Project - 3 Report

> **Team 7**

Github repository link: https://github.com/umangapatel123/CPLite

Task 1: Requirements and Subsystems

The CPLite system helps competitive programmers by creating personalized learning paths, using real-time performance data and tools for instructor-led mentorship.

Functional Requirements

1. **User Management:** Handles user authentication via OAuth2 (Google), Codeforces handle linking, role assignment (Mentor, Learner), mentor-learner relationships, role-based permissions, and profile management.

   Architectural Significance

   OAuth2 authentication requires a central Authentication Service for token management; all services must support stateless auth and verify tokens per request; affects API Gateway and frontend session handling. Role-Based Access Control (RBAC) demands a roles system in the database; governs permissions and data visibility across the system. Mentor-learner linking shapes the user data model and influences recommendation logic and frontend behavior. Codeforces handle linkage is essential for system functionality and frontend sync

2. **Codeforces Progress Tracking**: Weekly updates via API; dashboard with user rating, problems solved and other user stats

   Architectural Significance:
   Drives the dashboard UI, necessitates modular integration of CF API through a dedicated codeforces service, introduces scheduled jobs (cron) to automatically pull fresh data each week

3. **AI Recommendations**: Weekly problem suggestions generated tailored to learner performance using Gemini and CF API

   Architectural Significance:

   Initiates the foundation for learner-specific recommendation logic (the system's core value of personalized learning), justifying its central role in the architecture. Requires integration with LLMs and the CF API through their dedicated services. Requires asynchronous background scheduled tasks (cron job), a processing pipeline with service orchestration, and storage for recommended problems.

4. **Task Assignment & Progress Tracking**: Instructors manage/edit tasks, set deadlines, and track learner progress; synced with database for real-time updates accessible to both mentors and learners

Architectural Significance:
Requires data consistency mechanisms to maintain real-time updates for both mentors and learners. Affects role-based views, where mentors can edit tasks while learners only view them. Demands frontend synchronization to ensure accurate and timely updates.

5. **Gamification UI:** Frontend with badge displays, point tracking, and milestone highlights. (*not implemented*)
6. **LLM Insights**: Weekly summaries on learner performance (uses Gemini and CF API)

   Architectural Significance:

   Initiates the foundation for analysis of learner's progress (the system's core value of personalized learning). Requires integration with LLMs and the CF API through their dedicated services. Requires asynchronous background scheduled tasks (cron job), a processing pipeline with service orchestration, and storage for LLM insights.
7. **Notifications**: Real-time alerts for contests/deadlines

   Architectural Significance:

   Introduces a publish-subscribe architecture with RabbitMQ for event-driven coordination. Requires asynchronous processing for scalable notifications and persistent queues for message durability. Cross-service dependencies demand strict event schema validation to ensure compatibility. Relies on PostgreSQL for notification storage and WebSocket integration *(in future)* with Redis for real-time updates and horizontal scaling.
8. **Discussion Forum**: Q&A for learners/instructors (*not implemented*)

Non-Functional Requirements

1. Security (OAuth2, RBAC, Auditability)

Security requirements dictated the design of the **User & Relationship Management Subsystem**, which handles authentication, authorization, and user-role mappings. The use of **OAuth2 with Google** eliminates password management while ensuring trusted authentication. **Role-Based Access Control (RBAC)** is implemented system-wide, with roles enforced at API and service layers. These requirements shaped:

- The integration of external identity providers (Google).
- The need for JWT-based stateless session management.
- Clear separation of user privileges across microservices.
- Secure API gateway routing with restricted access points.

Security concerns also led to strict isolation between services, limiting the blast radius of potential vulnerabilities.

---

2. Availability (Resilient LLM & Notification Services)

Availability requirements resulted in the adoption of an **asynchronous, decoupled architecture**. Systems such as the **Notification Subsystem** are designed to operate independently of the other components, allowing the core platform to remain functional even if auxiliary systems face delays or errors. These requirements shaped:

- Use of **RabbitMQ** for non-blocking communication between services.
- **Heartbeat monitoring** for early detection of failure and potential support for auto-recovery.
- **Rate limiting via Nginx** to prevent overload from client traffic.

These availability-driven tactics allow services to fail gracefully without affecting user interaction or learning workflows.

---

3. Modifiability (Ease of Extension & Maintenance)

The platform is designed for continuous evolution, with anticipated changes to learning logic, UI components, and third-party API dependencies (like Codeforces or Gemini). This requirement led to:

- Adoption of the **Microservices Architecture**, enabling isolated updates to components such as the AI Service or Task Service without full re-deployments.
- Implementation of the **MVC pattern** to separate concerns within each service (model, controller, view logic), facilitating targeted development and testing.
- Centralized scheduling logic within the **Orchestration Service**, which allows AI/summary workflows to be modified without altering the learner or mentor interfaces.

This modularity supports both short-term flexibility and long-term maintainability.

---

4. Performance (Page Load < 3 seconds, concurrent users)

Why it's architecturally significant:

User responsiveness is a key UX driver, especially when presenting dynamic content such as dashboards, recommendations, and tasks.These requirements shaped:

- Use of **React Context** for optimized frontend rendering and minimal re-fetching.
- Efficient **API routing via Nginx**, minimizing latency between client and microservices.
- Targeted service optimization—e.g., splitting the task logic and AI processing into distinct services to avoid blocking.
- The platform can be potentially scaled to handle concurrent usage from learners and mentors — potentially across different time zones — interacting with tasks, dashboards, and recommendations in real time. Thus, **Message queues (RabbitMQ)** buffer background workloads to prevent synchronous overload and ensure eventual consistency. **Rate limiting** and **token-based session validation** protect against request flooding.
- Potential for caching and CDN use (for static frontend assets), enhancing client-side speed.

Performance goals enforced clean, lightweight service designs and shaped the separation of real-time vs scheduled logic.

---

5. Scalability (Handling Asynchronous Load & User Growth)

Why it's architecturally significant:

As the platform expands to support more learners and mentors, scalable systems are necessary to manage traffic surges and data growth.These requirements shaped:

- **RabbitMQ**, which allows asynchronous job processing (e.g., sending task reminders, generating summaries).
- **Microservices** that can scale horizontally—AI or Notification services can be scaled independently based on load.
- **Decoupling** of real-time user interaction (frontend) from backend batch jobs (orchestration and summarization).

These scalability-focused decisions future-proof the system against increased usage without requiring architectural overhauls.

Subsystem Overview

1. User & Relationship Management Subsystem

Role: Handles identity, authentication, user profile and mentor-learner relationships.

Functionality: Handles user login/registration with OAuth2 (Google), issues, validates JWT tokens for session management, supports various authentication strategies, manages user roles (learner, mentor) and permissions using Role-Based Access Control (RBAC), maintains mentor-learner relationships, manages user profiles by linking Codeforces handles to accounts, and stores related user data.

---

2. Task & Recommendation Subsystem

Role: Delivers adaptive learning content

Functionality: Generates personalized problem recommendations using performance data; Enables mentor customization/overrides of learning paths; Monitors task completion status and deadlines (workflow managed using CRUD API endpoints that the mentor can trigger and schedule API calls (via cron job))

Key Services Used: Codeforces Service, AI Service, Orchestration Service, Task Service

---

3. LLM Summary Subsystem

Role: Automated learner's progress reporting

Functionality: Analyzes and summarises learner statistics (rating, topics covered, problem attempts) every week (workflow managed using schedule API calls (via cron job)); Stores output in summary database for historical tracking

Key Services Used: Codeforces Service, AI Service, Orchestration Service

---

4. Notification Subsystem

Role: Delivers real-time alerts and updates.

Functionality: Uses RabbitMQ to process and dispatch notifications such as contest reminders, task updates (including "task of the day").

---

5. Discussion Forum Subsystem

Role: Enables community-based learning and peer support.

Functionality *(not implemented)*: Manages forum threads, posts, comments, and user interactions; Allows learners to ask questions and receive answers from both peers and mentors; Publishes forum activity to the notification system; Supports tagging, searching, and categorization of discussions by topic or difficulty level.

---

6. Frontend Subsystem

Role: Provides the user interface.

Functionality: Displays dashboards, AI recommendations, and task views, with tailored interfaces for mentors and learners, supported by React contexts for state management. In the future, it could also handle gamification elements like badges, points, and milestone highlights in the UI.

---

7. API Gateway (Nginx)

Role: Acts as primary entry point for traffic, routing requests between external clients and internal microservices to ensure smooth communication and service discovery.

Functionality: Directs traffic to the right service or frontend assets based on the URL structure; Controls the flow of traffic by enforcing rate limits, which helps prevent overuse or abuse of the system; Ensures security by adding headers to protect client information and making sure only authorized routes are exposed; Makes API documentation available, so developers can easily access relevant service details; Provides a health check endpoint to monitor the system's status; Optimizes traffic for better performance, speeding up response times and reducing unnecessary overhead.

