Remotix

TEAM 8

remotely operate system

Course:

Software Engineering (10177_001)

Instructor:

정옥란 교수님

Members:

김다인 (202234861)
김민재 (202138108)
박성언 (202334289)
박준혁 (202035522)
차준하 (202135584)

1. Introduction

1.1 Motivation

Teleoperation enables administrators to remotely control a physical robot, making it highly valuable in fields such as technical support, remote supervision, and smart infrastructure.

TurtleBot, a low-cost and extensible robot platform, is well-suited for teleoperation due to its compatibility with ROS (Robot Operating System) and ease of integration.

The **Remotix system** provides a **web-based interface** that allows an administrator to control a user's TurtleBot in real time, monitor its status, and write session reports. It is designed to be intuitive, responsive, and suitable for on-site robot supervision and control.

1.2 Scope of the system

The **Remotix system** is designed for scenarios where an administrator or supervisor needs to **remotely operate a user's TurtleBot**, particularly in support or monitoring settings.
The system offers real-time control, live status monitoring, and a way for the administrator to manually write and upload reports after each session.

In scope:

- Admin-only manual control of TurtleBot via web interface
- Live video and sensor feedback
- Manual report submission and cloud-based report storage

1.3 Objectives and success criteria of the project

Objectives:

- Provide a web-based interface for **administrators to remotely control** a user's TurtleBot
- Enable **real-time monitoring** of camera and sensor data during operation
- Allow the administrator to **manually create and submit control reports** after each session
- Support **remote operation, supervision, or troubleshooting** without requiring physical access to the robot

Success Criteria:

- TurtleBot responds immediately to admin commands via the web interface
- Live camera and sensor data are displayed with minimal latency
- Administrator-written reports are successfully submitted and stored on the cloud database
- The system operates reliably in multiple real-world test sessions

1.4 Technical Skill

The **Remotix system** is a web-based teleoperation platform that manages user authentication, **receives control-related requests from users**, and forwards commands to the robot and saves reports.

It consists of the following key components and features:

- **Web-based Control Interface**
    - A user interface that allows the administrator to control the TurtleBot's movement
    - Movement commands: forward, backward, turn left, turn right
    - Receives control-related requests from users and delivers them to the administrator for processing
    - Clean and intuitive layout for efficient remote operation
- **Real-Time Monitoring**
    - Live feedback from the TurtleBot's onboard camera and sensors
    - Visual monitoring via live camera feed
    - Sensor data such as speed, battery level, and other relevant metrics
- **Report Writing and Cloud Storage**
    - After each session, the administrator manually writes a report summarizing the operation
    - Reports are submitted via the web interface by the administrator
    - Submitted reports are stored in a **database** hosted on **Cloud Platform.**
- **Development & System Integration**
    - Robot control is implemented using ROS, running on a Raspberry Pi
    - The backend API server, built with FastAPI, manages user authentication, report submission, and command processing.
    - The frontend is developed using React, providing an intuitive web-based interface for administrators to control and monitor the robot.
    - Reports are written manually and submitted through the interface, then stored in a MySQL database server.
    - The entire system is containerized using Docker to ensure consistent development and deployment, and Git is used for version control and collaborative development.

2. Specification and Implementation

2.1 Development Process

Because our team is building an interface for an autonomous-driving system, a **safety-critical** application we chose **a plan-driven** development approach. Every subsystem was implemented from a detailed upfront design, strictly following the **waterfall model**.

2.2 Overview

Robot & System Integration

- ROS-based control of TurtleBot using rospy
- Robot-side communication via Raspberry Pi
- Web integration via rosbridge_suite and roslibjs

Frontend Development

- React-based web UI for control and monitoring
- The web interface displays live camera and sensor data during robot operation.
- Report writing and submission interface

Backend Development

- Backend server built with FastAPI
- User login and session management
- Report submission API for administrators to upload manually written reports
- Deployment on Naver Cloud Platform, including configuration of virtual machines, database access, and network security
- Integration with a MySQL database to store user credentials and report data
- Docker is used to containerize the backend server and database environment for easier deployment and consistency across systems.

