | clarify or discuss specific issues with my team |

| **Use Cases:** |
| --- |
| Threaded comments, sub-comments, typing indicators, attachments, reactions |

# 4. Details

## 4.1 Board and Task Management – AYESHA

### 4.1.1 User Story

- As a ***team member***, I want to create, update, and move tasks between columns (To Do, Doing, Done, etc).
- As a ***manager***, I want to assign tasks to the teammates so that work is distributed effectively.

### 4.1.2 Identified Use Case UML

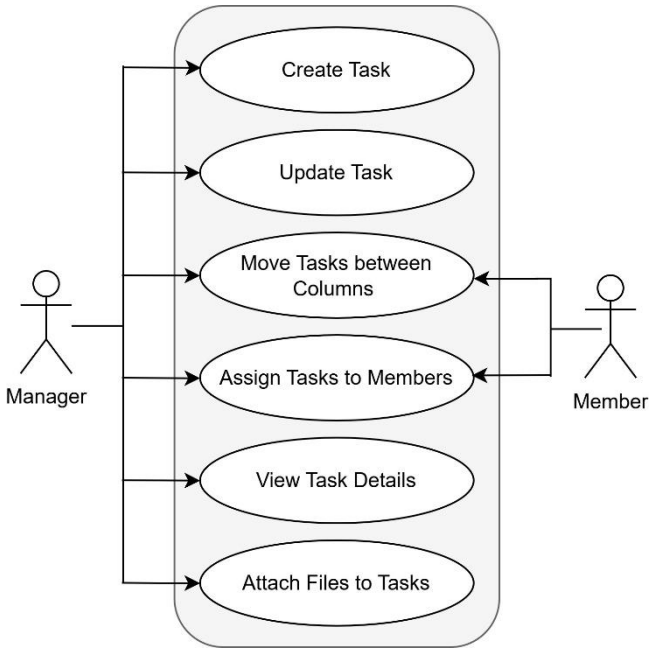


Fig1: Kanban UML Diagram

### 4.1.3 Actors

- **Project Managers**

Project Managers have elevated permissions and administrative capabilities within the system. They can:

- Create new tasks by providing title, description, deadlines, assignees, and file attachments.

- Edit all fields of any task across the project, regardless of assignee.
- Assign or reassign tasks to any team member.
- Delete tasks that are no longer relevant or completed.

Their role focuses on project planning, workload distribution, and ensuring task visibility across the team.

- **Team Members**

Team Members are regular users of the platform who are responsible for working on assigned tasks. They can:

- View task details relevant to them.
- Move tasks between Kanban columns (e.g., from “To Do” to “In Progress”).
- Access their own task list through the “My Tasks” view.

Their role ensures active participation in project execution while maintaining autonomy over their own assignments.

### 4.1.4 Description of Task with All Interactions (Actor ↔ Database)

- **Actor Interaction → Event Propagation**
- When a Project Manager or Team Member performs an action (create, move, update, delete), this is not just a direct DB update — it triggers a system-wide event.
- The backend logic is built to ensure that every significant task change emits an event via Redis Pub/Sub, decoupling the user action from the UI update.
- **From API to Database + Broadcast**

Each interaction follows this event pipeline:

1. User Action (via UI)
2. → API Call (Express route)
3. → Validation + DB Change (MongoDB)
4. → Redis Publish (e.g., taskUpdated)

5. → WebSocket Broadcast to all clients
6. → Real-Time UI Change (on every connected browser)

This makes the platform highly reactive and collaborative.

- **Synchronized Experience Across Clients**

- A task update made by a Manager is reflected instantly on a Team Member's board.
- Even deletions propagate instantly to avoid UI inconsistencies.

### 4.1.5 Data Flow (Step-by-Step Detailed for this Use Case)

- **Create Task Flow:**

1. Manager clicks "Add Task" button in the UI.
2. The frontend displays a modal with input fields: title, description, assignee, deadline, status, and file attachment.
3. Upon submission, React frontend sends a POST request to Node.js backend (/api/tasks).
4. The backend validates input, and if an attachment is included, it is processed using multer.memoryStorage().
    a. Files are stored in memory and encoded in Base64, then saved directly into MongoDB.
5. A new task document is created in the tasks collection in MongoDB, including metadata and optional file data.
6. The backend publishes a taskCreated event to Redis Pub/Sub.
7. All subscribed clients receive the event through WebSocket and update their UI in real-time.

- **Update Task Flow:**

1. Manager updates a task's title, description, assignee, deadline, or status.
2. The frontend sends a PATCH request to (/api/tasks/:id) with only the modified fields.

3. The backend removes any undefined fields and updates the corresponding task in MongoDB.
4. A taskUpdated event is published to Redis Pub/Sub.
5. Subscribed clients update the task in their UI in real-time.

- **Move Tasks Between Columns Flow:**

1. A Member or Manager drags a task from one Kanban column to another (e.g., from "To Do" to "In Progress").
2. The frontend sends a PATCH request to the backend to update the status field of the task.
3. The backend updates the task in MongoDB and emits a taskUpdated event via Redis.
4. Clients subscribed to the task updates render the new column position in real-time.

- **Assign Tasks to Members Flow:**

1. During task creation or update, the Manager selects a user to assign the task.
2. The frontend sends the assignee's user ID via the request body to the backend.
3. The assignee field is stored or updated in the MongoDB document.
4. A taskUpdated event is emitted via Redis, triggering real-time UI updates.

- **View Task Details Flow:**

1. A Member or Manager selects a task card.
2. The frontend sends a GET request to /api/tasks/:id.
3. The backend retrieves the full task document from MongoDB, including populated assignee details.
4. The frontend renders a detailed modal with task data, including attachments.

- **Attach Files to Tasks Flow:**

1. While creating or updating a task, the Manager can upload a file.
2. The file is handled by `multer` in memory and converted to a Base64 string.

3. File content and its MIME type are saved under the `attachment` field in the task document in MongoDB.
4. After saving, the task data is broadcast via Redis to ensure the file attachment is shown in real-time across clients.

- **Delete Task Flow:**

1. A **Manager** selects a task and clicks “Delete.”
2. The **frontend** sends a DELETE request to /api/tasks/:id.
3. The **backend** locates and deletes the task document in **MongoDB**.
4. A taskDeleted event is published to **Redis**, allowing clients to immediately remove the task from their view.

### 4.1.6 Sequence Diagrams

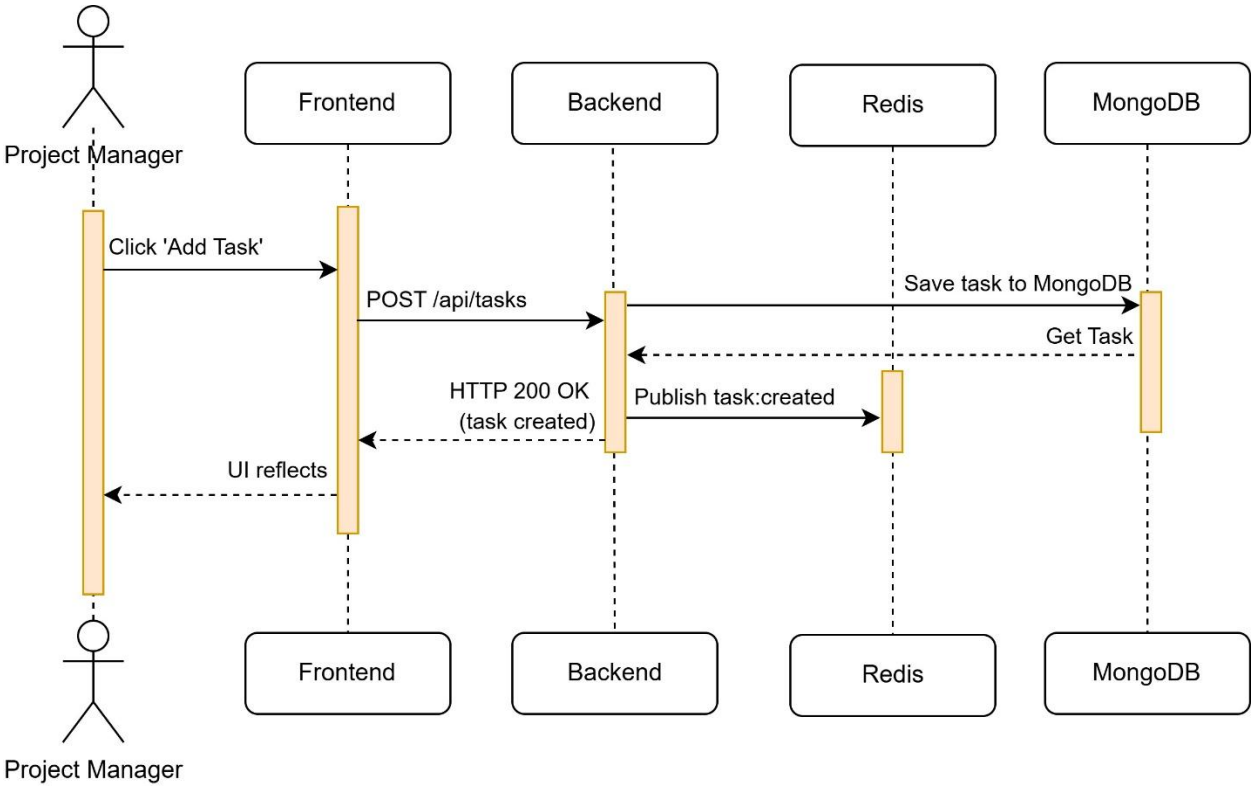


Fig2: Sequence diagram of Task Creation

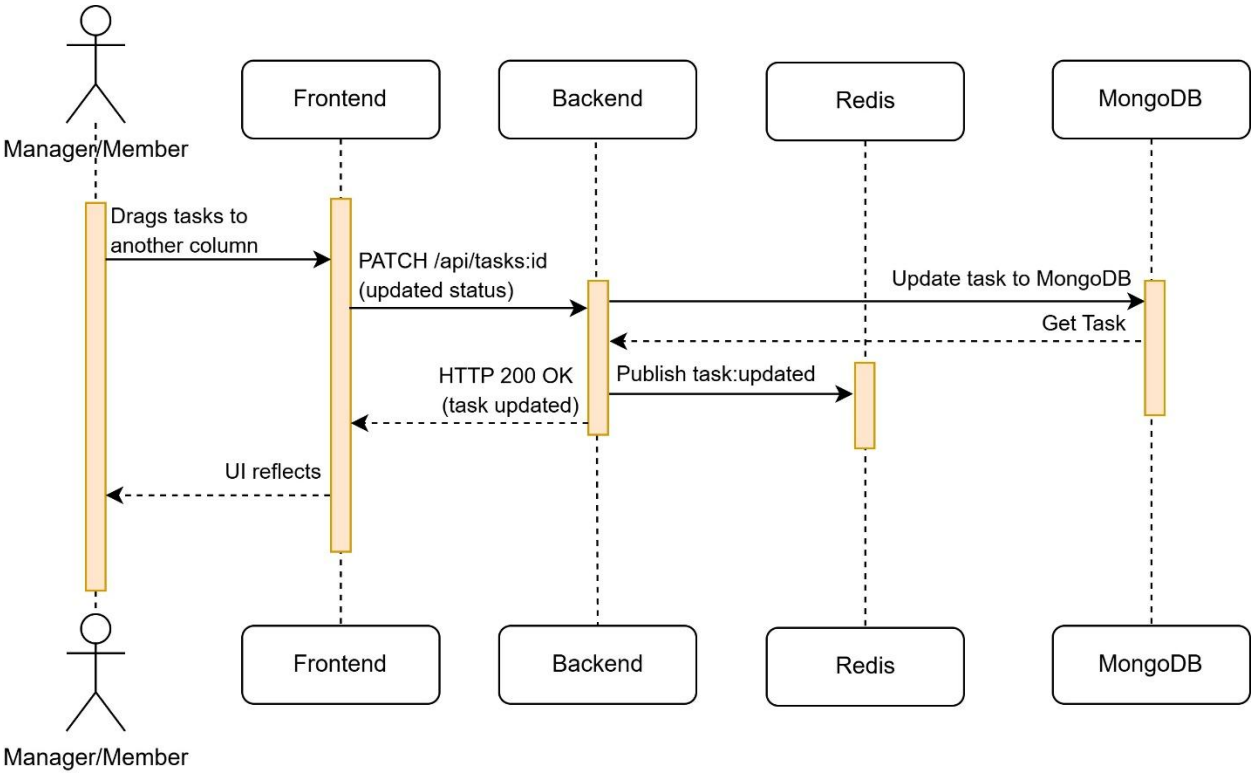


Fig3: Sequence diagram of Task Update

### 4.1.7 Databases

**MongoDB:**

- **Use:** Stores all structured, persistent task data: tasks, subtasks, attachments, due dates, statuses, etc.
- **Reason:** Flexible document model supports nested structures like subtasks, attachments.
- **ERD Diagram:**

