TaskFlow: A Distributed Tactical Mission Control System

Final Report for the Distributed Systems Course 2025/2026

Ahmed Trabelsi

ahmed.trabelsi2@studio.unibo.it

Lotfi Toumi

lotfitoumi@studio.unibo.it

April 23, 2026

Effective coordination between centralized command centers and distributed field operatives requires strict state synchronization across unpredictable network topologies. TaskFlow is a distributed workforce management platform utilizing a microservices architecture that separates rigid HTTP REST data operations from real-time WebSocket streams. This report outlines the system requirements, architectural design choices, distribution aspects, and fully containerized deployment strategies. By implementing an event-driven messaging layer via RabbitMQ, the system prioritizes high availability and graceful fault tolerance, ensuring deterministic data consistency for distributed mobile clients operating in volatile environments.

Disclaimer

During the preparation of this work, the authors utilized automated containerization tools (Docker) and API testing frameworks to validate network consistency. The authors reviewed and edited the configurations as needed and take full responsibility for the content of the final report and the deployed software artifact.

1 Concept

- **Type of product:** The system is a composite distributed platform consisting of a Web Application (GUI Dashboard) for Administrators, a Mobile Application for Field Operatives, and a backend ecosystem of autonomous Web-services.
- **Use case description:**
  - *What is the software doing?* It facilitates real-time tactical mission deployment, strict user authorization, and asynchronous chat coordination.
  - *Where are the users?* Users are geographically distributed; commanders operate in a central HQ, while operatives are deployed in varied field locations.
  - *When and how frequently?* Interactions are continuous and demand near zero-latency updates during active missions.
  - *How do they interact?* Administrators use desktop browsers (Angular Web Client), while operatives use mobile devices (Ionic App on 4G/5G).
  - *Data storage:* The system stores user credentials, task parameters, and message histories across four distinct, domain-isolated PostgreSQL databases.
  - *Roles:* There are strict asymmetric roles: Command Administrators and Field Operatives.
- **Why is distribution needed?**
  - *Geographically distributed environments:* Field operatives must receive intelligence across diverse and unstable mobile networks.
  - *Fault tolerance:* The system must gracefully handle network partitions without losing critical mission data.
  - *Computation and Load Isolation:* Heavy real-time WebSocket broadcasting must not block standard HTTP REST API threads, necessitating physical service separation.

2 Requirements Elicitation and Analysis

- **Functional Requirements:**
  1. **Role-Based Interfaces:** Provide a web dashboard for commanders and a mobile client for operatives. *Acceptance Criterion*: Administrators can access the web UI; operatives are restricted exclusively to the mobile UI.

2. **Asymmetric Access Control:** Operatives register via mobile but remain inactive until approved. *Acceptance Criterion*: Unapproved users receive an HTTP 403 Forbidden error upon login attempt.
3. **Task Lifecycle Management:** Centralized creation, assignment, and monitoring of tasks. *Acceptance Criterion*: Tasks persist in the Task DB and immediately appear on the targeted operative's device.
4. **Real-Time Event Delivery:** Tasks and alerts are pushed instantly via WebSockets. *Acceptance Criterion*: Operatives receive JSON payloads natively without manual HTTP polling.
5. **Secure Communications:** Bidirectional real-time chat must be supported. *Acceptance Criterion*: Messages are delivered successfully via the AMQP broker.

- **Non-Functional Requirements:**
  - **Consistency:** Task states must be reliably recorded without duplication.
  - **Availability:** The Gateway must remain active even under heavy messaging load.
  - **Fault Tolerance:** Mobile clients must recover from network disconnections seamlessly.
- **Implementation Requirements:**
  - The backend must utilize **Java 21** and Spring Boot to leverage modern virtual threads for I/O operations.
  - The entire stack must be orchestrated via **Docker** to ensure platform-agnostic deployment across different host environments.