Database

- Design and management of a MySQL database for storing user credentials and report data
- Use of SQLAlchemy for database interactions
- Support for report listing, searching, and structured storage

2.3 Requirements analysis

- **User and System Requirements**
  In the envisioned remote-assistance scenario, field personnel must be relieved of the burden of manual incident reporting: as soon as an issue arises, the system automatically submits an incident to the support center and provides the requester with immediate confirmation that their report was successfully received. On the other side, a trained operator accesses a consolidated list of incoming requests, selects a case, and establishes a live, two-way connection with the TurtleBot. During each session, the operator monitors real-time video feed from the robot's camera, issues motion commands—forward, backward, left, right, and stop—and, upon completion of the task, records a concise session report.

  To support these workflows, the backend must reliably receive, store, and forward each incident report to the appropriate operator, while continuously tracking the request's status. Simultaneously, it maintains a low-latency control link to the TurtleBot for sending commands and streaming video, and presents the robot's operational status (e.g., battery level, network quality) in real time.

- **Functional and Non-Functional Requirements**
  Functionally, the system exposes an interface through which operators can query all pending incident reports, accept a selected request, and initiate live remote control of the TurtleBot, including directional commands and video monitoring. It also supports post-session logging so that operators can document outcomes and any anomalies encountered.

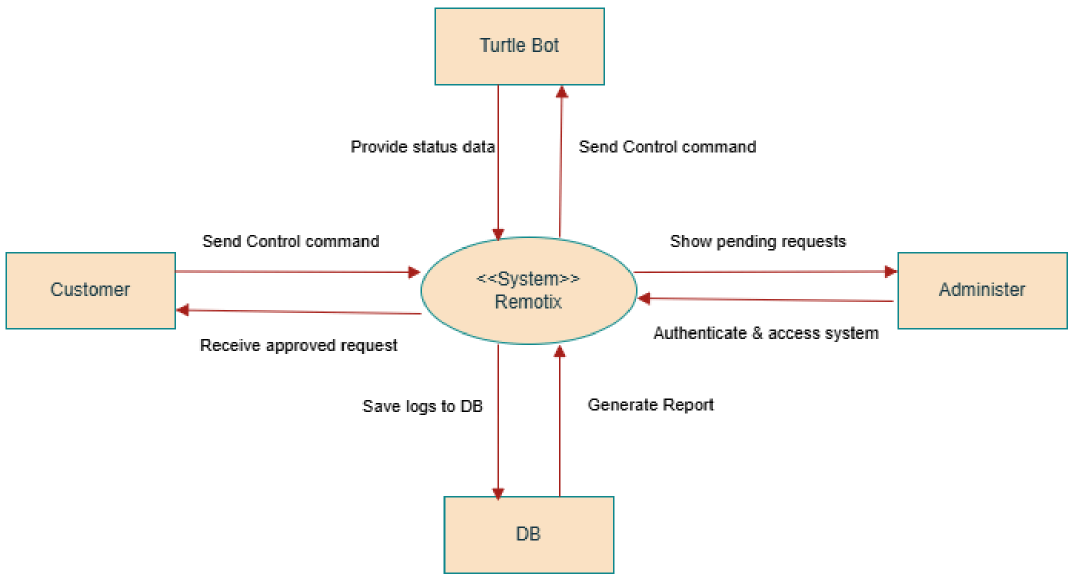
  Non-functionally, the end-to-end latency—from operator command to robot action, and from camera capture to display—must not exceed one second. The user interface is entirely web-based, requiring no specialized software or prior training

2.4 System modeling

This section presents the logical behavior of the Remotix system through system modeling diagrams. It focuses on how various actors including the operator,TurtleBot, and database interact with the system.

Context Model

diagram shows the external actors and their interactions



- **Customer**

  A **customer** submits a control request to the Remotix system.
  Once the request is approved, the customer receives the approved command result.

- **Administrator**

  The **administrator** first authenticates and accesses the system.
  The system shows all **pending control requests** submitted by customers.
  The administrator reviews and approves or rejects each request.