Supporting Service Components

1. **Codeforces Service**

   Role: Interfaces with the Codeforces API.

   Functionality: Fetches user stats (eg. rating, solved problems, etc.), contest data, and problem recommendations, and provides problem metadata.

2. **AI Service**

   Role: Powers intelligent insights and recommendations.

   Functionality: Integrates with the Gemini API to generate weekly summaries, suggest topics/difficulties, identify weak areas, and manages the stored output for future use.

3. **Orchestration Service**

   Role: Orchestrates the weekly recommendation and progres tracking workflow using Cron job.

Functionality: Runs scheduled jobs to gather stats from Codeforces, triggers the AI service, and assigns tasks via the Task Service.

4. **Task Service**

   Role: Manages assigned tasks.

   Functionality: Supports CRUD operations for tasks, tracks completions, allows instructor overrides, and handles deadlines.

Supporting Infrastructure

Role: Provides foundational components for the system.

Functionality: Uses Docker for containerization, PostgreSQL for persistent storage, and RabbitMQ for message-based communication.

Note: Orchestration Service and Recommendation Service refer to the same service and have been used synonymously in some places.

Task 2: Architecture Framework

Note: We have used Rozanski and Woods Viewpoint Set *(Rozanski, N. and Woods, E. (2011). "Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives")*

Reference: https://www.viewpoints-and-perspectives.info/home/viewpoints/

Stakeholders Identification and Architectural Viewpoints (IEEE 42010 Standard)

Stakeholders Identification

End Users

- **Mentors & Learners**
  - **Concerns**: System availability, Reliability, Performance, Task management, Personalization, Notifications, Analytics access, Easy Login/registration, Relationship management, Usability, Data security, System responsiveness
    - Mentor Concerns: Task assignment, Progress monitoring, Recommendation override
    - Learner Concerns: Progress tracking, Contest reminders, Performance insights

Technical Stakeholders

- **Developers (Software Engineers)**
  - **Concerns**: Code modularity and organization, testability, maintainability, scalability, impact analysis for changes, implementation efficiency, code reusability, portability across platforms
- **System Administrators**

- **Concerns**: System deployment, security protocols, monitoring capabilities, infrastructure dependencies, and network requirements

---

Viewpoints and Views

Functional Viewpoint (End Users: Mentors and Learners, Developers)

- **Views**: Context diagram, Use Case diagram
- **Addressed Concerns**: System availability, Reliability, Performance, Task management, Personalization, Notifications, Analytics access, Easy Login/registration, Relationship management
  - Mentor Concerns: Task assignment, Progress monitoring, Recommendation override
  - Learner Concerns: Progress tracking, Contest reminders, Performance insights

Context Viewpoint (End Users, Developers)

- **Views**: Context diagram
- **Addressed Concerns**: System scope and responsibilities, external entities and interfaces

Development Viewpoint (Developers)

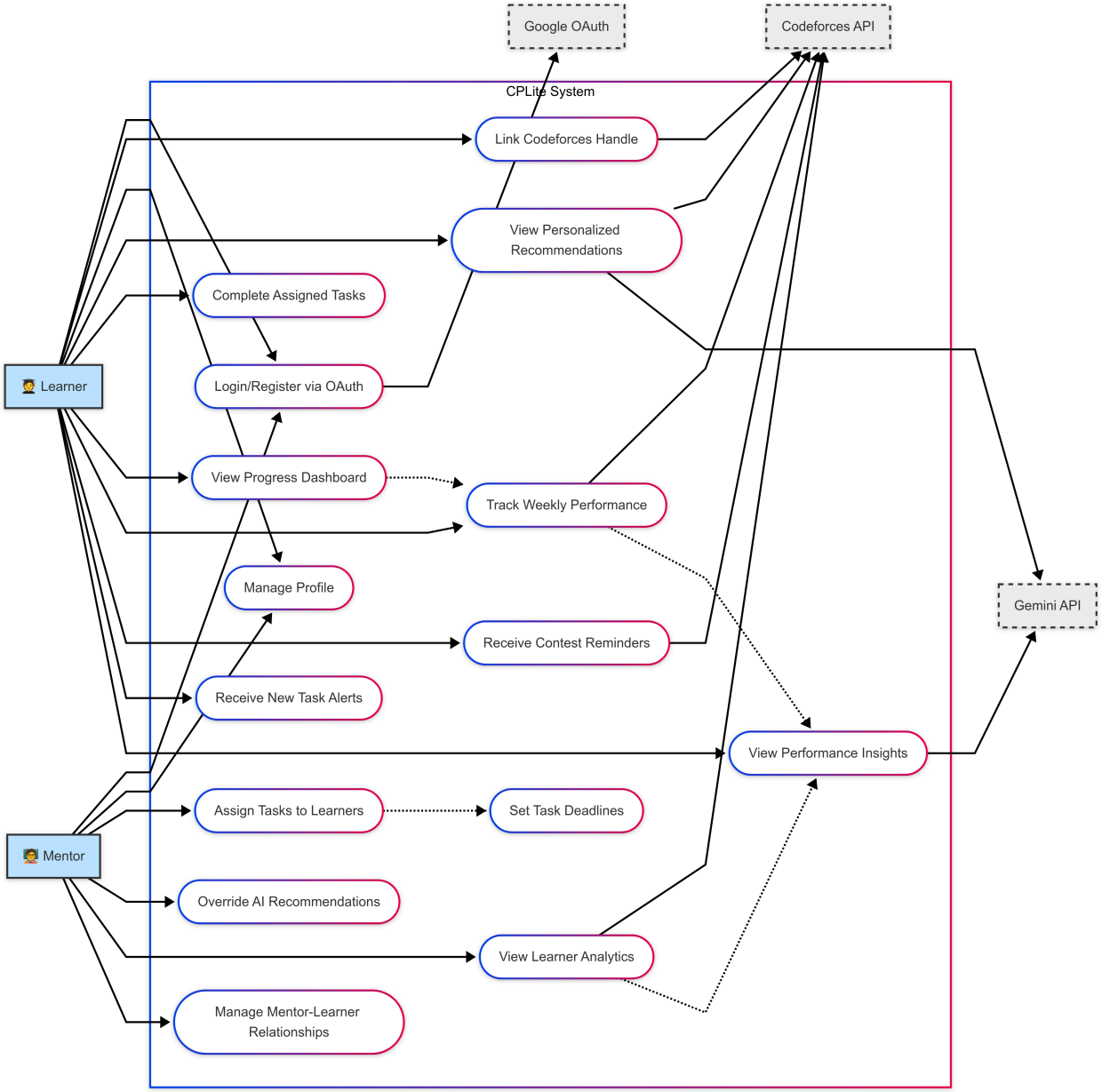
- **Views**: Component Diagram, Sequence Diagram, Code Repository Structure, Class Diagrams
- **Addressed Concerns**: Functional requirements implementation, code modularity, maintainability, testability, scalability, code organization, component reuse

Deployment Viewpoint (System Administrators)