2.1 Relevant Distributed System Features

- **Transparency:** Highly relevant. Mobile and Web clients interact solely with the API Gateway, completely unaware of the internal network IPs, Eureka registry, or database locations.
- **Fault Tolerance & Dependability:** Highly relevant. The system must prevent data loss during network partitions by allowing clients to perform a durable REST state-fetch upon reconnection.
- **Scalability:** Relevant. Using Eureka service discovery allows the system to handle an increasing number of operatives by dynamically scaling bottlenecked services.

- **Security & Trust:** Highly relevant. Sensitive tactical data is processed. The system requires strict authorization via stateless cryptographic tokens (JWT).
- **Performance & Concurrency:** Relevant. The separation of REST APIs and WebSocket STOMP channels ensures that high volumes of concurrent real-time messages do not exhaust the primary HTTP thread pools.

3 Design

3.1 Architecture

We selected a **Microservices Architecture**. A monolithic style was rejected because combining heavy, persistent WebSocket connections (for chat) with rigid, transactional HTTP requests (for authentication) creates severe resource contention. By isolating domains, we ensure that a spike in messaging traffic does not cause authentication timeouts.

As illustrated in fig. 1, the system abstracts the internal backend complexity from the users.

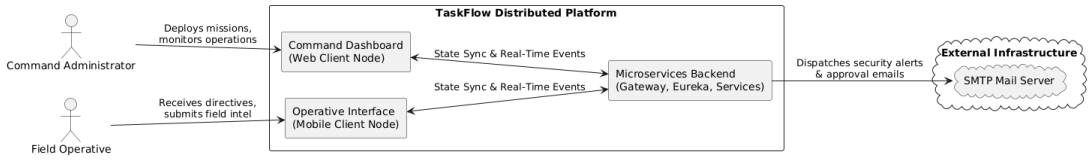

Figure 1: System Context Diagram highlighting actor interactions with the platform.

3.2 Infrastructure

The infrastructural components are distributed within an isolated Docker bridge network (fig. 2).

- *Components:* Spring Cloud Gateway (Proxy), Netflix Eureka (Service Discovery), RabbitMQ (Message Broker), and 4 separate PostgreSQL instances.
- *Distribution:* All containers sit on the same host network currently, but are logically segregated by internal IP and hostname routing.
- *Discovery:* Services find each other dynamically using the Eureka Service Registry, eliminating hard-coded IP addresses.

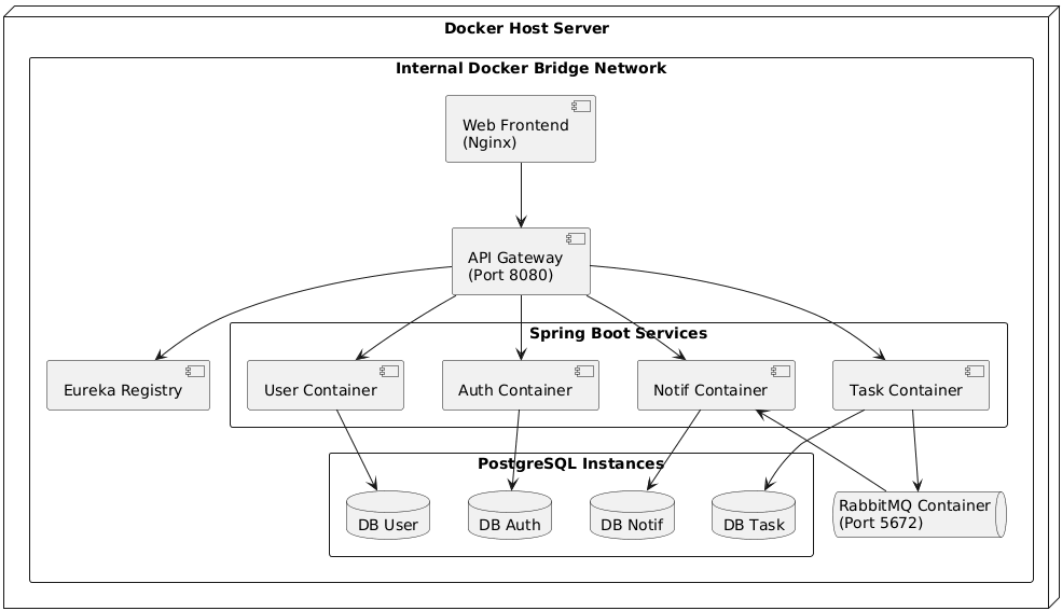

Figure 2: Docker Container Deployment Topology.