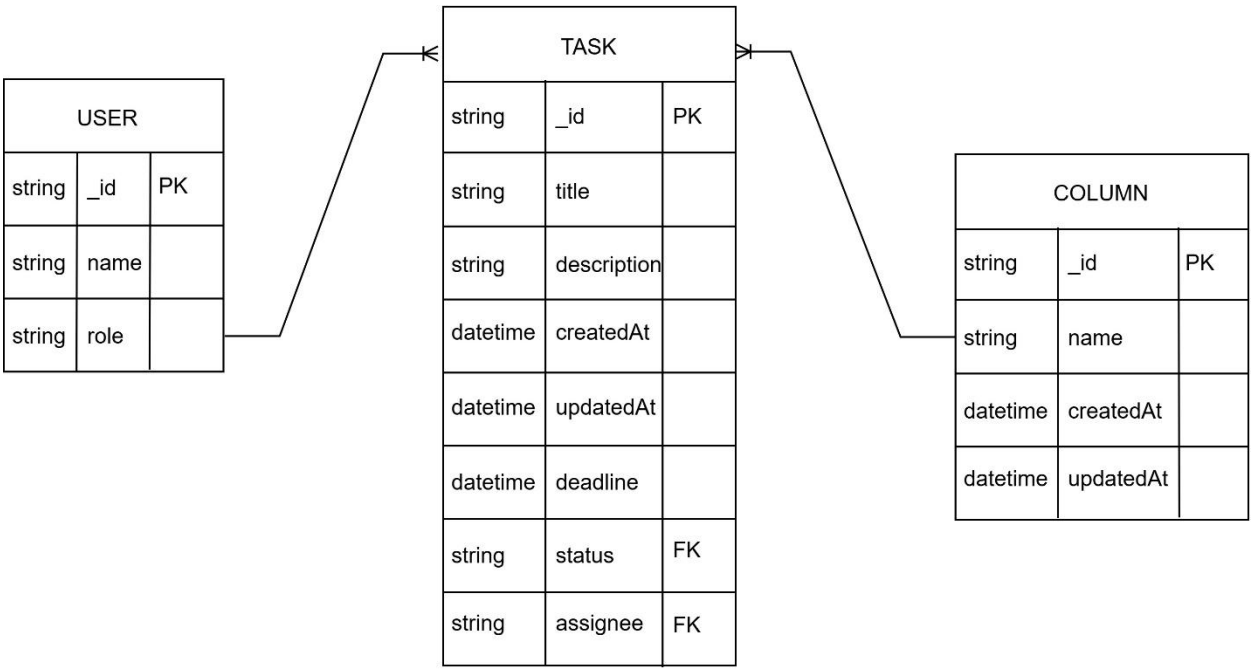


Fig4: ER Diagram

## Redis:

- **Use:** Used for real-time updates via Pub/Sub mechanism.
- **Reason:**
  - Extremely fast in-memory data store.
  - Pub/Sub model allows you to broadcast changes (e.g., new task, status change) without reloading the page.
  - Ensures quick, event-based UI updates across all connected clients.

## 4.2 Real-Time Code Collaboration – BILAL

### 4.2.1 User Story

- As a **developer**, I want to create a shared coding room, invite others, and edit code together in real-time. and I want to run code and see instant output with my team.
- As a **Collaborator**, I want to join a room with a link/ID and edit code live with others. and I want to see everyone's changes instantly.

### 4.2.2 Use Cases

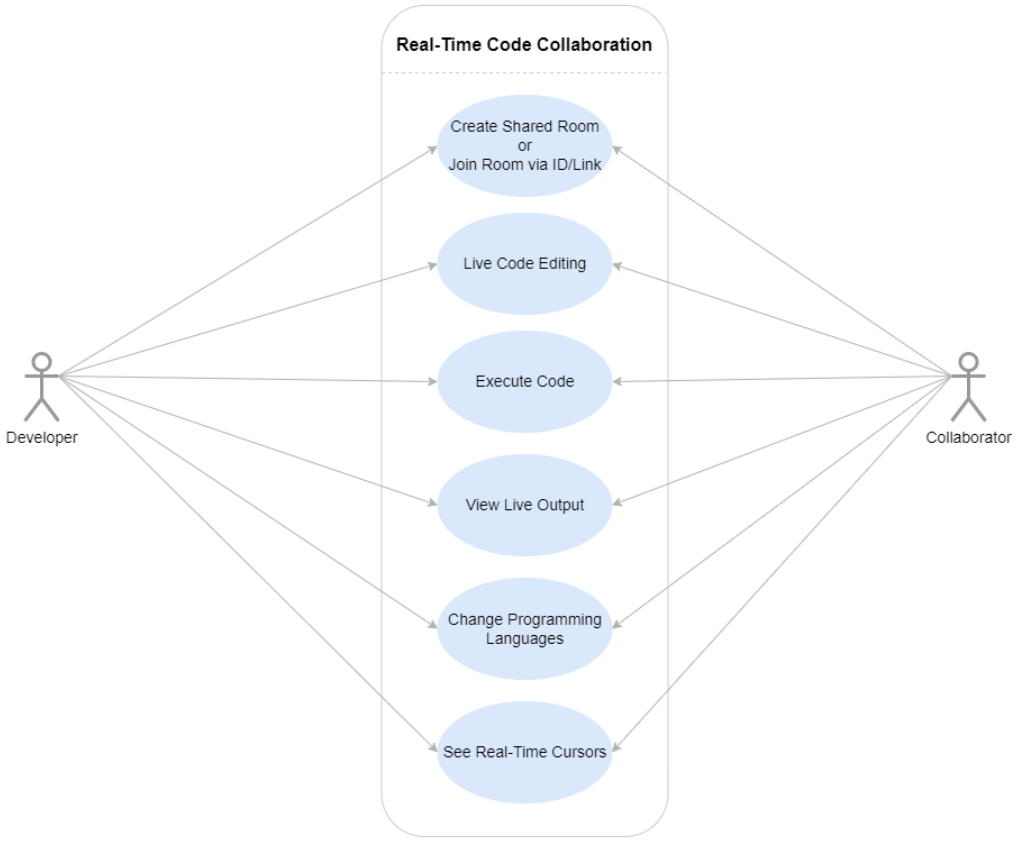


Fig3: Code Collaboration UML Diagram

### 4.2.3 Actors

- **Developer:** Creates a shared coding room with a unique ID/link, invites others via room ID/link, edits code in real-time with collaborators, and executes code while sharing live output with the team.

- **Collaborator:** Joins a room via ID/link, co-edits code in real-time with other participants, and executes code to test changes during the collaborative session.

### 4.2.4 Description of Task with All Interactions (Actor ↔ Database)

- **Database Role in Real-Time Synchronization**

Redis, an in-memory data store, acts as the temporary session state holder and event broker. Here's how it supports the interactions:

- **Session Storage:** When a user joins a room, their user ID and socket ID are stored in Redis against the room ID.
- **Code State:** The most recent version of the code in each room is cached in Redis, ensuring that reconnecting users receive the latest version instantly.
- **Event Dispatching:** Redis Pub/Sub channels are used to publish code changes and broadcast them to all WebSocket clients in the same room.
- **User Presence:** Redis tracks who is present in which room and handles joining/leaving logic without writing to any persistent database.

- **Actor-Driven Events and System Behavior**

Each action by a Developer or Collaborator (e.g., typing a line of code) becomes a system event. These events:

- Update the shared code state in Redis,
- Trigger a Pub/Sub notification,
- Are picked up by the backend's WebSocket layer, and
- Are immediately sent to all other participants in the room.

Redis allows these interactions to be extremely fast, with no I/O delays, enabling sub-second response times — critical for live collaboration.

### 4.2.5 Data Flow (Step-by-Step Detailed for this Use Case)

- **Join Room:**

1. Developer clicks "Join Room" or creates a new one.

2. Frontend sends request to backend (/api/rooms/join).
3. Backend checks/creates room, assigns room ID, and adds the user to the room session.
4. Room metadata is stored temporarily in Redis for fast access.

- **Real-Time Code Editing:**

1. Each keystroke triggers a WebSocket event (codeUpdate).
2. Server receives this, updates shared memory/Redis, and broadcasts to all room participants.
3. All users' editors reflect changes live (via WebSocket sync).

- **Code Execution:**

1. Developer hits "Run".
2. Code is sent to backend (/api/code/execute) with selected language.
3. Backend routes it to a sandboxed code runner (like Docker).
4. Output (stdout/stderr) is sent back via WebSocket and shown in terminal UI.

