Submission 2:
Software Architecture Document

Table of Contents:

1.Introduction

1.1 Purpose

This document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system.

1.2 Scope

This Software Architecture Document provides an architectural overview of the Accord Project Management System. The Accord Project Management System is being developed by Team 42 in Software Engineering @ Constructor University to support project and time management efforts.

1.3 Stakeholders

Role	Description
Project Manager	Oversees overall project delivery, manages team structures, and has full access to documents.
Administrator	Manages user permissions, folder hierarchies, and system configurations.
Team Leader	Coordinates the work of team members, uploads files, and monitors task progress.
Team Member(referred to as the Participant role)	Executes assigned tasks, uploads and views task files.
External Reviewer (referred to as the Stakeholder role)	Views project materials for auditing or external review purposes.

The previously outlined actors are further explained in section 4.1.

2. Architectural Representation

This document presents the architecture as a series of views; use case view, logical view, process view and deployment view. There is no separate implementation view described in this document. These are views on an underlying Unified Modeling Language (UML) model developed using Plant UML.

3. Architectural Goals and Constraints

There are some key requirements and system constraints that have a significant bearing on the architecture. They are:

3.1 The system must interface with an existing MongoDB database containing task creation details, and a MariaDB database containing structured information on users, teams, projects, stakeholders, and comments. The architecture must fully support the defined schema and data relationships.

3.2 The application must support differentiated user access and functionality based on roles: Administrators, Stakeholders, Project Managers, Team Leaders, and Participants. Each role should see a tailored dashboard and have specific privileges within the system.

3.3 All user functionality—including project creation and editing, team assignment, task updates, and stakeholder input—must be accessible from both on-campus and remote machines via a web browser.

3.4 The system must ensure data integrity and security. All access must be protected through login-based authentication, and role-based authorization must be enforced to restrict access to certain features.

3.5 The system will follow a web-based client-server architecture. The client interface will run in modern browsers and be built using HTML, CSS, and JavaScript, while the server logic will be implemented in Python using the Flask framework.

3.6 Performance requirements such as quick load times, efficient data queries, and smooth user experience under concurrent access must be considered throughout the architecture design.

4. Use Case View

The use-case view provides essential input for selecting scenarios and use cases that guide each development iteration. It highlights core system functionality and identifies scenarios that provide substantial architectural coverage—those that involve multiple components or illustrate sensitive areas of the design.

The key use cases for the project management software include:

- **Login**
- **Create Project** (Project Managers only)
- **Create and Edit Tasks, Add Tasks to Calendar** (Project Managers and Team Leader only)
- **View Assigned Projects and Tasks**
- **Update Task Status**
- **Comment on Tasks' Discussion Board**
- **Manage Team Composition** (Admin only)
- **View and Approve Project Charter** (Stakeholder only)
- **File Sharing**
- **Compute Task Cost and Total Cost**
- **Project Risk Assessment** (Project Managers only)
- **Budget Allocation** (Project Managers only)

These use cases are initiated by various actors: Project Managers, Team Leaders, Participants, and Administrators. Additionally, the system is designed to optionally interact with external services, such as the Google Calendar API, to provide scheduling and deadline tracking functionality.

4.1 Architecturally-Significant Use Cases

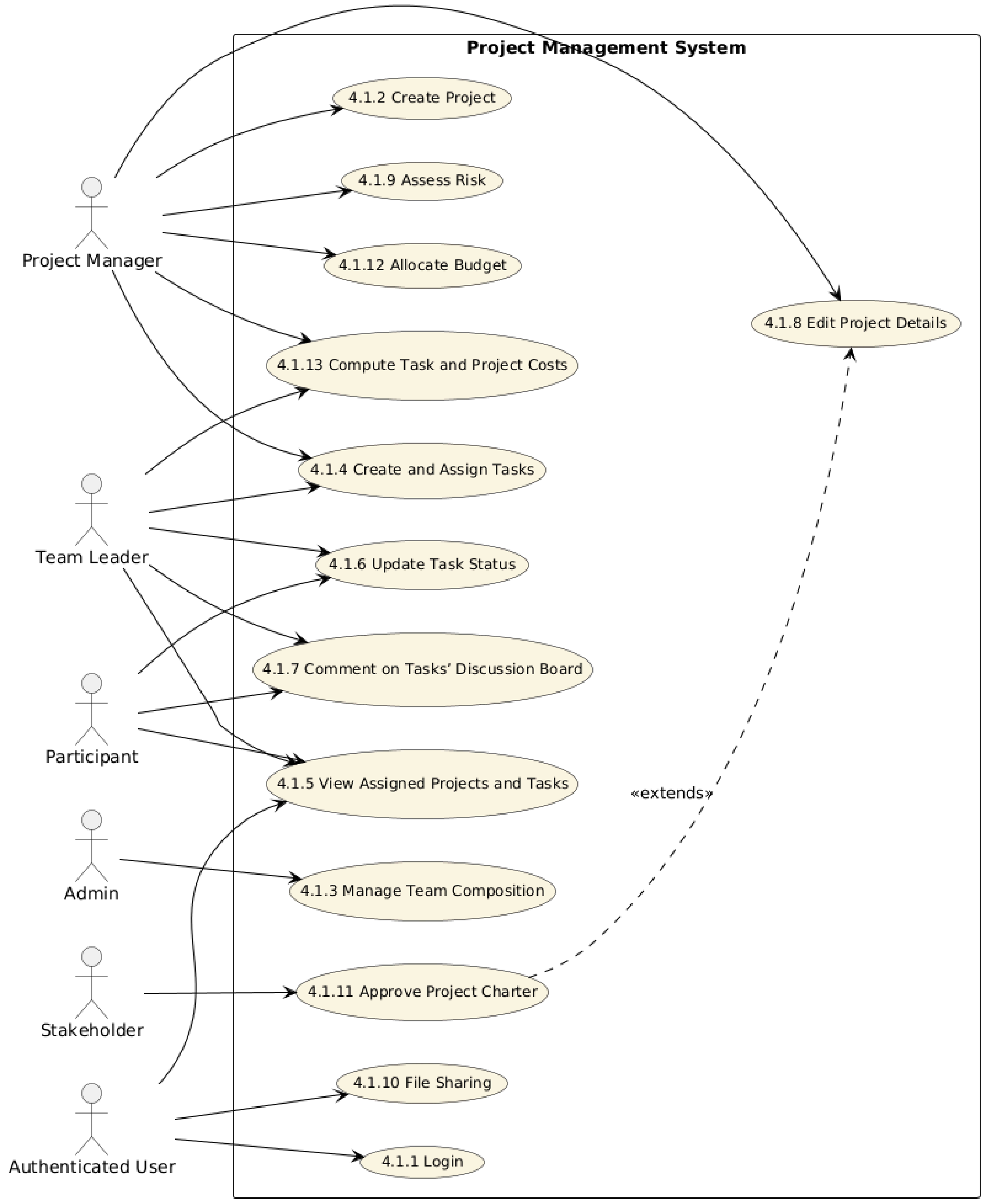

Project Management System
4.1.2 Create Project
4.1.9 Assess Risk
4.1.12 Allocate Budget
4.1.13 Compute Task and Project Costs
4.1.4 Create and Assign Tasks
4.1.6 Update Task Status
4.1.7 Comment on Tasks' Discussion Board
4.1.5 View Assigned Projects and Tasks
4.1.3 Manage Team Composition
4.1.11 Approve Project Charter
4.1.10 File Sharing
4.1.1 Login
4.1.8 Edit Project Details
«extends»
Project Manager
Team Leader
Participant
Admin
Stakeholder
Authenticated User

Diagram Name: Architecturally Significant Use-Cases

4.1.1 Login

Brief Description: This use case describes how a user logs into the project management system. Upon successful authentication, users are redirected to a dashboard tailored to their role: **Project Manager**, **Team Leader**, **Participant**, or **Admin**.

4.1.2 Create Project

Brief Description: This use case allows a **Project Manager** to create a new project. The manager assigns one or more existing teams to the project, selects teams, computes risks and costs, allocates budget, and fills out stakeholder information. Only users with the role of Project Manager can initiate this use case.

4.1.3 Manage Team Composition

Brief Description: This use case allows the **Admin** to view all teams, assign users who are currently unassigned, and designate a single team leader per team. Only Admins can perform this task.

4.1.4 Create and Assign Tasks

Brief Description: This use case allows a **Project Manager** to create and assign tasks within a project. Tasks are assigned to a team or specific users. Deadlines, descriptions, and dependencies may also be specified during task creation.
Subtasks can be created by **Team Leaders** and assigned to **Participants.**

4.1.5 View Assigned Projects and Tasks

Brief Description: This use case allows any authenticated user—**Project Manager**, **Team Leader**, or **Participant**—to view a list of their assigned projects and tasks. The data displayed depends on the user's role and team membership.

4.1.6 Update Task Status

Brief Description: This use case allows **Team Leaders** and **Participants** to update the status of a task (e.g., In Progress, Completed). They can also provide updates or flag issues as needed. Only users assigned to a task may update it.

4.1.7 Comment on Tasks' Discussion Board

Brief Description: This use case allows all assigned users to post comments on a specific task's discussion board as well as in private discussions with other team members. These comments facilitate communication and tracking of task progress. This is relevant to all users except **Stakeholders**.

4.1.8 Edit Project Details

Brief Description: This use case allows a **Project Manager** to update project metadata (e.g., deadlines, team assignments, stakeholders). The manager may also add or remove teams and modify task definitions if the project is still active.

4.1.9 Assess Risk

Brief Description: This use case allows a **Project Manager** to compute the Risk Score of a potential project. The total project risk score is computed using a weighted model across six categories: Budget, Timeline, Team Experience, Scope Complexity, External Dependencies, and Unforeseen Risks. Each is rated 0–100 and weighted accordingly in the formula:

$R = (B\times0.2) + (T\times0.2) + (E\times0.15) + (S\times0.15) + (D\times0.15) + (U\times0.15)$.

The final risk assessment percentages based on the model which now has a 97.3% Confidence Score are the following:

- **AI/ML Research Project:** 55.00% up to 90.00%. With a mean of 72.50%.
  Reasoning:

The high mean risk reflects factors like cutting-edge scope and data issues (Team Experience is weighted 20%). In practice, most AI/ML projects cluster in the upper half of the risk scale.
Example: A project with Budget 70%, Timeline 40%, TeamExp 60%, Scope 80%, External 60%, Unforeseen 70% yields a weighted risk ≈ $0.2\times70+0.15\times40+0.2\times60+0.15\times80+0.15\times60+0.15\times70 \approx 63.5\%$.