3.3 Modelling

- *Domain Entities:* User, Task, Notification, Message.
- *Mapping:* Each domain entity maps 1:1 to a specific microservice and its own isolated database. Client UI states are representations of these backend entities.
- *Domain Events: UserRegisteredEvent*, *TaskAssignedEvent*.
- *State:* The system state is strictly partitioned. No single database holds all information, reducing the blast radius of a potential database failure.

3.4 Interaction

Components communicate via three interaction patterns (fig. 3):

- **Synchronous Client-to-Service:** External HTTP requests routed by the Gateway.
- **Full-Duplex Streaming:** STOMP over WebSockets for continuous real-time signaling.
- **Asynchronous Service-to-Service:** Internal AMQP event publishing via RabbitMQ.

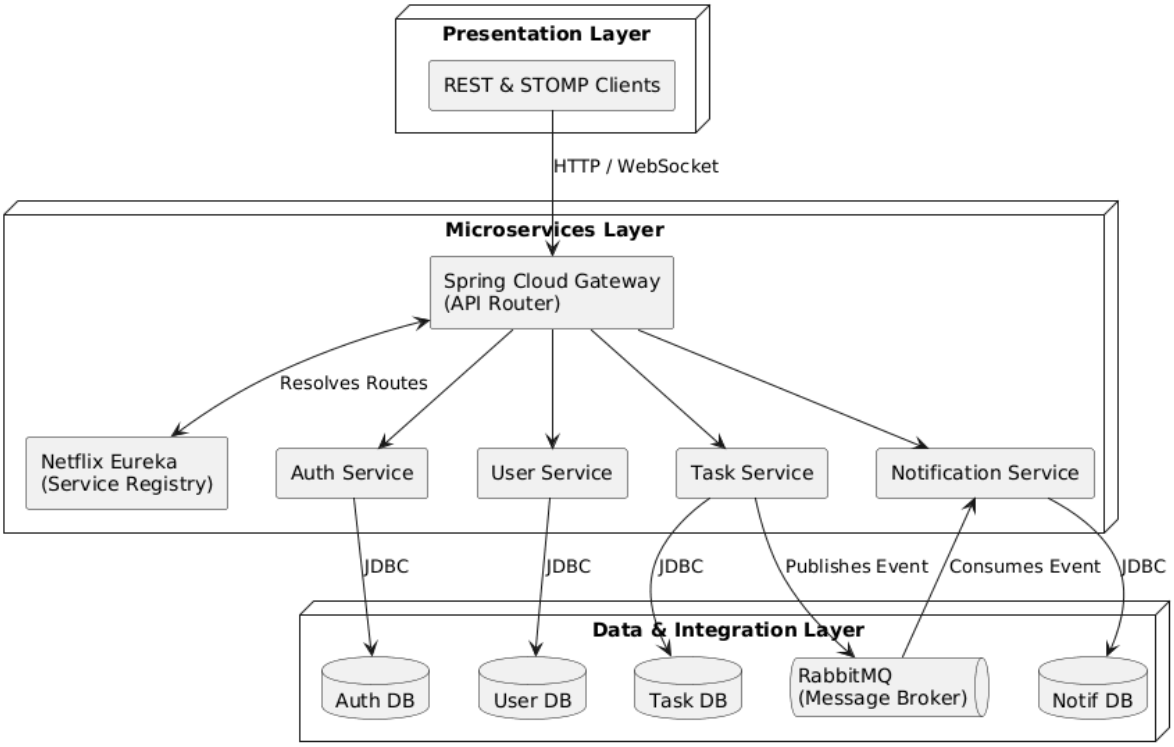

Figure 3: Logical Node Architecture Diagram.

3.5 Behaviour

Services are fundamentally **stateless** at the application layer. When a command is received (e.g., an Admin approves an operative), the User Service updates its specific database (Stateful component) and fires a "UserApproved" event to RabbitMQ. The Notification Service consumes this event and dispatches an email independently.