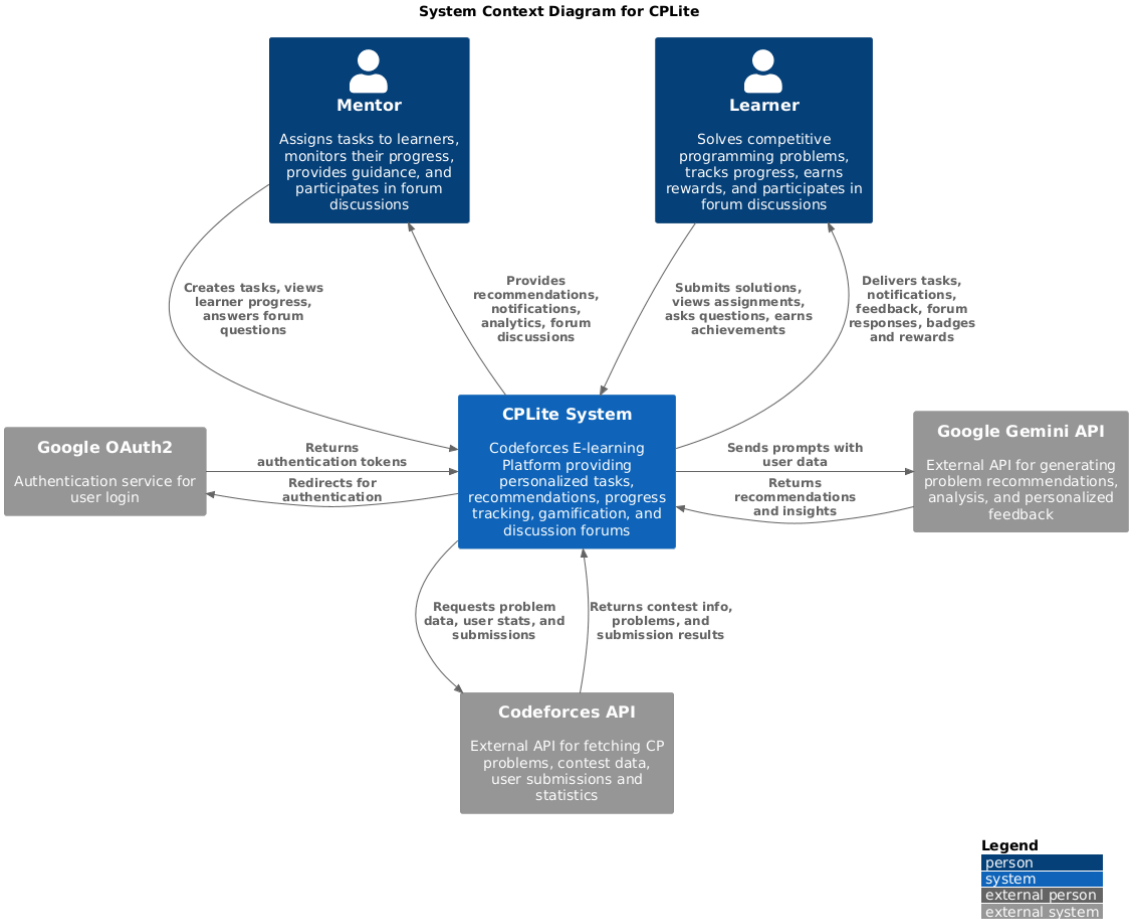
- **Views**: Container Diagram
- **Addressed Concerns**: Network requirements, security protocols, third-party software compatibility, deployment boundaries, and infrastructure dependencies

---

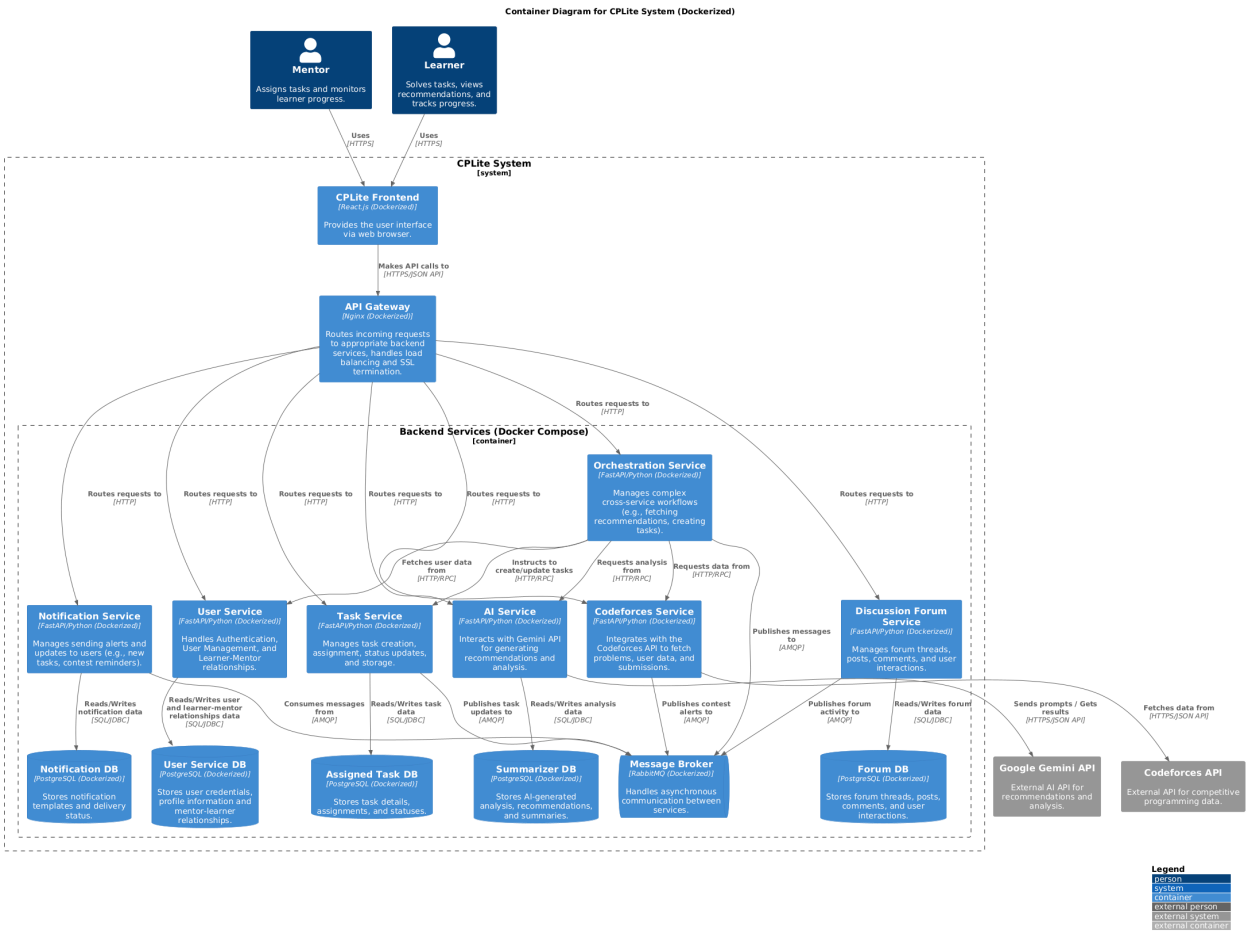
Use Case Diagram



Context Diagram:

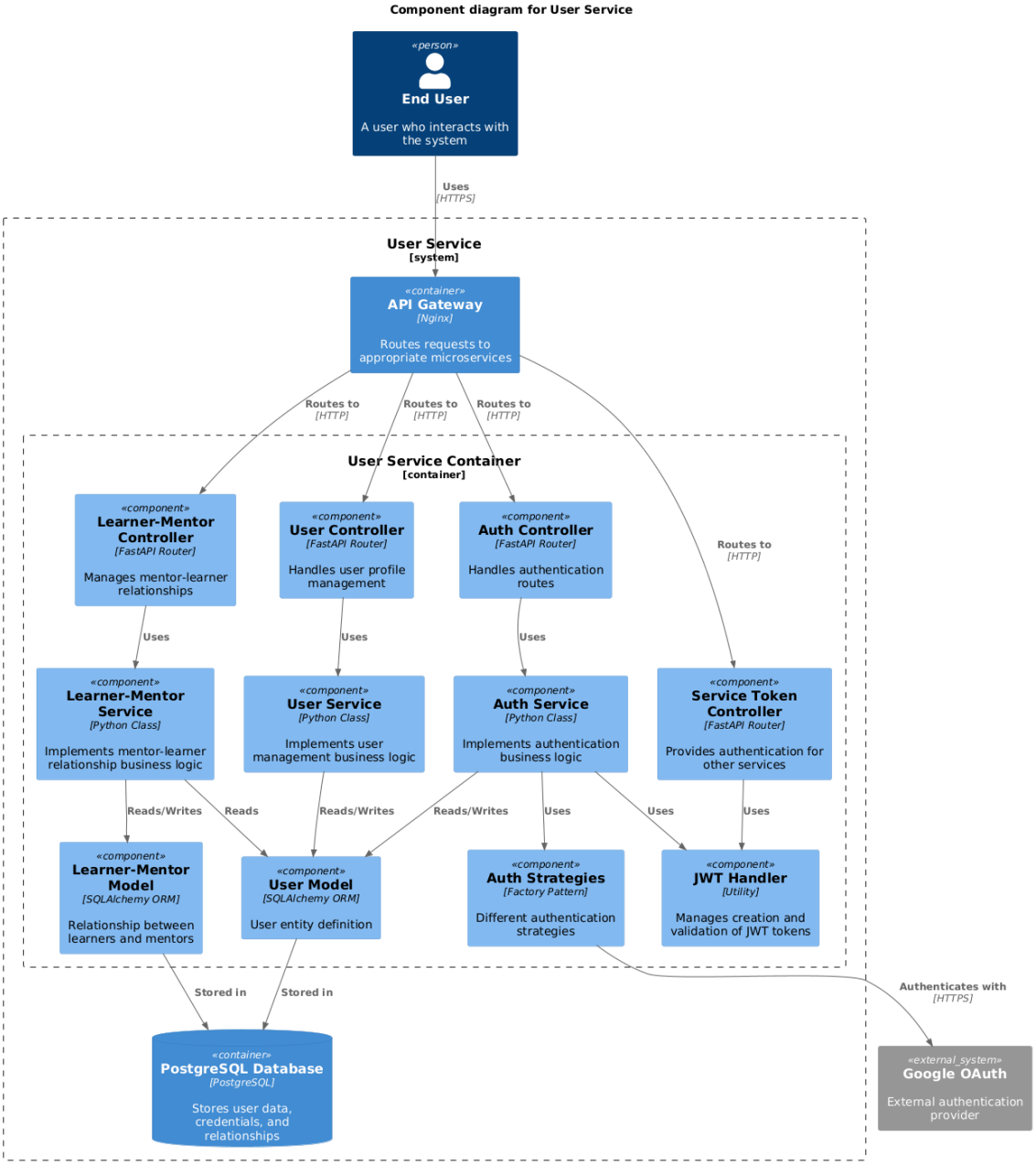


Container Diagram:

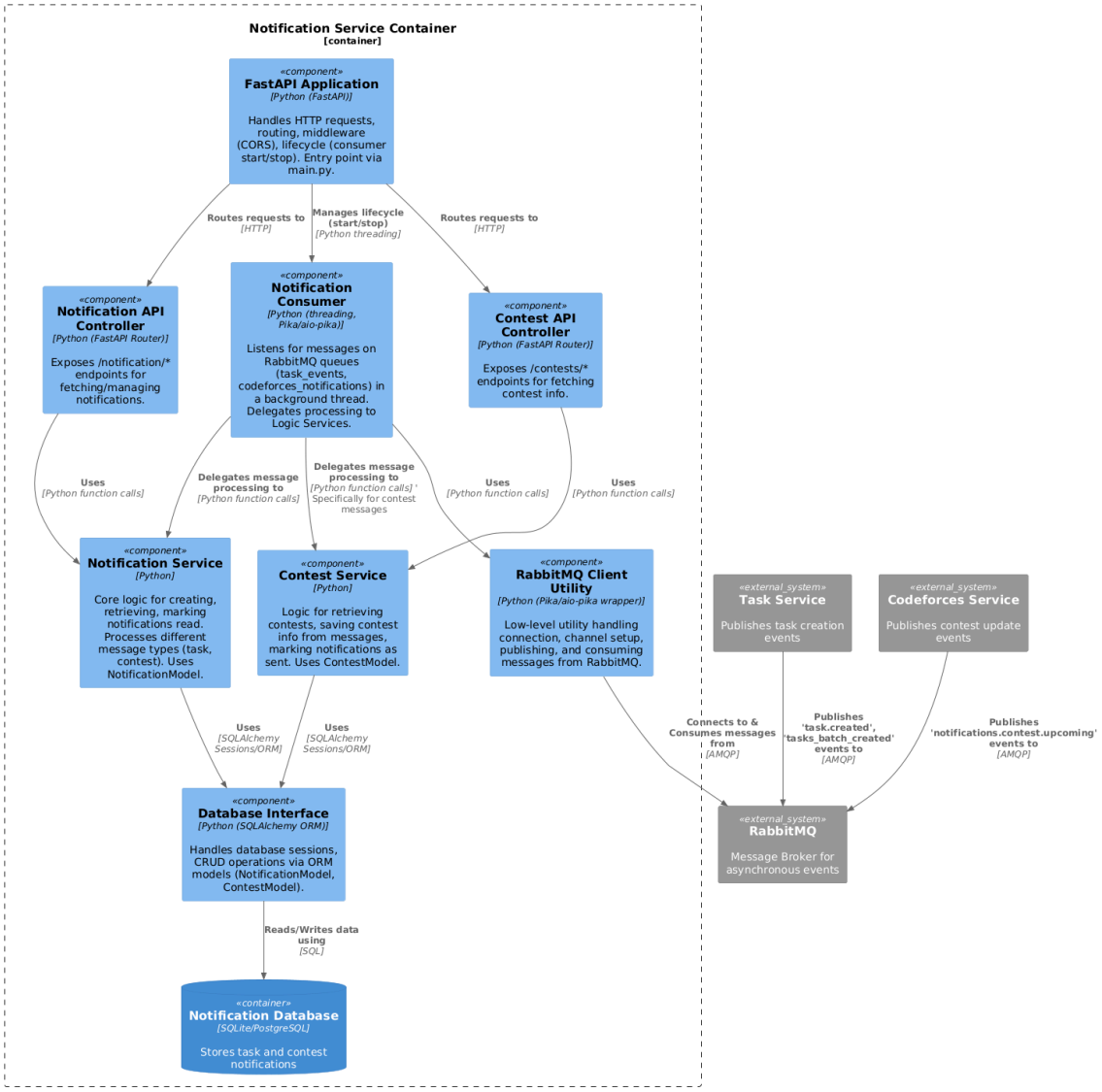


Component Diagrams:

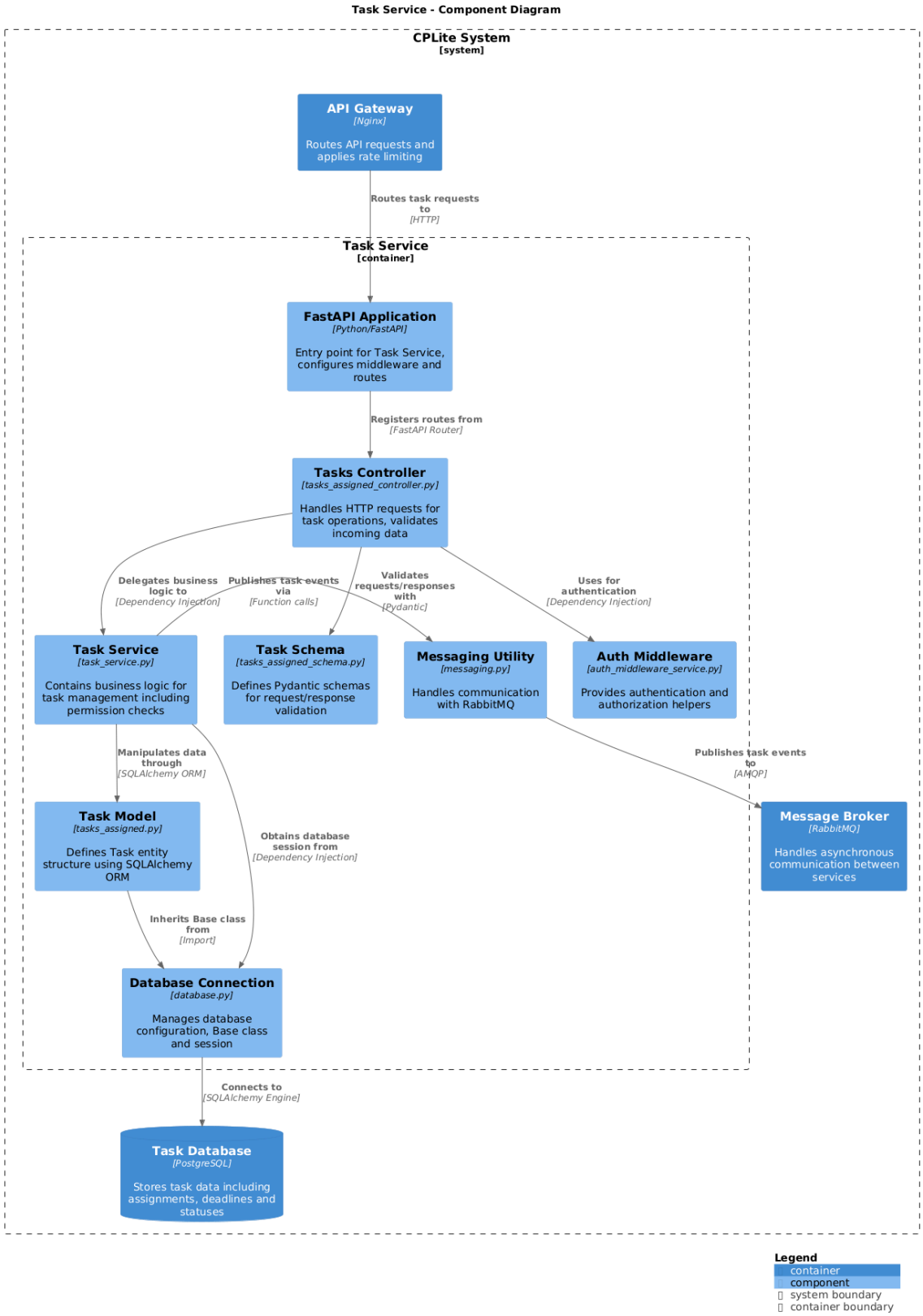
- **User Service (User Management Subsystem)**