- **TurtleBot**

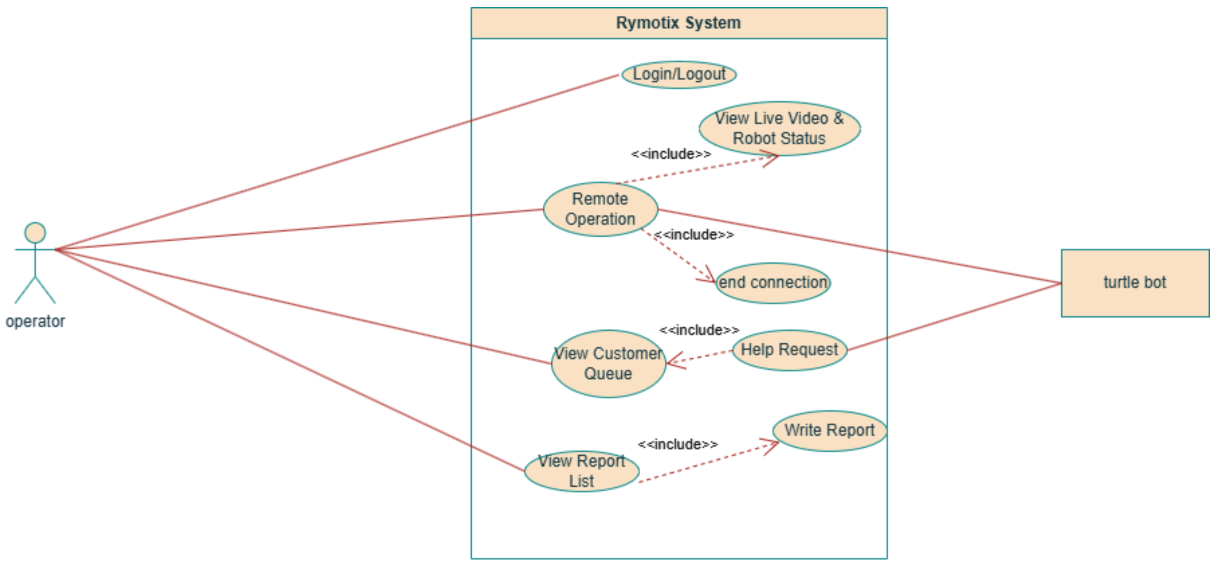
  TurtleBot provides real-time **status data** to the Remotix system.
  After approval, the system sends a **control command** to the TurtleBot to execute.

- **Database (DB)**

  The system **saves logs** of all actions and robot responses to the database.
  It also **generates reports** based on this stored data.