3.6 Data and Consistency Issues

- *Storage:* Relational data (PostgreSQL) is used to enforce strong ACID properties within each isolated microservice boundary.
- *Queries:* Only the service that "owns" the data can query it. There are no shared databases.
- *Consistency:* Across the distributed network, we rely on **Eventual Consistency** (BASE semantics). Events are published to the broker strictly *after* local database commits, ensuring downstream services and mobile clients eventually reflect the correct state.

3.7 Fault-Tolerance

- *Retry Mechanism:* RabbitMQ operates on an **At-Least-Once** delivery guarantee. If a consumer service crashes, unacknowledged messages remain in the queue and are redelivered upon restart.
- *Error Handling:* If a mobile client loses 4G connectivity, the WebSocket drops. Upon reconnection, the client catches the event and explicitly fetches missed state updates via REST.

3.8 Availability

- *Load Balancing:* Spring Cloud Gateway provides built-in client-side load balancing, routing traffic to active instances registered in Eureka.
- *Network Partitioning:* If a microservice node gets partitioned from the registry, Eureka removes its heartbeat, and the Gateway ceases routing traffic to it, ensuring clients do not face indefinite timeouts.

3.9 Security

- *Authentication & Authorization:* We utilize stateless JSON Web Tokens (JWT) for Role-Based Access Control (RBAC).
- *Cryptographic Schemas:* As seen in fig. 4, the JWT is signed with an HMAC SHA-256 secret. The API Gateway validates this cryptographic signature statelessly on every incoming request.

Additionally, the system requires an asymmetric trust lifecycle for registration, handled via the administrative mutation shown in fig. 5.

4 Implementation

- **Network protocols:** HTTP/REST for standard operations, WebSockets (STOMP) for client real-time feeds, and AMQP 0-9-1 for internal service coordination.
- **In-transit data:** All payloads and message queue events are represented as JSON.
- **Database Queries:** Managed via Spring Data JPA and Hibernate (SQL).
- **Authentication/Authorization:** Handled via encrypted JWTs passed in Bearer headers.

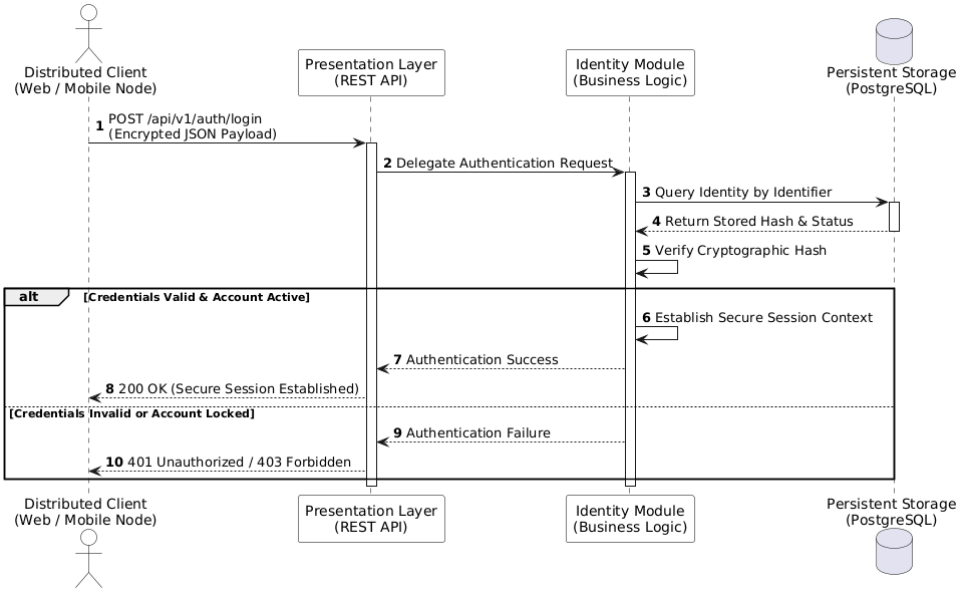

Figure 4: Stateless Authentication Sequence.

