Project 3 Report: Team 16

Task 1

Functional and Non-functional Requirements

Functional Requirements

Functional requirements define the specific behaviors and functionalities the system must support.

Unified Calendar Integration

- Aggregates events from Google Calendar, Outlook, Apple Calendar, and LMS (Moodle).
- Provides visual conflict indicators and priority-based event highlighting.
- Allows custom reminders and notifications.

AI-Powered Note-Taking System

- Supports both handwritten and typed notes.
- AI-based tagging and full-text search with OCR capabilities.
- Synchronization with lecture slides for contextual learning.

AI Integration for Learning Enhancement

- AI-generated quizzes based on course materials.
- AI-powered summarization of notes and lecture recordings.
- Concept mapping for topic visualization and retention.
- Performance analysis via task completion rates and quiz scores.

Resource Library

- Centralized repository for study materials, past papers, and reference documents.
- Categorized and searchable content with AI-based recommendations.

User Authentication and Personalization

- Authentication flow must comply with OAuth 2.0 standards and ensure secure handling of tokens and user data.
- Customizable user profiles & themes.

Non-Functional Requirements

Non-functional requirements define the quality attributes and architectural constraints of the system.

Performance and Scalability

- System must process user requests within **500ms**.
- New data must sync across devices within **≤5 seconds**.
- Supports at least **100 simultaneous users** and handle **100 API requests per second**.
- Storage must dynamically scale to support **1GB per user**.

Security and Compliance

- Must enforce **HTTPS** for all communications.
- Secure APIs with **OAuth 2.0 authentication**.

Usability and Accessibility

- Web application accessible on all modern browsers with an **intuitive UI**.
- Onboarding process must complete within **≤5 minutes**.
- 100% screen-reader accessible with adjustable themes (**light** and **dark**).

Maintainability and Modularity

- Modular architecture with **microservices** for AI, calendar, and core functionalities.
- Independent updates possible without affecting the entire system.
- Uses **asynchronous operations and message queues** to reduce latency.

Reliability and Availability

- System must maintain **99.9% uptime**.
- Automatic failover mechanisms and redundant data backups.

Architecturally Significant Requirements

1. **Scalability & Performance** → Ensures smooth operation under high user loads. Constrained by the Firebase Spark plans limitations.
2. **Security & Compliance** → Protects user data and aligns with regulations.
3. **AI-Powered Features** → Differentiates the product in the ed-tech space.
4. **Usability & Accessibility** → Enhances user experience for diverse student needs.
5. **Maintainability & Modularity** → Allows future enhancements and adaptability.

Subsystem Overview

In this section, we list the main subsystems of SOS and how they work together to help students organize their academic lives, especially focusing on note management and exam preparation.

User Authentication & Security Subsystem

- **Role:** Manages user identity, secure access to the system, and protects user data. Acts as the gatekeeper for all other subsystems.
- **Functionality:**
  - Handles user registration and login using Single Sign-On (SSO).
  - Manages user sessions and authorization, ensuring users only access their data and system functionalities based on their roles (though roles are currently student-focused).
  - Enforces security policies (like password complexity and data encryption hashing).
- **Integration with other subsystems:**
  - **All subsystems:** Every subsystem relies on this subsystem to authenticate and authorize users before granting access to features or data. For example, before a user can access the Note-Taking

Subsystem, they must be authenticated by the User Authentication & Security Subsystem.

Resource Library Subsystem

- **Role:** Provides a central repository to store, organize, and manage all types of study materials. This is the core for storing notes, lecture slides, and other resources.
- **Functionality:**
  - Allows users to upload and categorize study materials related to their courses (notes, documents, PDFs, links, past papers, etc.).
  - Provides a structured organization system (e.g., by course, topic, date, type).
  - It offers robust search functionality to help you find materials quickly.
  - Implements AI-based tagging and recommendations to help users discover relevant resources.
- **Integration with other subsystems:**
  - **Note-Taking Subsystem:** Notes created in the Note-Taking Subsystem are stored and managed within the Resource Library. Notes can be linked to specific courses or resources within the library.
  - **AI Learning & Assessment Subsystem:** Utilizes resources from the library to generate quizzes, summaries, and concept maps.

Note-Taking Subsystem

- **Role:** Provide the tools for students to create, edit, and enhance their notes, both during lectures and for personal study.
- **Functionality:**
  - Supports both typed note input and handwritten note input (via tablet, stylus, or uploaded images).
  - Offers rich text editing features for typed notes (formatting, lists, headings, etc.).
  - Implements OCR (Optical Character Recognition) on handwritten notes to make them searchable.

    - Allows users to link notes to specific lecture slides or resources from the Resource Library for contextual learning.
    - Provides AI-based tagging suggestions to help categorize notes effectively.
- **Integration with other subsystems:**
    - **Resource Library Subsystem:** Stores and retrieves notes from the Resource Library. Notes are a type of resource within the library.
    - **AI Learning & Assessment Subsystem:** Provides note content to the AI for summarization and quiz generation.

Calendar Management Subsystem

- **Role:** Aggregates all important events from various sources into a unified calendar view, ensuring students don't miss deadlines or important events.
- **Functionality:**
    - Integrates with external calendar services (Google Calendar, Outlook Calendar, Apple Calendar) and the Moodle Calendar.
    - Displays events from all integrated calendars in a unified SOS view.
    - Identifies and visually highlights scheduling conflicts.
    - Allows users to prioritize academic events and set custom reminders and notifications.

AI Learning & Assessment Subsystem

- **Role:** Leverages Artificial Intelligence to enhance the learning process, particularly for study and exam preparation, using notes and course resources.
- **Functionality:**
    - **AI-Generated Quizzes:** Creates quizzes based on notes, lecture materials, and resources in the Resource Library. Quizzes can be customized by topic, difficulty, etc.
    - **AI-Powered Summarization:** Summarizes notes, lecture recordings (if integrated in the future), and lengthy documents from the Resource Library, providing concise overviews.
- **Integration with other subsystems:**

- **Resource Library Subsystem:** Retrieves notes and course materials to be used for quiz generation, summarization, and concept mapping.
- **Note-Taking Subsystem:** Utilizes the content of notes for AI processing.
- **User Authentication & Security Subsystem:** Tracks user performance data securely.