- **Leave Room:**

1. On exit, client emits a leaveRoom signal.
2. Server removes user from Redis room list and notifies others.

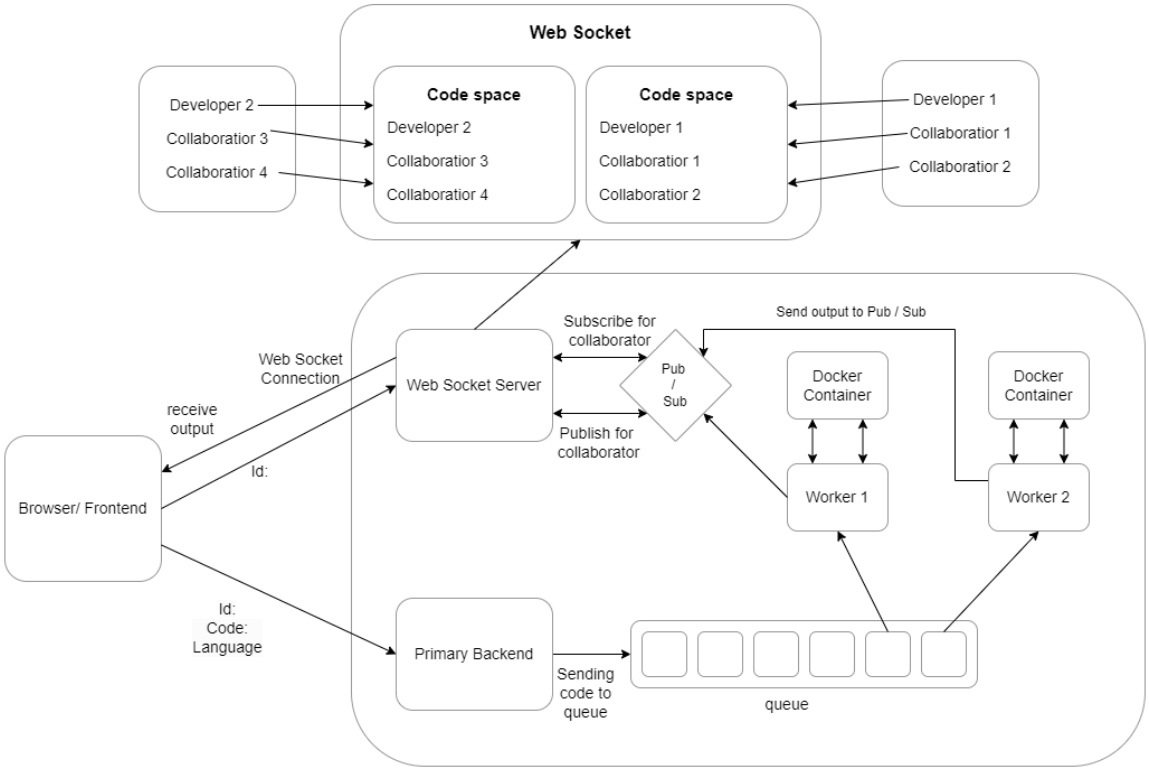


Fig4: Data Flow Diagram

## 4.2.6 Activity Diagram

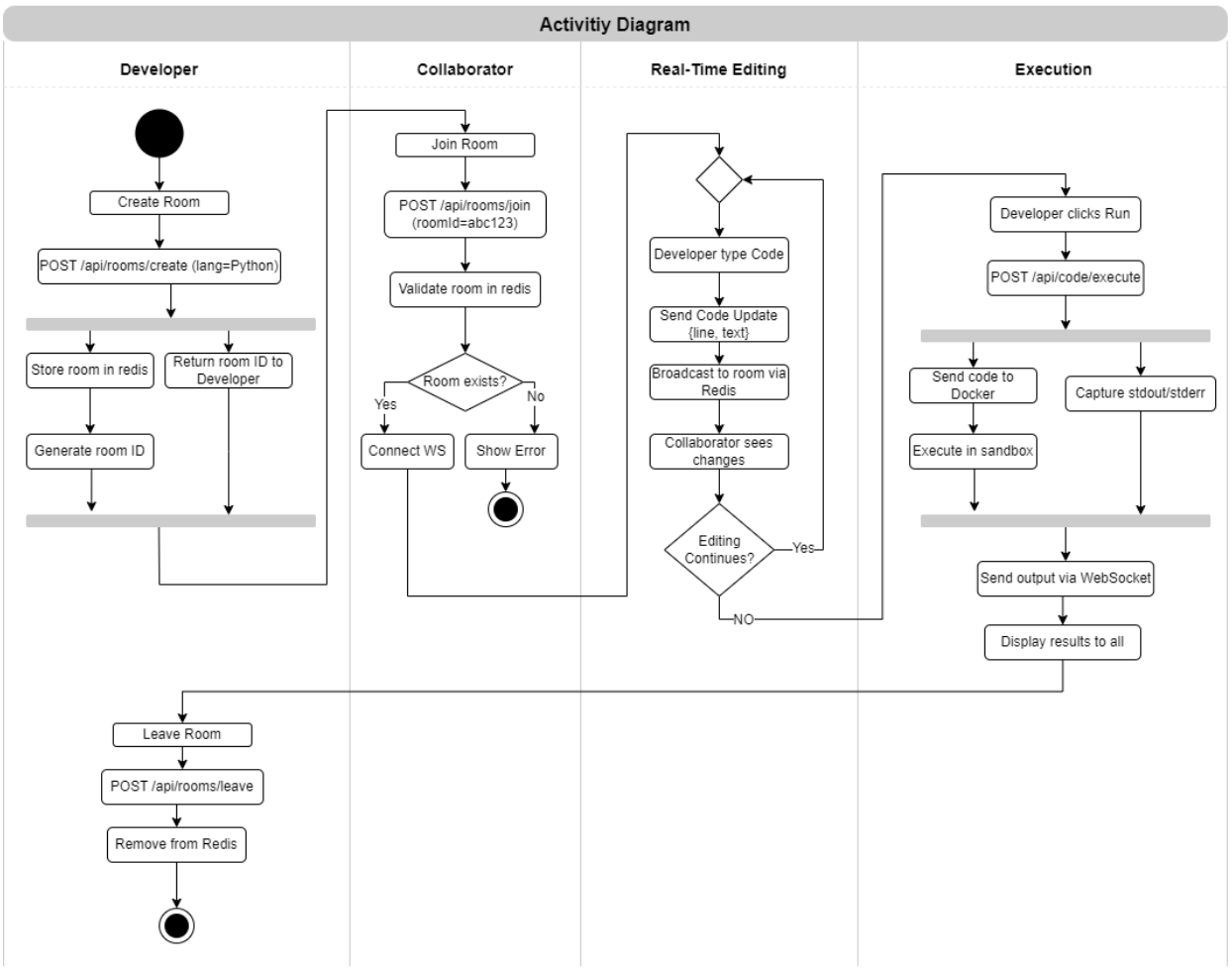


Fig5: Activity Diagram

### 4.2.7 Sequence Diagram

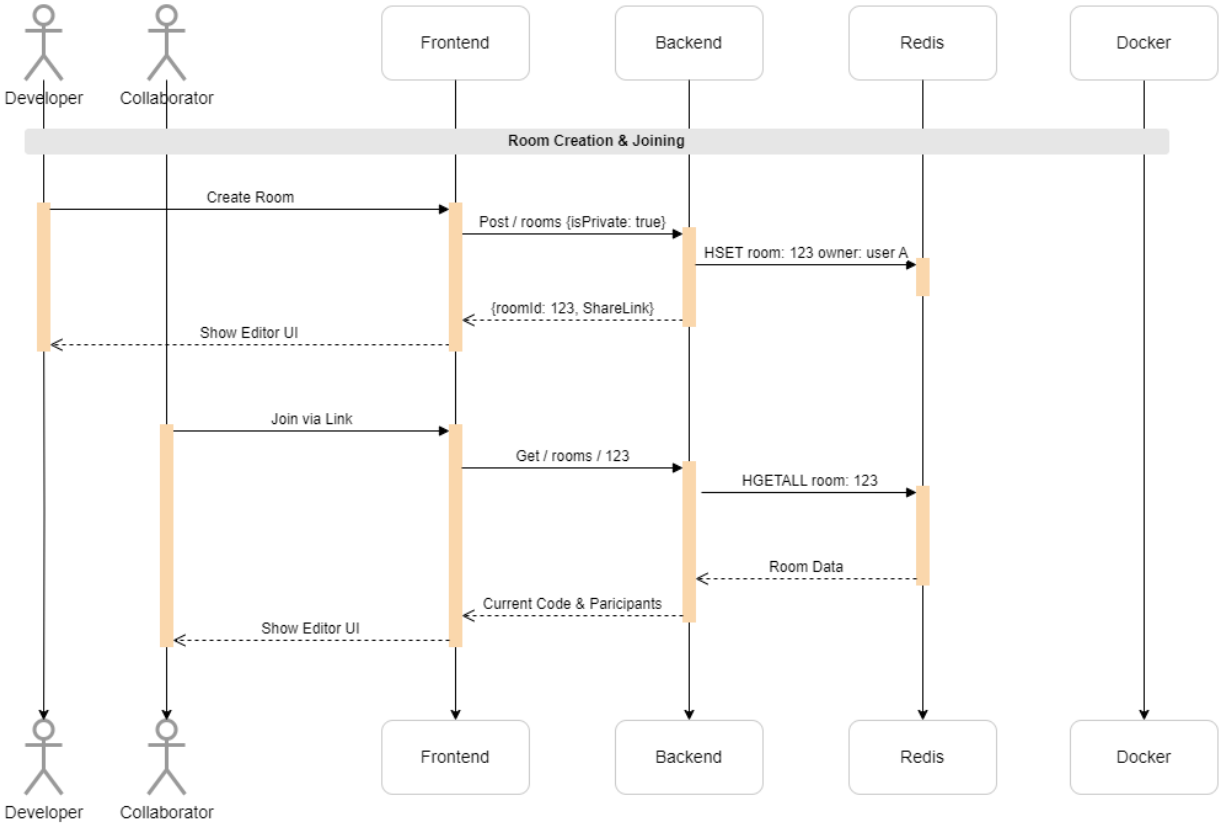


Fig6: Sequence diagram of room creation

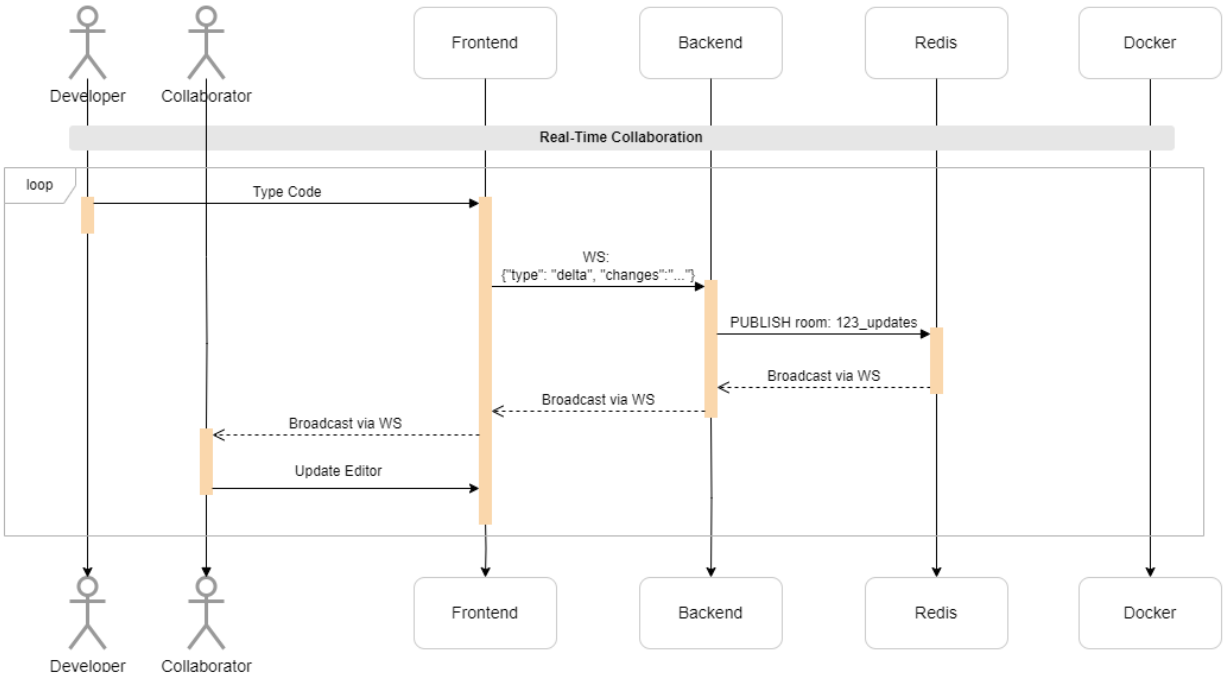


Fig7: Sequence diagram of Real time code collaboration

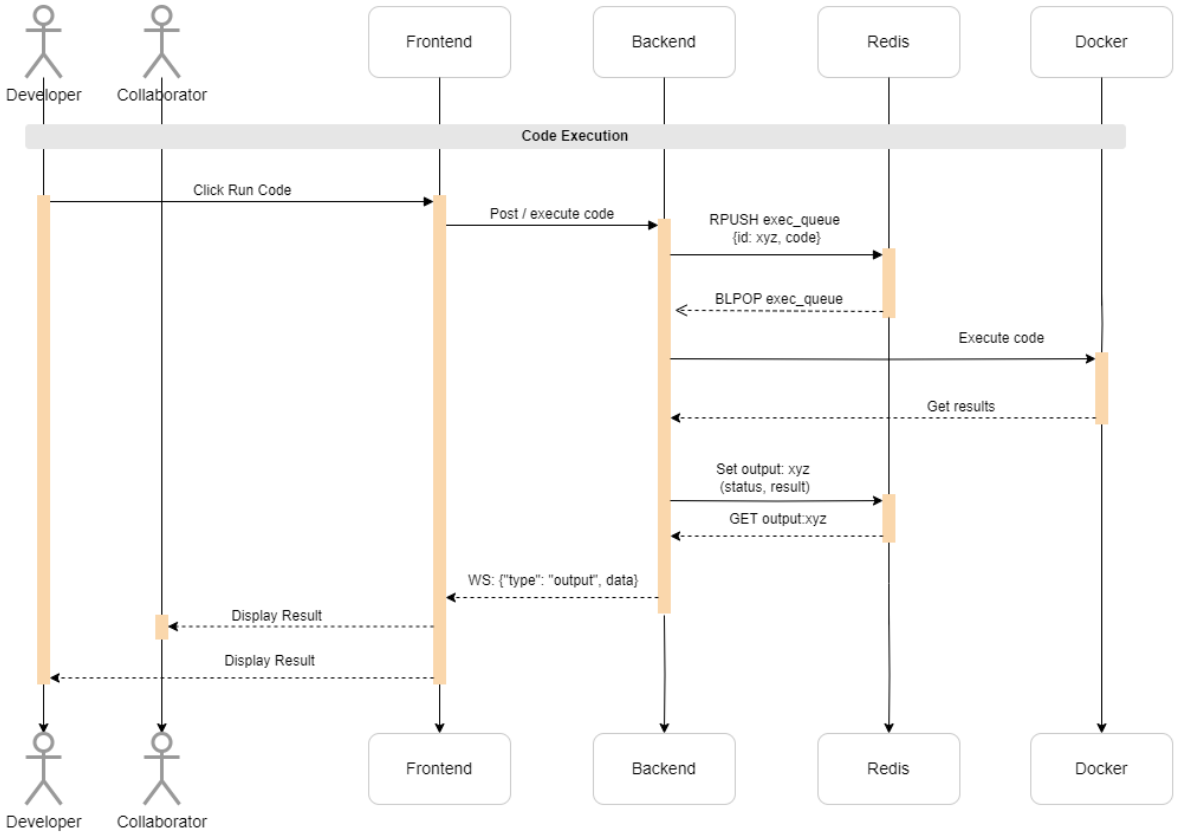


Fig8: Sequence diagram of code execution

### 4.2.8 Databases

**Redis:**

- **Use:** Manage real-time user sessions, room lists, and broadcast socket messages.
- **Why:** Redis is fast, supports Pub/Sub, and can handle in-memory room state efficiently.

## 4.3 Graph-Based Project View – REVISHA

### 4.3.1 User Story

- As a ***project member***, I want to view a graph of tasks, users, and statuses so that I can understand task assignments, progress stages, and dependencies at a glance.

### 4.3.2 Use Cases

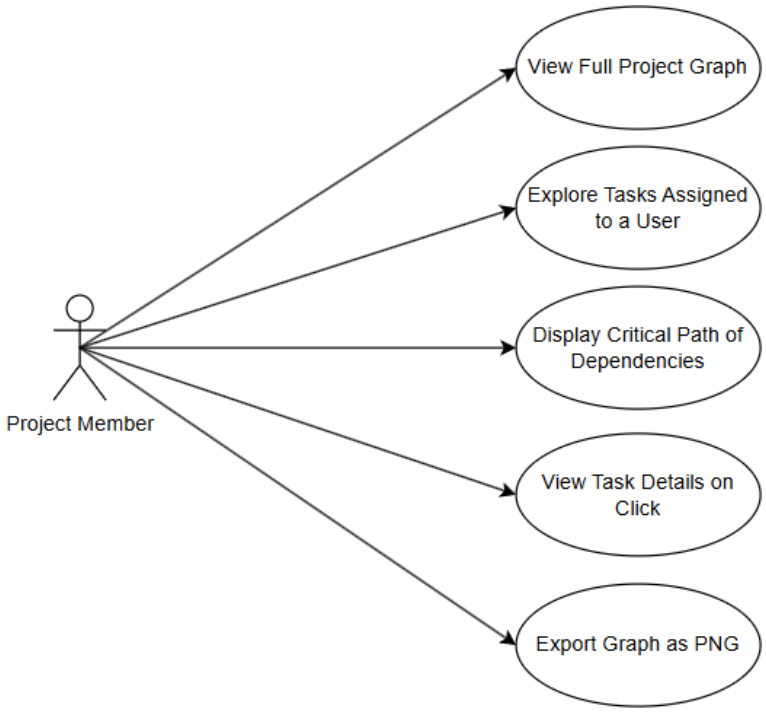


Fig 9: Project-Graph use case diagram

### 4.3.3 Actors

**Project Manager:** Uses the visualization dashboard to explore the overall project structure. They interact with filters to view tasks based on status, assigned user, or task dependencies. The graph helps them assess the team's workload and understand the flow of tasks across the project.

### 4.3.4 Description of Task with Interaction (Actor ↔ Database)

**Task Title:** Graph-Based Project Visualization using Neo4j

**Planned Goal:** This feature enables a clear, interactive visualization of the project's internal structure using a pre-filled Neo4j graph database. Instead of viewing data in static tables or lists, project members can:

- Visually inspect which user is assigned to which task.
- Understand task statuses such as "To Do," "In Progress," or "Done."
- Explore dependencies between tasks to identify blockers.
- Assess workload distribution across the team.
- Discover patterns like bottlenecks, isolated tasks, or overloaded users.

**Actor Interaction → Graph Exploration via Dashboard**

When a **Project Member** opens the Visualization Dashboard, they are greeted with a default general graph and related KPIs. From there, they can:

- Filter nodes by type (Task, User, Status).
- Click nodes to inspect task details (e.g., title, due date, assignee).
- Export the graph as a PNG.
- Switch between views (e.g., user-specific or critical path).
  This interactive mode helps users perform sprint planning, track milestones, and identify project risks more intuitively.

**Sample node and relationship structure in Neo4J**

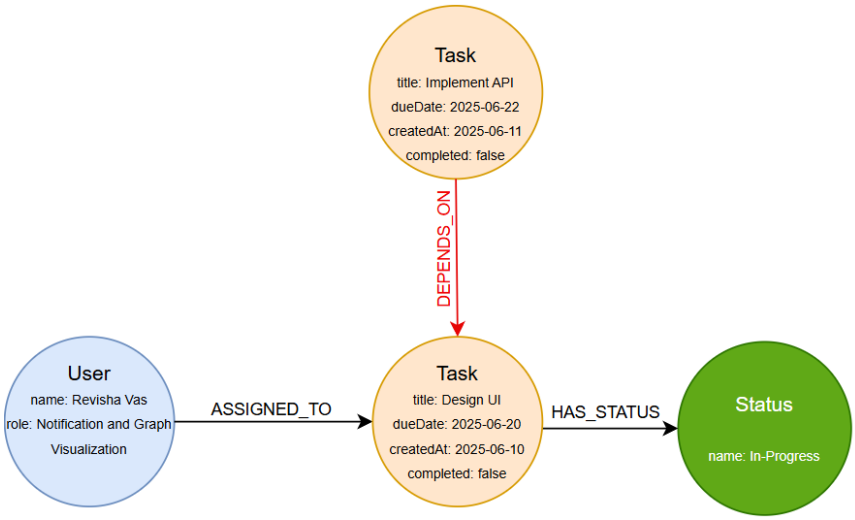