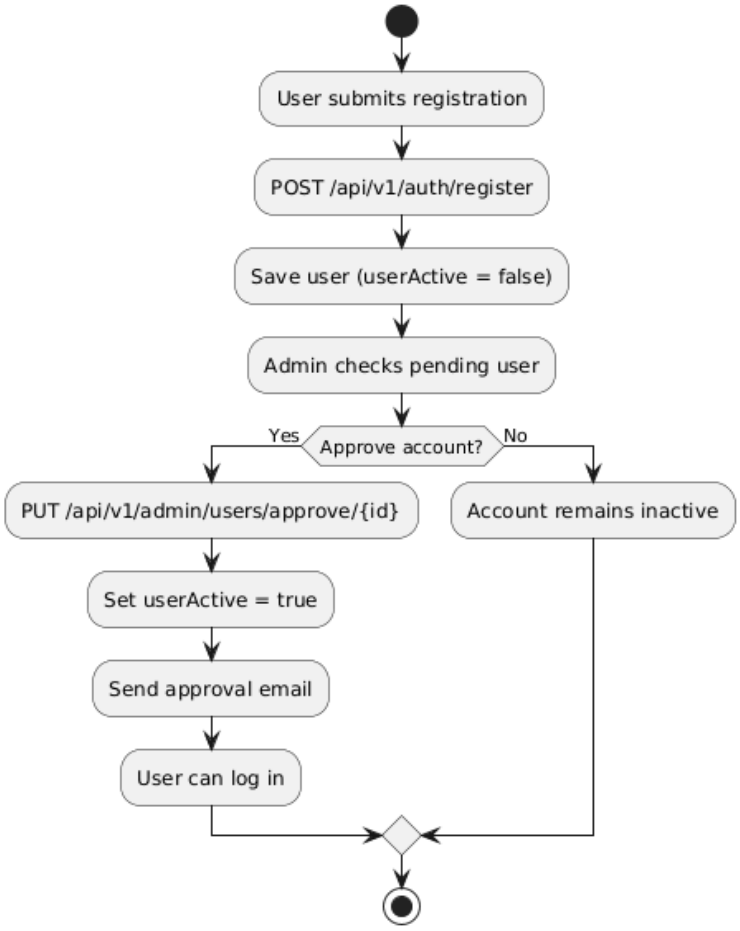

Figure 5: User Registration Activity Flow.

4.1 Technological details

The backend relies on Java 21, Spring Boot 3.2, and Spring Cloud. The web interface utilizes Angular, while the mobile interface utilizes Ionic and Capacitor.

5 Validation

5.1 Automatic Testing

- **Unit testing:** Implemented using JUnit 5 and Mockito. Services are tested in isolation by mocking database repositories and RabbitMQ templates.
- **Automation:** Tests are automated via Maven (`mvn test`). They verify that JWT validation logic, core REST controllers, and AMQP event publishing trigger the correct payload generations.
- **Requirement tested:** Verifies the strict data consistency and role-based access control requirements programmatically.

5.2 Acceptance test

- *Manual Testing:* We performed manual end-to-end testing of the mobile client.
- *Why manual?:* Simulating physical 4G network drops and testing the native Capacitor device APIs for reconnection events is extremely difficult to mock accurately in a CI/CD pipeline. We verified that switching the mobile device to airplane mode and back successfully triggered the REST resynchronization routines.

6 Deployment

As requested, the deployment instructions from the project's README are provided below. The system is designed to be executed with a minimal Docker setup.

Requirements

- Docker + Docker Compose
- Node.js 18+ and npm

Quick Start

1. Start the backend

```
cd backend
docker-compose up --build -d
```

This starts all microservices, databases, API Gateway, Eureka, and RabbitMQ. Wait ∼60 seconds for all services to register.

2. Start the web frontend

```
cd frontend-web/app
npm install
npm start
```

Opens at ***http://localhost:4200***

3. Start the mobile frontend

```
cd frontend-mobile/app
npm install
ionic serve
```

Opens at ***http://localhost:8100***

Default Admin Account

```
Email:    lotfitoumi56@gmail.com
Password: admin123
```

7 User Guide

The following interfaces demonstrate the practical usage and expected outcomes of the system requirements.