These subsystems are designed to work in concert to provide a comprehensive Student Operating System. The core workflow for a user-focused on note organization and exam preparation would be:

1. **Import Notes:** The user imports existing notes (typed or scanned) into the **Resource Library**.
2. **Organize Notes:** The user organizes notes within the Resource Library, potentially using AI-suggested tags from the **Note-Taking Subsystem**.
3. **Enhance Notes:** User can use the **Note-Taking Subsystem** to further edit, link, and enhance their notes.
4. **Study with AI:** User utilizes the **AI Learning & Assessment Subsystem** to generate quizzes and summaries from their notes and related course materials in the **Resource Library**, actively preparing for quizzes and exams.
5. **Stay Organized & Focused:** The **Calendar** ensures the user is aware of deadlines and follows them properly.

Task 2

Stakeholder Identification

Stakeholders and Their Concerns

Stakeholder	Concerns
Students (End Users)	Usability, accessibility, performance, integration with existing tools, AI-powered features, and data privacy.
University Faculty	Ease of course material management, quiz assistant, and academic engagement.

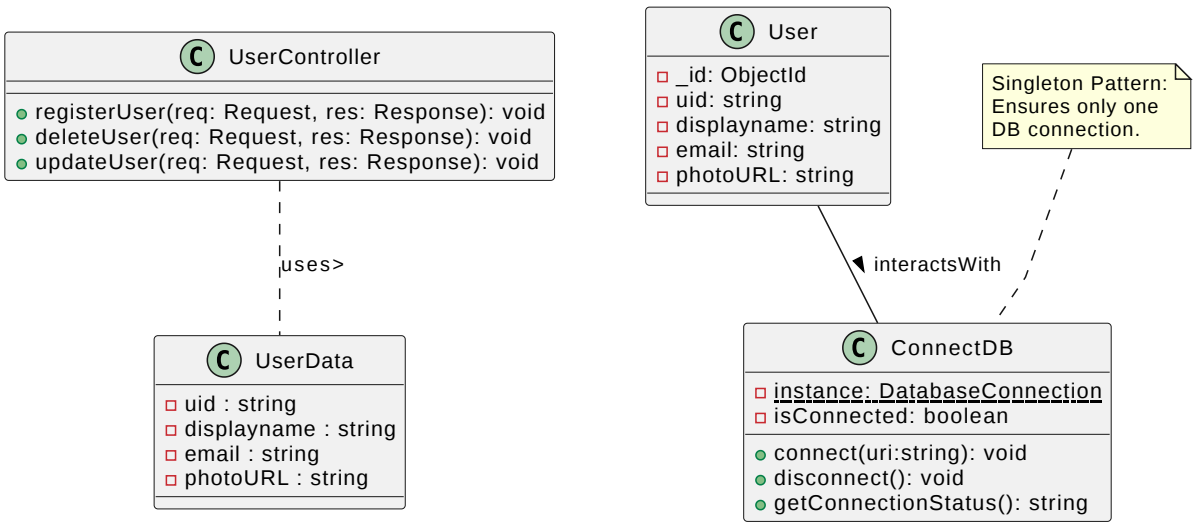
Developers & Architects	Modular architecture, AI model integration, and security.
Maintenance Team	Scalability, maintainability, system reliability, API security, and extensibility.

Viewpoints and Views Addressing Stakeholder Concerns

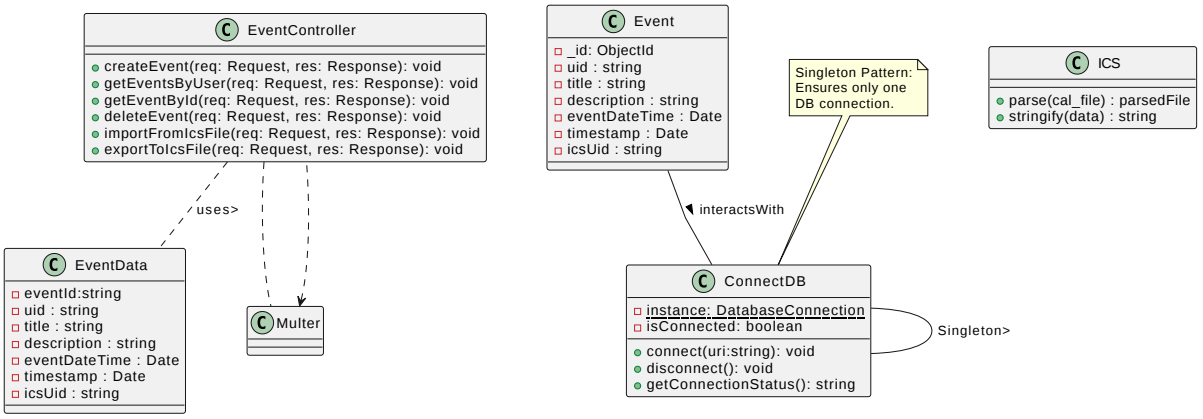
Viewpoint	View
Logical Viewpoint	Class diagrams
Process Viewpoint	Sequence diagrams
Deployment Viewpoint	Deployment diagrams
Security Viewpoint	Access control diagrams
Operational Viewpoint	Monitoring dashboards