- **Notification Service (Notification Subsystem)**

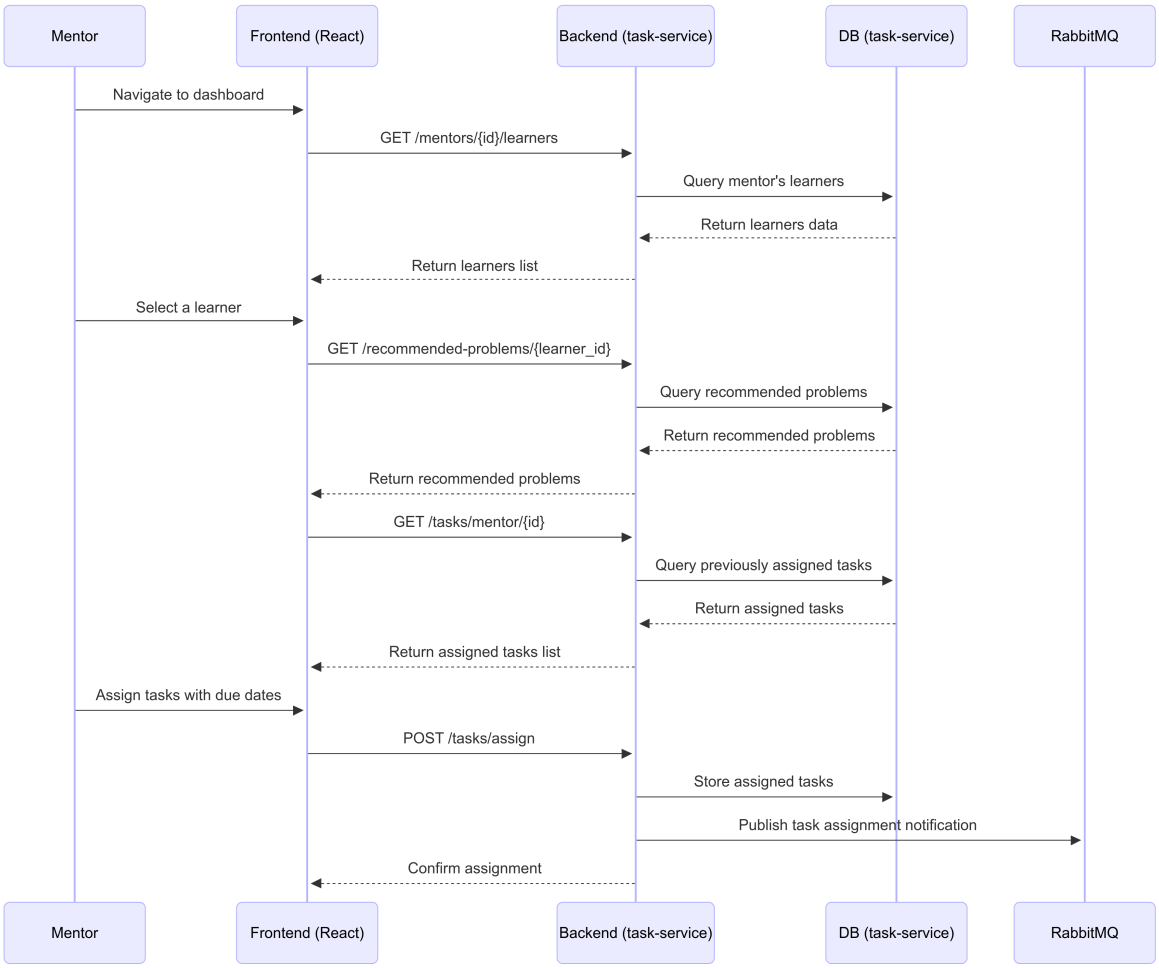


- **Task Service**

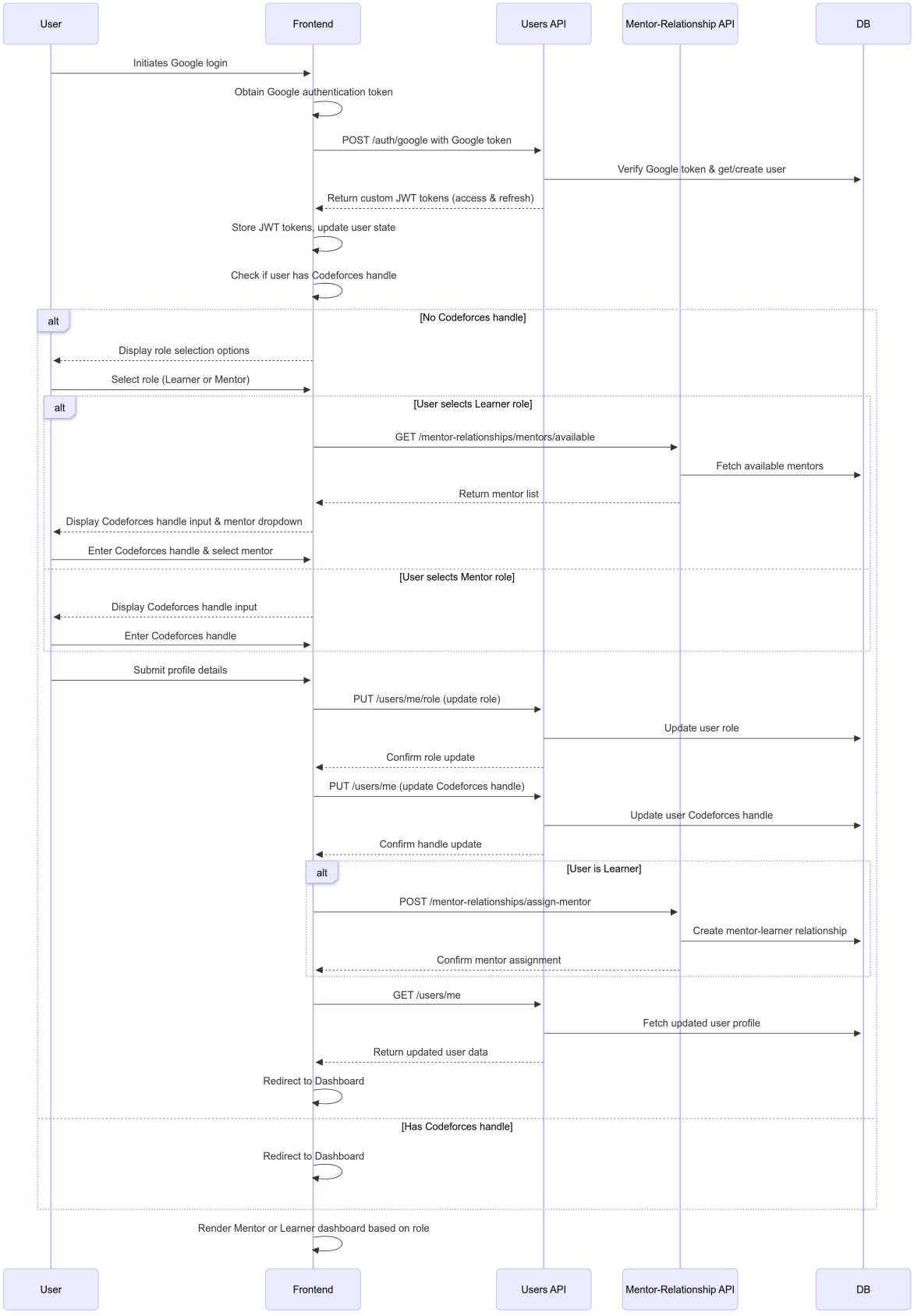

Task Service - Component Diagram
CPLite System
[system]
API Gateway
[Nginx]
Routes API requests and applies rate limiting
Routes task requests to
[HTTP]
Task Service
[container]
FastAPI Application
[Python/FastAPI]
Entry point for Task Service, configures middleware and routes
Registers routes from
[FastAPI Router]
Tasks Controller
[tasks_assigned_controller.py]
Handles HTTP requests for task operations, validates incoming data
Delegates business logic to
[Dependency Injection]
Publishes task events via
[Function calls]
Validates requests/responses with
[Pydantic]
Uses for authentication
[Dependency Injection]
Task Service
[task_service.py]
Contains business logic for task management including permission checks
Task Schema
[tasks_assigned_schema.py]
Defines Pydantic schemas for request/response validation
Messaging Utility
[messaging.py]
Handles communication with RabbitMQ
Auth Middleware
[auth_middleware_service.py]
Provides authentication and authorization helpers
Manipulates data through
[SQLAlchemy ORM]
Publishes task events to
[AMQP]
Task Model
[tasks_assigned.py]
Defines Task entity structure using SQLAlchemy ORM
Obtains database session from
[Dependency Injection]
Message Broker
[RabbitMQ]
Handles asynchronous communication between services
Inherits Base class from
[Import]
Database Connection
[database.py]
Manages database configuration, Base class and session
Connects to
[SQLAlchemy Engine]
Task Database
[PostgreSQL]
Stores task data including assignments, deadlines and statuses
Legend
container
component
system boundary
container boundary

