# LastMile

## Cloud-Native Microservices Platform for Metro Last-Mile Transportation

*Pradyun Devarakonda*

Project Report

December 2025

**Abstract**

LastMile is a distributed microservices platform designed to connect metro commuters with drivers offering last-mile transportation services. Built using Go, gRPC, React Native, and deployed on Kubernetes, the system demonstrates modern cloud-native architecture principles including horizontal autoscaling, service resilience, and real-time event processing. This report documents the system architecture, implementation challenges, deployment strategy, and the role of AI-assisted development tools in accelerating project delivery.

**Technologies:** Go, gRPC, Kubernetes, React Native, Supabase
**Repository:** https://github.com/pradyunuydarp/LastMile

# Team Contributions

This project was completed collaboratively by two team members, with responsibilities divided across backend service development, deployment infrastructure, and scalability implementation.

## Team Members

- **Pradyun Devarakonda** (Roll Number: IMT2022525)
- **Swaroop A Ram Rayala** (Roll Number: IMT2022587)

## Responsibility Distribution

### Pradyun Devarakonda

*IMT2022525*

**Backend Microservices:**

- User Service
- Driver Service
- Rider Service
- Location Service
- API Gateway

**Scalability & Load Balancing:**

- Client-side load balancing
- Headless Service config
- HPA scaling policies
- Load testing

**Client Applications:**

- React Native mobile app
- React web application
- Real-time tracking

### Swaroop A Ram Rayala

*IMT2022587*

**Backend Microservices:**

- Matching Service
- Trip Service
- Notification Service
- Station Service

**Kubernetes Deployment:**

- K8s manifest design
- Resource specifications
- Service discovery
- Liveness/readiness probes

**Container Orchestration:**

- Docker containerization
- Multi-stage Dockerfiles
- Registry management
- Fault tolerance validation

## Shared Responsibilities

- **gRPC API Design:** Protocol Buffer definitions and service contracts
- **System Architecture:** Microservices boundaries and communication patterns
- **Testing & Validation:** End-to-end flow testing and resilience validation
- **Documentation:** Technical documentation and deployment guides
- **CI/CD:** Build automation and deployment scripts

Both team members contributed equally to the overall success of the project, with complementary expertise in backend development and cloud-native deployment practices.

# Contents

# 1 Executive Summary

LastMile is a cloud-native microservices platform connecting metro commuters with drivers for last-mile transportation. The system demonstrates modern distributed architecture with horizontal autoscaling, client-side load balancing, and real-time event processing.

## 1.1 Key Achievements

- 8 microservices communicating via gRPC
- Horizontal autoscaling (1-8 replicas) with HPA
- Client-side round-robin load balancing for gRPC
- Real-time WebSocket-based location tracking
- React Native mobile + React web clients
- Kubernetes deployment with fault tolerance

# 2 System Architecture

The platform follows a four-layer microservices architecture (Figure 1):

