Live Chat Queue Balancer

A Web-Based Simulation using React, Vite, and a Circular Queue

Authors: 12301183 - Deepak Rana, 12304053 - Debargha Debnath, 12321218 - Arijit Debnath, 12316615 - Dhairya Sharma, 12321525 - Kishan Singh

Date: July 2025

Why do we need a Chat Queue Balancer?

Optimise Agent Utilisation

Without an efficient system, user requests can be dropped or handled inefficiently, leading to poor customer satisfaction and underutilised agents.

Enhance Customer Satisfaction

In modern customer service, multiple users often try to connect with a limited number of support agents simultaneously. A queueing system ensures fairness and efficiency.

Fairness and Efficiency

A well-implemented queueing system ensures fair handling of requests using First-In, First-Out (FIFO) principles, maximising agent productivity and reducing wait times.

What We Set Out to Build

Our primary objectives for this project were multifaceted, aiming to create a robust and realistic simulation of a live chat queueing system.

Real-World Simulation

To accurately simulate a real-world chat queueing system, reflecting common scenarios in customer support.

Circular Queue Implementation

To implement a **Circular Queue** data structure for efficient management of incoming chat requests.

Dynamic React Frontend

To build a dynamic, real-time frontend dashboard utilising **React** and **Vite** for a responsive user experience.

Backend REST API

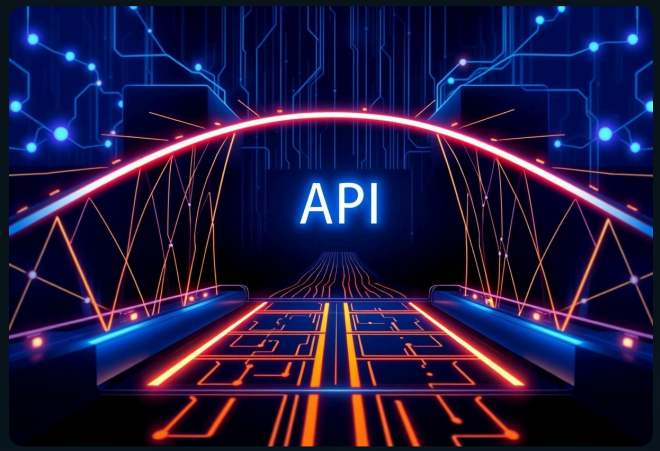
To create a robust backend server to manage the simulation logic and expose data via a reliable REST API.

High-Level Design

Our system architecture is designed for clear separation of concerns, ensuring scalability and maintainability.

React Frontend (Client)

The user's interactive dashboard, built with React and Vite, responsible for visualising real-time data from the backend.



REST API (Bridge)

A JSON-based API (/api/status) acting as the communication bridge, facilitating seamless data exchange between the frontend and backend.

Python Backend (Server)

The core simulation engine, managing the circular queue, agent logic, and providing the central data processing for the application.