- **Administrative Web Client:** fig. 6 shows the core dashboard, while fig. 7 displays the asymmetric access control queue. fig. 8 demonstrates task deployment.
- **Operative Mobile Client:** fig. 9 and fig. 10 show the expected outcomes of the STOMP/AMQP real-time communication bridges successfully delivering data to a mobile node.

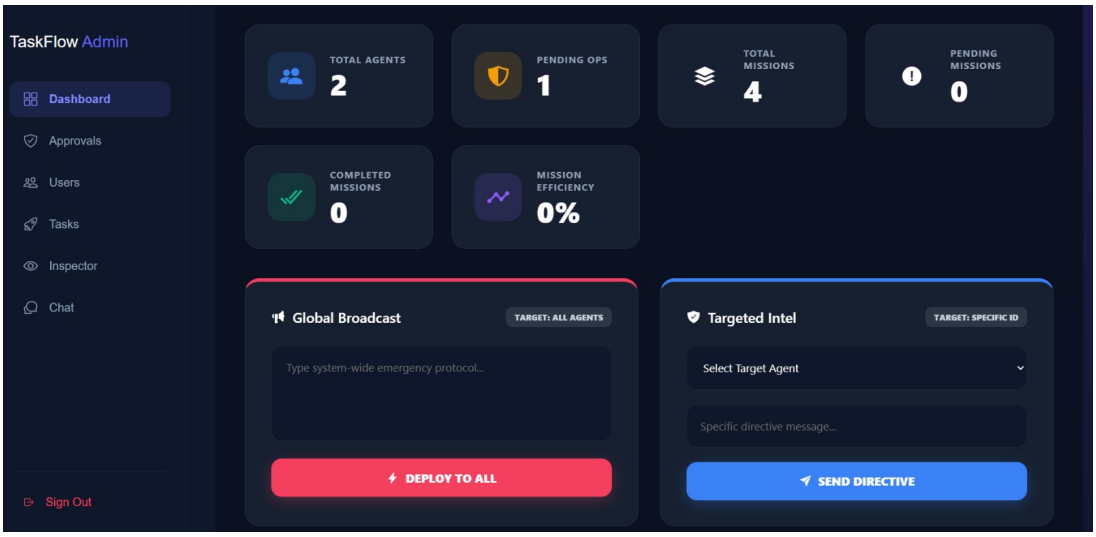

Figure 6: Angular Web Client displaying synchronized DB state.

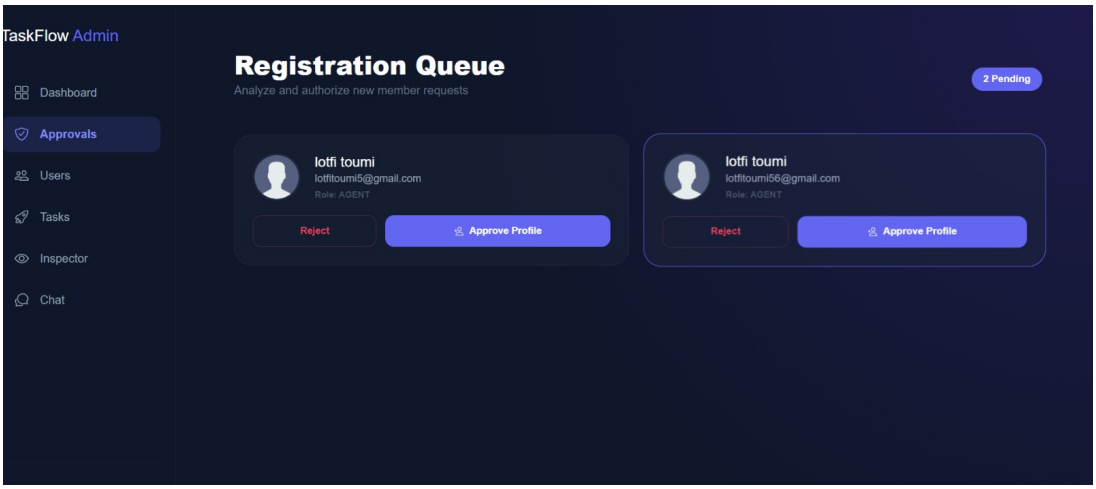

Figure 7: Administrative authorization interface.