- **Game Development:** 60.00% up to 99.00%. With a mean of 79.50%.
  Reasoning:
  The range covers relatively lower-risk AAA scenarios up to near-certain failure for troubled indie projects.

- **Mission-Critical Software:** 45.00% up to 85.00%. With a mean of 65.00%.
  Reasoning:
  General IT data (Standish Group) show ~66% of projects end in partial or total failure which matches our findings.
- **Mobile App Development:** 45.00% up to 80.00%. With a mean of 62.50%.
  Reasoning:
  Mobile projects often have lighter scope (Scope weight 10%) but tight timelines (Timeline weight 25%).

- **Embedded Systems**: 60.00% up to 95.00%. With a mean of 77.50%.
  Reasoning:
  Embedded/electronic development is very risky: industry analyses cite ~90% of new electronic products fail to reach the market.

Risk Budget Allocation (Annotation based on findings):
6% for Low Risk (<50%)
9% for Medium Risk (50–75%)
15% for High Risk (>75%)

4.1.10 File Sharing

Brief Description: This use case allows any authenticated user—**Project Manager**, **Team Leader**, or **Participant** to share files using Google Drive API in an integrated file storage and management system.

4.1.11 Approve Project Charter

Brief Description: This use case allows a **Stakeholder** to review and approve the Project Charter submitted by the **Project Manager**. Once approved, the project moves from the planning to the execution phase. Only users with the **Stakeholder** role can perform this action.

4.1.12 Allocate Budget

Brief Description: This use case allows a **Project Manage**r to define and allocate the budget for a project. The **Project Manager** can distribute the budget across tasks and teams. Budget adjustments can be made as the project progresses by editing the project charter and submitting it for approval.

4.1.13 Compute Task and Project Costs

Brief Description: This use case allows **Project Managers** and **Team Leaders** to estimate the cost of the project and introduce costs when managing tasks respectively. Cost computation may include various goods and services. The system may provide projections to assist in budgeting decisions.

5. Logical View

A description of the logical view of the architecture. Describes the most important classes, their organization in service packages and subsystems, and the organization of these subsystems into layers. Also describes the most important use-case realizations, for example, the dynamic aspects of the architecture.
The internal architecture (described below) is deployed as detailed in Section 7. While this section focuses on logical responsibilities and class-level relationships, the physical deployment, including service locations and scalability plans, is discussed separately.

The logical view is composed of three major layers:

- **Presentation Layer**: Handles UI components like forms and dashboards for different roles.
- **Application Logic Layer**: Processes requests (e.g., assigning tasks, managing teams) using Flask routes and control logic.
- **Data Layer**: Interfaces with MariaDB for structured data and MongoDB for task tracking.

Each entity (User, Team, Task, Project) is modeled as a class or collection.

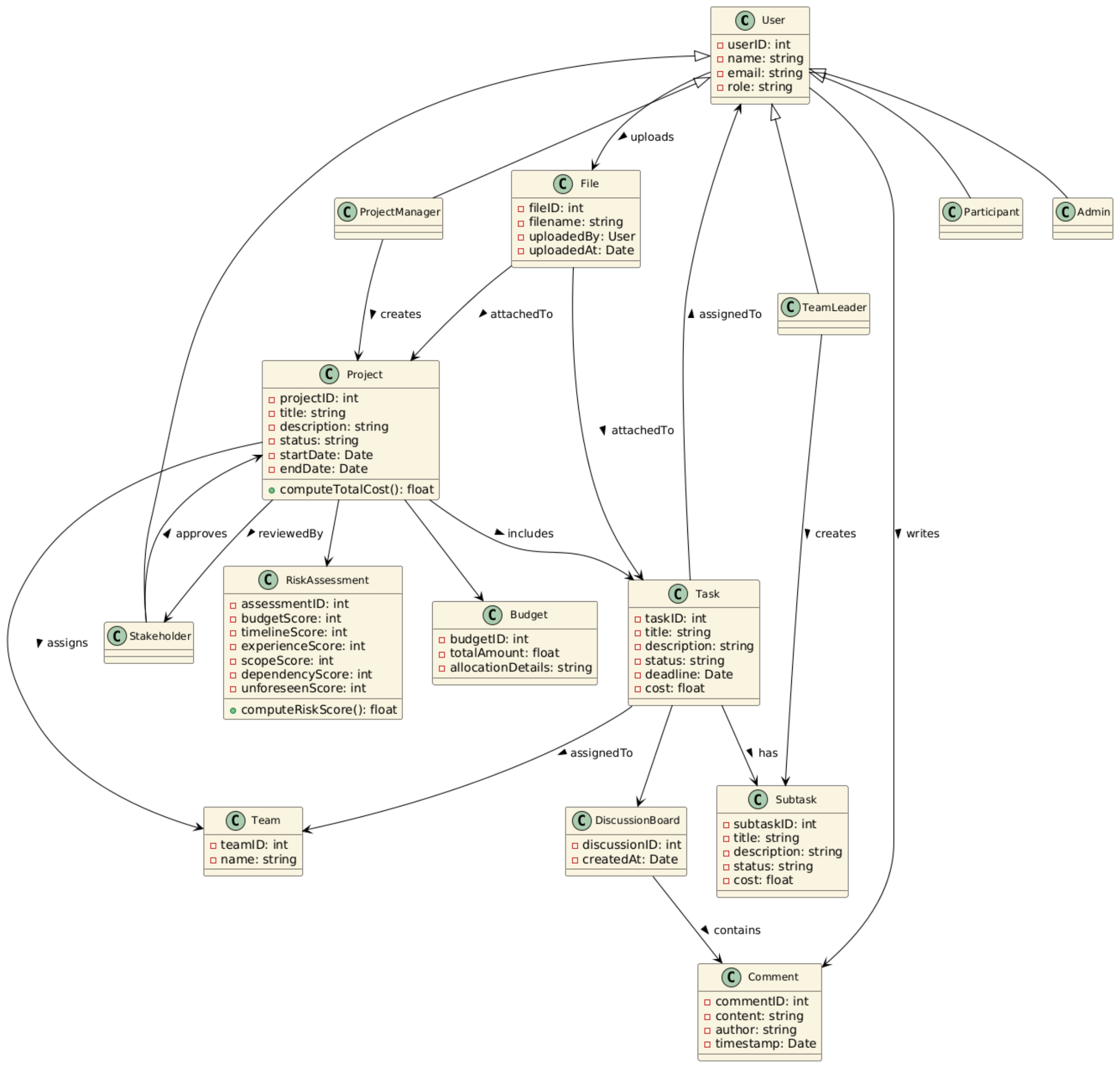


Diagram Name: Class Diagram

5.1. Presentation Layer

The Presentation Layer is responsible for delivering the user interface and ensuring a smooth user experience across all roles in the system. It is built using HTML, CSS, and JavaScript. It renders role-specific dashboards and forms dynamically based on the user's authentication state and permissions.

- **Role-Specific Dashboards:** Each user—Admin, Project Manager, Team Leader, Participant, or Stakeholder—sees a different UI that reflects their privileges and tasks.

- **Form Handling:** This layer includes forms for login, task updates, project creation, team management, and stakeholder input. Form validation is done both client-side (via JavaScript) and server-side (via Flask).

- **API Communication:** It sends HTTP requests (GET/POST) to Flask endpoints to fetch or submit data. API documentation is described further in section 9.

5.2. Application Logic Layer

The Application Logic Layer serves as the brain of the system. It is implemented using the Flask framework in Python and is responsible for interpreting user requests, enforcing business logic, and coordinating interactions between the Presentation Layer and the Data Layer.

- **Routing and Controllers:** Flask routes are mapped to specific functions that handle various requests such as task assignment, user authentication, file uploads, and project creation.

- **Business Logic: Includes logic for:**
    - Assigning roles and privileges
    - Managing team compositions
    - Evaluating project risk
    - Enforcing access control and validation rules
    - Computing Cost
    - Allocating Budget
    - Posting comments
    - Sharing Files

- **Session Management:** Utilizes Flask's session handling or Flask-Login to manage active sessions, keep users logged in, and enforce role-based views.

- **External APIs Integration:** This layer also interacts with external services, such as:
    - Google Calendar API for scheduling and deadline visualization
    - Google Drive API for document uploads and storage

5.3. Data Layer

The Data Layer is responsible for persistent data storage and retrieval. It interacts with two separate databases to handle structured and semi-structured information:

MariaDB (Relational)

Used for storing structured, normalized data that includes:

- User credentials and roles
- Project metadata
- Team compositions and leadership assignments
- Stakeholder records
- Comments and discussion threads

MongoDB (NoSQL)

Used for managing dynamic and flexible task data, especially where subtasks, updates, and history logs vary in structure.

- Stores:
    - Task documents with nested subtasks
    - Task status updates
    - Progress logs
    - Activity timelines
    - Task specific budgeting and cost calculation information

Benefits:

- Flexible schema suits frequently-changing task structures
- Efficient for document retrieval and hierarchical data models
- Can scale horizontally if needed

Data from both databases is abstracted and accessed through service modules or helper functions within the application layer to ensure loose coupling and easier maintenance.

6. Process View

A description of the process view of the architecture. Describes the tasks involved in the system's execution, their interactions, workflows, and configurations. Also describes the allocation of objects and classes to tasks.

6.1 Processes

6.1.1 Login

When a user accesses the login page through their browser, they are prompted to enter their username and password. Upon submission, the login credentials are sent to the Flask backend for authentication. The system then queries the MariaDB database to retrieve the stored credentials associated with the provided username. The entered password is hashed and compared against the stored hashed password in the database.

If the credentials are valid, the system establishes a user session and redirects the user to a role-specific dashboard based on their assigned role.

