Contents

[(1) Architecture Design: 1](#_Toc210003739)

[a) API Structure and Services: 1](#_Toc210003740)

[b) Database Selection: 2](#_Toc210003741)

[c) Scalability Considerations: 2](#_Toc210003742)

[d) Architecural diagram: 2](#_Toc210003743)

[(2) Development Workflow and Ci/CD: 2](#_Toc210003744)

[a) Development workflow: 2](#_Toc210003745)

[b) Git branching strategy: 3](#_Toc210003746)

[c) Testing and deployments: 3](#_Toc210003747)

[(3) Security and performance: 3](#_Toc210003748)

[a) Security measures: 3](#_Toc210003749)

[b) Monitor and optimize system performance: 4](#_Toc210003750)

### Architecture Design:

### API Structure and Services:

For the MVP, I chose a monolithic architecture using FastAPI for the backend, SvelteKit for the frontend, and MongoDB as the database. All services are containerized with Docker and orchestrated using Docker Compose. This setup is simple to develop, easy to deploy, and ideal for rapid prototyping.

The backend exposes RESTful endpoints for core features like distance calculation and historical queries. The code is organized by routers, making it modular and maintainable. Rate limiting and input validation are built in for security and reliability.

All core features like order management, user management, delivery tracking, and distance calculation are implemented as RESTful endpoints within a single codebase. At the same time, I tried to make the code as modular as I can so that it’s easy to expand as the feature grows. This approach enables rapid development, simple deployment, and easier maintenance.

As the product scales, the backend can be refactored into microservices:

* Order Service: Handles order creation and management.
* User Service: Manages users and drivers.
* Delivery Service: Tracks deliveries and optimizes routes.
* Notification Service: Sends SMS/email updates.

For real-time updates and decoupling, an event-driven architecture can be introduced using message queues (e.g., RabbitMQ, Kafka).

### Database Selection:

I selected MongoDb (NoSQL) for the MVP because it offers flexible schema design, which is great for evolving requirements. It’s easy to scale horizontally and works well for storing semi-structured data like addresses and delivery events. If the business needs strong transactions or complex reporting a SQL database like PostgreSQL can be added for those components.

### Scalability Considerations:

To handle 100x growth, I’d keep the backend stateless and scale out with more containers or cloud instances. MongoDB can be sharded and replicated for high availability. Adding caching (Redis), asynchronous processing (message queues), and a load balancer would help manage increased traffic. I’d set up monitoring and logging tools for observability.

### Architectural diagram:

Backend (FastAPI)

* Rest API
* Rate limiting
* Input validation
* Logging and retry

Frontend (Svelte)

User’s browser

MongoDB Database

### Development Workflow and Ci/CD:

### Development workflow:

### For this project, I’d structure the workflow to keep things fast, collaborative, and reliable:

* **Local Development:**
  1. - Each developer works on their own feature or bugfix branch, cloned from main.
  2. - The app is containerized, so everyone can run the full stack locally using Docker Compose. This ensures consistency.
* **Feature Branches & Pull Requests:**
  1. - New work starts in a feature branch (e.g., feature/add-rate-limiting).
  2. - Once the feature is ready, a pull request is opened for review. This allows for feedback, discussion, and automated checks before merging.
* **Code Review & Testing:**
  1. - All code is reviewed by at least one other team member.
  2. - Automated tests (unit and integration) run on every pull request via a CI service (like GitHub Actions).
  3. - If tests pass and the code is approved, it’s merged into main.
* **CI/CD (with AWS):**
  1. - Every push to the repository triggers the CI pipeline.
  2. - The pipeline builds Docker images, runs tests, and checks for code quality issues.
  3. - When code is merged to main, the CI/CD pipeline builds the latest Docker images and pushes them to AWS ECR.
  4. - AWS ECS automatically deploys the new containers, updating the running services with zero downtime.
  5. - This means every change that passes review and testing can be deployed quickly and safely.
* Environment Management:
  1. - Environment variables and secrets are managed securely (e.g: AWS Secrets Manager).
  2. - Staging and production environments are kept separate, so new features can be tested before going live.

This workflow makes it easy to collaborate, catch bugs early, and deploy updates quickly. Docker ensures everyone’s environment is the same, and AWS provides scalable, reliable hosting. Automated testing and deployment mean we can move fast without sacrificing quality.

### Git branching strategy:

I’d use a feature-branch workflow. The main branch always contains stable, production-ready code. For new features or bug fixes, I’d create a separate branch like backend/delete\_history or bugfix/cors. Once the work is done, I’d open a pull request for review and merge it back into main after automated tests pass. This keeps the codebase organized and makes collaboration easy.

### Testing and deployments:

* I’d write unit tests for backend logic and input validation, integration tests for API endpointsn and Cypress tests for Browser integration testing. All tests run locally and in CI before merging.
* On every push or pull request, a CI service (like GitHub Actions) builds docker images and runs tests.
* When code is merged to main, the pipeline builds and pushes images to AWS ECR.
* AWS ECS automatically deploys the new containers, updating the running services.

This setup ensures deployments are automated, reliable, and only happen after successful tests.

### (3) Security and performance:

### Security measures:

* All API endpoints would validate and sanitize input to prevent injection attacks (already implemented in my code).
* Sensitive data, like database credentials and API keys, would be stored in environment variables or a secrets manager (e.g., AWS Secrets Manager), never hardcoded.
* I’d use HTTPS for all communication in production to encrypt data in transit.
* Rate limiting is already in place to prevent abuse and brute-force attacks.
* For authentication, I’d implement secure user sessions or JWT tokens and always hash passwords if storing them.

### Monitor and optimize system performance:

* I’d add structured logging to track errors and key events. For example, in my code I’m using Python’s logging module to log important events, warnings, and errors in the FastAPI endpoints.
* Use monitoring tools like AWS CloudWatch or Grafana to keep an eye on resource usage, response times, and error rates.
* Set up alerts for unusual activity or performance drops.
* To optimize, I’d analyze the backend code to find slow endpoints, add caching (like Redis) for frequent queries, and scale out with more containers if needed.