UML Diagrams

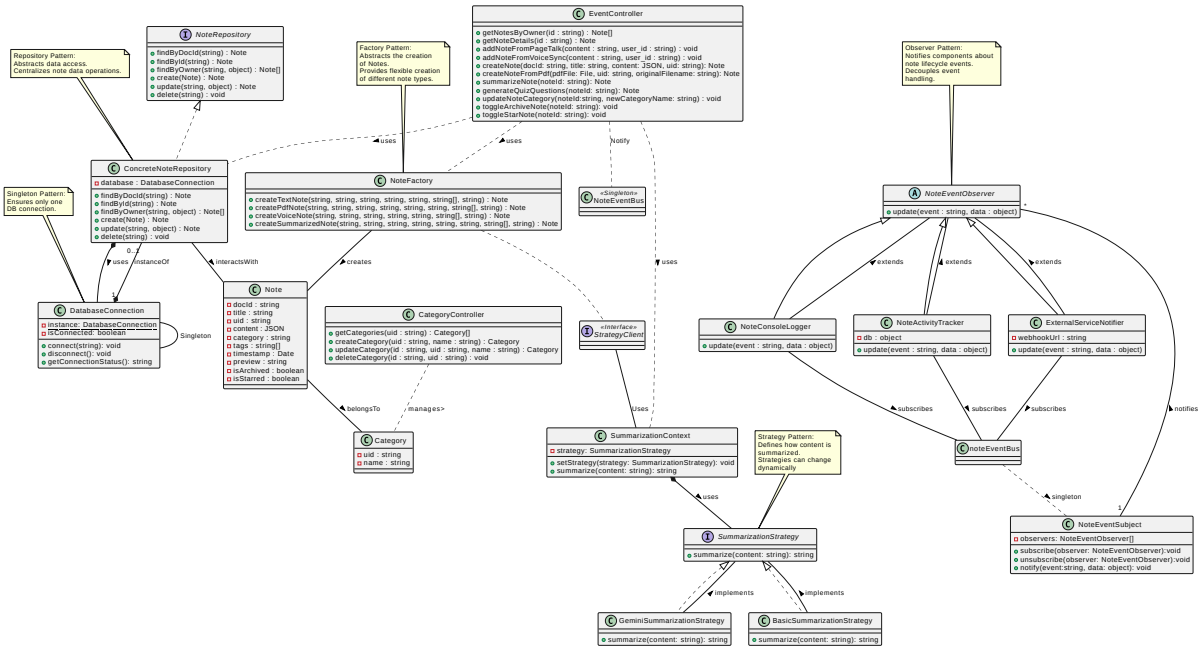
- User Subsystem



- Event Control Subsystem

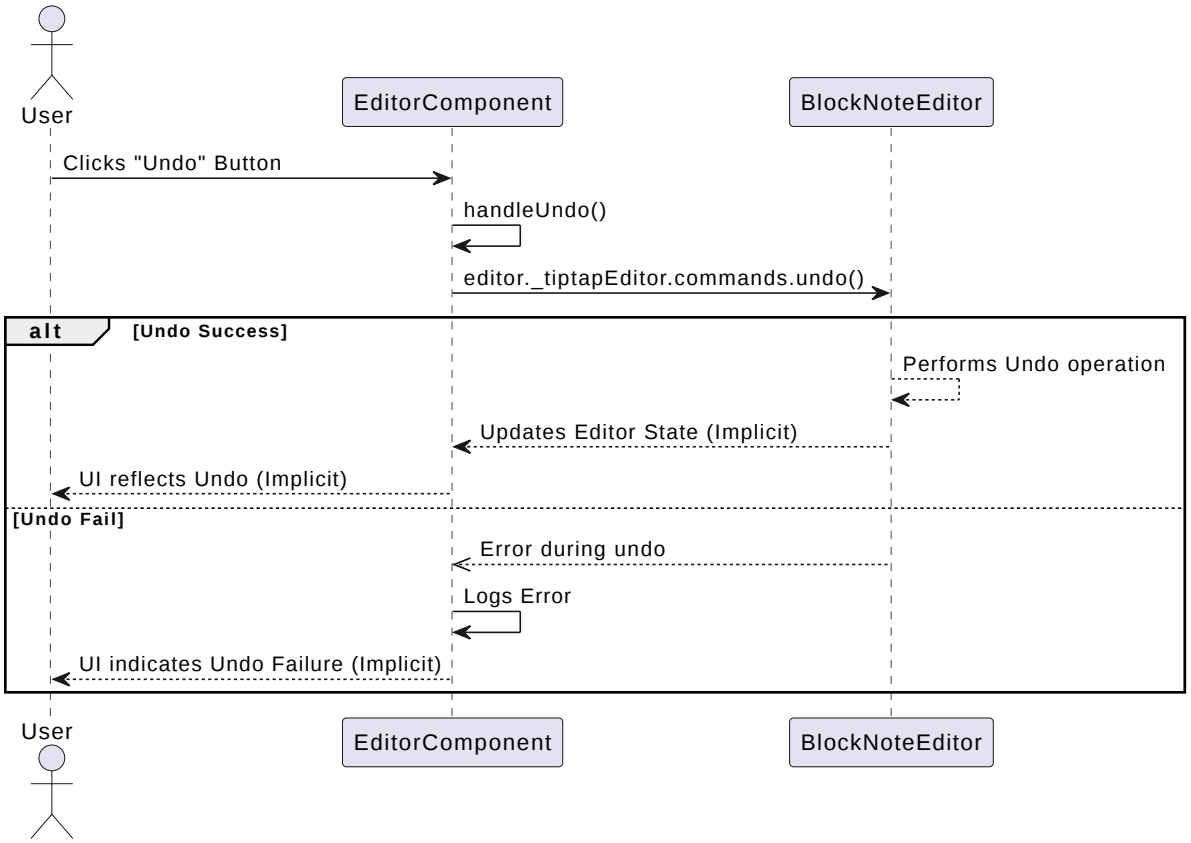


- Note taking Subsystem

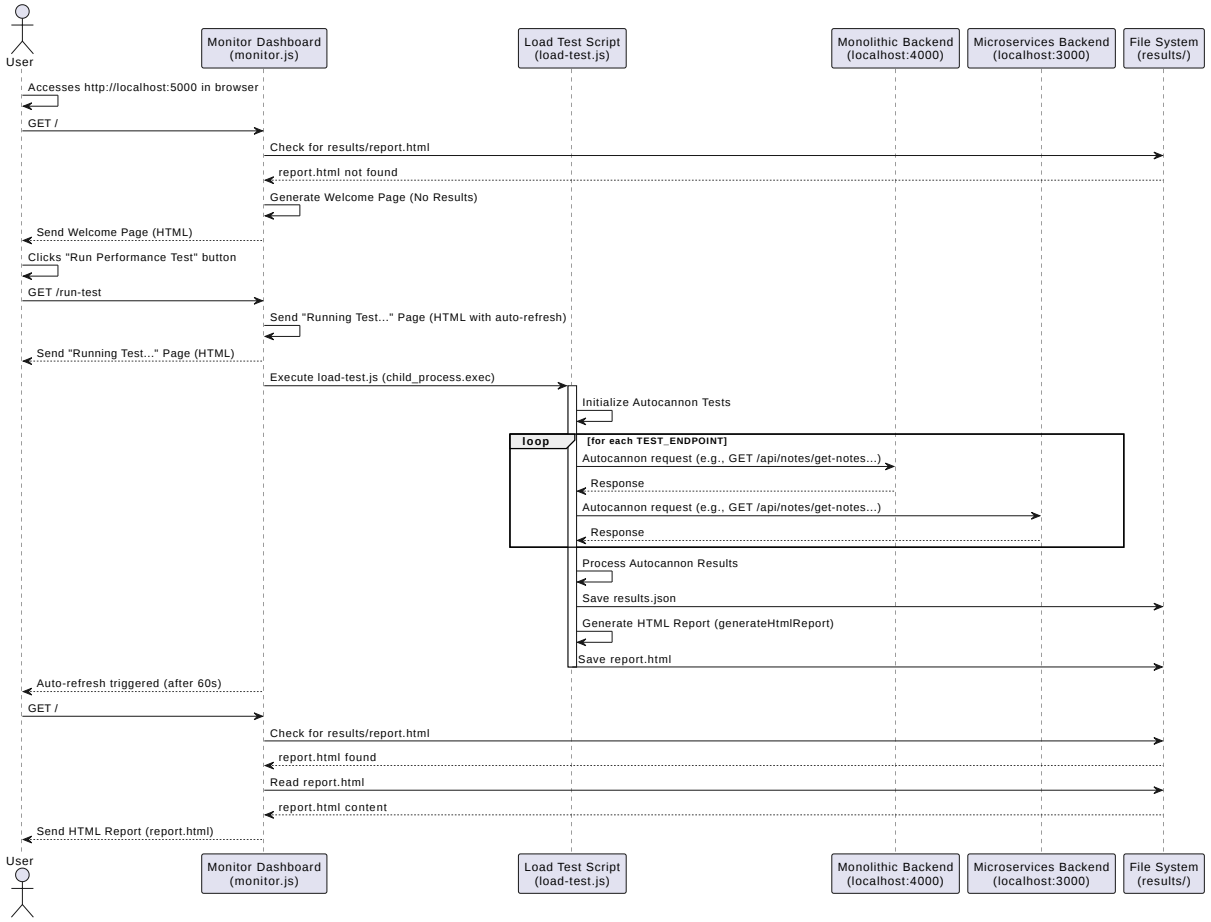


Sequence Diagrams

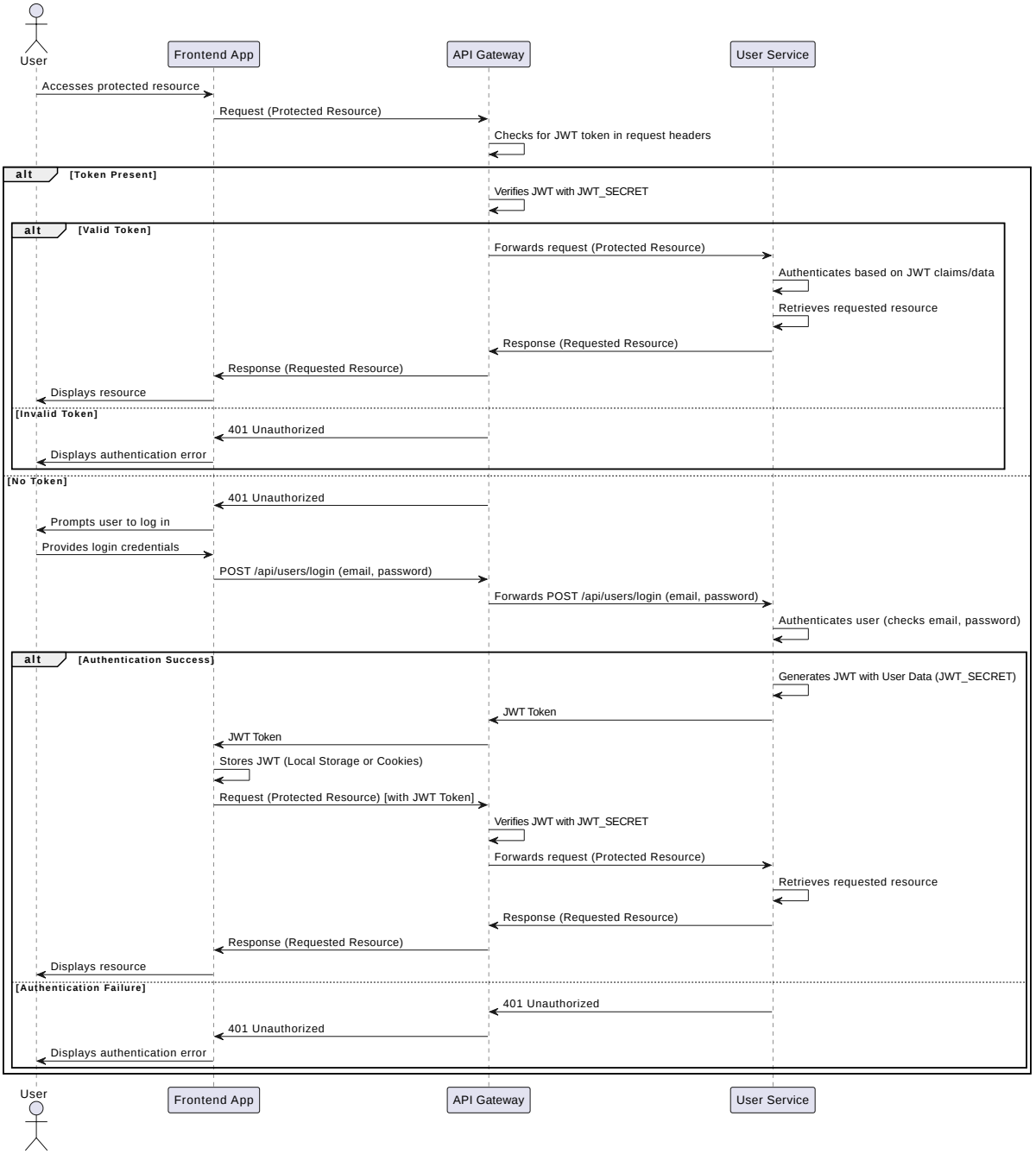
- Undo sequence diagram



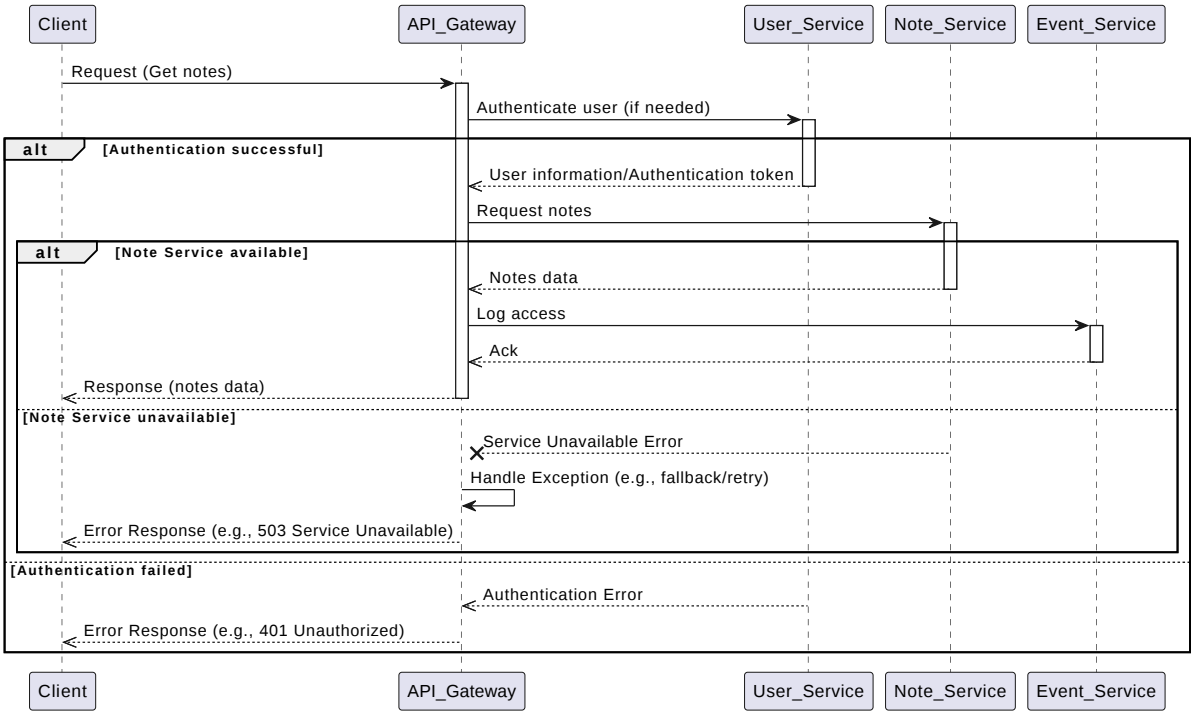
- Built-in Monitoring sequence diagram



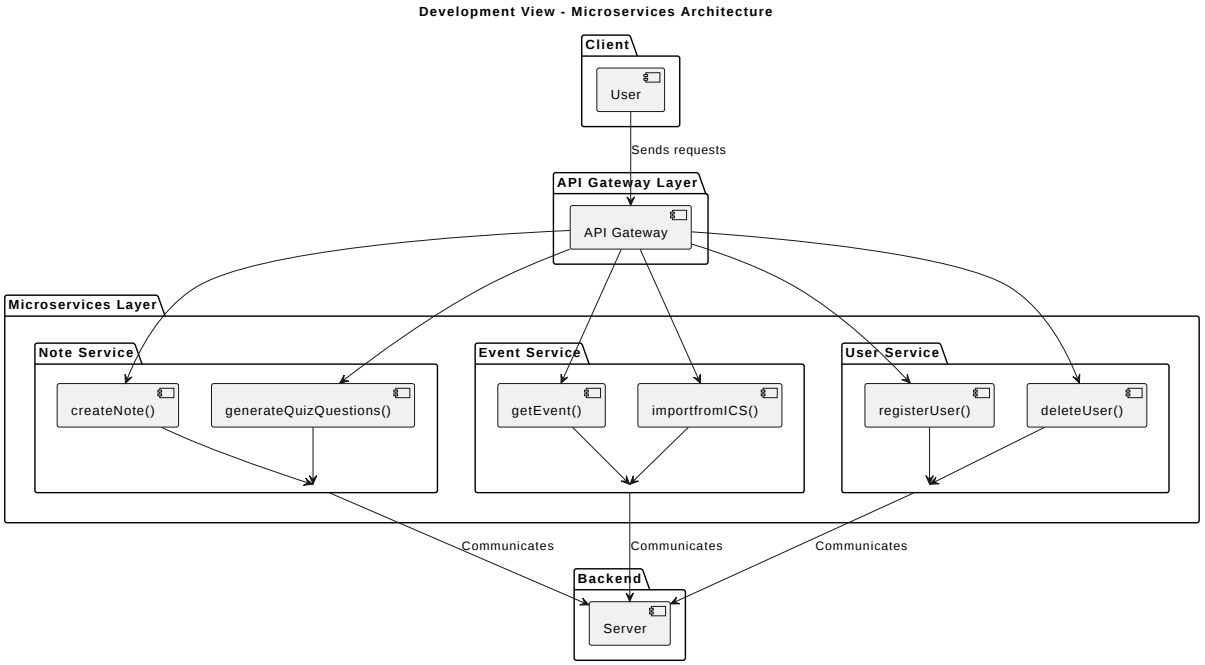
- Authentication sequence diagram



- Exception sequence diagram



Development View



Major Design Decisions (ADRs)

MongoDB usage as database

Choosing **MongoDB** for the SOS platform aligns well with the app's architectural and functional goals. Here's why it makes sense:

1. **Schema Flexibility for Evolving Data Models:** Our app includes diverse data types like notes, deadlines, quizzes, social updates, and AI-generated content. MongoDB's **document model (BSON)** allows flexible schemas, making it ideal for fast iteration and heterogeneous data.
2. **Seamless Scalability:** With a target of supporting 100+ users and 1,000+ requests/sec, MongoDB's **horizontal scaling and sharding** capabilities support this kind of growth efficiently.
3. **Fast Development Cycle:** MongoDB's JSON-like structure maps closely to how data is handled on the frontend (e.g., JavaScript), reducing serialization/deserialization overhead — ideal for our project's rapid 4-week development timeline.
4. **Built-in Replication and High Availability:** MongoDB provides **replica sets** for data redundancy and automatic failover, helping meet our **99.9% uptime** requirement.
5. **Great Support for Microservices Architecture:** Each microservice (e.g., calendar, notes, AI analytics) in our app can operate with its own MongoDB collection or database without strict joins, keeping services loosely coupled.