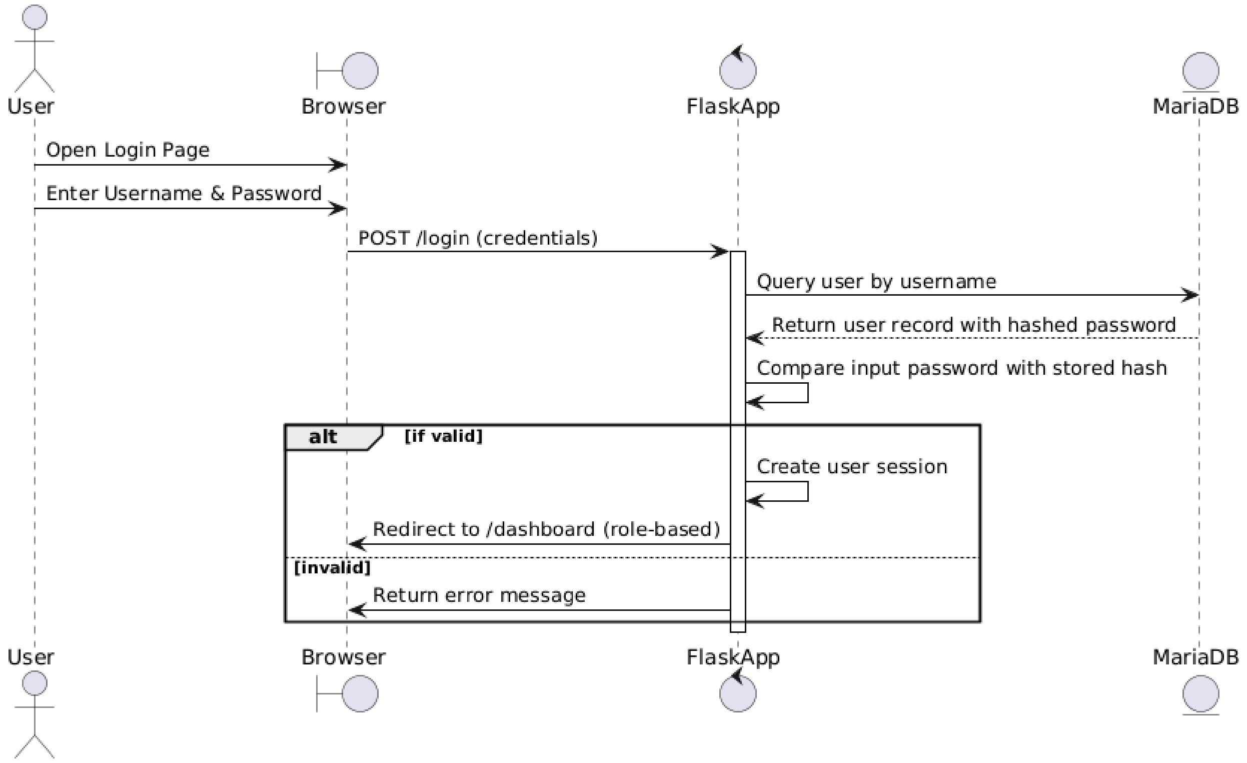

User
Browser
FlaskApp
MariaDB
Open Login Page
Enter Username & Password
POST /login (credentials)
Query user by username
Return user record with hashed password
Compare input password with stored hash
alt
[if valid]
Create user session
Redirect to /dashboard (role-based)
[invalid]
Return error message
User
Browser
FlaskApp
MariaDB

Diagram Name: Login Workflow

6.1.2 Task Creation

When a Project Manager wants to create a task, they navigate to the project dashboard and open the “Create Task” form. They input relevant details such as the task name, description, deadline, assigned team, and any dependencies or priority levels.

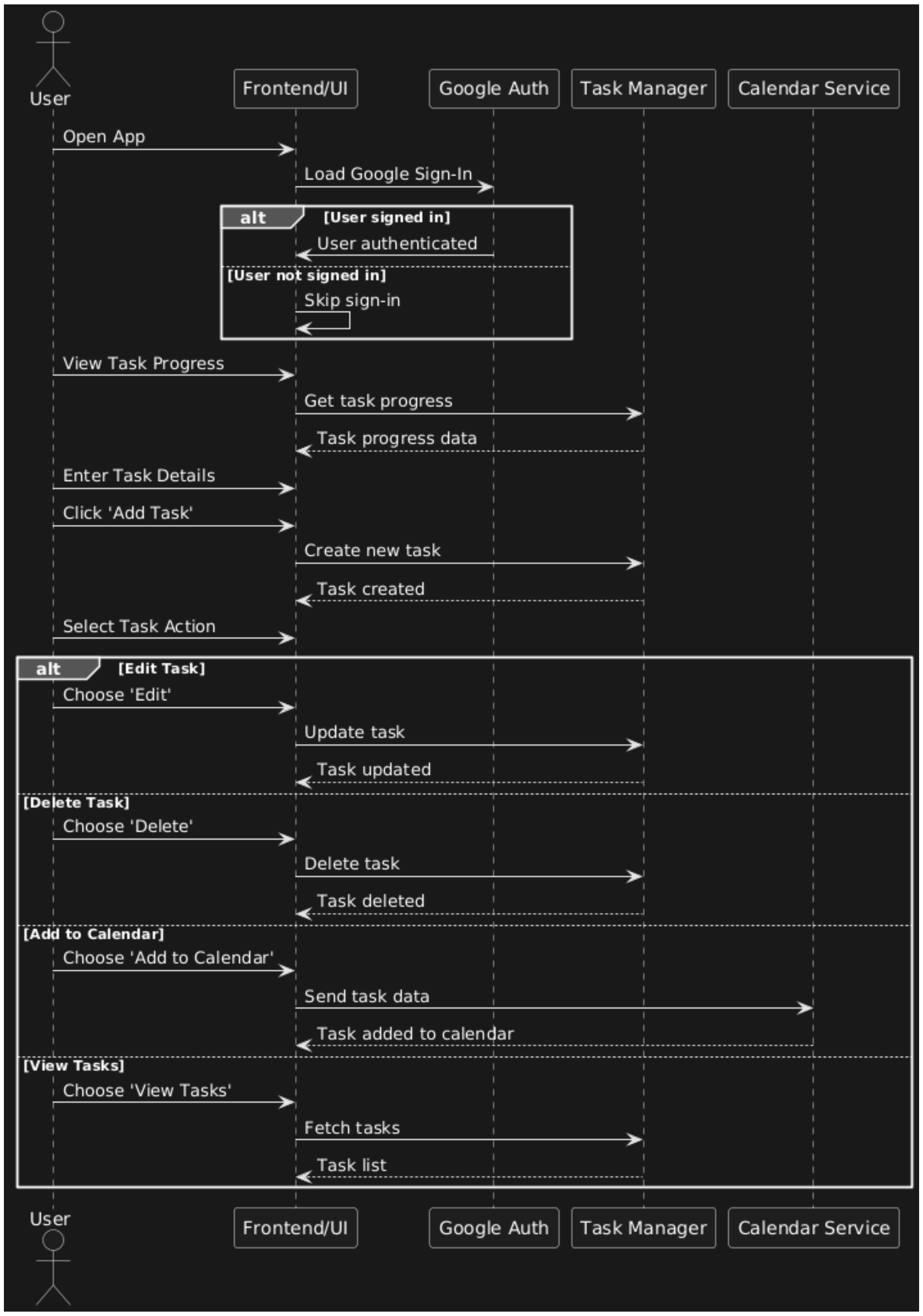


Diagram Name: Task Creation Workflow

6.1.3 Task Progress

Once a task is created, the task's progress needs to be tracked. The Team Leader or Participant assigned to the task will update the status as they work on it.

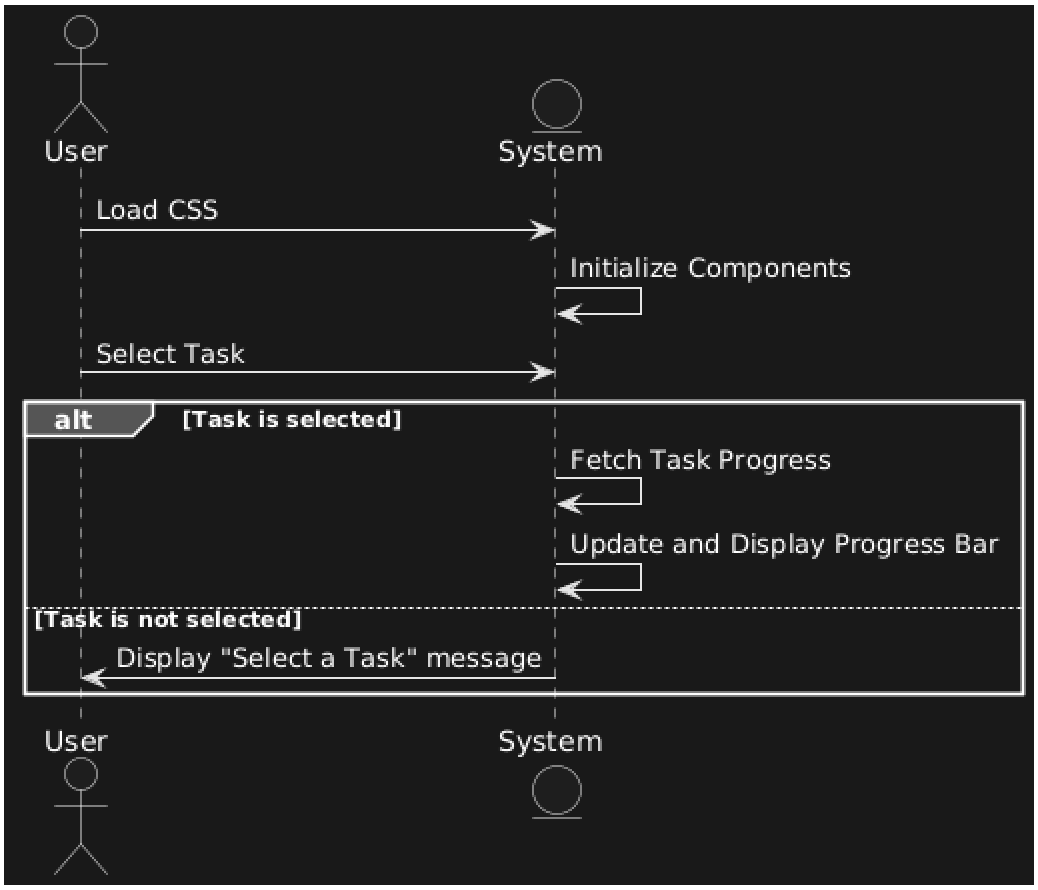


Diagram Name: Task Progress Workflow

6.1.4 Subtask Creation Workflow

A Team Leader can create subtasks for a task to break it down into smaller, more manageable units.