Fig 10: Graph view of Neo4J

In the Neo4j graph shown above, the task node "Design UI" is linked to the user "Revisha" via the ASSIGNED_TO relationship and is associated with the status "In Progress" through HAS_STATUS. Another task, "Implement API", is connected to "Design UI" by a DEPENDS_ON relationship, indicating that it cannot proceed until "Design UI" is completed. This graph structure models real-world task dependencies, where relationships between task nodes define execution order and ownership.

### 4.3.5 Data Flow (Step-by-Step detailed for this use case)

**Step 1: Load General Graph**

- User opens the dashboard → /api/status/graph
- Backend queries Neo4j and returns all nodes and relationships.
- Frontend renders the graph with interactive features.

**Step 2: Filter by User**

- User selects a specific user → /graph/user/:name
- Neo4j filters tasks assigned to that user and connected DEPENDS_ON tasks
- Renders focused workload + dependency view

**Step 3: Show Critical Path**

- User activates critical path mode → /graph/critical-path
- Backend computes the longest DEPENDS_ON path
- Frontend renders key blocker chain with interactive edges

**Graph view and Cypher Logic**

**1. General Graph View**

- **API: /api/status/graph**
- **Cypher Logic**:

  ```
  MATCH (n)-[r]->(m) RETURN n, r, m
  ```

- **Purpose**: Fetches the complete graph including all task nodes, users, statuses, and their relationships.
- **Frontend Actions**:
  - Filter by node type (e.g., Task, User, Status)
  - Export full graph as PNG
  - Click nodes to inspect details (title, assignee, due date)

**2. User-Specific Graph View**

- **API**: `/api/status/graph/user/:name`
- **Cypher Logic**:

  ```
  MATCH (u:User {name: $username})-[:ASSIGNED_TO]->(t:Task)
  OPTIONAL MATCH (t)-[r:DEPENDS_ON]->(d:Task)
  RETURN t AS n, r, d AS m
  ```

- **Purpose**: Shows only tasks assigned to a specific user, along with any blocking dependencies.
- **Frontend Actions**:
  - Dropdown to select user
  - Graph filters to toggle extra dependencies
  - Focused visualization for workload assessment

### 2.3 Critical Path Graph View

- **API**: /api/status/graph/critical-path
- **Cypher Logic:**
  MATCH path = (a:Task)-[:DEPENDS_ON*]->(b:Task)
  WITH path, length(path) AS len
  ORDER BY len DESC
  LIMIT 1
  UNWIND relationships(path) AS r
  WITH startNode(r) AS n, endNode(r) AS m, r
  RETURN n, r, m
- **Purpose**: Displays the longest chain of dependent tasks, useful for identifying the project's bottleneck or "critical path."
- **Frontend Actions:**
  - Toggle between graph views
  - Highlights blockers and enables focused sprint prioritization