The Heart of the System: Circular Queue

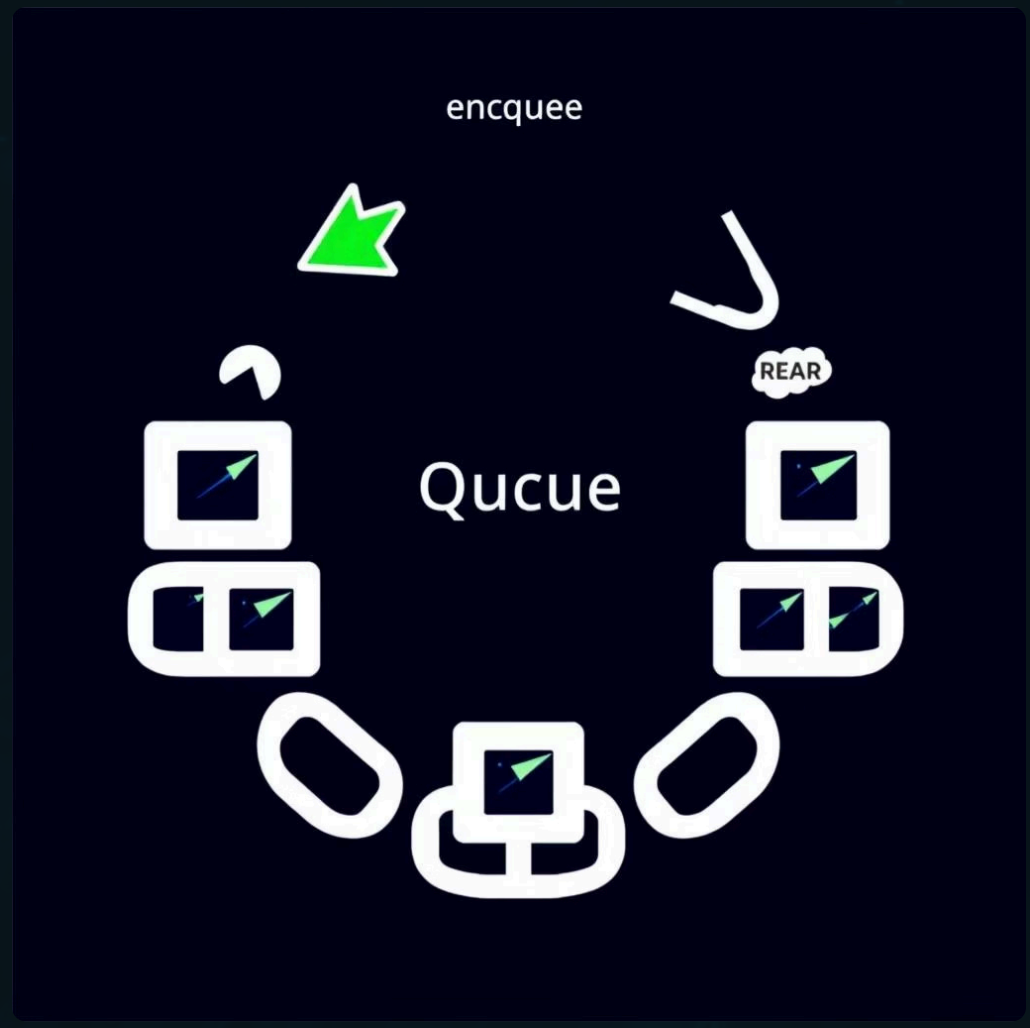
The circular queue is fundamental to our system's efficiency.

What is it?

- A linear data structure operating on the First-In, First-Out (FIFO) principle.
- It connects the last element to the first, forming a circle.

Why use it?

- Highly efficient for fixed-size buffers, preventing memory waste.
- It reuses empty space, unlike a standard queue that might require resizing.



Key Operations:

- **enqueue():** Adds a new chat request to the rear.
- **dequeue():** Removes a chat from the front for an available agent.
- **isFull():** Checks if the queue is at capacity.
- **isEmpty():** Checks if the queue contains any chats.

The Simulation Engine: Backend Implementation

Our backend powers the entire simulation and data management.

1

Technology Stack

Developed using **Python** with **Flask** (or FastAPI for larger scale), offering a robust and lightweight server framework.

2

Core Simulation Logic

Features a simulation loop that periodically generates new chat requests. It employs a **Round-Robin** algorithm for fair distribution among available agents.

3

API Endpoint

A single **GET /api/status** endpoint provides the entire system state as a comprehensive JSON object to the frontend, ensuring data consistency.

The Real-Time Dashboard: Frontend Implementation

The frontend is designed for dynamic, responsive visualisation of the queue and agent statuses.

Technology: React + Vite

Leverages React for component-based UI development and Vite for fast development server and build optimisation.

Component Architecture

Built using reusable components: **QueueDisplay** for queue visualisation, **AgentDashboard** for agent status, and **EventLog** for system activities.

State Management

Utilises React's native **useState** hook to efficiently manage and update queue, agent, and log data across the application.

Live Updates

useEffect and **setInterval** poll the backend API every 2 seconds, ensuring the dashboard remains perfectly synchronised with the simulation state.

Live Demonstration

Observe the Live Chat Queue Balancer in action, demonstrating real-time updates and efficient queue management.