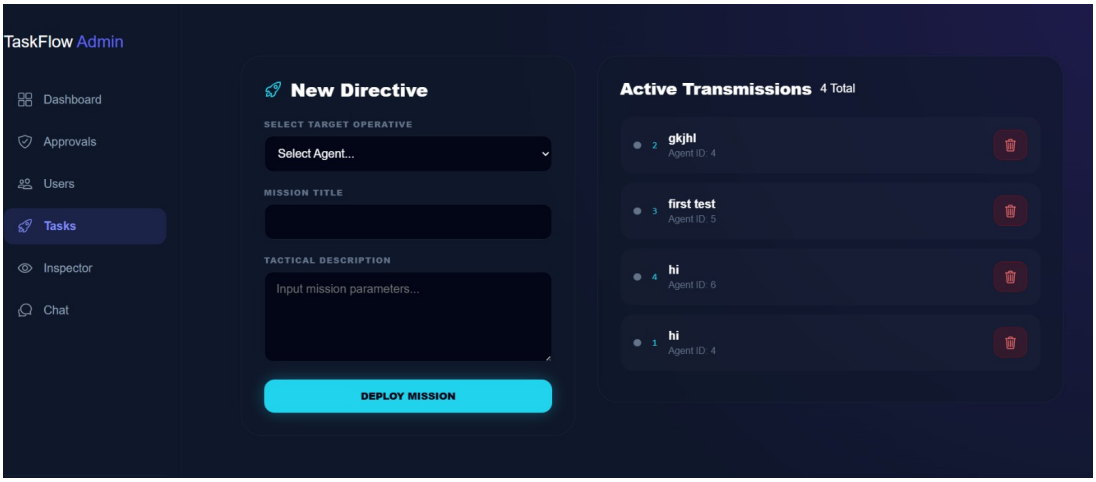

Figure 8: Command Dashboard for deploying directives.

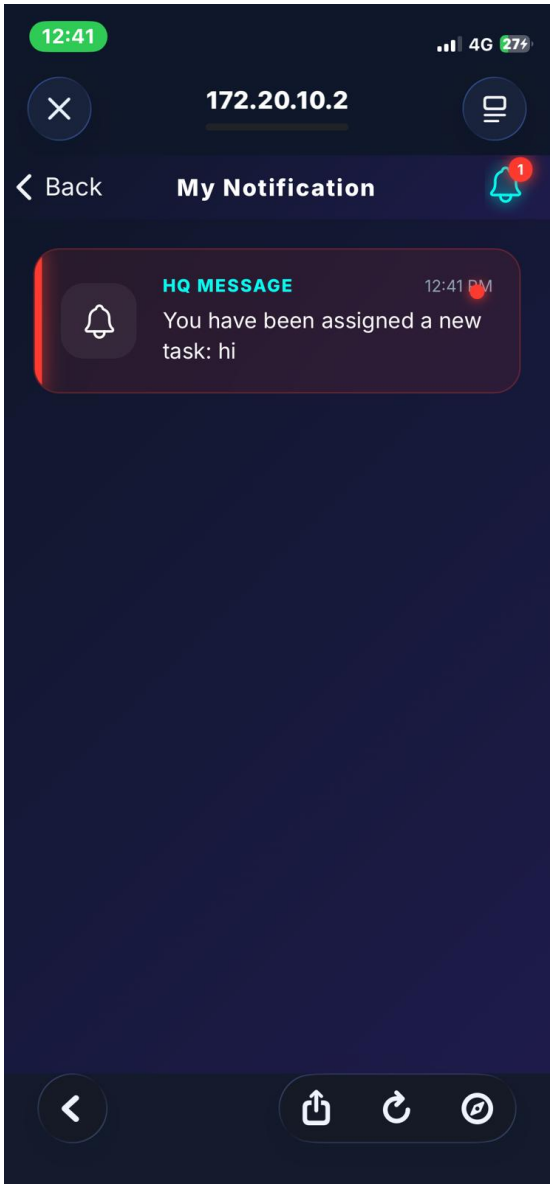

Figure 9: Mobile Node receiving real-time AMQP task notifications.