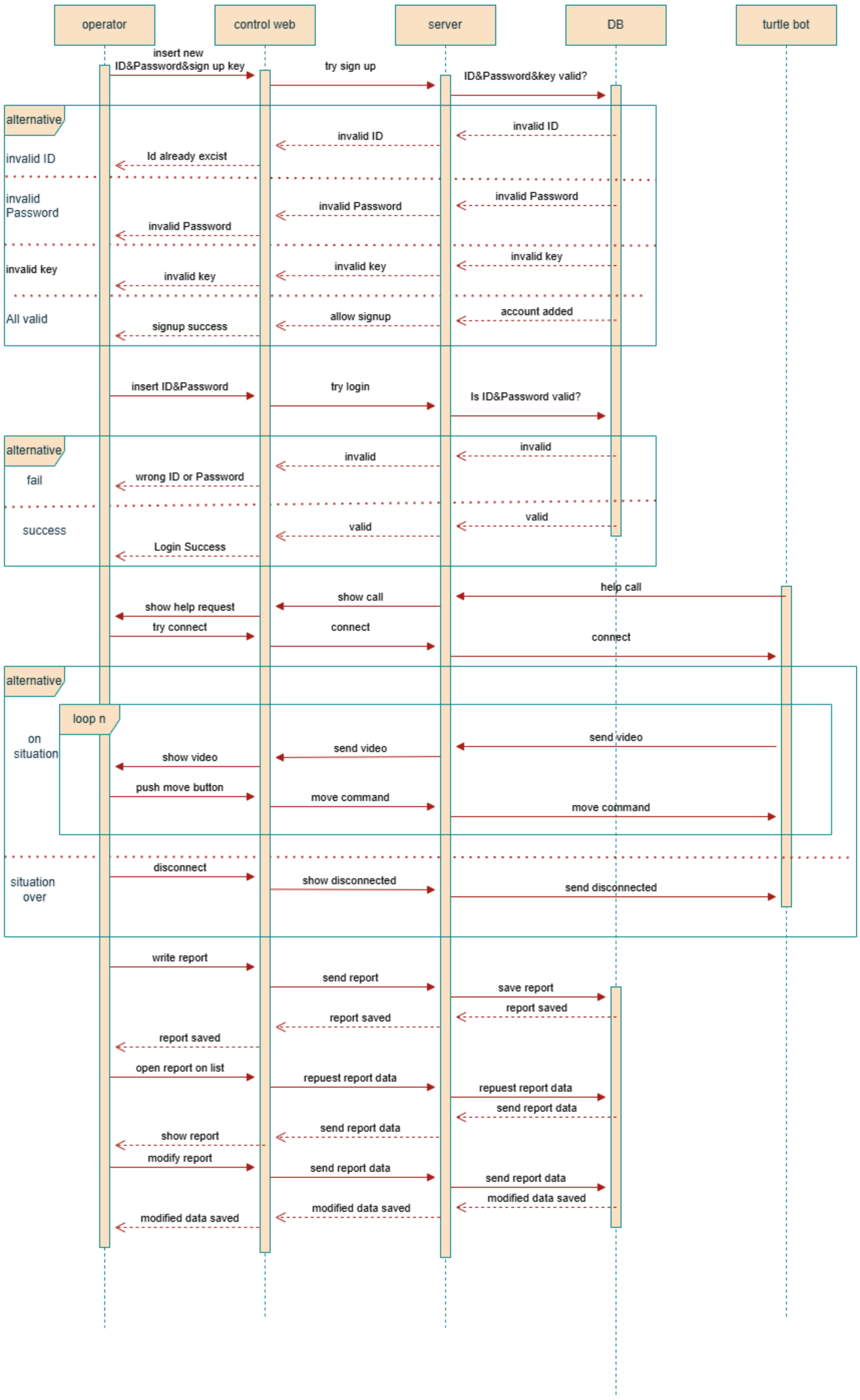
Use Case Diagram

Use case diagram shows functionalities available to the operator, including login, request acceptance, real-time teleoperation, monitoring, and report writing.



Sequence Diagram

Sequence diagram shows the detailed scenario from start to finish: the TurtleBot sends a help request, the operator connects and issues commands via the control web, and finally submits a report after the situation is resolved.


operator
control web
server
DB
turtle bot
insert new ID&Password&sign up key
try sign up
ID&Password&key valid?
alternative
invalid ID
invalid ID
invalid ID
Id already excist
invalid Password
invalid Password
invalid Password
invalid Password
invalid key
invalid key
invalid key
invalid key
All valid
account added
allow signup
signup success
insert ID&Password
try login
Is ID&Password valid?
alternative
invalid
invalid
fail
wrong ID or Password
valid
valid
success
Login Success
help call
show call
show help request
try connect
connect
connect
alternative
loop n
on situation
send video
send video
show video
push move button
move command
move command
disconnect
show disconnected
send disconnected
situation over
write report
send report
save report
report saved
report saved
report saved
open report on list
repuest report data
repuest report data
send report data
send report data
show report
modify report
send report data
send report data
modified data saved
modified data saved
modified data saved

- **Overall Flow**

  The TurtleBot sends a help request to the server.

  The server forwards the help request to the operator via the control web.

  The operator views the request and attempts to connect.

  The server establishes a connection with the TurtleBot

  While the emergency situation is ongoing, the following actions occur repeatedly:

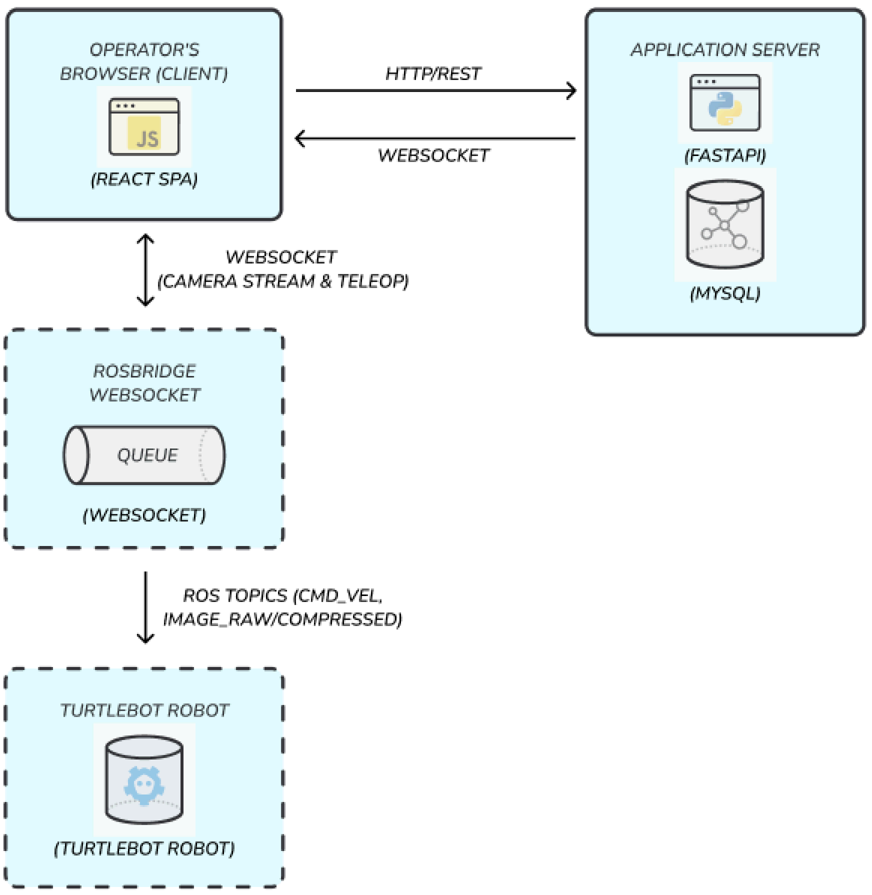
  > The TurtleBot streams real-time video to the server, forwarded to the control web.
  >
  > The operator views the video and sends movement commands via the web interface.
  >
  > Commands are delivered to the TurtleBot through the server, and the TurtleBot moves accordingly.