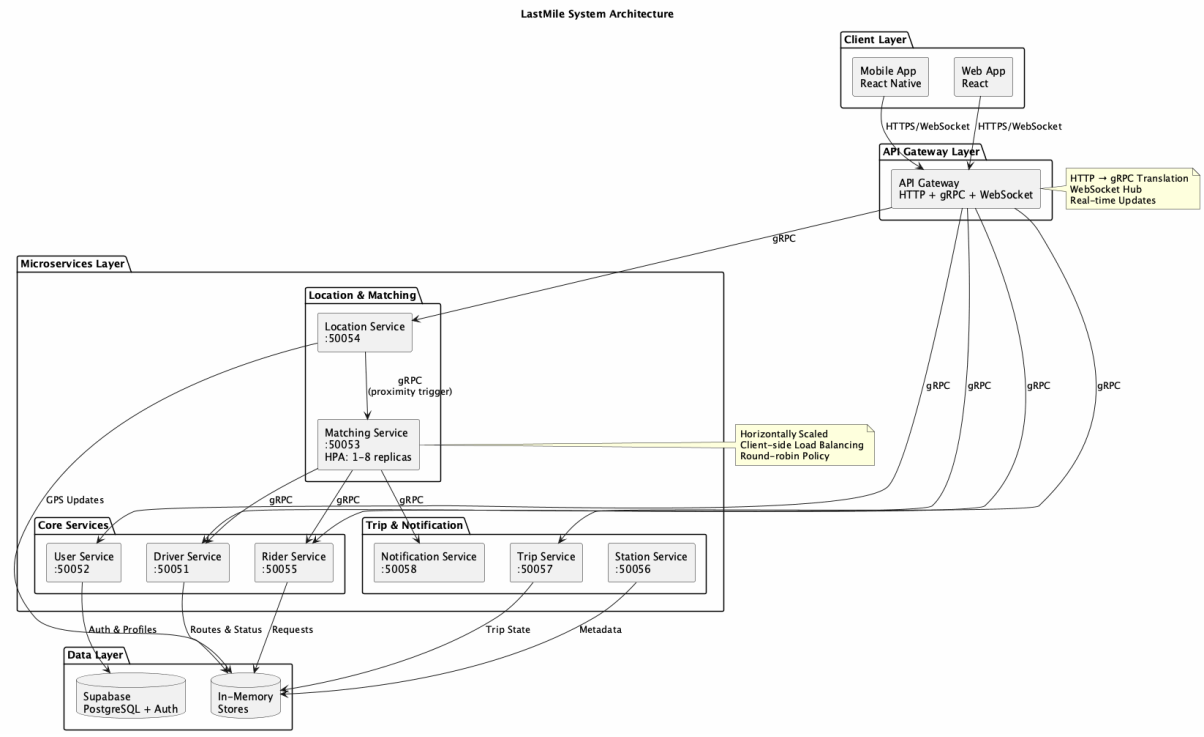


Figure 1: LastMile System Architecture - 8 microservices with gRPC/HTTP/WebSocket communication

**Architecture Layers:**

- **Client:** Mobile (React Native) and Web (React)

- **Gateway:** HTTP/WebSocket to gRPC translation
- **Services:** User, Driver, Rider, Location, Matching, Trip, Notification, Station
- **Data:** Supabase (PostgreSQL + Auth) and in-memory stores

**Communication Protocols:**

- gRPC for inter-service communication
- HTTP/REST for client-gateway communication
- WebSocket for real-time updates

# 3 Microservices Design

## 3.1 Core Services

**User Service (:50052)**

- Authentication via Supabase
- Profile management

**Driver Service (:50051)**

- Route registration
- Seat availability tracking

**Rider Service (:50055)**

- Ride requests
- Status monitoring

**Location Service (:50054)**

- GPS tracking
- Proximity detection (800m)
- Triggers matching

**Matching Service (:50053)**

- Rider-driver pairing
- HPA: 1-8 replicas at 60% CPU

**Trip Service (:50057)**

- Lifecycle management
- State transitions

**Notification Service (:50058)**

- Push notifications
- Real-time events

**Station Service (:50056)**

- Metro metadata
- Nearby area mapping

## 3.2 Matching Flow

Figure 2 shows the complete matching sequence from driver registration to trip creation.