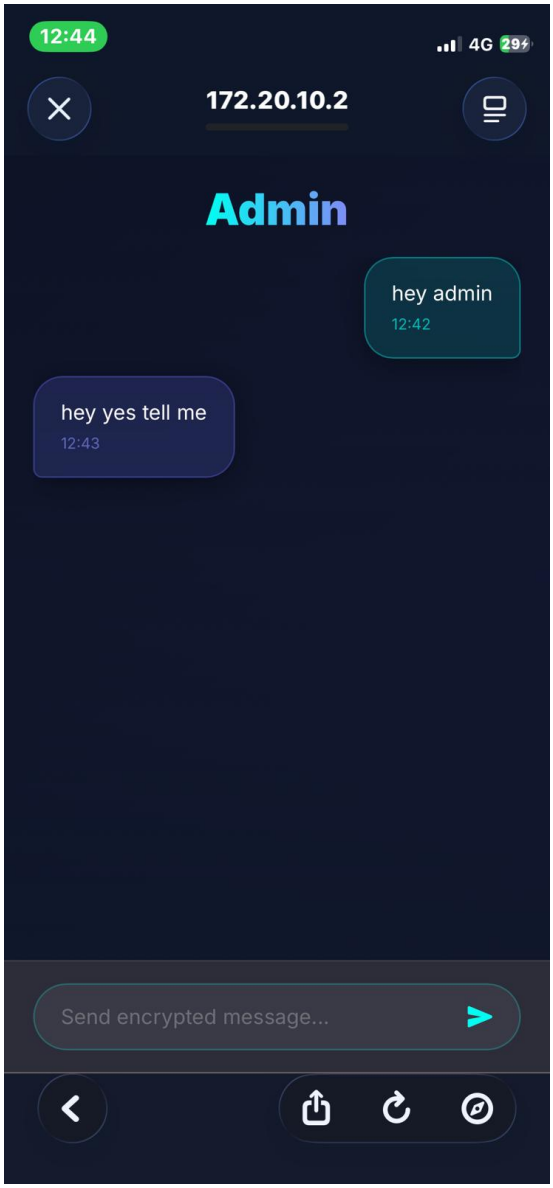

Figure 10: Mobile Node demonstrating real-time AMQP chat.

8 Self-evaluation

Ahmed Trabelsi: I led the development of the presentation layer, bridging the clients with the distributed backend. I engineered both the Angular web dashboard and the cross-platform Ionic application. A strength of my work was successfully implementing the STOMP WebSocket client to achieve zero-latency real-time synchronization on mobile devices. A significant challenge overcome was managing complex CORS configurations across the distributed Gateway.

Lotfi Toumi: My primary role focused on engineering the distributed backend infrastructure. I implemented the Spring Cloud API Gateway, the Eureka Service Registry, and the core microservices. A major strength of the product is the robust decoupling achieved via RabbitMQ for asynchronous events. A potential weakness is the current reliance on a single Docker host, limiting true physical node distribution.

9 Future works

Future iterations will focus on migrating the deployment topology to a Kubernetes cluster to achieve true physical node distribution and automated pod scaling. Additionally, implementing database replication (Leader-Follower) will eliminate the databases as single points of failure.

Note on Evaluation: To fulfill the demonstration requirement, a complete, pre-recorded video demonstrating the live deployment and real-time execution of the system has been prepared and will be provided to Prof. Omicini.

References

[1] Rabbitmq documentation: Reliability guide and amqp 0-9-1 model explained, 2025.

[2] Spring framework documentation: Messaging with stomp over websockets, 2025.

[3] Eric A. Brewer. Towards robust distributed systems. In *PODC*, 2000.

[4] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design.* Addison-Wesley, 5th edition, 2011.