Why MongoDB is Better than PostgreSQL

1. **Schema Flexibility:** MongoDB allows dynamic, schema-less documents — ideal for evolving features like AI-generated quizzes or notes, whereas PostgreSQL requires predefined schemas and migrations, slowing rapid iteration.
2. **Faster Prototyping with JSON:** MongoDB stores data in BSON (JSON-like), which directly maps to frontend structures in JavaScript/React, making development smoother than PostgreSQL's relational tables.
3. **Microservices Compatibility:** Each microservice in SOS (calendar, notes, AI) can operate independently on MongoDB without the complexity of foreign keys and joins required in PostgreSQL.

Why MongoDB is Better than Firebase Realtime Database

1. **Advanced Querying:** MongoDB supports complex filtering, aggregation pipelines, and indexing — Firebase Realtime DB has limited querying

options and poor support for compound queries.

2. **Better Control Over Data Modeling:** MongoDB offers structured collections and documents with deep nesting and indexing. Firebase Realtime DB can become messy with deeply nested structures that are hard to maintain at scale.
3. **More Mature Ecosystem for Large-Scale Apps:** MongoDB provides robust CLI tools, enterprise features, and better local development support. Firebase Realtime DB is optimized for small-to-medium mobile apps and lacks granular control.

Firebase usage

Including Firebase Authentication in the SOS system is a **strong architectural choice** because:

1. **Out-of-the-Box Security:** Firebase handles encryption, token management, and secure user data storage, helping to avoid reinventing security logic.
2. **Scalable Identity Management:** It scales to a decent number of users, matching our project's non-functional goal of supporting 50+ simultaneous users.
3. **Cross-Platform SDKs:** Firebase supports web and mobile platforms natively, which aligns well with our project's multi-device accessibility requirement.
4. **Reduced Time to Market:** Since Firebase abstracts complex authentication flows, development time is significantly reduced.
5. **** Easy Third-party Integration:** Students can sign in using Google or Facebook — platforms they already use — making onboarding smoother and more intuitive.
6. **Anonymous Authentication for Try-before-Login:** Users can explore features before committing, improving engagement and retention.
7. **Strong Community and Documentation:** Firebase is well-supported, documented, and widely adopted — lowering the maintenance and onboarding curve for developers.

Why Firebase Auth is Better than Auth0

1. **Simpler Integration for MVP:** Firebase offers ready-to-use SDKs for web and mobile with minimal setup, ideal for a fast-moving 4-week project like

SOS. Auth0, while powerful, often needs more configuration for basic flows.

2. **Anonymous Authentication Support:** Firebase uniquely supports anonymous sign-ins, letting users explore the app features before committing. Auth0 doesn't support this natively.
3. **Generous Free Tier with Unlimited Users:** Firebase allows larger number of users in the free plan (with quotas), whereas Auth0 caps free usage at 7,000 active users/month and becomes expensive quickly beyond that.

Why Firebase Auth is Better than AWS Cognito

1. **Developer-Friendly and Easier Onboarding:** Firebase has a smoother learning curve and a clean, intuitive console. Cognito often requires wrestling with AWS IAM roles, policies, and complex configuration for basic use cases.
2. **Quicker Setup for Web Apps:** Firebase is designed for frontend-heavy apps like SOS and can be set up in minutes. Cognito's integration is more backend-oriented and slower to prototype.
3. **Out-of-the-Box Email, Google, and Social Logins:** Firebase handles email/password, Google, Facebook, and others natively with minimal configuration. Cognito often requires custom code or AWS Lambda triggers for a similar experience.

Modular backend

Here is why using microservices architecture is better than using a monolithic architecture for our project:

1. **Scalability**: Scale specific microservices (e.g., calendar syncing, AI quiz generation) based on usage without affecting the core system.
2. **Heterogeneity**: Use the best language/framework for each microservice (e.g., Python for AI, Node.js/React for UI) and experiment with new technologies.
3. **Resilience**: If one service crashes (e.g., quiz generator), others (e.g., calendar, notes) continue working. Prioritize fault tolerance for critical services.
4. **Organizational Alignment**: Assign ownership of microservices to different teams (e.g., Notes Team, AI Team), reducing coordination needs.

5. **Composability**: Combine services to offer compound features (e.g., "Smart Study Mode" using calendar, notes, and AI).
6. **Replaceability**: Easily replace services (e.g., upgrading note-taking system) without reworking the entire application.
7. **Ease of Deployment**: Use continuous deployment pipelines for safe, fast deployments and rollbacks without downtime.

Why Microservices is better than Monolith for our project

1. **Faster Time to Market**: Teams can work in parallel on calendar integration, AI summarization, and social feed parsing — without stepping on each other's toes.
2. **Better Fault Isolation**: A crash in the social media parser won't take down the entire SOS system — only that one module fails.
3. **Tech Freedom**: We're not tied to one language or database; MongoDB can power some services while others could use Redis or SQL if needed.
4. **Simplified Scaling**: Instead of scaling the whole system, we can scale just the AI engine when quiz generation demand spikes before exams.
5. **Easier Maintenance**: Bugs and performance issues are easier to isolate, debug, and fix because services are loosely coupled and independently deployable.
6. **Cleaner Codebases**: Each service is small, focused, and easier to understand — avoiding the bloated "God object" problem of monoliths.

Task 3

Architectural Tactics

We outline the key architectural tactics used in the SOS system, focusing on how each tactic contributes to non-functional requirements such as **modifiability**, **maintainability**, **performance**, **scalability**, and **security**.

Repository Pattern

- **Tactic Type**: Modifiability, Maintainability
- **Intent**: Abstract and encapsulate data access logic.

Description:

- Separates data access from business logic.
- Provides a clean, well-defined API for interacting with data sources.
- Allows changes to database implementation without affecting core logic.

Implementation:

- **Files**:
  - `monolithic_backend/src/notes/repositories/noteRepository.js`
  - `microservices/note-service/repositories/noteRepository.js`
- **Usage**:
  - Implements methods like `create`, `update`, `delete`, `findByDocId`, `findByOwner`, etc.
  - Encapsulates Mongoose queries behind repository methods.
  - Supports easier testing, refactoring, and backend migration.

Method Specialization (Within Repositories)

- **Tactic Type**: Performance, Usability
- **Intent**: Optimize data retrieval and clarify data access intent.

Description:

- Adds purpose-built query methods beyond generic CRUD.
- Improves query efficiency and code readability.