  When the situation ends:

  > The connection with the TurtleBot is terminated.
  >
  > The operator writes a report through the control web.
  >
  > The report is sent to the server and saved in the DB.

2.5 Architectural design

Architecture Design



Deployment Diagram of the Remotix Dashboard System

Show the full deployment of our software components and runtime nodes:

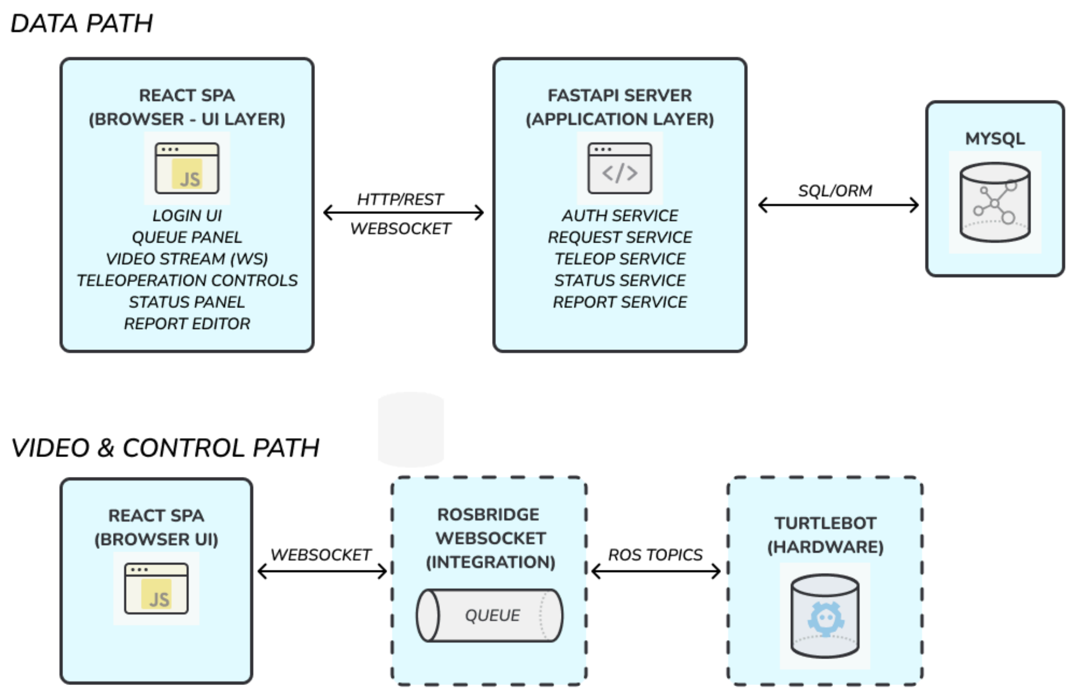
- Operator's Browser (React SPA) – the presentation layer running in the user's browser.
- Application Server (FastAPI + MySQL) – hosts all business logic (authentication, request handling, reporting, status updates) and persists data via an ORM to MySQL.
- ROSBridge WebSocket – an external integration node translating between WebSocket messages and ROS pub/sub.
- TurtleBot Robot – the physical ROS node that subscribes to /cmd_vel and publishes camera, battery, and odometry topics.