### 2.4 KPIs and Analytical Insights

The dashboard is enriched with real-time analytics using Cypher queries. These appear above or below the graph to enhance decision-making:

| Metric | Description | API Endpoint | Cypher Query (Summary) |
|---|---|---|---|
| **Total Tasks** | Count of all task nodes | `/totalTasks` | `MATCH (t:Task) RETURN count(t)` |
| **Completed Tasks** | Tasks linked to "Done" status | `/completedTasks` | `MATCH (t)-[:HAS_STATUS]->(s:Status {name: "Done"}) RETURN count(t)` |
| **Overdue Tasks** | Due date passed & not marked "Done" | `/overdueTasks` | `MATCH (t)-[:HAS_STATUS]->(s) WHERE t.dueDate < date() AND s.name <> "Done"` |

| | | | |
|---|---|---|---|
| **Next Milestone** | Earliest task deadline (excluding "Done") | `/nextMilestone` | `MATCH ... ORDER BY dueDate ASC LIMIT 1` |
| **Current Milestone** | Earliest task “In Progress” | `/currentMilestone` | `MATCH ... WHERE s.name = "In Progress" ORDER BY dueDate ASC LIMIT 1` |
| **Top Blocked Tasks** | Tasks blocked by most others | `/top-blocked-tasks` | `MATCH (t)<-[:DEPENDS_ON]-(d) RETURN t.title, count(d)` |
| **Task Load per User** | Number of tasks assigned per user | `/tasks-per-user` | `MATCH (u:User)-[:ASSIGNED_TO]->(t:Task) RETURN u.name, count(t)` |
| **Overdue by Status** | Overdue task count grouped by status | `/overdue-by-status` | `MATCH (t)-[:HAS_STATUS]->(s) WHERE due < date() RETURN s.name, count(t)` |
| **Task Creation Timeline** | Tasks created per day (for line chart) | `/tasks-by-date` | `RETURN apoc.date.format(t.createdAt, "ms", "yyyy-MM-dd") AS createdDate, count(*)` |

### 2.5 Technologies Used

- **Neo4j** – Graph DB for task/user/status modeling
- **Cypher** – Declarative graph query language for subgraph retrieval and analysis
- **Express.js** – Node backend API layer
- **React** – Frontend component structure
- **react-force-graph** – D3-based graph visualization
- **Recharts** – For rendering KPIs and analytical charts
- **Tailwind CSS** – UI styling
- **Docker Compose** – Container orchestration

## 4.3.6 Sequence Diagrams

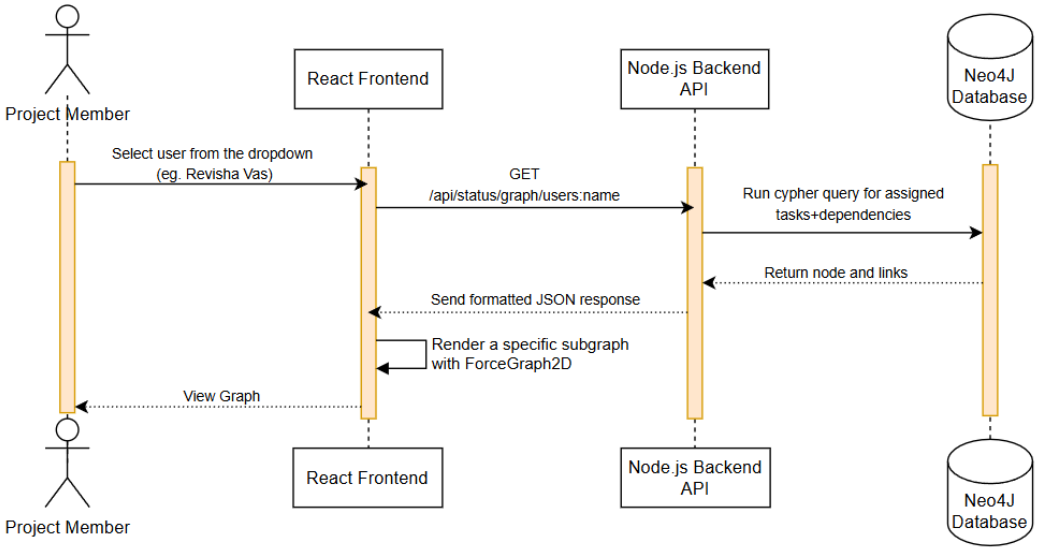


Fig 11: Sequence diagram – User-specific graph

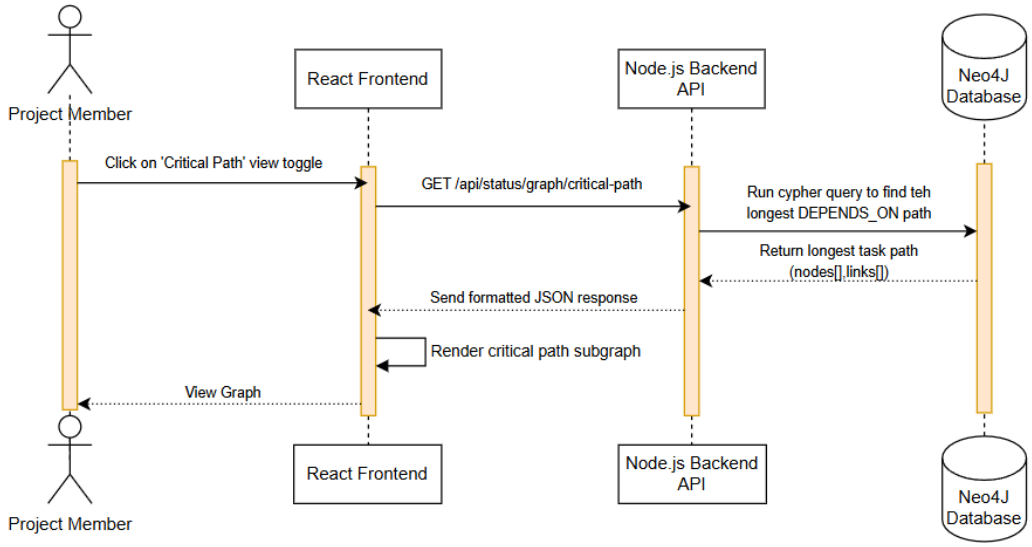


Fig 12: Sequence diagram – Display critical path dependency

## 4.3.7 Database

### Neo4j

- **Use**: Powers the static task graph visualization and supports dashboard insights like task dependencies, blockers, and status-based analytics.

- **Why**: To represent complex task relationships (like status flows, dependencies, and assignments) in a flexible graph structure that’s hard to model with traditional databases.

## 4.4 Comments – NAYANA

### 4.4.1 User Story

As a project manager or team member, I want to actively participate in discussions on task-related threads by having the ability to post, edit, delete, and reply to comments; reply to teammates using @mentions; respond to comments with emojis; and upload and share file attachments with my comments. Thus allowing me to actively participate in team discussions and maintain clear, timely, collaborative, and organized communication.

### 4.4.2 Use Cases

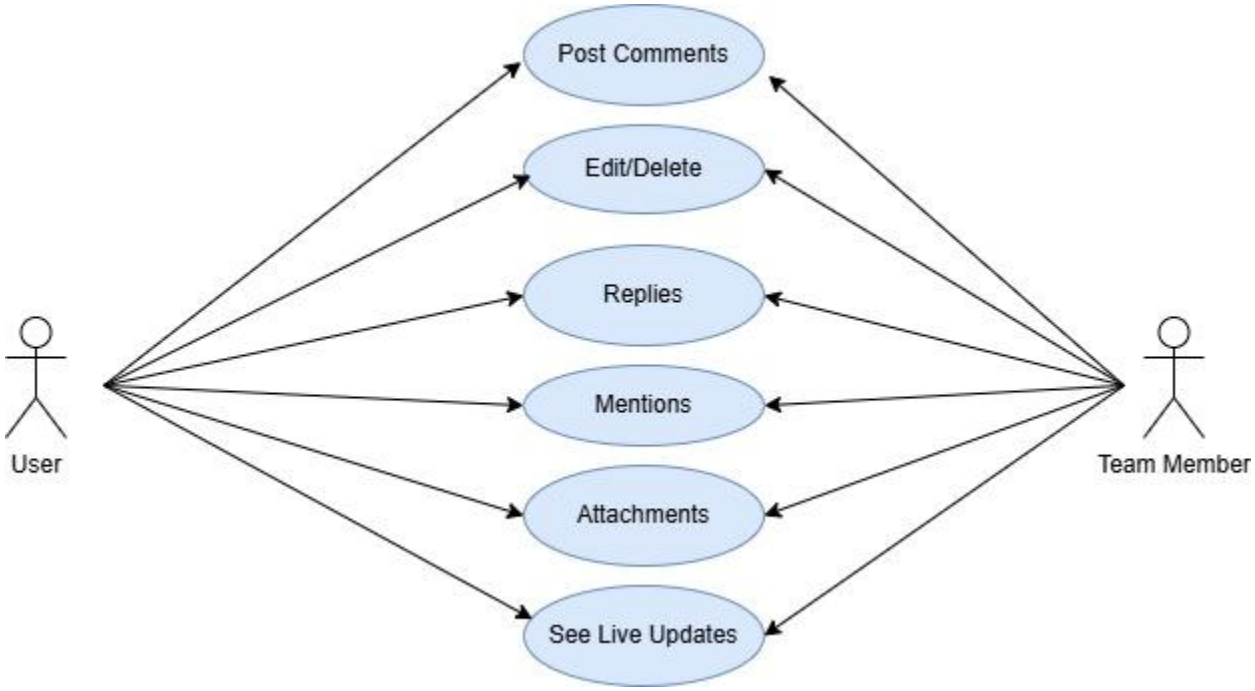


Fig13: Use Case UML diagram

### 4.4.3 Actors

- **Users (Project Managers & Team Members):**

  Users actively participate in the communication flow of the project. Project managers and team members are included in this since they have the same access to the comment system's features. They are able to:

  - Post new comments on tasks.
  - Edit or delete their own comments to make sure they are clear and appropriate.

- Reply to existing comments, while maintaining discussions threaded and organized.
- Use @username when mentioning teammates to get their attention or request input.
- Respond to comments with emojis to provide prompt recognition or feedback.
- Upload file attachments that are saved locally on the server and accessed via a URL.

### 4.4.3. Description of Task with All Interactions (Actor ↔ Database)

- **Actor Interaction → Event Propagation**

The backend does more than simply update the database when a user (Project Manager or Team Member) calls for a comment-related action, including posting a new comment, editing an existing one, replying, reacting with an emoji, deleting, or uploading an attachment. In order to keep all connected users in sync, it additionally uses Socket.IO to emit events in real time. Because of its architecture, the comment system is very collaborative, dynamic, and responsive

- **From API to Database + Broadcast**

Each interaction follows this event pipeline:

User Action (via UI)

→ API Call (Express route)

→ Validation + MongoDB Operation

→ Socket.IO Emit (e.g., `commentAdded`, `commentUpdated`, `commentDeleted`)

→ Frontend Listener Receives Update

→ Real-Time UI Update on All Connected Clients

This process guarantees that all users always see the most recent comment activity, including new messages, replies, edits, and deletions.

- **Synchronized Experience Across Clients**

- Real-Time Updates: All users will see updates in real time without having to refresh since Socket.IO instantaneously broadcasts comments, replies, edits, reactions, and deletions.