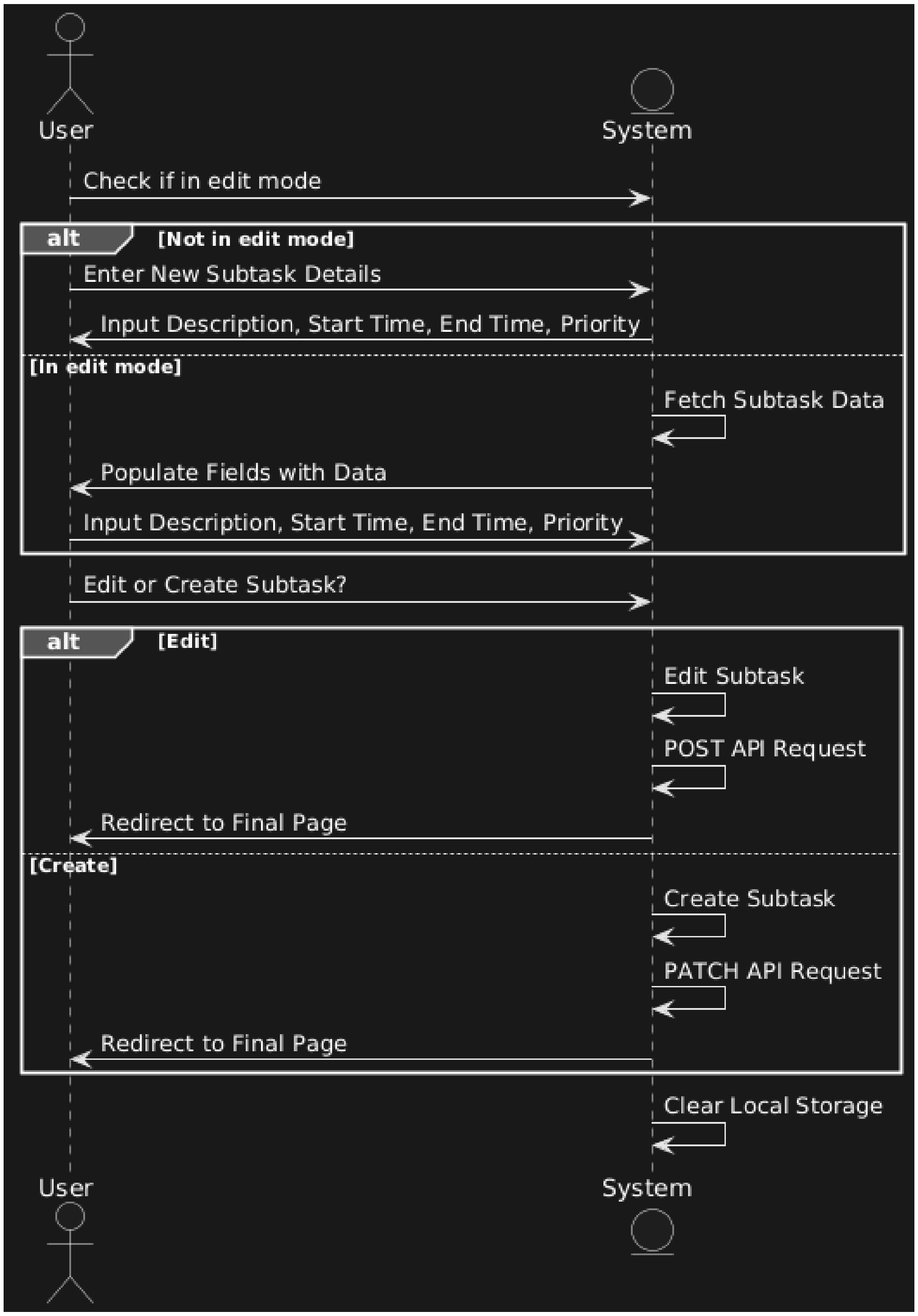


Diagram Name: Subtask Creation Workflow

6.1.5 Risk Score Computation

The Risk Score computation allows the Project Manager (PM) to evaluate the potential risks of a project before starting it. This is done by selecting a project type from a dropdown list. The system then calculates a risk percentage based on pre-established values that were trained using data from previous projects or companies.

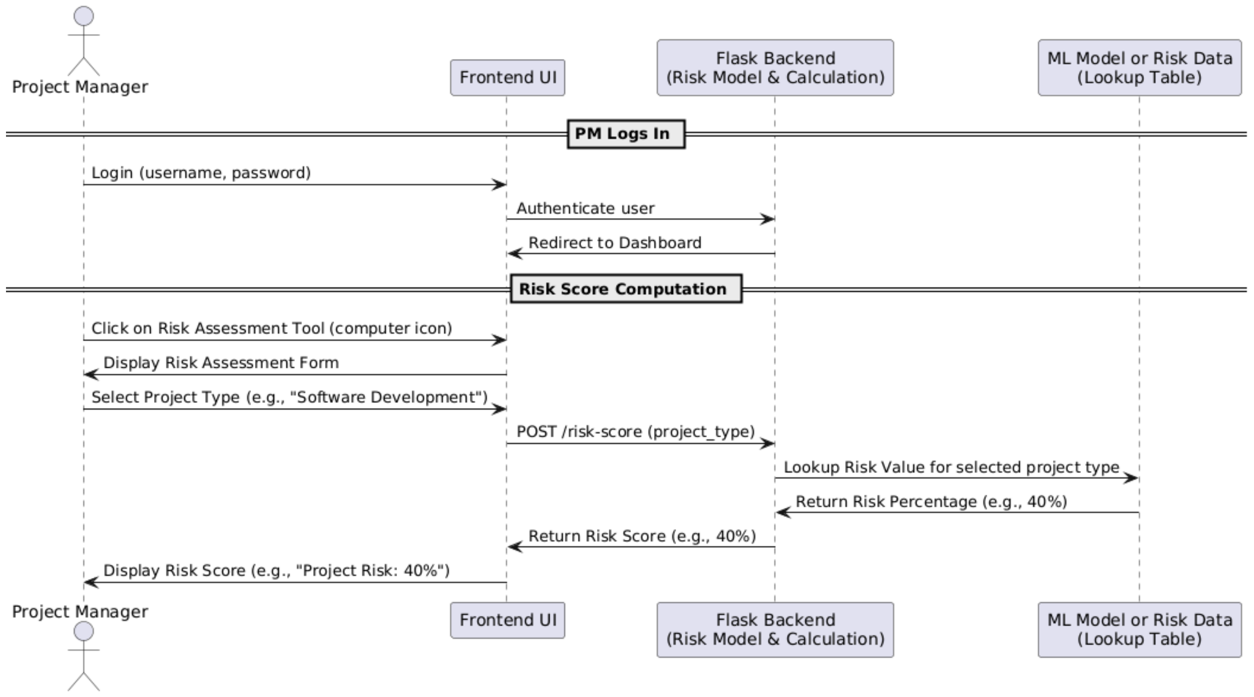

Project Manager
Frontend UI
Flask Backend (Risk Model & Calculation)
ML Model or Risk Data (Lookup Table)
PM Logs In
Login (username, password)
Authenticate user
Redirect to Dashboard
Risk Score Computation
Click on Risk Assessment Tool (computer icon)
Display Risk Assessment Form
Select Project Type (e.g., "Software Development")
POST /risk-score (project_type)
Lookup Risk Value for selected project type
Return Risk Percentage (e.g., 40%)
Return Risk Score (e.g., 40%)
Display Risk Score (e.g., "Project Risk: 40%")

Diagram Name: Risk Assessment Workflow

6.1.6 Project Creation, Approval and Editing

When the project is created by the Project Manager, a Project Charter is Created and sent for review to the Stakeholder in charge of that. That Stakeholder can either Approve or Request Revision within a specified time window. The charter is then modified accordingly. In case of the Project Manager editing the charter at any time, it is once again sent for approval and the process repeats as described.

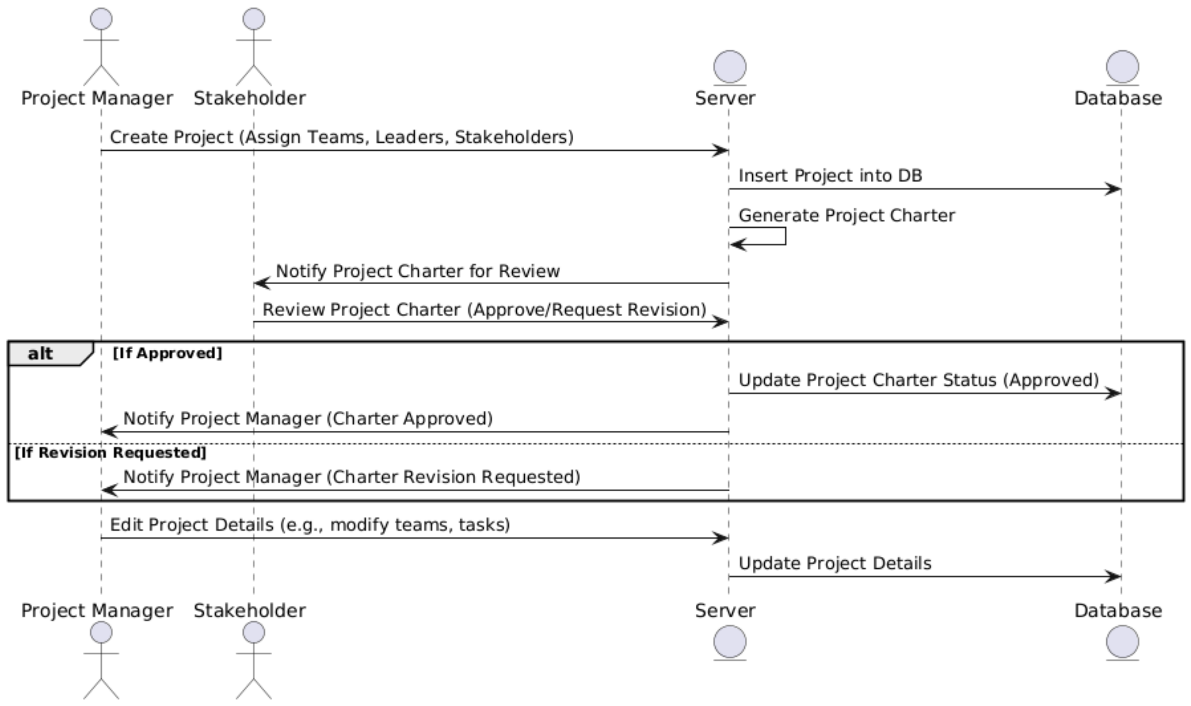

Project Manager
Stakeholder
Server
Database
Create Project (Assign Teams, Leaders, Stakeholders)
Insert Project into DB
Generate Project Charter
Notify Project Charter for Review
Review Project Charter (Approve/Request Revision)
alt
[If Approved]
Update Project Charter Status (Approved)
Notify Project Manager (Charter Approved)
[If Revision Requested]
Notify Project Manager (Charter Revision Requested)
Edit Project Details (e.g., modify teams, tasks)
Update Project Details
Project Manager
Stakeholder
Server
Database

Diagram Name: Project Creation, Approval and Editing Workflow

6.1.7 File Sharing

Users share files by navigating to the specific Task, and clicking on the File Sharing button. Then they upload directly to google drive in a folder created for the specific project and task, which all team members have access to.