Implementation:

- **Files**:
  - `monolithic_backend/src/notes/repositories/noteRepository.js`
  - `microservices/note-service/repositories/noteRepository.js`
- **Methods**:
  - `findArchivedByOwner(uid)`: Retrieves only archived notes.
  - `findStarredByOwner(uid)`: Retrieves only starred notes.
- **Impact**:
  - Avoids full note retrieval and manual filtering.

- Directly queries specific subsets from the database.

Microservices Architecture

- **Tactic Type**: Scalability, Resilience
- **Intent**: Decompose the system into independently deployable services.

Description:

- Separates business domains into standalone services.
- Enables independent development, deployment, and scaling.
- Improves fault isolation.

Implementation:

- **Service Directories**:
  - `microservices/api-gateway/`
  - `microservices/user-service/`
  - `microservices/note-service/`
  - `microservices/event-service/`
- **Structure**:
  - Each service has its own source code, database, and config.
  - Services communicate via HTTP or messaging.

API Gateway Pattern

- **Tactic Type**: Security, Maintainability
- **Intent**: Centralize external access and cross-cutting concerns.

Description:

- Acts as a single entry point for client requests.
- Handles routing, authentication, logging, and rate-limiting.
- Hides internal service complexity.

Implementation:

- **File**: `microservices/api-gateway/index.js`

- **Technology**: `http-proxy-middleware`
- **Routing Logic**:
  - Routes `/api/users` to User Service.
  - Routes `/api/notes` to Note Service.
  - Uses `pathRewrite` to strip `/api` prefix before forwarding.

Database Indexing

- **Tactic Type**: Performance
- **Intent**: Accelerate query execution by indexing frequently queried fields.

Description:

- Reduces query latency for frequent lookups and filtering.
- Ensures uniqueness and optimizes sorting.

Implementation:

- **Schema Files**:
  - `note.js`, `category.js` in both monolith and microservices.
  - `event.js` in `event-service`.
- **Examples**:
  - Index on `uid`, `docId`, `timestamp` in `noteSchema`.
  - Compound unique index on `uid` and `name` in `categorySchema`.
- **Collation**:
  - Case-insensitive uniqueness for category names using Mongoose collation.

Asynchronous Processing

- **Tactic Type**: Performance, Scalability
- **Intent**: Enable non-blocking operations for I/O-heavy tasks.

Description:

- Uses `async/await` to prevent blocking in Node.js.
- Improves system responsiveness and concurrency.

Implementation:

- **Files**:
  - `noteController.js`, `noteRepository.js` in both monolithic and microservices.
- **Usage**:
  - `await` used for database queries (`createNote`, `getNotesByOwner`, etc.).
  - Allows the event loop to serve other requests while waiting.

The SOS system applies a combination of architectural tactics tailored to specific quality attributes:

Tactic	Type(s)
Repository Pattern	Modifiability, Maintainability
Method Specialization	Performance, Usability
Microservices Architecture	Scalability, Resilience
API Gateway Pattern	Security, Maintainability
Database Indexing	Performance
Asynchronous Processing	Performance, Scalability

These tactics are reflected clearly in the codebase structure and implementation strategy, supporting a robust and adaptable software architecture.

Implementation Patterns

The design patterns implemented in our microservices-based note-taking application are documented here. The application consists of a Next.js frontend and a microservices backend with separate services for users, notes, and events. Design patterns were strategically implemented to improve code maintainability, scalability, and reusability.

Factory Pattern

Implemented in the Note Service (`/microservices/note-service/factories/noteFactory.js`) to create different types of notes (text notes, PDF notes, voice notes, summaries) with consistent structure and metadata.

Implementation:

```
class NoteFactory {
  static createTextNote(docId, uid, title, content, category, tags = [], preview = "No preview available.") {
    return new Note({
      docId,
      uid,
      title,
      content,
      category,
      tags,
      timestamp: Date.now(),
      preview,
      type: 'text'
    });
  }

  static createPdfNote(docId, uid, title, content, originalFilename, category = 'Generated', tags = ['pdf-import'], preview = "No preview available.") {
    // Implementation for PDF notes
  }

  static createVoiceNote(docId, uid, title, content, category, tags = [], preview = "No preview available.") {
    // Implementation for voice-based notes
  }

  static createSummarizedNote(docId, uid, originalNoteId, title, content, category = 'Generated', tags = [], preview = "No preview available.") {
    // Implementation for summarized notes
  }
}
```

Benefits:

- Encapsulates object creation logic
- Enforces consistent note structure
- Makes note creation extensible

- Simplifies creation of specialized note types

Singleton Pattern

Implemented for database connections in each service (`/microservices/note-service/models/connect.js` , `/microservices/event-service/models/connect.js` , `/microservices/user-service/models/connect.js`) to ensure only one database connection instance exists.

Implementation:

```
class DatabaseConnection {
  constructor() {
    if (DatabaseConnection.instance) {
      return DatabaseConnection.instance;
    }

    this.isConnected = false;
    DatabaseConnection.instance = this;
  }

  async connect(uri) {
    // Connection logic with reuse of existing connections
  }

  // Other methods...
}

// Create and export singleton instance
const dbConnection = new DatabaseConnection();
```

Benefits:

- Prevents multiple database connections
- Provides a global access point
- Controls resource usage
- Facilitates connection management

Observer Pattern

Implemented in the Note Service (`/microservices/note-service/observers/noteObserver.js`) to handle events and notifications when notes are created, updated, deleted, etc.

Implementation:

```
class NoteEventSubject {
  constructor() {
    this.observers = [];
  }

  subscribe(observer) {
    if (!this.observers.includes(observer)) {
      this.observers.push(observer);
    }
  }

  notify(event, data) {
    this.observers.forEach(observer => {
      observer.update(event, data);
    });
  }
}

// Example usage in controller
const savedNote = await noteRepository.create(newNote);
noteEventBus.notify('create', savedNote);
```

Benefits:

- Loose coupling between components
- Support for event-driven architecture
- Extensible notification system
- Simplified integration with external systems

Strategy Pattern

Implemented for note summarization (`/microservices/note-service/strategies/summarizationStrategy.js`) to support different summarization algorithms.

