Scalability & Availability Considerations

Handling Increased Users or Data

Our backend is designed using Express.js with asynchronous I/O, allowing it to handle multiple concurrent requests efficiently. The PostgreSQL database is normalized and supports indexing on frequently queried fields, helping reduce response time even as the data grows. While current usage is modest (suitable for a university-scale dorm shopping app), the architecture supports horizontal scaling by adding more backend instances in response to traffic spikes.

We also use pagination for certain endpoints (e.g., product lists, order history) to avoid loading large datasets into memory at once. This prevents client-side and server-side bottlenecks and improves overall user experience.

Load Balancing Strategy

While we did not implement a load balancer in this sprint, our system architecture is compatible with standard load balancing solutions like NGINX, Render Autoscaling, or platform-native solutions such as AWS Application Load Balancer. The backend is stateless, so scaling horizontally by adding more instances is feasible without significant architectural changes. This allows us to support a larger number of users by distributing the traffic across multiple server processes.

Deployment Strategy and Uptime Considerations

Our application uses Render for the backend and Netlify for the frontend. Both platforms provide continuous deployment, automatic restarts, and basic health monitoring, which help ensure high availability. We use GitHub Actions to trigger CI/CD pipelines automatically on pushes to the main branch, reducing manual deployment errors and downtime. Our deployment strategy supports zero-downtime deploys, meaning user access remains uninterrupted during updates.

To prepare for potential scale-up scenarios, we ensured that:

- The system is stateless, allowing for easy deployment scaling
- Our environment variables and .env setup allow for environment-specific configuration
- We keep production and development builds separate to avoid cross-environment failures

Future Scalability Considerations

Although current demand is manageable, the system is designed with scalability in mind. In future sprints or post-MVP, we could:

- Introduce monitoring tools like UptimeRobot or Grafana for live uptime and performance tracking
- Add rate limiting to API endpoints to prevent overload or abuse
- Use database connection pooling for improved performance under concurrent load
- Separate services by domain (e.g., cart service, order service) to allow modular scaling