Arrows are labeled with their transport protocols:

- HTTP/REST and WebSocket between React SPA and FastAPI
- SQL/ORM between FastAPI and MySQL
- WebSocket between React SPA and ROSBridge
- ROS Topics between ROSBridge and TurtleBot

This deployment view ensures that we understand not just "what" components exist, but "where" and "how" they run in our production environment.

Entire System Structure



Component/Container View of Data Path & Video & Control Path

Zoom in on the two primary interaction chains that drive our application:

1. Data Path
   - React SPA ↔ FastAPI ↔ MySQL
   - Covers login, customer‑queue management, and report CRUD
   - Uses HTTP/REST for discrete requests and WebSocket for real-time updates
2. Video & Control Path
   - React SPA ↔ ROSBridge ↔ TurtleBot
   - Powers live video streaming and teleoperation
   - Uses WebSocket from browser to ROSBridge and ROS Topics between ROSBridge and TurtleBot

By splitting the view this way, we highlight how high‑latency data operations are isolated from the low‑latency video/control loop, meeting both responsiveness and reliability requirements.

Applied Patterns

To underpin this structure with solid engineering principles, we apply:

1. Server–Client
   - React SPA (client) ↔ FastAPI + MySQL (server)
   - Clear separation of UI and backend, with REST for CRUD and WebSocket for live events
2. Model–View–Controller (MVC)
   - Model: FastAPI's ORM entities (CustomerRequest, ControlSession, Report, RobotStatus)
   - View: React components (LoginModal, QueuePanel, StreamPanel, StatusPanel, ReportEditor)

- Controller: FastAPI routers + React hooks/services (useAuth, useQueue, useTeleop) orchestrate user input, business logic, and UI updates

These patterns guarantee modularity, maintainability, and the low‑latency behavior critical for safe, real‑time robot teleoperation.

3. Validation

3.2 Usability Testing (Validation)

We conducted usability testing with three real users, and based on their feedback, designed the system so that once a customer is connected no additional connections can be made, and for safety reasons vehicle control cannot be obtained unless teleoperation mode is activated.

3.3 Verification

We verified the integrity of the ROS system by logging published values to ensure accuracy, capturing WebSocket connection details—including the number of connected clients and server initialization messages—and confirming via the browser's developer console that our React client successfully received ROS-network transmissions. Additionally, we performed a manual human check of the database storage component to confirm that all data were recorded correctly.

4. Evolution

Code Refactoring - Web socket connection

In our original WebSocket setup, every React component panel opened its own connection for the topic it needed. Each page refresh therefore spawned multiple connections, wasting resources and making maintenance increasingly difficult. We resolved this by introducing a single WebSocket context that centralizes the connection, resulting in a far more efficient architecture.

Code Refactoring - Layered Architecture

Initially, the FastAPI-based backend was implemented as a single Python file, which made development difficult due to the increased complexity and lack of modularity. **To address this, we plan to refactor the backend into a layered architecture**, separating concerns across Controller, Service, and Repository layers. This change is expected to improve the maintainability, scalability, and testability of the system.

IDEA1. backend CICD

To improve development efficiency and deployment reliability, we planned to implement a CI/CD pipeline for the backend service using **GitHub Actions** and **Docker.**

When code is pushed to the repository, **GitHub Actions** will automatically build a Docker image and push it to **Docker Hub**.

Then, the cloud server (hosted on Naver Cloud Platform) will automatically pull the latest image and restart the backend service, enabling smooth and consistent deployment without manual invention.

As a result, the system can quickly reflect changes made by users or developers without manual deployment steps.

IDEA2. Multi-robot control

Due to current **hardware limitations**, the system is operated by a **single robot**, and the customer incident-handling workflow has been partially simulated with **virtual data**; moving forward, this technology must be deployed in real-world environments and enhanced to enable real-time processing of large volumes of customer reports.

5. Team Activity

- T1. Team Front : UI design (김민재(20%), 박성언(20%))
- T2. Team Back : DB and connection design (김다인(20%), 박준혁(20%))
- T3. Team robot : H/W setting and connection (차준하(20%))

6. Review

In conclusion,

customers demand a fully autonomous, steering-wheel-free driving experience in which even the smallest errors are minimized through automation. Companies operating autonomous vehicles are keen to meet these needs and require a powerful tool to pinpoint exactly where the system falls short or which specific models are problematic. Finally, our system is fully capable of covering all of these requirements.