- Live Attachment Sharing: Files uploaded are saved locally and shared via accessible URLs,

### 4.4.4. Data Flow (Step-by-Step Detailed for this Use Case)

**Post Comment Flow**

A user clicks "Post" after typing a comment.

- A POST request including the content, task/project reference, and optional file is sent by the frontend to /api/comments.
- The new comment is saved into the MongoDB comments collection after the backend verifies the input.
- The URL of the attached file is recorded in the comment document and the file is uploaded to the server (stored locally).
- The collection's insert event is detected by a MongoDB Change Stream.
- All users who are connected receive a commentAdded event from the backend via Socket.IO once it has listened to the change stream.
- The change is sent to clients, who instantly see the new comment.

**Reply to Comment Flow**

To respond to an existing comment, a user clicks "Reply."

- Using the parentCommentId, the frontend submits a POST request to /api/comments.
- This response is saved in the comments collection by the backend and is connected to the parent comment.
- The MongoDB Change Stream takes up the modification.
- Clients display the response in real-time beneath the appropriate thread after Socket.IO emits a commentAdded event.

**Edit/Delete Comment Flow**

The user edits or deletes one of their own comments.

- A PATCH or DELETE request is sent to /api/comments/:id by the frontend.
- The MongoDB comment is updated or deleted by the backend.
- Update or delete operations are detected by MongoDB Change Stream.
- Using Socket.IO, the backend sends out a commentUpdated or commentDeleted event.

- The modification is immediately reflected in the user interface by connected clients.

**React to Comment Flow**

To respond to a comment, the user clicks an emoji .

- A PATCH request is sent to /api/comments/:id/reactions by the frontend.
- The MongoDB reactions array is updated by the backend.
- An update event is triggered by the MongoDB Change Stream.
- A commentReacted event is emitted by Socket.IO, and all clients instantly change the emoji display.

**Mention User Flow**

The user adds @username when typing a comment.

- For backend use (such asmentions lists), the mention is optionally labeled and saved in the comment body.
- The Change Stream is triggered when the new comment is added.
- The frontend highlights the specified username.

**Attachment Flow**

While posting or editing a comment, the user uploads a file.

- The file is saved locally on the server, and the MongoDB comment document contains the file's public URL.
- The change (update or insert) is detected by MongoDB Change Stream.
- Users can view the file attachment

## 4.4.5. Data Flow diagram

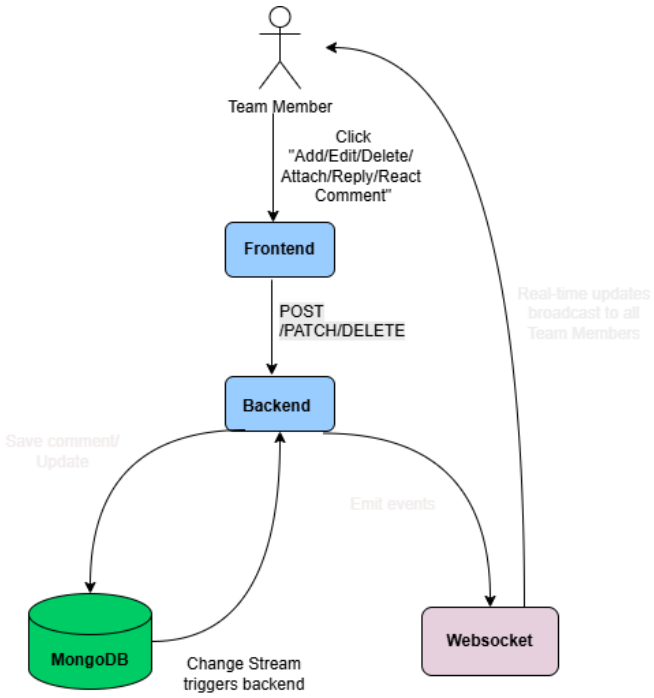


Fig14: Data flow diagram

### 4.4.6. Sequence Diagram

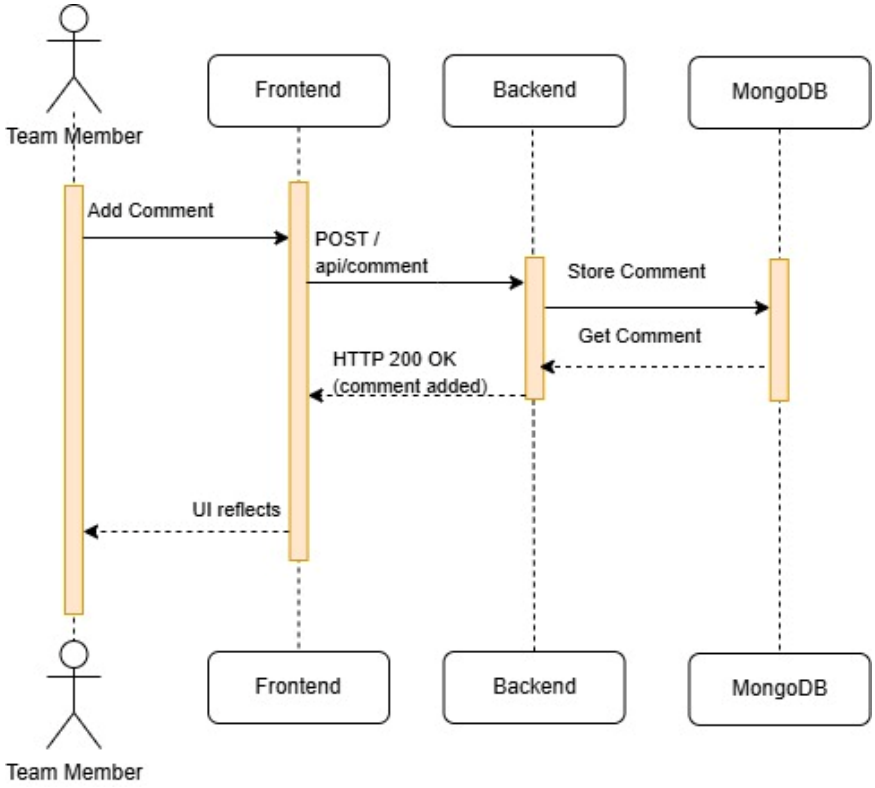


Fig15: Add Comment Sequence Diagram

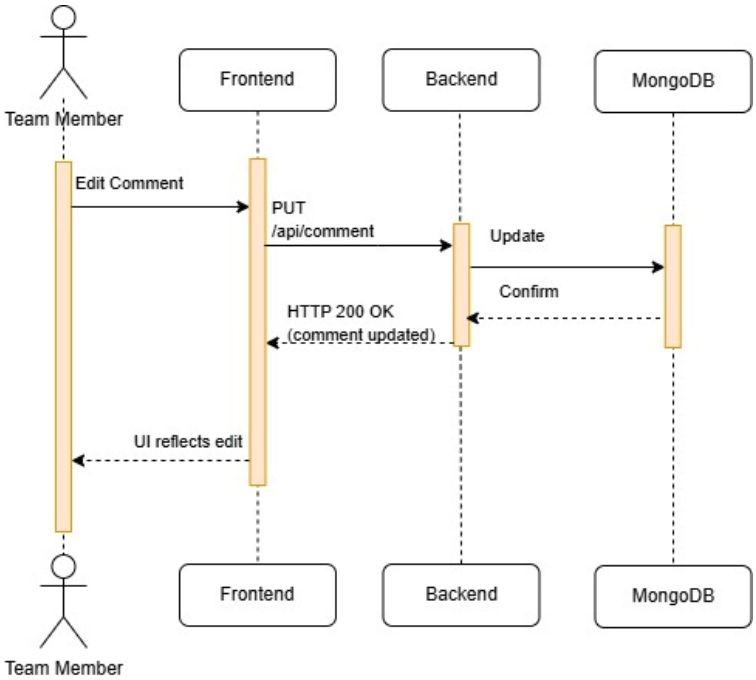


Fig16: Edit Comment Sequence Diagram

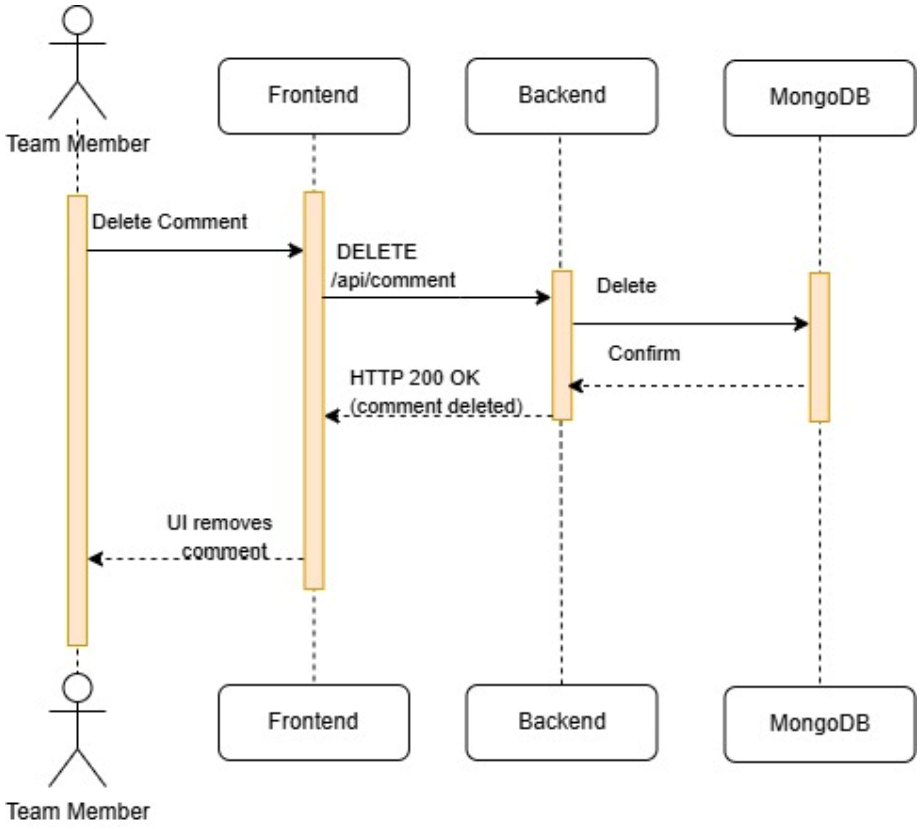


Fig17: Delete Comment Sequence Diagram

### 4.4.6. Database

**MongoDB:**

- **Use:** Saves all structured and persistent comment data, such as author references, attachments, mentions, reactions, content, and parent-child responses.
- **Reason:** MongoDB's flexible document model allows for nested structures (e.g., threaded replies), dynamic fields (e.g., emoji reactions), and embedded attachment metadata.
- **ER Diagram:**