Sequence Diagram (key flows)

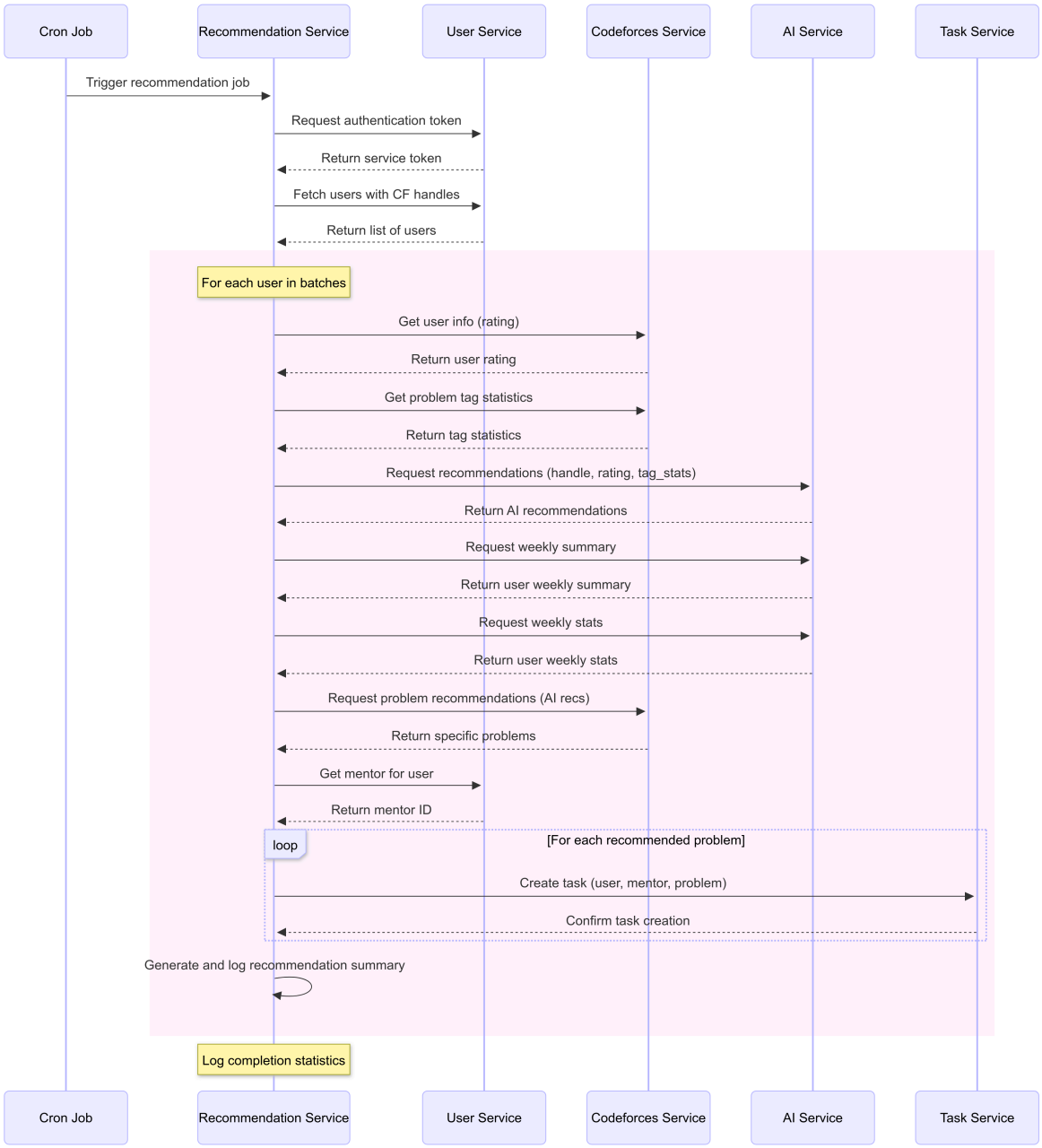
- **Mentor view**



- **User Login Flow**


User
Frontend
Users API
Mentor-Relationship API
DB
Initiates Google login
Obtain Google authentication token
POST /auth/google with Google token
Verify Google token & get/create user
Return custom JWT tokens (access & refresh)
Store JWT tokens, update user state
Check if user has Codeforces handle
alt
[No Codeforces handle]
Display role selection options
Select role (Learner or Mentor)
alt
[User selects Learner role]
GET /mentor-relationships/mentors/available
Fetch available mentors
Return mentor list
Display Codeforces handle input & mentor dropdown
Enter Codeforces handle & select mentor
[User selects Mentor role]
Display Codeforces handle input
Enter Codeforces handle
Submit profile details
PUT /users/me/role (update role)
Update user role
Confirm role update
PUT /users/me (update Codeforces handle)
Update user Codeforces handle
Confirm handle update
alt
[User is Learner]
POST /mentor-relationships/assign-mentor
Create mentor-learner relationship
Confirm mentor assignment
GET /users/me
Fetch updated user profile
Return updated user data
Redirect to Dashboard
[Has Codeforces handle]
Redirect to Dashboard
Render Mentor or Learner dashboard based on role
User
Frontend
Users API
Mentor-Relationship API
DB

- **Problem Recommendation Flow**



Architecture Decision Records (ADRs)

ADR 1: Adoption of Microservices Architecture

Decision: Adopt a microservices architecture for CPLite.

Status: Accepted

Context:

CPLite supports diverse functionality including user management, AI-powered recommendations, progress tracking, Codeforces API integration, and real-time notifications. These features need to evolve independently, scale selectively, and be maintained with minimal interdependencies.

Considered Options:

1. **Monolithic Architecture:** All functionalities bundled into a single codebase and deployed as a single unit.
2. **Microservices Architecture:** Independent, loosely coupled services for each major subsystem (user, task, recommendation, etc.)
3. **Layered Architecture:** The application is divided into logical layers (e.g., presentation, business logic, data access), usually within a single codebase or process boundary.

Decision:

Adopt a microservices architecture for all core system functionalities.

Rationale:

- **Scalability:** Individual services can scale independently (e.g., AI service during peak recommendation generation).
- **Maintainability:** Easier isolation and debugging of issues per service.When one service is disrupted others are not directly affected.
- **Modifiability:** Future changes to specific modules (e.g., LLM updates) won't affect others.
- **Team Velocity:** Parallel development becomes possible across teams or individuals.

Consequences:

- Supports future scalability and modular enhancements.(Pro)
- Facilitates modular development and independent scaling.(Pro)
- Requires service discovery, inter-service communication management, and deployment orchestration.(Con)
- Adds complexity in debugging and tracing cross-service workflows.(Con)

ADR 2: Integration with Codeforces via Dedicated Service

Decision: Create a dedicated `codeforces-service` for handling all interactions with the Codeforces API.

Status: Accepted

Context:

Multiple parts of the platform require data from Codeforces—user stats, problem metadata, and contest schedules. Direct integration in every service would lead to code duplication, tight coupling, and rate-limit issues.

Considered Options:

1. **Direct Calls from All Services:** Each service independently accesses the Codeforces API.
2. **Central Codeforces Service:** A unified service interfaces with Codeforces and exposes internal APIs to other services.

Decision:

Create a centralized `codeforces-service` to handle all Codeforces API interactions.

Rationale:

- **Decoupling:** Abstracts Codeforces API from the rest of the system.
- **Maintainability:** Central place for managing caching, retries, rate-limiting.
- **Extensibility:** Simplifies introducing additional Codeforces data in future features.

Consequences:

- Cleaner architecture and reuse of logic across services.(Pro)
- Single point of failure unless redundancy is added.(Con)
- Requires careful design of internal APIs for flexibility.(Con)

ADR 3: Use of Strategy Pattern for Authentication Mechanisms

Decision: Implement the Strategy Pattern for authentication logic to support multiple auth providers (e.g., Google OAuth, Email/Password).

Status: Accepted

Context:

The platform requires user authentication as a core feature, starting with Google OAuth to simplify login and enhance trust. However, supporting other strategies (e.g., email/password, enterprise SSO) in the future may become necessary as the user base diversifies. Hardcoding a single authentication flow would restrict flexibility and add overhead during future expansion.