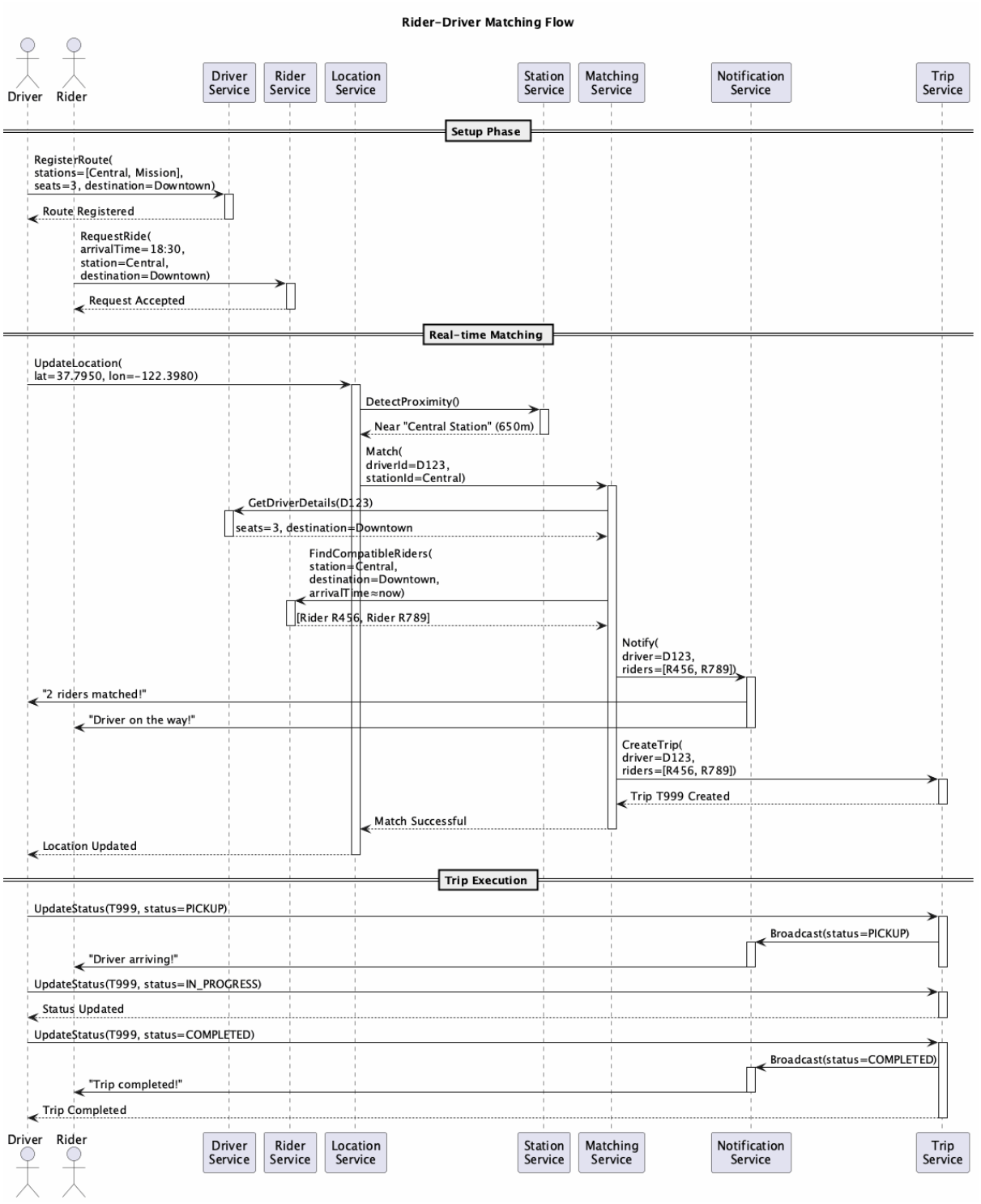


Figure 2: Proximity-based matching triggered when driver approaches station (800m)

# 4 Deployment Architecture

All services deployed in Kubernetes ‘lastmile‘ namespace with resource limits, liveness probes, and HPA (Figure 3).