Implementation:

```
class SummarizationStrategy {
  async summarize(content) {
    throw new Error('summarize method must be implemented by concrete s
trategies');
  }
}

class GeminiSummarizationStrategy extends SummarizationStrategy {
  async summarize(content) {
    // AI-based summarization implementation
  }
}

class BasicSummarizationStrategy extends SummarizationStrategy {
  async summarize(content) {
    // Simple text-based summarization implementation
  }
}

class SummarizationContext {
  constructor(strategy = null) {
    this.strategy = strategy || new GeminiSummarizationStrategy();
  }

  setStrategy(strategy) {
    this.strategy = strategy;
  }

  async summarize(content) {
    return await this.strategy.summarize(content);
  }
}
```

Benefits:

- Enables runtime algorithm switching
- Encapsulates algorithm implementations

- Facilitates adding new algorithms
- Decouples client code from specific implementations

Repository Pattern

Implemented in the each Service (`/microservices/note-service/repositories/noteRepository.js`) to abstract data access operations from business logic.

Implementation:

```
class NoteRepository {
  async findByDocId(docId) {
    try {
      return await Note.findOne({ docId });
    } catch (error) {
      console.error('Error in findByDocId:', error);
      throw error;
    }
  }

  async create(noteData) {
    try {
      const note = new Note(noteData);
      return await note.save();
    } catch (error) {
      console.error('Error in create:', error);
      throw error;
    }
  }

  // Other repository methods...
}
```

Benefits:

- Centralizes data access logic
- Decouples business logic from data access
- Simplifies unit testingImproves code maintainability

- Allows for easy switching of data sources (e.g., from MongoDB to a different database)

The implementation of these design patterns has significantly improved the architecture of our microservices-based application. The patterns enable:

1. **Modularity**: Components can be developed, tested, and maintained independently.
2. **Extensibility**: New features can be added without significant changes to existing code.
3. **Maintainability**: Clear separation of concerns makes the codebase easier to understand and modify.
4. **Testability**: Decoupled components can be tested in isolation.
5. **Scalability**: Services can be scaled independently based on demand.

Task 4

Prototype Development

The prototype system demonstrates the fundamental workflows and architectural decisions across a full-stack application using both Monolithic and Microservices architectures. The prototype validates the system's practicality by implementing the following key features:

Core Functional Workflows

User Authentication

1. Integrated Firebase Authentication for Google and Email/Password login.
2. Automatically creates user records in the backend upon first login.

Note Management System

1. Users can create, edit, categorize, archive, and star notes using a block-based editor.
2. Notes can be uploaded as PDFs or exported back to PDF.
3. AI-powered note summarization and quiz generation via Google Gemini API.

Calendar & Event Workflow

1. Events are created, edited, and viewed using a full calendar UI.
2. ICS file import/export feature to support interoperability.

Architectural Demonstration

Monolithic Architecture

1. All features (users, notes, events) are handled in a single Node.js/Express backend with direct MongoDB communication.
2. Showcases tight integration and low-latency responses.

Microservices Architecture

1. Features are split into isolated services (user, note, event), routed via an API Gateway.
2. Demonstrates independent scalability and modularity.
3. Implements multiple design patterns (e.g., Repository, Observer, Strategy) in the Note Service.

Performance Validation

Performance Monitoring Suite

1. Automated load testing using Autocannon to compare throughput, latency, and cold start times.
2. HTML report generation for visual comparison of Monolithic vs Microservices performance.
3. Observed trade-offs: Monolithic had better latency/throughput; Microservices had better cold start and scalability potential.

Architecture Analysis

We implemented the monolith and microservices architecture for our backend. This report evaluates their behavior under load using key metrics like throughput, latency, cold start time, status codes, and error rates.

Key Metric Highlights

- **Throughput**
  - Monolithic: 77.90 requests/sec

    - Microservices: 68.90 requests/sec
    - Monolithic handled ~11.55% more requests per second
- **Latency**
    - Monolithic: 1450.82 ms
    - Microservices: 3085.22 ms
    - Monolithic had ~112.65% lower latency, offering faster responses
- **Cold Start Time**
    - Monolithic: 70 ms
    - Microservices: 26 ms
    - Microservices performed ~62.86% better in cold start conditions

---

Detailed Endpoint Analysis

Throughput (Requests Per Second)

- **Notes - Get All**
    - Monolithic: 112.10
    - Microservices: 98.20
    - Monolithic outperformed by ~12.4%
- **Events - Get All**
    - Monolithic: 43.70
    - Microservices: 39.60
    - Monolithic outperformed by ~9.38%

Latency (Milliseconds)

- **Notes - Get All**
    - Monolithic: 883.05 ms
    - Microservices: 2872.60 ms
    - Microservices had over 3x higher latency
- **Events - Get All**
    - Monolithic: 2018.60 ms

- Microservices: 3297.83 ms
- Latency was significantly higher in microservices

HTTP Status Codes

- All endpoints returned `200 OK` in both Monolithic and Microservices architecture
- Confirms stable request handling in both systems

Error Count

- No errors recorded for any endpoints in either architecture
- Indicates reliability across the board

Architecture Trade-offs

Monolithic Architecture

Advantages

- Lower latency due to minimal internal communication overhead
- Easier to develop, test, and deploy as a single cohesive unit
- Better transactional integrity and data consistency
- Lower resource consumption, ideal for smaller teams or MVPs

Disadvantages

- Scaling requires the entire application to scale together
- Difficult to adopt different tech stacks for different components
- Risky deployments — any change requires redeploying the full app
- Can become a bottleneck as the codebase and team grow

Microservices Architecture

Advantages

- Services can be scaled independently based on demand
- Allows teams to use different technologies per service
- Isolates failures — one failing service won't crash the entire system
- Enables parallel development by autonomous teams

Disadvantages

- Increased latency from inter-service communication
- Higher operational and monitoring complexity
- Difficult to enforce data consistency across distributed systems
- Each service adds its own overhead, increasing total resource usage

---

Monolithic architecture performed better in terms of latency and throughput under the tested conditions. However, microservices shine when scalability, flexibility, and modularity are priorities. Choose your architecture based on long-term goals, team size, and performance needs.

Contributions

Member	Contribution
Yash	Finalizing requirements, Note taking subsystem, Monolithic and microservices backend, Backend monitoring system, Minor features, Architecture analysis
Sreenivas	Finalizing requirements, Resource library subsystem, API gateway system, Architectural Tactics
Sreyas	Finalizing requirements, Calendar management subsystem, Implementation patterns
Krishna Chaitanya	Finalizing requirements, User authentication subsystem, Subsystem overview, Stakeholders Identification
Sankalp	Finalizing requirements, AI assessment and learning subsystem, Architectural Design Records