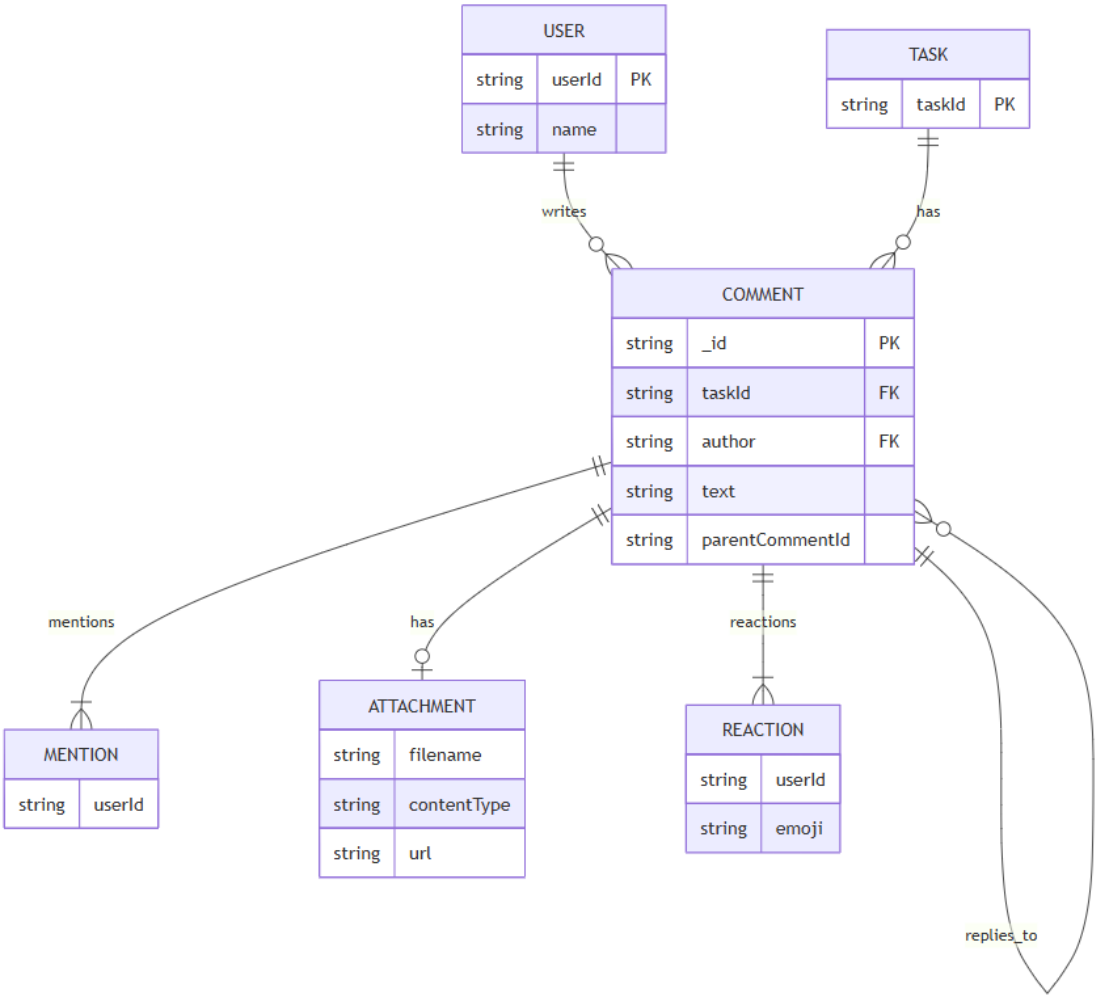


Fig18: ER Diagram

## 4.5 Chat System – PRAHARSHA

### 4.5.1 User Story

- As a *user*, I want to send and receive real-time messages so that I can communicate instantly with other team members.
- As a *user*, I want to view my chat history with other users so I can keep track of past conversations.
- As a *user*, I want to initiate new chats by selecting users from a list.

### 4.5.2 Use Cases

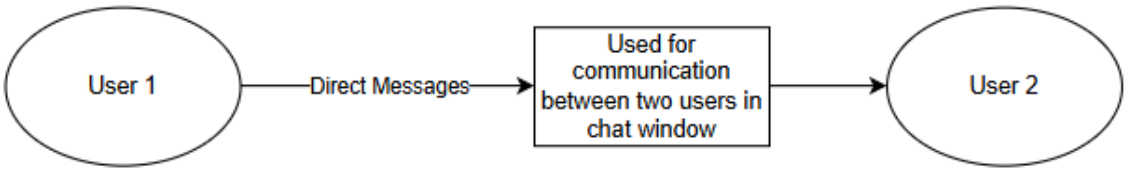


Fig19: Use case diagram of chat

### 4.5.3 Actors

All Users can:

Start a new chat with any other user.

View chat history with individuals.

Send and receive messages in real-time.

Use the chat interface (sidebar + chat window) to manage conversations.

### 4.5.4 Description of task with all interactions

Actors

Every message sent by a user not only updates the MongoDB database but also triggers a Redis Pub/Sub event.

This decouples message storage from UI rendering and enables real-time WebSocket broadcasting across all connected clients.

End-to-End Data Flow

Each interaction follows a systematic event pipeline:

User Action throuht UI

→ API Call (POST/GET on /api/chat)

→ DB Interaction via Mongoose (MongoDB Message model)

→ Redis chat-messages Publish

→ WebSocket Broadcast (via Socket.IO)

→ Frontend Real-Time Update

This ensures a consistent and reactive chat experience across clients.

### 4.5.5. Data Flow (Step-by-Step for Chat Use Cases)

Send Message Flow

User types a message and clicks "Send".

React frontend triggers sendMessage mutation (/api/chat/send).

Backend creates a message document (Message.create(...) in MongoDB).

Redis chat-messages channel publishes this message.

All connected clients subscribed to WebSocket receive the message and update the chat window instantly.

Receive Message Flow

Backend emits newMessage via Socket.IO after Redis receives the event.

The frontend listens using chatSocket.on('newMessage') in ChatWindow.jsx.

Relevant messages are filtered and appended to liveMessages state.

UI updates in real-time without a page reload.

View Chat History Flow

When a user selects a chat partner, a GET request hits /api/chat/history/:user1/:user2.

Backend fetches all messages between these two users from MongoDB and populates sender/receiver info.

React frontend renders the past chat history using ChatWindow.

Start New Chat Flow

User opens the "New Chat" modal (NewChatModal.jsx).

All available users (excluding current user) are fetched via /api/chat/users.

On selecting a user, the modal closes and switches the main view to that user's chat.

Load Conversations Flow

On page load or refresh, /api/chat/conversations/:userId is called.

Backend identifies all distinct users the current user has chatted with.

Sidebar renders these users, showing a list of conversations.

### 4.5.6. Dataflow Diagram

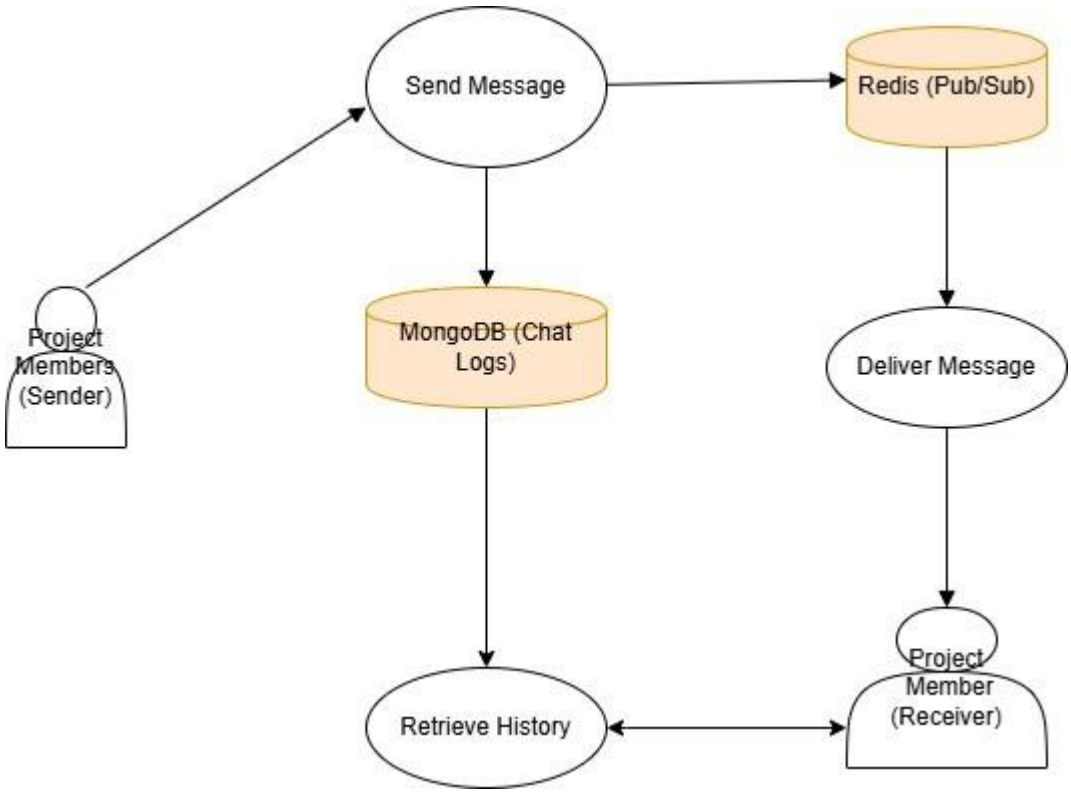


Fig 20: Dataflow diagram of chat

### 4.5.7. Sequence Diagram:

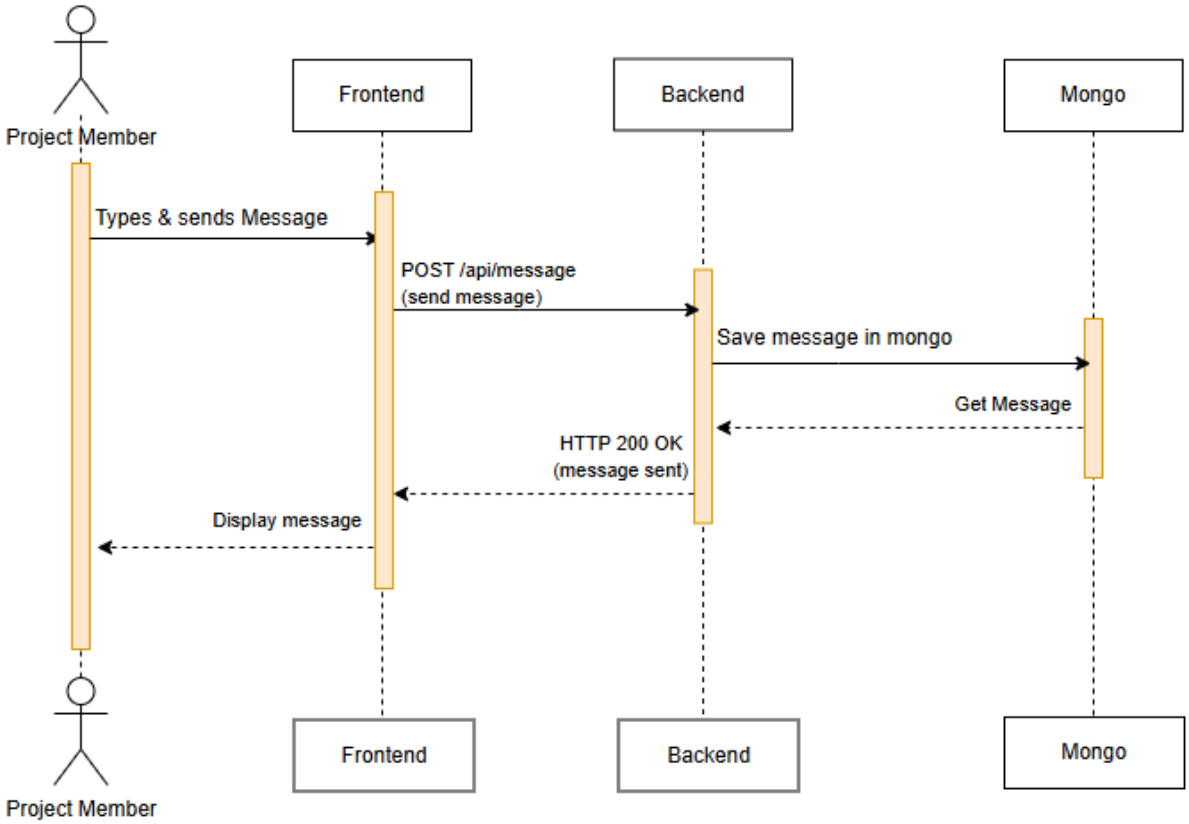


Fig 21: Sequence diagram of message

### 4.5.8 Database

Redis:

Use: Real-time message broadcasting, before database persistence

Reason:

- In-memory, low-latency Pub/Sub for instant delivery.
- Efficiently handles small, fast-expiring updates.