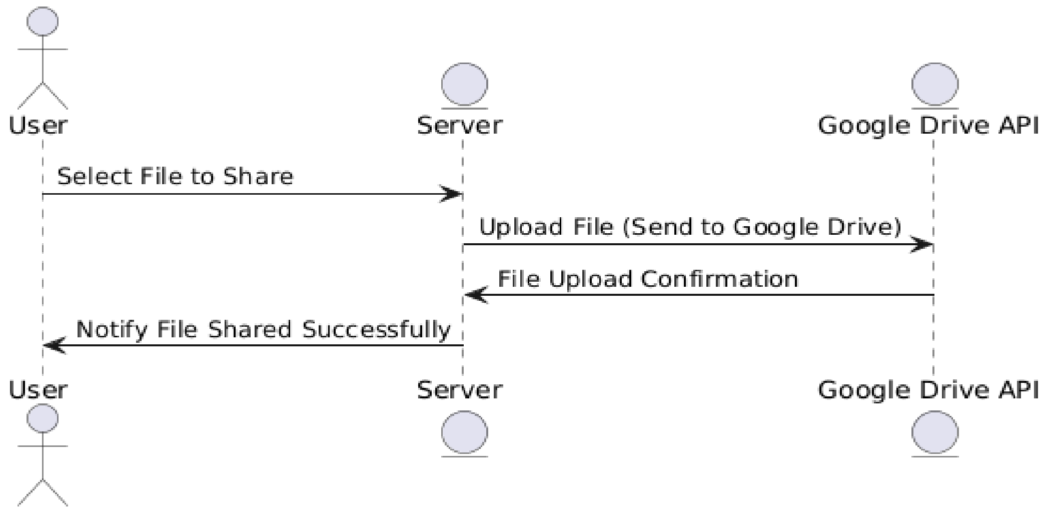


Diagram Name: File Sharing Workflow

6.1.8 Computing Cost

Cost computation starts at the subtask level during project charter creation. Each subtask is classified as either a **good** or a **service**, and different parameters are used to compute the estimated cost:

- For **services**, the estimated cost is calculated as:
  `Estimated Cost = Time (hours) × Number of People × Cost Rate per Hour per Person`

- For **goods**, the estimated cost is calculated as:
  `Estimated Cost = Quantity × Price per Piece`

These individual subtask costs are aggregated to compute the **Base Budget**, which serves as the foundation for project budgeting. Actual costs are entered later during task tracking, especially by team leaders, and can differ from initial estimates.

The system also computes **actual service costs** dynamically based on hours worked and people involved, which are updated during task progress reviews.

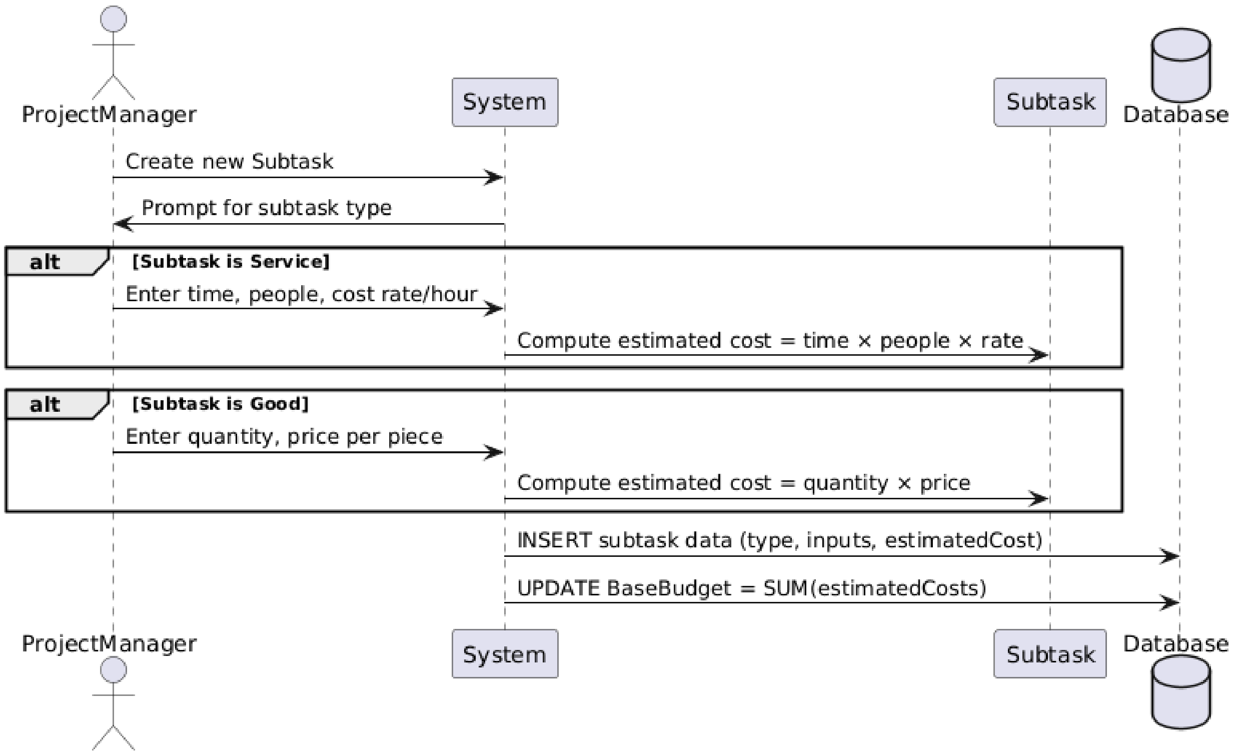

ProjectManager
System
Subtask
Database
Create new Subtask
Prompt for subtask type
alt
[Subtask is Service]
Enter time, people, cost rate/hour
Compute estimated cost = time × people × rate
alt
[Subtask is Good]
Enter quantity, price per piece
Compute estimated cost = quantity × price
INSERT subtask data (type, inputs, estimatedCost)
UPDATE BaseBudget = SUM(estimatedCosts)
ProjectManager
System
Subtask
Database

Diagram Name: Cost Computation Workflow

6.1.9 Budget Allocation

After computing all estimated subtask costs, the **Base Budget** is determined. To prepare for unexpected scenarios, a **Risk Buffer** is added to the base budget, resulting in the **Total Project Budget**.

Project Budget = Base Budget + Risk Buffer

During project execution, each task's budget is dynamically linked to the subtasks it contains. Team leaders update **actual costs**, allowing the system to track variances in real time.The system distinguishes between:

- **Project Savings**: When actual cost < estimated cost
- **Project Extra Cost**: When actual cost > estimated cost

Savings can be reallocated to cover overruns, and the project manager can intervene to approve extra costs or reallocate resources. This ensures flexible and responsive budget management.

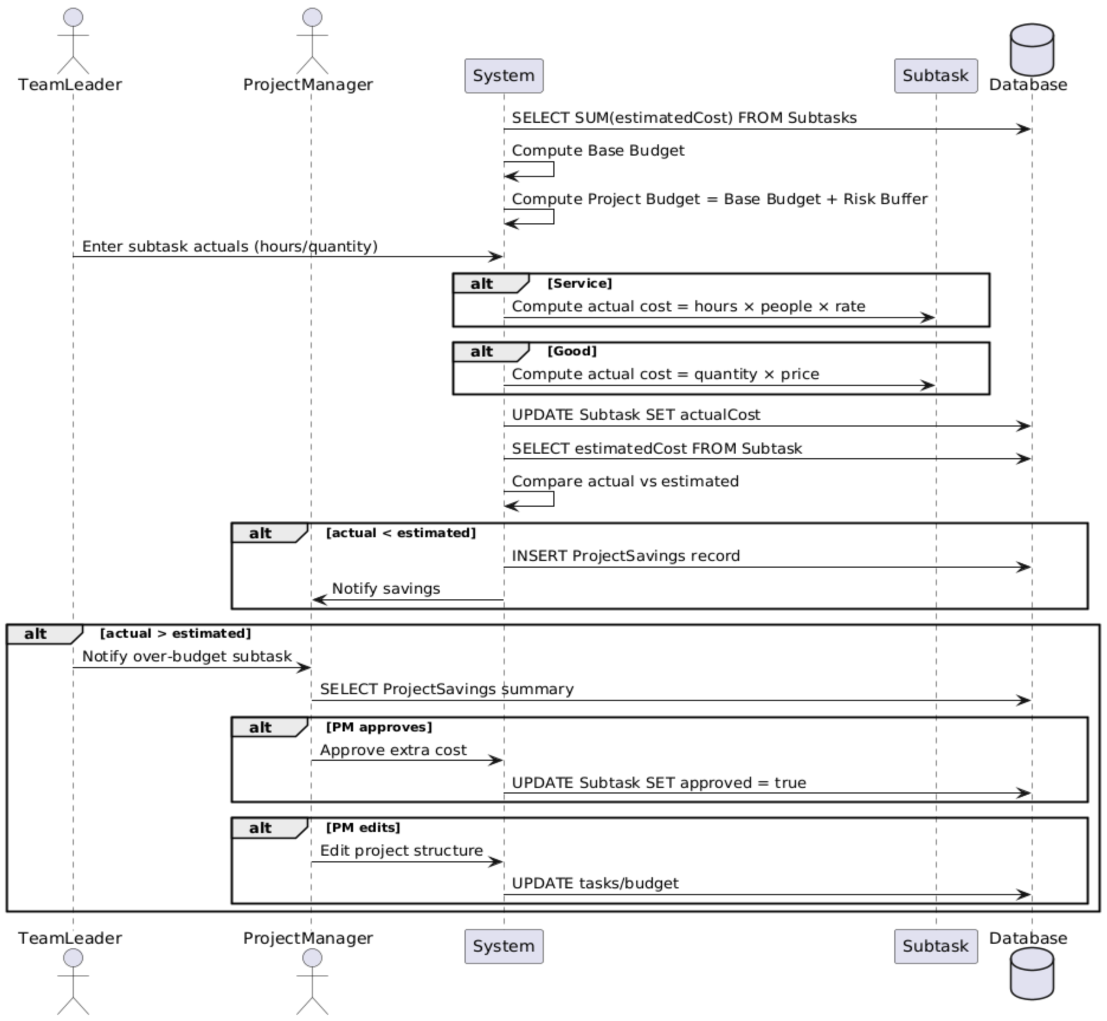

TeamLeader
ProjectManager
System
Subtask
Database
SELECT SUM(estimatedCost) FROM Subtasks
Compute Base Budget
Compute Project Budget = Base Budget + Risk Buffer
Enter subtask actuals (hours/quantity)
alt [Service]
Compute actual cost = hours × people × rate
alt [Good]
Compute actual cost = quantity × price
UPDATE Subtask SET actualCost
SELECT estimatedCost FROM Subtask
Compare actual vs estimated
alt [actual < estimated]
INSERT ProjectSavings record
Notify savings
alt [actual > estimated]
Notify over-budget subtask
SELECT ProjectSavings summary
alt [PM approves]
Approve extra cost
UPDATE Subtask SET approved = true
alt [PM edits]
Edit project structure
UPDATE tasks/budget

Diagram Name: Cost Computation Workflow

6.1.10 Discussion Board

When a project is created by the Project Manager, a private chat channel is automatically established with each assigned Team Lead for communication on strategy, timelines, and feedback. Both parties are notified of unread messages. When a team is created within the project, a group chat can be initiated that includes the Team Lead and all team members, enabling team-wide coordination. In addition, the Team Lead may create smaller, task-specific chats by selecting individual participants. These chats are limited to only the chosen members, with unread message indicators ensuring timely responses.

