Concurrent System Design for Optimization API

1. Introduction

This document explains how we can build a backend system that can handle many requests at the same time.
The system will help optimize container movements at a port, even when there are changes like arrival rates, berth limits, and priority rules.

Our main goals are:

- Keep response time fast (low latency)
- Handle a lot of users (high load)
- Keep working even if something fails (graceful degradation)

2. System Overview

We will build the system with these main parts:

- **Load Balancer:** Sends incoming requests to different servers so that no single server gets overloaded.
- **API Gateway:** Manages user requests, handles security, and controls how many requests come in.
- **Optimization Service:** This is where the optimization logic runs. It will be **stateless** so we can easily add more copies when needed.
- **Cache (Redis or Memcached):** Saves previous results so we can quickly give answers without doing heavy calculations again.
- **Database (MongoDB or PostgreSQL):** Stores all the old requests and results for record-keeping.
- **Monitoring (Prometheus, Grafana, ELK Stack):** Watches the system to make sure everything is working well.

3. Key Components

Component	What it does
Load Balancer	Shares incoming traffic among different servers
API Gateway	Controls access and traffic flow

Component	What it does
Optimization Service	Runs the main optimization logic
Cache	Stores frequently used results to answer faster
Database	Saves all data safely
Monitoring Tools	Keeps track of system health and problems

4. How We Scale

- **Horizontal Scaling:** Add more servers when traffic increases.
- **Auto-scaling:** Set rules to automatically add or remove servers based on usage.
- **Database Sharding:** Split the database into smaller parts to handle more data and traffic.

5. How We Handle Failures

- **Graceful Degradation:** If the system is too busy, show cached results or simple alternatives.
- **Retry Mechanism:** Try again if something fails temporarily.
- **Circuit Breaker:** Stop sending requests to services that are already failing to avoid bigger problems.
- **Monitoring and Alerts:** Get notified if something goes wrong, like slow responses or errors.

6. Conclusion

This design helps the optimization API work smoothly even when there are lots of requests. By using caching, scaling, and monitoring, the system can handle real-world port conditions without slowing down or crashing.