ISO-IEC-IEEE-42010
System Architecture Design

1. Executive Summary

Project Name - Planning and Logistics for Unified Project Management (PLUMP)
Objective - To design and implement a web based issue tracking and project management tool allowing employees to streamline and manage their goals.
Stakeholders - Project Managers, Team Members, Clients, Professor Aguado, Joaquin
Core Features -

- Project, Task, Sprint Management
- User roles and Permissions
- Kanban Boards
- Notifications
- Activity Logs

2. System Architecture Description

2.1 Stakeholders and Respective Concerns

Stakeholders	Concerns
Product Owner	Scalability
Team Members	Performance + Usability
Security Team	Security + Authentication

2.2 Viewpoints

Functional View: Task lifecycle, user flows

Deployment View: Microservices + frontend hosted via Docker containers on cloud (e.g., AWS/GCP)

Development View: Modular monorepo with Git branching

Data View: PostgreSQL schema, object storage for attachments

Security View:

- Authentication
- HTTPS everywhere
- Role-Based Access Control
- Input validation and sanitation

2.3 Architecture Views

Context Diagram

```
User -->|Uses| Frontend
Frontend -->|API Calls| Backend
Backend -->|Reads/Writes| Database
Backend -->|Stores| ObjectStorage
Backend -->|Sends| NotificationService
```

Component Diagram

```
Frontend -->|REST API| Backend
Backend --> AuthService
Backend --> ProjectService
Backend --> TaskService
Backend --> NotificationService
Backend --> FileStorageService

AuthService --> Database
ProjectService --> Database
TaskService --> Database
NotificationService --> MessageQueue
FileStorageService --> ObjectStorage
```

Creating a Task: Sequence Diagram

```
participant U as User
participant FE as Frontend
participant BE as Backend
participant DB as Database

U->>FE: Fill task form
FE->>BE: POST /tasks
BE->>DB: INSERT INTO tasks
DB-->>BE: Success
BE-->>FE: 201 Created
FE-->>U: Show success message
```

2.4 Architectural Decisions and Reasoning

Frontend: React + TailwindCSS + Vite

Why choose React?

- **Industry standard** for modern frontend development.
- Huge ecosystem and community support.
- Excellent support for **component-based architecture**, reusable UI.
- Works well with libraries like Redux, React Router, etc.

Compared to alternatives:

- vs. Angular: React has a lighter learning curve and better flexibility.
- vs. Vue: React has broader job market and library ecosystem.

Why choose Tailwind CSS?

- Utility-first: No need to write custom CSS classes unless needed.
- Enforces **design consistency** across your UI.
- Extremely customizable and works great with dark/light mode, responsiveness, and themes.

Compared to alternatives:

- vs. Bootstrap: Tailwind gives more **design flexibility**, not locked into prebuilt styles.
- vs. plain CSS/SCSS: Much faster styling and better scalability.

Why choose Vite?

- Fast dev server and builds (thanks to native ES modules).
- Out-of-the-box support for **hot module replacement** and optimized React support.
- Modern tooling that's simpler and faster than Webpack.

Compared to alternatives:

- vs. CRA (Create React App): Vite is faster, lighter, and more customizable.

Backend: Node.js (NestJS) + NEXT.js + REST API

Why choose Node.js?

- Built on JavaScript → **shared language** between frontend and backend.
- Fast I/O and event-driven → great for handling many API requests efficiently.

Why choose NestJS?

- Scalable and **structured framework** built on top of Express.
- Inspired by Angular → **strong typing with TypeScript**, dependency injection, and modular architecture.
- Ideal for building enterprise-grade backends with **clean code and maintainability**.

Compared to alternatives:

- vs. Express.js alone: NestJS is opinionated and easier to scale.
- vs. Django/Rails: NestJS is better for JS/TS stacks, more flexible API design.

Why include Next.js?

- Can be used for **server-side rendering (SSR)** or API routes.

- You might use it **alongside or ahead of NestJS** if you want SEO-ready pages or server-rendered components.
- Helpful for **hybrid rendering**, dynamic content, and performance optimization.

Why choose REST API?

- Universally understood, easy to debug, and well-supported across platforms.
- Simple for CRUD operations and great if you're not doing real-time updates or complex graph traversals.

Compared to alternatives:

- vs. GraphQL: REST is easier to implement and test; less overhead for simple APIs.
- vs. gRPC: REST is more web-friendly and doesn't require extra tooling for most frontend use.

Database: Prisma

Why choose Prisma?

- **Type-safe ORM** with autocompletion and database schema syncing.
- Modern syntax and seamless integration with PostgreSQL, MySQL, SQLite, etc.
- Easy database migrations and clear data modeling with `schema.prisma`.

Compared to alternatives:

- vs. Sequelize: Prisma is more modern, less verbose, and fully TypeScript-native.
- vs. TypeORM: Prisma is faster, has more active development, and better DX (developer experience).

Authentication: JWT with Refresh Tokens

Why choose JWT + Refresh Tokens?

- **Stateless authentication**, great for REST APIs.
- Tokens can be stored client-side (e.g., in httpOnly cookies or localStorage).
- Refresh tokens improve **security and user experience** by allowing silent re-authentication.