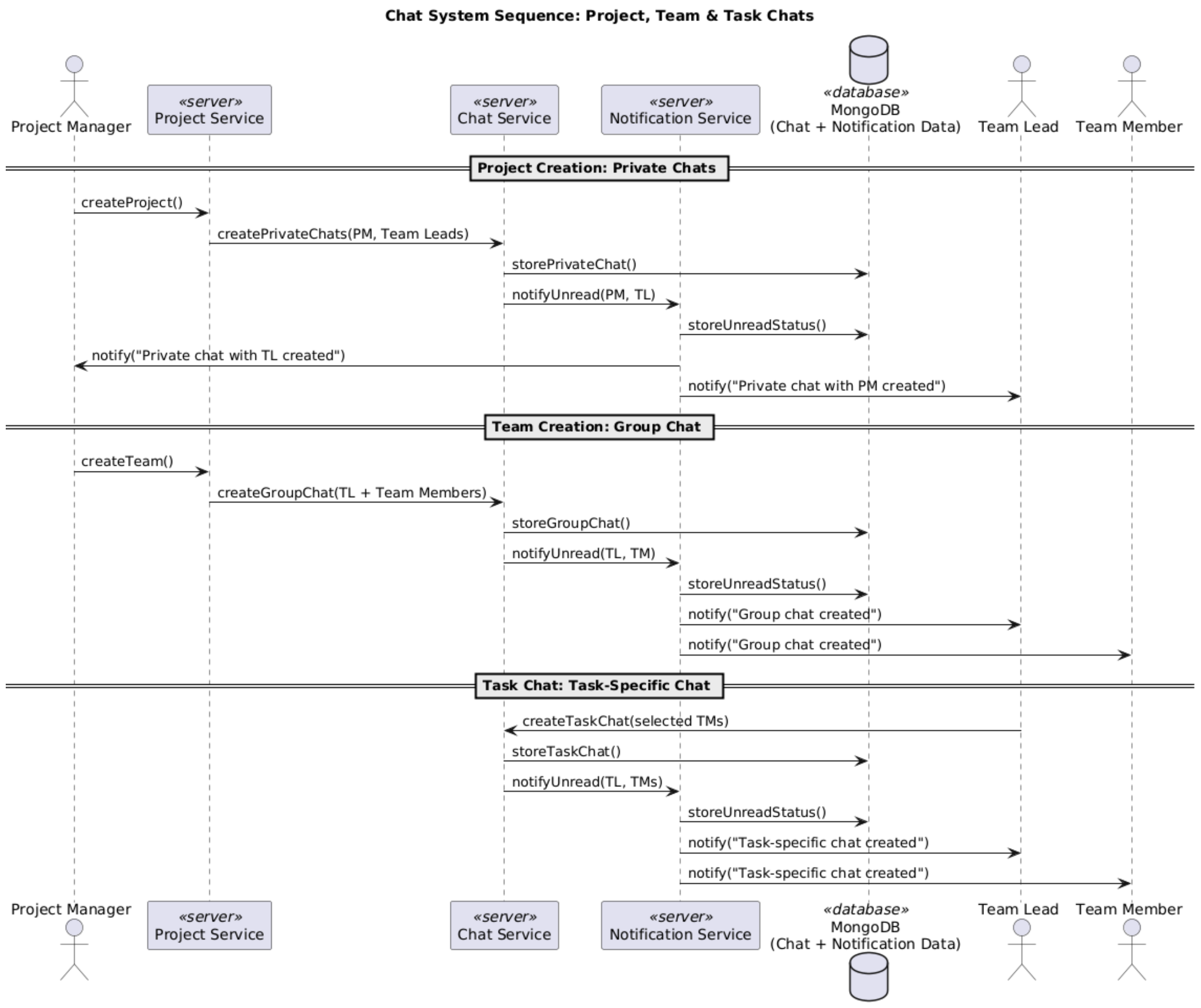

Chat System Sequence: Project, Team & Task Chats
Project Manager
«server» Project Service
«server» Chat Service
«server» Notification Service
«database» MongoDB (Chat + Notification Data)
Team Lead
Team Member
Project Creation: Private Chats
createProject()
createPrivateChats(PM, Team Leads)
storePrivateChat()
notifyUnread(PM, TL)
storeUnreadStatus()
notify("Private chat with TL created")
notify("Private chat with PM created")
Team Creation: Group Chat
createTeam()
createGroupChat(TL + Team Members)
storeGroupChat()
notifyUnread(TL, TM)
storeUnreadStatus()
notify("Group chat created")
notify("Group chat created")
Task Chat: Task-Specific Chat
createTaskChat(selected TMs)
storeTaskChat()
notifyUnread(TL, TMs)
storeUnreadStatus()
notify("Task-specific chat created")
notify("Task-specific chat created")

Diagram Name: Discussion Board Workflow

7. Deployment View

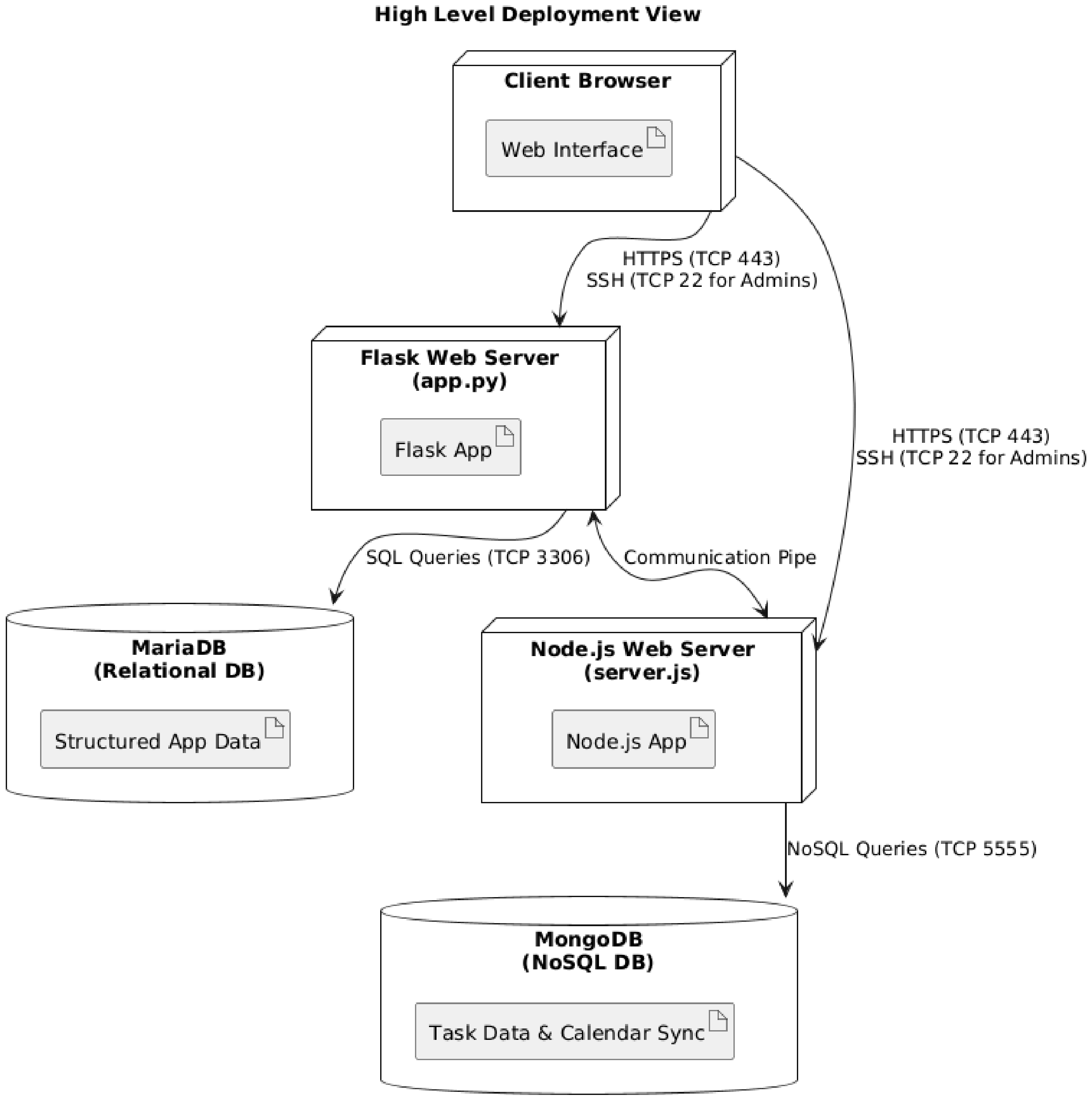


Diagram Name: High Level Deployment View

7.1 Components and Deployment Details

- **Flask Web Server (Application Layer)**
    - Deployed on a Python-compatible web server.
    - Hosts the core backend application (`app.py`), which handles routing, business logic, and API communication with the database layers.
    - Uses Flask extensions for ORM, validation, and Google Drive integration. Communicates with MariaDB.
    - Supports TCP/IP requests from the client browser (443, 22(for **Admins**))

- **Server.js (Application Layer)**
    - Deployed on a javascript-compatible web server.
    - Hosts the core backend application (`server.js`), which handles business logic, and API communication with the database layers.
    - Uses Node.js frameworks for ORM, validation, and Google Calendar API integration. Communicates with MongoDB.
    - Supports TCP/IP requests from the client browser ((443, 22(for **Admins**))
    - Communicates with Flask through a communication pipe.

