# **Readme.MD file**

Overview

Welcome to the Zangoh AI Agent Supervisor Workstation Challenge! In this challenge, you will build a workstation interface that allows human supervisors to monitor, intervene, and improve AI customer service agents.

Zangoh is developing autonomous AI agents for enterprise customer service. These agents handle routine customer inquiries, but human supervisors need to monitor their performance, step in when necessary, and continuously improve the system. Your task is to build the supervisor workstation that makes this human-in-the-loop workflow possible.

Challenge Duration

You have 3 hours to complete this challenge. Please manage your time accordingly.

**Note**: You may receive an additional requirement during the challenge. Be prepared to adapt your solution.

Tech Stack

The challenge uses the following technologies:

Backend

* Node.js with Express
* MongoDB (provided via Docker)
* WebSockets for real-time updates
* Mock LLM API (for simulating AI responses)
* Qdrant Vector Database (for knowledge retrieval)

Frontend

* React with hooks
* Chakra UI component library
* WebSocket client for real-time updates
* Recharts for data visualization

Environment Setup

Prerequisites

* Docker and Docker Compose
* Node.js (v16 or later)
* npm or yarn
* Git
* A modern web browser (Chrome or Firefox recommended)

Setup Steps

**Clone the starter repository** git clone https://github.com/zangoh/supervisor-challenge.git

cd supervisor-challenge

**Start the Docker environment** The challenge environment is containerized to ensure consistency. Start the Docker containers:  
  
 docker-compose up -d

1. This will start:  
   * MongoDB database
   * Qdrant vector database
   * Backend API server
   * Frontend development server
   * Mock LLM API
   * Conversation simulator

**Verify services are running** Check that all services are up and running:  
  
 docker ps

1. You should see containers for backend, frontend, mongodb, qdrant, and simulator.
2. **Access the application**
   * Frontend: http://localhost:3000
   * Backend API: http://localhost:8080
   * API Documentation: http://localhost:8080/api-docs

**Initialize test data (if needed)** The environment comes pre-loaded with test data, but if you need to reset it:  
  
 docker-compose exec backend npm run seed

Project Structure

The starter code is organized as follows:

supervisor-challenge/

├── backend/ # Backend API server

│ ├── data/ # Sample data

│ ├── middleware/ # Express middleware

│ ├── models/ # MongoDB models

│ ├── routes/ # API routes

│ ├── websocket/ # WebSocket handlers

│ ├── utils/ # Utility functions

│ └── server.js # Main server file

├── frontend/ # React frontend

│ ├── public/ # Static assets

│ ├── src/

│ │ ├── api/ # API client

│ │ ├── components/ # React components

│ │ ├── context/ # React context providers

│ │ ├── pages/ # Page components

│ │ ├── utils/ # Utility functions

│ │ └── App.js # Main App component

├── docker/ # Docker configuration

├── mockups/ # UI design screenshots

├── docker-compose.yml # Docker Compose configuration

└── README.md # This file

UI Design Reference

All UI designs are provided in Figma and as static images in the mockups folder. Your implementation should follow these designs.

* **Figma Link**: [FIGMA\_LINK\_HERE]

The designs include:

1. **Dashboard/Overview Screen**: Showing key metrics and conversation lists
2. **Conversation Detail/Intervention Screen**: For monitoring and intervening in conversations
3. **Agent Configuration Screen**: For adjusting AI agent parameters and capabilities

Focus on implementing the functionality according to these designs. Pixel-perfect reproduction is not required, but your implementation should closely match the overall layout and user experience.

Available APIs

The backend provides the following API endpoints:

Conversations

* GET /api/conversations: Get all conversations with filtering
* GET /api/conversations/:id: Get a specific conversation
* POST /api/conversations/:id/messages: Add a message to a conversation
* PATCH /api/conversations/:id/status: Update conversation status
* POST /api/conversations/:id/tags: Add tags to a conversation

Agents

* GET /api/agents: Get all agents
* GET /api/agents/:id: Get a specific agent
* PATCH /api/agents/:id/config: Update agent configuration
* GET /api/agents/:id/metrics: Get agent performance metrics

Intervention

* POST /api/intervene: Intervene in a conversation
* POST /api/intervene/release: End intervention and return to agent

Knowledge Base

* GET /api/knowledge-base: Get all knowledge bases

LLM Mock API

* POST /api/llm/generate: Generate LLM response
* POST /api/llm/sentiment: Analyze text sentiment

WebSocket API

The WebSocket server provides real-time updates for:

* New conversations
* Message updates
* Metrics changes
* Agent status changes

Refer to the API documentation at http://localhost:8080/api-docs for detailed information.

Testing Your Implementation

A testing script is provided to help you validate your implementation. Run it with:

cd testing

npm install

node test-runner.js

This will check the core functionality of your implementation and provide feedback on what's working and what needs attention.

Implementation Requirements

Your implementation should include:

1. Agent Monitoring Dashboard

* Real-time view of active customer conversations with AI agents
* Key metrics display (resolution rate, average response time, customer satisfaction)
* Alert system for potentially problematic conversations
* Filtering options (by agent, status, alert level)

2. Intervention Interface

* Ability to view full conversation history for any active session
* "Take over" functionality to assume control from the AI agent
* Agent performance feedback mechanism
* Option to return control to the AI with notes/guidance

3. Agent Configuration

* Adjust AI parameters (temperature, max tokens, etc.)
* Enable/disable specific capabilities or knowledge domains
* Configure automatic escalation thresholds
* Save and load configuration presets

Tips for Success

* **Start with core functionality**: Implement the basic features first before adding polish
* **Use the provided components**: The starter code includes basic components you can build upon
* **Test incrementally**: Use the testing script to validate your progress
* **Manage your time**: Allocate time for each major feature, leaving time for integration and testing
* **Read the API docs**: Understand the available endpoints before implementation
* **Study the designs**: Familiarize yourself with all screens before starting to understand the complete system

Submission

When you're done, submit your solution as follows:

**Ensure all code is committed**:  
  
 git add .

git commit -m "Final submission"

**Create a zip file of your solution**:  
  
 # From the project root

zip -r submission.zip . -x "node\_modules/\*" "\*/node\_modules/\*" ".git/\*"

1. **Upload your submission** to the platform or provide a GitHub repository link as instructed
2. **Include in your submission**:  
   * Complete source code
   * Setup instructions if you've modified the environment
   * Brief description of your implementation, challenges faced, and solutions
   * Any notes for the evaluator

Evaluation Criteria

Your solution will be evaluated based on:

* **Functionality** (40%): Does it fulfill all the requirements?
* **Code Quality** (25%): Is the code well-structured, maintainable, and efficient?
* **Design Implementation** (15%): How well does it match the provided designs?
* **Technical Decisions** (20%): Are the technical choices appropriate and efficient?

Need Help?

If you encounter any issues with the environment setup:

1. Check the Docker logs: docker-compose logs <service-name>
2. Ensure all services are running: docker ps
3. Restart the environment if needed: docker-compose down && docker-compose up -d

Good luck!