Compared to alternatives:

- vs. Session-based auth: JWT is scalable, especially for APIs and microservices.
- vs. OAuth2 alone: JWT is simpler for internal apps or systems not relying on third-party login providers

Summary Table

Layer	**Choice**	**Why it's better**
Frontend	React + Tailwind + Vite	Fast dev, modern design, reusable components
Backend	Node.js + NestJS + (Optional Next.js SSR)	Type-safe, scalable, and well-structured
API	REST	Simpler and cleaner for CRUD apps
Database	Prisma	Type-safe ORM with fast migrations and better DX
Auth	JWT + Refresh Tokens	Secure, scalable, and stateless login management

3. Modules and Features

3.1. Authentication and Authorization

- JWT login/register/reset
- Role-based access control (Admin, Manager, Developer)

3.2. User and Team Management

- User profiles
- Team creation/invite
- Permissions management

3.3 Project Management

- Create/edit/delete projects
- Assign team members
- View team dashboards

3.4 Issue and Task Tracking

- Set priority, status, labels
- Drag-and-drop Kanban board
- Sprint assignment

3.5 Admin Panel

- User roles
- Data insights
- System logs

4. Database Schema (Prisma)

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = "file:./dev.db"
}

enum Role {
  USER
  ADMIN
  MANAGER
}

enum Type {
  INTERNAL
  EXTERNAL
}

enum Phase {
  INITIATING
  PLANNING
  EXECUTING
  MONITORING_CONTROLLING
}

enum Status {
  PROPOSED
  IN_PROGRESS
  COMPLETED
  APPROVED
  CANCELED
}




enum HealthColours {
  GREEN
  YELLOW
  RED
}


model User {
  userID          Int       @id @default(autoincrement())
  firstName       String
  lastName        String
  email           String    @unique
  phone           String
  address         String
  unit            String
  unitManager     String
  activationDate  DateTime  @default(now())
  active          Boolean   @default(false)
  primaryRole     Role      @default(USER)
  type            Type      @default(INTERNAL)

  // Relationships
  teams           Team[]    // Many to Many: one user can be in many teams
  tasks           Task[]
}

model Team {
  teamID    Int           @id @default(autoincrement())
  name      String
  createdOn DateTime       @default(now())

  //Relationships
  users     User[]         // Many-to-Many: one team can have many users

  // Team–Project remains as one-to-one.
  project   Project?       @relation("TeamProject")
}

model Project {
  projectID   Int          @id @default(autoincrement())
  title       String
  status      Status
  phase       Phase
  //projectLead User?

  // // Foreign Keys
  teamID      Int          @unique  // Enforces one to one
  // healthID    Int          @unique
  // budgetID    Int          @unique
  // dateID      Int          @unique

  // Relationships
  team        Team?        @relation("TeamProject", fields: [teamID], references: [teamID])
  health      HealthStatus? //@relation(fields: [healthID], references: [healthID])
  budget      Budget?       //@relation(fields: [budgetID], references: [budgetID])
  dates       ProjectDates? //@relation(fields: [dateID], references: [dateID])

  tasks       Task[]
```

```
}

model Budget {
  budgetID      Int    @id @default(autoincrement())
  projectID     Int    @unique
  totalBudget   Float
  actualCost    Float
  forecastCost  Float

  // Relationships
  project     Project @relation(fields: [projectID], references: [projectID])
}

model HealthStatus {
  healthID  Int    @id @default(autoincrement())
  projectID Int    @unique
  scope     HealthColours @default(GREEN)
  schedule  HealthColours @default(GREEN)
  cost      HealthColours @default(GREEN)
  resource  HealthColours @default(GREEN)
  overall   HealthColours @default(GREEN)

  // Relationships
  project   Project @relation(fields: [projectID], references: [projectID])
}

model ProjectDates {
  dateID           Int      @id @default(autoincrement())
  targetDate       DateTime
  startDate        DateTime
  actualCompletion DateTime?

// Relationships
  project          Project  @relation(fields: [projectID], references: [projectID])
  projectID        Int      @unique
}
model Task {
  taskID            Int    @id @default(autoincrement())
  projectID         Int
  dateID            Int    @unique
  title             String
  percentageComplete Float
  priority          String
  userID            Int
  details           String
  status            String

  // Relationships
  project      Project @relation(fields: [projectID], references: [projectID])
  dates        TaskDates? @relation(fields: [dateID], references: [dateID])
  user         User? @relation(fields: [userID], references: [userID])
}

model TaskDates {
  dateID          Int      @id @default(autoincrement())
  targetDate      DateTime
  startDate       DateTime
  actualCompletion DateTime?

// Relationships
  task            Task?
  taskID          Int      @unique }
```

5. Sample User Stories

- **As a user**, I want to create a new task so that I can track my work.

- **As a project manager**, I want to assign team members to projects.

- **As a developer**, I want to comment on tasks so I can communicate blockers.

Specific Information on the Related Components and and the Views Affected with each user story is given in the IEEE-1016 document under 10. Mapping of Sample User Stories to exact System Components and Flows

6. Documentation

README, API_DOCS, Contribution, Schema can all be found on the team github.

7. Future Enhancements

- AI assistant for auto-tagging and sprint planning
- Mobile app (React Native)
- GitHub/Slack integrations
- Public/Private project visibility
- Plugin system for custom extensions