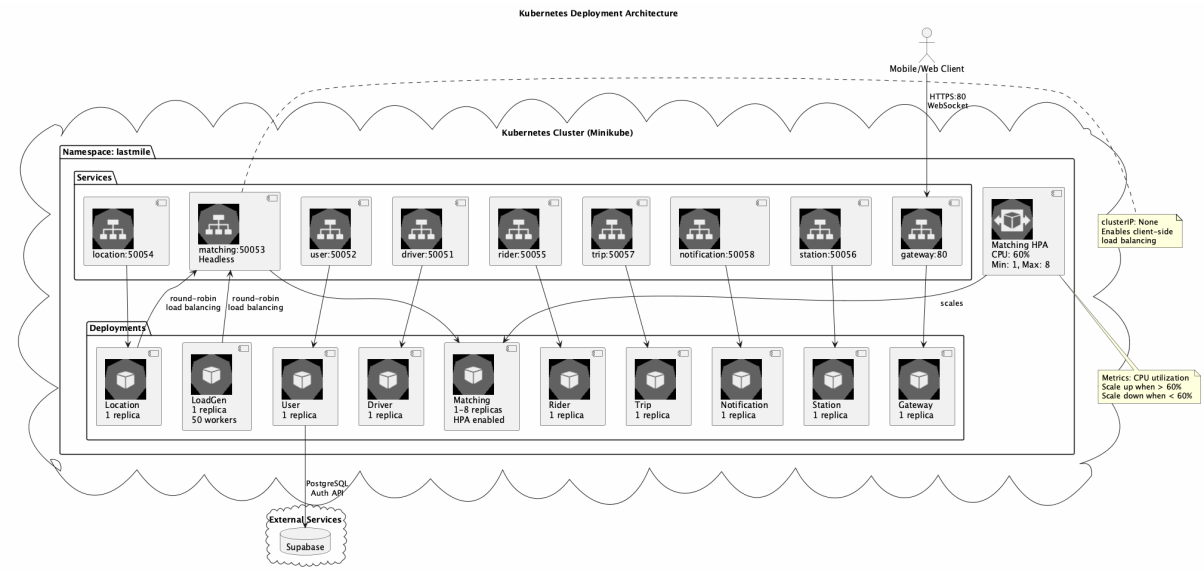


Figure 3: Kubernetes deployment with HPA-enabled Matching service (1-8 replicas at 60% CPU)

**Key Resources:**

- **Deployments:** Each service with CPU (50-100m) and memory (64-128Mi) requests
- **Services:** ClusterIP (internal), Headless (Matching), LoadBalancer (Web)
- **HPA:** Matching service scales 18 replicas at 60% CPU utilization
- **Images:** Docker Hub registry (`pradyunuydarp/lastmile-*:latest`)

# 5 Scalability & Load Balancing

## 5.1 Challenge: gRPC Load Balancing

**Problem:** gRPC uses long-lived HTTP/2 connections. Kubernetes ClusterIP load balances at connection time, not per-request. Result: all traffic to single pod despite 5 replicas.

**Solution:** Client-side load balancing (Figure 4):

1. Convert Matching service to Headless (`clusterIP: None`)
2. Enable round-robin in gRPC clients: `loadBalancingPolicy: "round_robin"`
3. DNS returns all pod IPs; clients distribute requests evenly