Considered Options:

1. **Single Hardcoded OAuth2 Implementation:** Implement Google OAuth directly in the login flow without abstraction.
2. **Pluggable Strategy Pattern for Authentication:** Define an `AuthStrategy` interface with interchangeable strategies for different auth providers (Google, Email, etc.).

Decision:

Use the **Strategy Pattern** to encapsulate authentication behavior. Implement `GoogleAuthStrategy` as the initial strategy and support seamless addition of other strategies like `EmailAuthStrategy` through a factory (`AuthStrategyFactory`).

Rationale:

- **Extensibility:** Makes it easy to support additional authentication methods without rewriting core logic.
- **Clean Separation:** Keeps each provider's logic encapsulated and independently testable.
- **Maintainability:** Adding new strategies (e.g., enterprise SSO, mobile login) becomes a drop-in effort via the factory pattern.
- **Current Fit:** Google OAuth remains the default for MVP, but this decision future-proofs the design.

Consequences:

- Improves flexibility for future authentication methods.(Pro)
- Promotes clean, modular code architecture.(Pro)
- Introduces slightly more boilerplate for initial setup.(Con)
- Factory logic must be updated to register any new strategy.(Con)

ADR 4: Adoption of RabbitMQ for Asynchronous Event Handling

Decision: Use RabbitMQ as the message broker for async workflows (e.g., notifications, task assignments).

Status: Accepted

Context:

Many workflows, like notifying users of tasks, summarizing learner data, or generating insights, should not block the main application. The system should remain responsive even under load.

Considered Options:

1. **Synchronous Polling or Direct Service Calls:** Use HTTP-based calls between services or polling mechanisms where one service periodically queries another (e.g., frontend polling task status from backend).
2. **Message Broker (RabbitMQ):** Adopt a pub/sub architecture using message queues.

Decision:

Use RabbitMQ as the message broker for asynchronous event-driven communication.

Rationale:

- **Decoupling:** Emit and consume events independently across services.
- **Scalability:** Supports load buffering and horizontal scaling of subscribers.

- **Reliability:** Messages can be persisted, retried, and handled gracefully even during temporary failures.

Consequences:

- Enables resilient and scalable background processing.(Pro)
- Requires infrastructure setup, monitoring, and schema versioning.(Con)
- Introduces complexity in debugging distributed asynchronous workflows.(Con)

ADR 5: Use of PostgreSQL for Persistent Storage

Decision: Use PostgreSQL as the primary database for all microservices.

Status: Accepted

Context:

The platform stores structured data such as user profiles, mentor-learner mappings, task states, recommendations, and notification logs. Ensuring relational integrity and robust querying are critical.

Considered Options:

1. **MongoDB (NoSQL):** Flexible schema, document-based model.
2. **PostgreSQL (Relational DB):** Strong schema support and SQL querying.

Decision:

Adopt PostgreSQL for all microservices that require persistent data storage.

Rationale:

- **Relational Strength:** Mentor-learner relationships, task dependencies, and access control logic map well to relational models.
- **Data Consistency:** ACID compliance helps prevent inconsistencies during concurrent writes (e.g., task progress updates).
- **Developer Familiarity:** The team is comfortable with SQL and ORM tools.

Consequences:

- Reliable and robust for structured, interrelated data.(Pro)
- Strong support for schema migrations and constraints.(Pro)
- Less flexible for highly nested or schema-less data (e.g., LLM outputs).(Con)
- Potential performance overhead for read-heavy analytics unless optimized.(Con)

Task 3: Architectural Tactics and Patterns

Architectural Tactics

1. Security Tactics

1.1. Authentication via OAuth (Google)

Non-functional Requirements Addressed: Confidentiality, Integrity

Implementing OAuth integration with Google delegates user authentication to a trusted external provider. This eliminates the need to store sensitive password data internally while providing a familiar, secure, and streamlined login experience for users.

1.2. Authorization via Role-Based Access Control (RBAC)

Non-functional Requirements Addressed: Access Control, Security

Our system enforces RBAC by clearly defining user roles with specific permissions. Access control checks are embedded at both API and service layers, ensuring granular permission management. This structured authorization minimizes the risk of unauthorized access and reinforces the principle of least privilege.

---

2. Availability Tactics

2.1. Heartbeat Monitoring

Non-functional Requirements Addressed: System Uptime

These checks enable real-time alerts, ensuring that failures are detected and addressed promptly for continuous system availability.

2.2. Rate Limiting via Nginx

Non-functional Requirements Addressed: Availibility, Reliability

Nginx is configured for rate limiting at API endpoints to prevent service degradation caused by excessive requests. This tactic protects against denial-of-service attacks and enforces fair resource allocation among users while keeping the system robust under high load.

---

3. Modifiability Tactics

3.1. MVC Pattern Implementation → Modifiability (Design Time Separation)

Non-functional Requirements Addressed: Adaptability, Maintainability

The Model-View-Controller (MVC) pattern separates data (Model), presentation (View), and business logic (Controller), allowing independent development, testing, and maintenance. This separation not only simplifies updates to specific components (such as UI enhancements) but also optimizes resource usage across the system.

---

4. Usability Tactics

4.1. Containerization with Docker

Non-functional Requirements Addressed: Consistency, Efficiency, Operational simplicity

Docker containerization standardizes environments across development, testing, and production. This tactic simplifies deployment and scaling, reduces configuration complexity, and improves the overall experience, leading to a more streamlined operational workflow.

---

5. Scalability Tactics

5.1. Microservices Architecture

Non-functional Requirements Addressed: Scalability, Resilience

By decomposing the system into independent microservices, we enable the individual scaling of components based on demand. This isolation of functions improves overall fault tolerance and allows updates to be deployed to specific services without impacting the entire system.

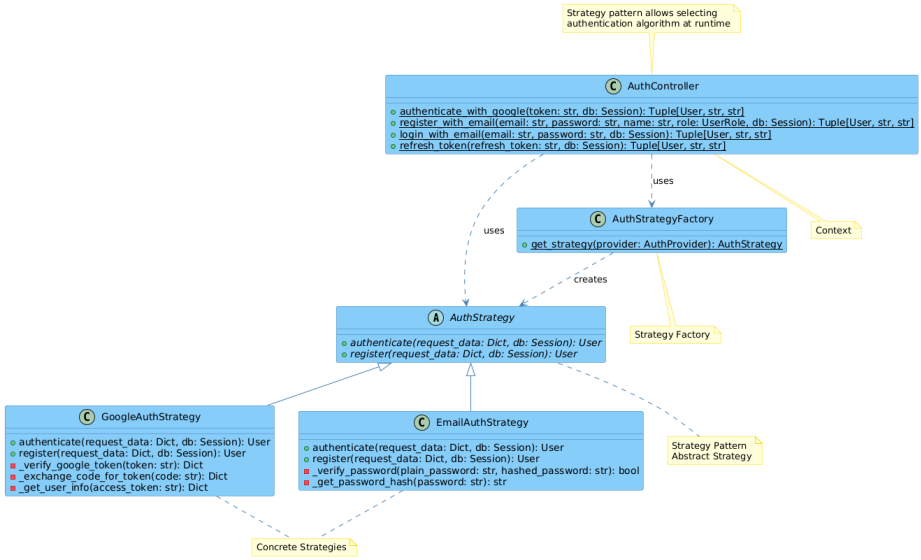
5.2. Message Broker (RabbitMQ)

Non-functional Requirements Addressed: Traffic Management, Scalability, Decoupled Communication

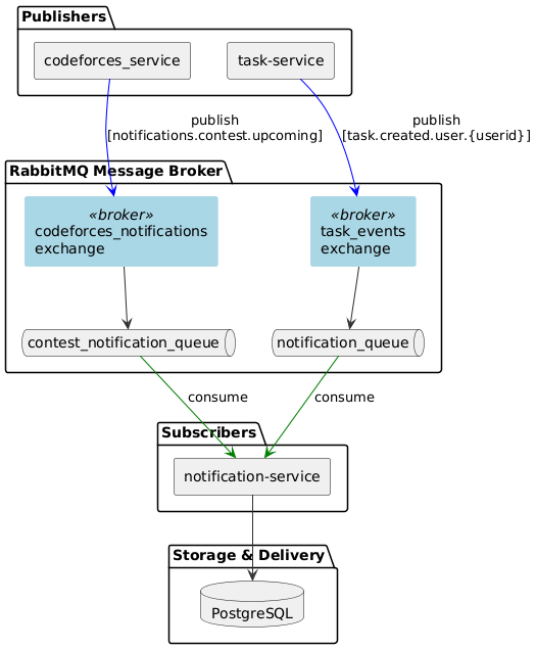
Implementing RabbitMQ for asynchronous pub/sub messaging decouples service communications. This approach manages traffic spikes effectively, facilitates asynchronous processing, and enhances system resilience by ensuring messages are reliably persisted and delivered even during high-load periods.