- **MariaDB (Data Layer: Relational)**
  Stores structured application data such as:
    - User information and roles
    - Projects and teams
    - Tasks metadata (if applicable outside Mongo)
      Stakeholders, comments, and discussions
    - Ensures transactional integrity and supports SQL queries.
    - Accepts requests from TCP/IP ports (3306)

- **MongoDB (Data Layer: NoSQL)**
  Used specifically for:
    - Task creation and management
    - Task-related activities
    - Google Calendar integration (syncing task events)
    - Accepts requests from TCP/IP ports (5555)

  Chosen for its flexibility in handling unstructured task data and fast document retrieval for activity logs and calendar syncing.

7.2 Communication and Data Flow

The **Flask application** and the **Server.js** application serve as a server hub for all clients.

- They interact with:
    - **MariaDB** for structured queries and relationships.
    - **MongoDB** for CRUD operations related to tasks and calendar events.
- MongoDB enables fast reads/writes for task-related operations that benefit from a schema-less structure, whereas MariaDB supports the normalized structure needed for managing complex project relationships and users.

7.3 Hosting & Scalability Considerations

- The current deployment assumes all services (Flask server, Server.js, MariaDB, MongoDB) are hosted on a **single machine** or **local network**, suitable for development and testing.
- For production deployment:
    - Flask can be served behind a WSGI server like **Gunicorn** or **uWSGI**, with **Nginx** as a reverse proxy.
    - **MariaDB** and **MongoDB** can be containerized using Docker for easier deployment, backup, and scalability. MariaDB could be hosted through a service like **Supabase**.
    - Authentication credentials and API keys (e.g., for Google Calendar) should be securely stored using environment variables or a secrets manager.

8. Size and Performance

8.1 Current Restrictions

As mentioned in section 7.3, the system is running in a local development environment, which limits its scalability and restricts concurrent access to a single user or limited local users. This means performance testing under realistic multi-user conditions has not yet been conducted.

In a production environment, once the application is hosted on a proper web server, the system will be capable of handling multiple simultaneous users (8 ports) through proper server configuration and resource allocation.

The performance will also depend on the efficiency of request handling within the Flask app and Server.js, database query optimization (especially for MariaDB), and the responsiveness of the MongoDB-based task and calendar system. Load balancing and caching strategies can be explored in future iterations to support high user volume. All in all, we expect the size of our relational database to increase proportionally with the number of users, while the NoSQL database may grow faster due to task-related interactions and calendar data, which can be frequent and dense.

9. API Documentation

9.1 Google Calendar API

9.1.1 Overview

The Google Calendar Integration allows users to sync their tasks and deadlines with Google Calendar, providing a unified view of their schedule and reminders. It uses OAuth 2.0 for authentication and the Google Calendar API for event management.

9.1.2 Architecture

The Google Calendar API architecture is built around RESTful endpoints that allow developers to interact with Google Calendar data. It provides access to manage calendars, events, and settings, using resources like Event, Calendar, CalendarList, Setting, and ACL. The API is designed to be scalable and robust, supporting various programming languages and platforms.

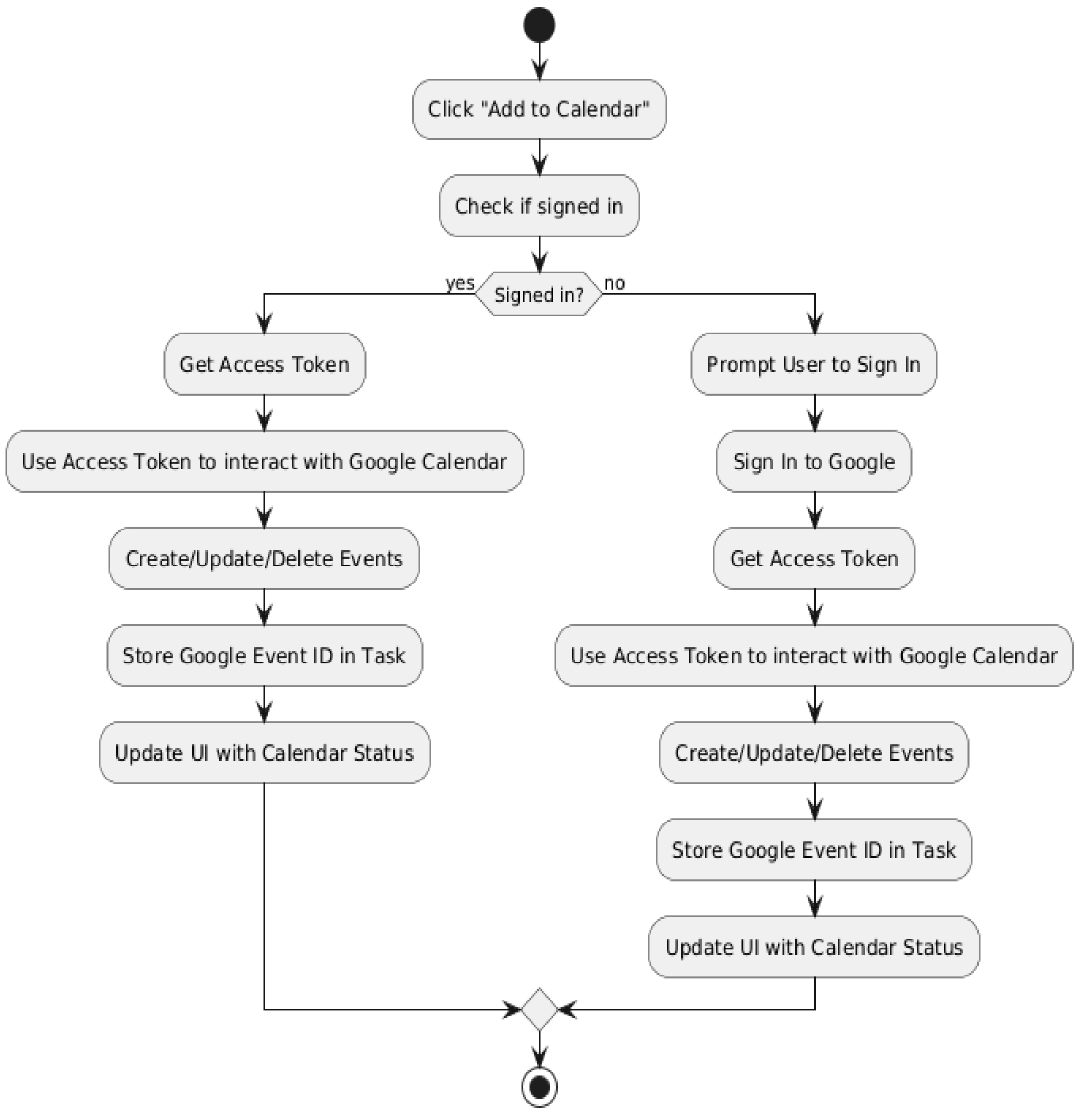
9.1.3 Authentication

Google Sign-In is used to authenticate users with their Google account.
OAuth 2.0 tokens are used to authorize the app to manage Google Calendar events on behalf of the user.

OAuth 2.0 Authentication Flow

1. The user clicks "Add to Calendar" to trigger Google Sign-In.
2. If the user is not signed in, they are prompted to log in.
3. Once authenticated, the system obtains an access token for the Google Calendar API.
4. The access token is used to interact with the Google Calendar API to create, update, and delete events.

Credential Storage:
`.env` includes all user info
SCOPES = ['https://www.googleapis.com/auth/calendar.readonly']



Click "Add to Calendar"
Check if signed in
Signed in?
yes
no
Get Access Token
Use Access Token to interact with Google Calendar
Create/Update/Delete Events
Store Google Event ID in Task
Update UI with Calendar Status
Prompt User to Sign In
Sign In to Google
Get Access Token
Use Access Token to interact with Google Calendar
Create/Update/Delete Events
Store Google Event ID in Task
Update UI with Calendar Status

Diagram Name: Google Calendar OAuth 2.0 Authentication Flow

9.1.4 Features

1. Add tasks to Google Calendar
2. Sync task deadlines
3. Set reminders
4. Update calendar events
5. Remove calendar events
6. Automatic event deletion on task deletion

9.1.5 API Functions

Function Name	Description
`loadSavedCredentialsIfExist()`	AReads previously authorized credentials from the save file
`saveCredentials(client)`	Serializes credentials to a file compatible with GoogleAuth.fromJSON @param {OAuth2Client} client
`authorize()`	Load or request or authorization to call APIs
`listEvents(auth)`	Lists the next 10 events on the user's primary calendar. @param {google.auth.OAuth2} auth An authorized OAuth2 client.

9.1.7 Permission Model

Role	Access Level	Description
Project Manager	Writer	Full access to task management system
Team Leader	Writer	Access to team tasks
Team Member	Writer - restricted	Completion access to assigned subtasks

9.1.8 Error Handling

1. Re-authentication on failure
2. Warning on invalid event data input
3. Permission errors logged but non-blocking

9.2 Google Drive API

9.2.1 Overview

This documentation covers the integration of Google Drive API in our Project Management System. The integration enables automatic creation of team and task folders in Google Drive, with appropriate access permissions for project managers, administrators, and team members.

9.2.2 Architecture

The Google Drive API integration in our project follows a service-oriented architecture. The integration is encapsulated in a dedicated module (`google_drive.py`) which provides a set of functions that handle various Google Drive operations such as authentication, folder creation, file uploads, and permission management.

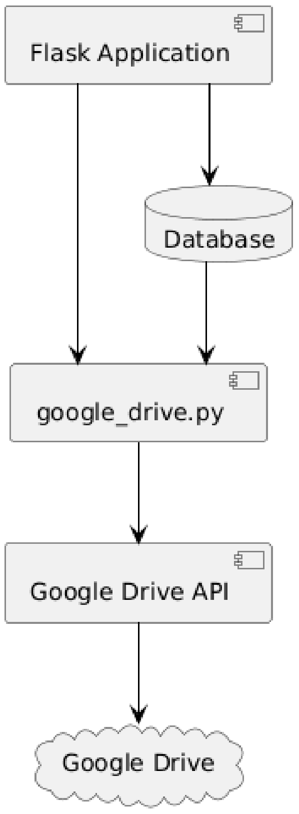


Diagram Name: Google Drive API Architecture

9.2.3 Authentication

OAuth 2.0 Authentication Flow

5. Check for existing credentials.
6. If none, initiate OAuth 2.0 flow using `credentials.json`.
7. Prompt user to grant permissions.
8. Store access token in `token.json`.
9. Use stored token for future requests.