MongoDB:

- Use: Used for storing full message history between users and Persisting.
- Reason: Document model fits chat schema supports search & store history

# 5. Application

## 5.1 Languages Used

BitBoard is built using JavaScript and TypeScript along with three core technologies: MongoDB, Redis, and Neo4j.

- **Frontend:** React (JavaScript)
- **Backend:** Node.js (JavaScript/TypeScript)
- **Databases:**
    - **MongoDB:** Document-based NoSQL database used to store structured data like tasks, comments, users, and messages.
    - **Redis:** In-memory data store used for Pub/Sub and fast data retrieval in real-time collaboration, chat, and typing indicators.
    - **Neo4j:** Graph database used to visualize task-user-status relationships and project flow.

**Why JavaScript + Node.js + MERN Stack:**

- **Full Stack Consistency:** Using JavaScript on both frontend and backend allows for seamless communication, reusable logic, and easier developer handoff.
- **Real-Time Capabilities:** Node.js is optimized for event-driven, asynchronous operations — a perfect match for WebSocket-based updates and Redis Pub/Sub.
- **Ecosystem Strength:** The availability of mature libraries for sockets, routing, and database integration significantly accelerated development.

## 5.2 GitHub Link

https://github.com/mbilalhussain15/BitBoard-Real-Time-Code-Collaboration

## 5.3 Methods/Functions

**Add a task:**

This code snippet defines a POST API endpoint to create a new task, optionally including a file upload. It uses multer middleware to handle the file and encodes it in base64 if provided. The task data, including title, description, assignee, deadline, and status, is collected from the request body. After saving the task to the database, it populates the assignee's name

and publishes a taskCreated event. If an error occurs, it logs the error and sends a 400 response with an error message.

```
/**
 * POST: Create Task with optional file upload
 */
router.post('/', upload.single('attachment'), async (req, res) => {
  try {
    const taskData = {
      title: req.body.title,
      description: req.body.description,
      assignee: req.body.assignee,
      deadline: req.body.deadline,
      status: req.body.status,
    };

    if (req.file) {
      taskData.attachment = {
        data: req.file.buffer.toString('base64'),
        contentType: req.file.mimetype,
      };
    }
    const task = new Task(taskData);
    await task.save();

    const populatedTask = await Task.findById(task._id).populate('assignee', 'name');

    await pub.publish('taskCreated', JSON.stringify(populatedTask));

    res.status(201).json(populatedTask);
  } catch (err) {
    console.error('Task creation failed:', err);
    console.error('Failed data:', req.body);
    res.status(400).json({ error: err.message || 'Task creation error' });
  }
});
```

**Create a new room:**

This function handles the creation of a new room with a unique ID and a generated name. It starts by validating the presence of a name in the request body and then generates a user ID. The user is removed from all previous rooms before a new room is created and stored in Redis along with its participants. A WebSocket token is generated for secure communication, and a success response is returned with relevant room and user details.

```
export const createRoom = async (req, res) => {
  let { name } = req.body;
  if (!name) return res.status(400).json({ error: "Name is required" });

  const userId = await generateUserId(name);
  await removeUserFromAllRooms(userId);

  const roomId = Math.floor(100000 + Math.random() * 900000).toString();
  const roomName = await generateRoomName();

  await redis.set(`room:${roomId}`, JSON.stringify([{ userId, name }]));
  await redis.set(`roomName:${roomId}`, roomName);


  const token = generateWebsocketToken(userId, roomId);
    console.log("JWT_SECRET in token generation:", JSON.stringify(process.env.JWT_SECRET));

  res.status(200).json(
    {
        userId,
        name,
        roomId,
        roomName: roomName,
        websocketToken: token,
        message: "Room created" });
};
```

**Overdue task cypher query:**

This route handles the /overdueTasks endpoint and is responsible for calculating the number of tasks that are overdue. When the route is accessed, it opens a Neo4j session and runs a Cypher query to find all tasks that have a due date set and are not marked as "Done." It then checks whether each task's due date is in the past by converting the stored value (either a string date or a timestamp) into a proper date format. If the due date is earlier than today, the task is considered overdue. The total count of such tasks is returned as a JSON response. If any error occurs during the query execution, it logs the error and responds with a 500 status code. Finally, the Neo4j session is properly closed to free up resources.

```
router.get("/overdueTasks", async (req, res) => {
  const session = getSession();
  try {
    const result = await session.run(`
      MATCH (t:Task)-[:HAS_STATUS]->(s:Status)
      WHERE t.dueDate IS NOT NULL AND s.name <> "Done"
      WITH t,
        CASE
          WHEN toString(t.dueDate) CONTAINS "-" THEN date(t.dueDate)
          ELSE date(datetime({ epochMillis: toInteger(t.dueDate) }))
        END AS due
      WHERE due < date()
      RETURN count(t) AS overdue
    `);
    res.json({ overdue: result.records[0].get("overdue").toInt() });
  } catch (error) {
    console.error("Error in /overdueTasks:", error);
    res.status(500).json({ error: "Failed to get overdue tasks" });
  } finally {
    await session.close();
  }
});
```

**Post Comment:**

This Express.js route handles the creation of a new comment or reply for a specific task, supporting both text and file attachments. It extracts the `author`, `text`, `parentCommentId`, and `mentions` from the request body and the `taskId` from the URL. Before proceeding, it validates the task's existence by making an API call to the Kanban service. If a file is uploaded, it constructs an `attachment` object containing metadata like filename, MIME type, and file URL for later use or storage.

```
//Add comment or reply, with mentions,attachment
router.post('/:taskId/comments', upload.single('attachment'), async (req, res) => {
  const { author, text, parentCommentId, mentions } = req.body;
  const taskId = req.params.taskId;

  try {

    await axios.get(`${KANBAN_URL}/tasks/${taskId}`);

    let attachment;
    if (req.file) {
      attachment = {
        filename: req.file.originalname,
        contentType: req.file.mimetype,
        url: `/uploads/${req.file.filename}`
      };
    }
```

**Chat history:**

This route handles GET requests to fetch chat history between two users. It extracts user1 and user2 from the request parameters and queries the database for messages exchanged between them in both directions. The messages are populated with sender and receiver details, including their name and role. If successful, the history is returned in JSON format; otherwise, an error message is sent with a 500 status code.

```
router.get('/history/:user1/:user2', async (req, res) => {
  try {
    const { user1, user2 } = req.params;
    const history = await Message.find({
      $or: [
        { sender: user1, receiver: user2 },
        { sender: user2, receiver: user1 }
      ]
    }).populate('sender receiver', 'name role');
    res.json(history);
  } catch (err) {
    console.error(' Error loading history:', err);
    res.status(500).json({ error: 'Failed to load chat history' });
  }
});
```

# 6. Evaluation

## 6.1 Outlook

BitBoard showcases a successful integration of modern technologies like MongoDB, Redis, and Neo4j to deliver a seamless real-time collaborative platform. As remote and hybrid development becomes more common, tools like BitBoard can greatly improve productivity, transparency, and team coordination. The foundational work laid in this project opens doors for future expansion into enterprise-level Agile tools, educational platforms for coding bootcamps, or even developer-oriented social networks.

## 6.2 Lessons Learned

- **Collaborative Development:** Real-time systems demand precise planning and synchronization. Through regular sync-ups and Git conflict resolutions, we learned the value of modular design and proactive communication.
- **Database Integration:** Working with three different databases (MongoDB, Redis, Neo4j) taught us how to match technology to data needs — structured persistence, in-memory speed, and relationship modeling.
- **Event-Driven Architecture**: Implementing Redis Pub/Sub and MongoDB Change Streams provided hands-on understanding of reactive systems and how to ensure data consistency across clients.
- **Frontend-Backend Coordination:** Keeping the UI in sync with real-time changes across sockets and APIs required robust event propagation patterns, which deepened our understanding of WebSocket and state management.

## 6.3 Possible Extensions

- **Authentication and Role-Based Access Control:** Currently, BitBoard assumes users are authenticated. Integrating OAuth or JWT-based secure access control would be a logical next step.
- **Offline Support & Caching:** Using local storage or service workers, BitBoard can be enhanced to allow limited offline capabilities with sync-on-reconnect logic.
- **Real-Time Video/Audio Rooms:** Embedding voice or video calls using WebRTC could take live collaboration to the next level for remote teams.
- **CI/CD Pipeline:** Integration with GitHub Actions or other CI tools to support automated code testing/deployment directly from the code collab module.

## 6.4 Reflections

Looking back, BitBoard represents a well-coordinated team effort where each feature component came together to form a powerful real-time platform. The initial scope was ambitious, but consistent progress, mutual support, and technical learning made the vision achievable. Each member brought unique strengths — from UI design and code execution to database optimization and socket communication. This project not only honed our software engineering skills but also simulated a startup-like product development experience. With a few more iterations and refinements, BitBoard could be developed into a publicly usable platform.

# 7. References


- Delikumar, E. (n.d.). *Building real-time applications with Node.js, MongoDB, and Redis*. Medium. https://medium.com/@edsin.delikumar/building-real-time-applications-with-node-js-mongodb-and-redis-f500fd272e9a
- Rayan. (n.d.). *Realtime collaborative whiteboard using Node.js, MongoDB, Nginx LB, and Redis*. Medium. https://medium.com/@rayancr/realtime-collaborative-whiteboard-using-nodejs-mongodb-nginx-lb-and-redis-a0f1d29a1462
- Mazaheri, N. A. (n.d.). *Using Redis as a cache for MongoDB with Node.js*. Medium. https://medium.com/@na.mazaheri/using-redis-as-a-cache-for-mongodb-with-node-js-aaf303cfb513
- Batra, S. (2021). *A complete guide to MERN stack development*. GeeksforGeeks. https://www.geeksforgeeks.org/a-complete-guide-to-mern-stack-development/
- Chodorow, K. (2013). *MongoDB: The definitive guide* (2nd ed.). O'Reilly Media.
- Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph databases: New opportunities for connected data* (2nd ed.). O'Reilly Media.
- Tilkov, S., & Vinoski, S. (2010). Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 14(6), 80–83. https://doi.org/10.1109/MIC.2010.145
- Desikan, S. (2020). *WebSocket vs REST: A comparison for real-time communication*. Towards Data Science. https://towardsdatascience.com/websockets-vs-rest-apis-for-real-time-communication-d7f89c78c048
- Neo4j, Inc. (2024). *Neo4j Graph Database Platform: Developer Manual v5.14*. https://neo4j.com/docs/