Implementation Patterns

1. Strategy Pattern (Low-level Design Pattern):



2. Publish-Subscribe with RabbitMQ (Architectural Design Pattern):



The system adopts a **publish-subscribe (Pub/Sub) architecture** using **RabbitMQ** as the central **message broker** for decoupled, event-driven communication. Publishers (e.g., `codeforces_service`, `task-service`) emit structured events to **topic exchanges**, while subscribers (e.g., `notification-service`) consume from **bound queues** using **routing key patterns**.

- **Event Types**:
  - **Broadcast Events** (e.g., upcoming contests): Published to a topic exchange (`codeforces_notifications`) with a specific routing key like `notifications.contest.upcoming`.
  - **Targeted Events** (e.g., task assignments): Published to another exchange (`task_events`) using dynamic keys like `task.created.user.{user_id}`.
- **Message Flow**:
  1. Publishers serialize structured messages (e.g., `TaskCreatedMessage`, `ContestNotificationMessage`) and publish them with a `type` header to RabbitMQ exchanges.
  2. RabbitMQ routes messages to queues (e.g., `contest_notification_queue`, `notification_queue`) based on binding keys.
  3. Subscribers (e.g., `NotificationConsumer`) process messages asynchronously, validate schema, and persist them to **PostgreSQL**.
- **Benefits**: **Loose coupling** between services; **Scalable and resilient** via persistent queues and asynchronous consumers; **Flexible routing** using **topic exchanges** and wildcard keys; Enforces **strict schema validation** for compatibility and traceability.

3. Models, Controllers, Services (MCS) Pattern (Architectural Design Pattern):

The Models-Controllers-Services (MCS) pattern is a modular software architecture style that helps us implement separation of concerns, increases maintainability and scalability.

It consists of 3 core components:

- **Models:** define the structure of the data and how it's stored.
- **Controllers:** handle incoming requests and act as intermediaries between the services and the different endpoints.
- **Services:** encapsulate business logic and core operations, keeping controllers lean and focused.

This structure ensures that:

- Logic isn't duplicated or scattered,
- Each file has a single responsibility,
- It's easier to debug, test, and maintain the codebase.

We have implemented the MCS pattern in `ai_service` , `task-service` , `user-service` and `notification-service` .

- **Models:**

  These define schemas or ORM models that represent entities in the application, such as users, contests, or tasks. They typically interact with the database layer.

  Example: `user_model.py` and `user_stats.py` define the user structure.

- **Controllers:**

  Controllers manage API logic and request-response flows. They receive HTTP requests, invoke the appropriate service functions, and return responses.

  Example: `auth_controller.py` handles authentication logic, `contest_controller.py` routes contest-related logic.

- **Services:**

  Services are where business logic resides. This includes things like data aggregation, validation, notifications, and third-party API calls.

  Example: `auth_service.py` handles token creation and user authentication, `notification_service.py` sends out notifications using messaging queues, `ai_recommendations.py` generates AI-based suggestions.

Benefits: Separation of concerns into models, business logic and request handling leads to **higher readability**, **loose coupling** between components, along with following the **single responsibility principle.** This makes the system highly **maintainable, scalable** and **testable.** Individual parts of the system can be updated without needing to make changes in the whole system.

3. Microservices Pattern (Architectural Pattern/Style):

Our system employs the microservices architectural pattern to decompose the application into loosely coupled, independently deployable services. Each service is responsible for a specific business capability and communicates through well-defined APIs. This approach offers several advantages:

- **Modularity**: Services can be developed, deployed, and scaled independently, allowing the team to focus on specific business domains.

- **Technology Flexibility**: Different services can use appropriate technologies based on their specific requirements.
- **Resilience**: Failure in one service doesn't cascade to the entire system.
- **Scalability**: High-demand components can be scaled independently without scaling the entire application.

In our implementation, services (Codeforces Service, AI Service, Orchestration Service, User Service, Notification Service and Task Service) support various subsystems while maintaining separation of concerns. These services communicate via REST APIs and message queues (RabbitMQ), with the API Gateway handling service discovery and request routing.

Task 4

Prototype Development

Github repository link: https://github.com/umangapatel123/CPLite

Architecture Analysis

This report compares two architectural patterns for handling event notifications within our system, specifically focusing on the notification subsystem responsible for alerting users about events like new task creation.

- **Implemented Architecture:** The original implementation used a **Publish/Subscribe (Pub/Sub)** pattern facilitated by RabbitMQ. Event-producing services published messages to a RabbitMQ exchange, and the notification service (`NotificationConsumer`) subscribed to relevant queues to receive these messages in near real-time.
- **Alternative Architecture:** The current notification subsystem utilizes a **Polling** mechanism. A dedicated service (`NotificationPoller`) periodically queries an API endpoint to check for new events (e.g., tasks created since the last check).

Aspect	Original Architecture (RabbitMQ Pub-Sub)	New Architecture (Polling)
Communication Model	Asynchronous messaging	Periodic synchronous HTTP polling
Technology Used	RabbitMQ	REST API via HTTP + Python asyncio
Message Flow	Producer pushes event to broker; consumer subscribes and reacts in real-time. The consumer's `callback` is invoked immediately upon receiving a message, triggering the	Poller periodically pulls new data via REST API. The `NotificationPoller` receives the list of new events (if any), processes each one (formats and sends notifications via `NotificationService`), and updates

Aspect	Original Architecture (RabbitMQ Pub-Sub)	New Architecture (Polling)
	notification logic via `NotificationService`.	its `last_poll_time` marker for the next cycle.
Triggering Mechanism	Event-driven	Time-driven (based on `poll_interval`)
Dependencies	Relies on the availability and performance of the RabbitMQ broker. Requires producers to integrate RabbitMQ client libraries to publish messages.	Relies heavily on the availability and performance of the polled API endpoint and the underlying service/database. Requires logic to handle authentication (fetching service tokens) and manage the `last_poll_time` state.

Non-Functional Requirements

Pub-Sub Architecture Pattern (Event Driven):

https://p.ip.fi/

Polling Architecture Pattern (Client-Server) :

https://p.ip.fi/

Comparison

a. Response Time

Architecture	Average Response Time (to be filled)	Description
Pub-Sub	3.48 ms	Near-instantaneous delivery of messages once events are published
Polling	7667.57 ms	Latency introduced by polling interval and HTTP request overhead

- **Outcome**: Pub-sub generally yields better response time because messages are pushed immediately. Polling has delays depending on the polling frequency.

b. Throughput

Architecture	Max Throughput (to be filled)	Description
Pub-Sub	0.21	Designed to handle high-frequency event flow, decouples producer/consumer
Polling	0.08	Limited by request-response cycle and frequency of polling

- **Outcome**: RabbitMQ handles burst traffic more efficiently as it buffers messages and allows consumers to scale. Polling can bottleneck under high load if REST endpoints are not optimized or rate-limited.

Trade Off Analysis

Trade-Off	RabbitMQ Pub-Sub	Polling
Scalability	Highly scalable via consumer groups, queues	Limited by polling interval and HTTP load
Real-time Capability	Near real-time notifications	Depends on polling interval
Operational Complexity	Requires RabbitMQ setup and maintenance	Easier deployment, fewer moving parts
Network Load	Efficient (push-based)	Higher (repeated polling, even with no updates)
Failure Tolerance	High (messages buffered in broker)	Risk of missing updates if polling fails
Service Coupling	Decoupled (producer and consumer isolated)	Tightly coupled to availability of API
Debugging & Observability	Requires specialized tools (e.g., RabbitMQ UI)	Easier (plain HTTP + logs)

We chose RabbitMQ over HTTP polling because it provides near real-time notification delivery with significantly lower latency, reduces system load by eliminating continuous database queries, and offers better scalability as notification volume increases. Our performance tests confirm RabbitMQ delivers higher throughput while consuming fewer resources.

Reflections and Lessons Learned

Working on this project taught us a lot about building scalable and modular systems. Designing with microservices helped us clearly separate concerns, making the system easier to extend and maintain.

One of the key lessons was how important it is to think about non-functional requirements (like scalability, security, and availability) early in the design process. These shaped many of our architectural decisions and ensured the system could handle real-world usage.

We also realized how important teamwork and clear communication are, especially when working on different services that depend on each other. Collaboration and regular check-ins helped us stay aligned and avoid integration issues.

Lastly, though some features (like gamification and the discussion forum) weren't fully implemented, planning for them in advance showed us how future-ready design can make adding features easier down the line.

Overall, this project helped us connect theory to practice and better understand what goes into building a real-world software system.

Contributions

Everyone contributed equally to the project.