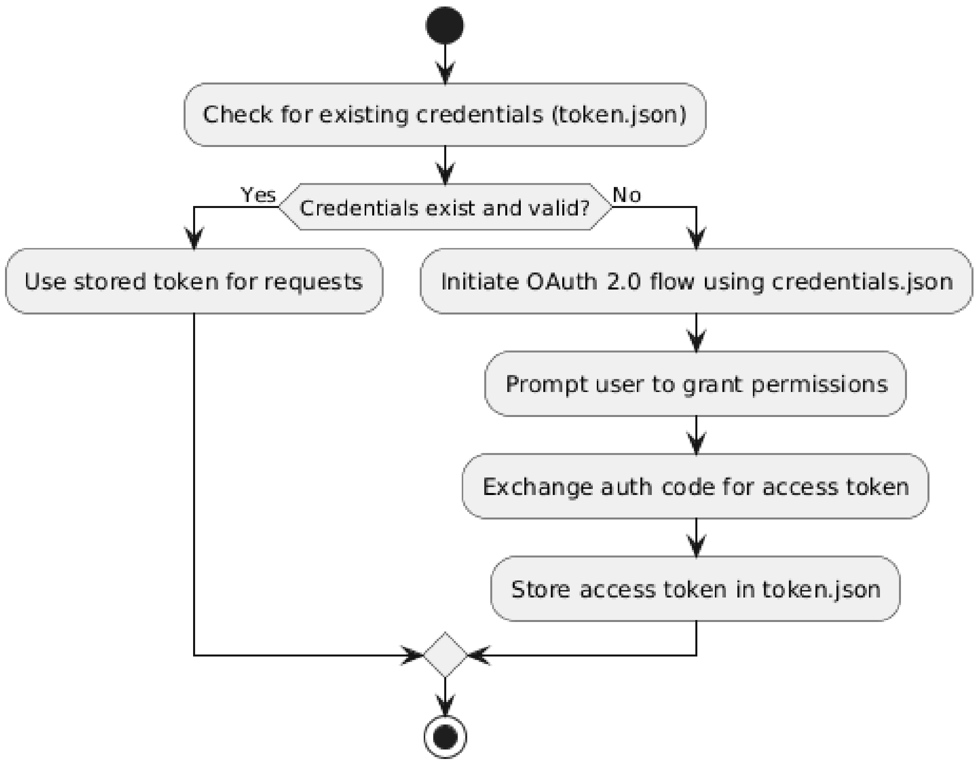

Check for existing credentials (token.json)
Yes
Credentials exist and valid?
No
Use stored token for requests
Initiate OAuth 2.0 flow using credentials.json
Prompt user to grant permissions
Exchange auth code for access token
Store access token in token.json

Diagram Name: Google Drive OAuth 2.0 Authentication Flow

Credential Storage(stored in .env for security purposes)

- `credentials.json` for client secrets
- `token.json` for OAuth tokens

SCOPES = ["https://www.googleapis.com/auth/drive.file"]

9.2.4 Features

7. Hierarchical folder structure
8. Role-based access control
9. File upload to task folders
10. File sharing with appropriate permissions
11. Task-linked folder management

9.2.5 API Functions

- Core Functions

Function Name	Description
`authenticate_google_drive()`	Authenticates and returns credentials
`create_folder()`	Creates a folder
`get_folder_by_name()`	Fetches folder ID by name
`upload_file()`	Uploads file to Drive
`share_file_with_user()`	Shares file/folder with a user

- Project-Specific Functions

Function Name	Description
`create_team_folder()`	Creates folder for a team
`create_task_folder()`	Creates folder under team folder
`upload_task_file()`	Uploads file to a task folder

9.2.6 Folder Structure

```
Google Drive Root
└── Team_{TeamID}_{TeamName}
        └── Task_{TaskName}
        ├── file1.pdf
        ├── file2.docx
```

9.2.7 Permission Model

Role	Access Level	Description
Project Manager	Writer	Full access to team/task folders
Admin	Writer	Full access to team/task folders
Team Member	Reader	Read-only access to relevant folders

9.2.8 Error Handling

4. Re-authentication on failure
5. File operation exceptions handled
6. Permission errors logged but non-blocking

10. Quality

The system is designed with a strong focus on key software quality attributes to ensure not only functional correctness but also long-term usability, maintainability, and security. Below is a breakdown of the most important quality factors and how they are addressed:

10.1. Security

- **Authentication & Authorization**:
  The system enforces role-based access control (RBAC). Users are authenticated via secure login sessions, and their roles (Admin, Project Manager, Team Leader, Participant) determine their access scope within the application.

- **Password Handling**:
  Passwords are stored in hashed form using a secure hashing algorithm *(werkzeug.security - generate_password_hash, check_password_hash)* .
  User credentials are never stored or transmitted in plaintext. Forgotten passwords are handled through **Admin** communication.

- **Form Validation & Input Sanitization**:
  All form inputs are validated both client-side and server-side to prevent injection attacks (e.g., SQL injection, XSS). Parameterized queries are used with MariaDB and MongoDB to avoid malicious inputs.

10.2. Maintainability

- **Modular Design**:
  The codebase follows a clear separation of concerns between the presentation (HTML/CSS/JS templates), application logic (Flask routes and control flow, Server.js), and data access layers (database interaction).

- **Consistent Naming & Structure**:
  Files and functions are organized according to standard conventions, making it easier for new developers to understand and contribute.

- **Version Control**:
  Git is used for tracking changes and collaboration. Feature branches and pull requests ensure that development is organized and reversible.

- **Code Documentation**:
  Docstrings and inline comments explain the purpose of functions, classes, and complex logic. A developer setup guide is also available to help onboard new contributors.

10.3. Reliability

- **Error Handling**:
  Try-except blocks are used to handle exceptions gracefully. Users receive informative error messages without exposing sensitive technical details.

- **Data Consistency**:
  Operations that modify the database are wrapped in transactions to ensure consistency. In case of failure, rollbacks are performed to prevent partial updates.

- **Logging & Monitoring**:
  Application logs track user actions, system errors, and database queries. These logs assist in debugging and are useful for ongoing maintenance.

10.4. Usability

- **User-Centered Design**:
  The UI is intuitive and responsive. Role-specific dashboards show only relevant data, and forms are designed to minimize user effort.

- **Accessibility**:
  Semantic HTML and keyboard navigation support are implemented to improve accessibility. Visual feedback is provided for user actions.

- **Feedback Loops**:
  Actions like submitting a form or updating a task provide immediate visual confirmation (e.g., success messages, loading indicators).

10.5. Testability

- **Unit & Integration Testing**:
  Critical parts of the application (e.g., login logic, task creation, team assignments) have associated tests.

- **Mocking External Services**:
  Where APIs are involved (e.g., Google Calendar), mock interfaces are used to simulate calls during testing.

- **Continuous Integration**:
  The system is configured locally on different machines to run tests and lint code regarding different features before merging.

- **Traceability Matrix**
  Requirements related to the Google Drive API integration will be marked as green, those related to the Google Calendar API integration will be marked as orange. Requirements detailing User Role based functionality will be marked with red.

Traceability Matrix

Requirement	Feature / Functionality	Test Case ID	Test Description	Status
Task Creation	Create new tasks with description, timing, and priority	TC001	Test task creation with valid input data	☑
Task Creation	Create new tasks with description, timing, and priority	TC002	Test task creation with invalid data (missing fields, incorrect format)	☑
Task Editing	Update existing tasks	TC003	Test task update with valid changes	☑
Task Editing	Update existing tasks	TC004	Test task update with invalid changes (e.g., invalid time format)	☑
Task Deletion	Delete tasks	TC005	Test task deletion and ensuring it is removed from the database	☑

Task Status Management	Set task status (In Progress, Completed, Frozen)	TC006	Test task status change between 'In Progress', 'Completed', and 'Frozen'	☑
Task Filtering & Sorting	Filter and sort tasks by priority, status, and date	TC007	Test filtering tasks by priority, status, and date	☑
Task Search	Search tasks by description	TC008	Test search functionality for tasks based on description	☐
Google Calendar Integration	Sync task events with Google Calendar	TC009	Test adding a task to Google Calendar	☑
Google Calendar Integration	Sync task events with Google Calendar	TC010	Test task removal from Google Calendar after task deletion	☑
Sub-Task Creation	Create Sub-Tasks for tasks with start/end time, description, and priority	TC011	Test Sub-Task creation with valid input data	☑

Sub-Task Creation	Create Sub-Tasks for tasks with start/end time, description, and priority	TC012	Test Sub-Task creation with invalid data (missing fields, incorrect format)	☑
Sub-Task Completion	Mark Sub-Task as completed	TC013	Test marking an Sub-Task as completed	☑
Sub-Task Status Management	Set Sub-Task priority and manage its status	TC014	Test changing Sub-Task priority and marking it as completed	☐
Progress Tracking	Visual representation of task and Sub-Task progress	TC015	Test progress bar display and update during task and Sub-Task completion	☑
Google Calendar Event Updates	Update events in Google Calendar when task is modified	TC016	Test updating Google Calendar event when task details change	☐
Google Calendar Authentication	Handle authentication errors, token expiration, and API limits	TC017	Test OAuth authentication and token expiration handling	☑

Google Drive Integration	Create folders and set permissions on Google Drive	TC018	Test Google Drive folder creation and access permissions	☑
User Login	Handle authentication errors and password hashing	TC019	Test Login Feature with valid and invalid data	☑
Role-Based Dashboard	Display different dashboard based on User Role	TC020	Login with different users and compare dashboards	☑
Admin functionality	Add users, delete users, create teams, assign team members, assign roles	TC021	Test relevant flask methods and compare with DB queries	☑
Project Creation (Project Manager)	Create project, define charter, define stakeholders	TC022	Test project creation with valid and invalid data, using valid and invalid user logins	☑
Delete Project (Project Manager)	Delete all project data	TC023	Test deletion feature	☑

Approve Project Charter (Stakeholder)	**Display project charter and approve**	**TC023**	**Test approval of charter**	☑

10.6. Portability

- **Minimal Dependencies**:
  Only essential libraries are included, and they are listed for reproducibility.
- **Cross-Platform Compatibility**:
  Works on Linux, macOS, and Windows development environments.