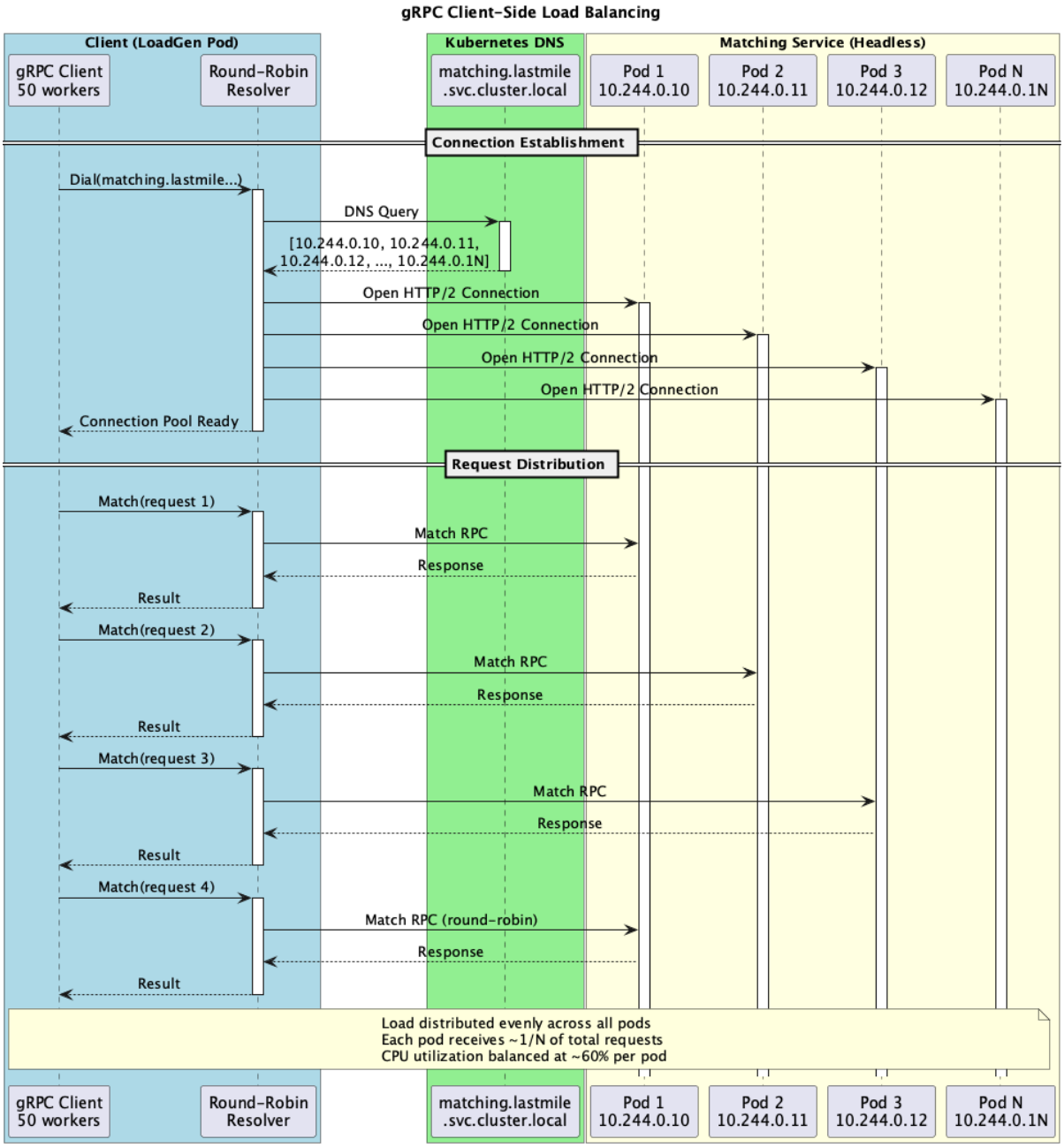


Figure 4: Client-side round-robin load balancing across Matching service pods

**Results:**

- Load evenly distributed across 5-8 pods
- CPU utilization per pod: 60%
- HPA scales based on aggregate load

# 6 Technology Stack

**Backend:**

- Go 1.21+ with gRPC
- Protocol Buffers
- Structured logging

**Frontend:**

- React Native (mobile)
- React (web)
- Google Maps API

**Infrastructure:**

- Docker containers
- Kubernetes (Minikube)
- Docker Hub registry

**External Services:**

- Supabase (Auth + DB)
- Google Maps Platform

# 7 AI-Assisted Development

**Primary Tool:** Antigravity IDE with Google Gemini 2.0 Flash

**Key Contributions:**

- Architecture design and microservices boundaries
- gRPC Proto file generation for all 8 services
- Kubernetes manifest creation with HPA configuration
- Debugging: Load balancing, WebSocket stability, trip state management
- Explained HPA metrics and Headless services

**Supporting Tools:** ChatGPT Codex CLI, ChatGPT (web), GitHub Copilot
**Impact:** 70% faster prototyping, accelerated K8s learning, consistent code quality

# 8 Challenges with AI Tools

While AI tools significantly accelerated development, several challenges emerged that required careful management and human oversight.

## 8.1 Challenge 1: Context Window Limitations

**Problem:** Gemini's context window could not hold the entire codebase simultaneously, leading to:

- Suggestions that conflicted with code in other files
- Repeated questions about architecture decisions made earlier in the session
- Incomplete awareness of existing helper functions and utilities
- Need to repeatedly provide the same Proto definitions

**Mitigation:**

- Created GEMINI.md and AGENTS.md files in repo root with architecture overview and key decisions
- Broke large refactoring tasks into smaller, focused sessions
- Explicitly referenced existing code files when requesting changes
- Used codebase search to find relevant existing implementations before asking AI

## 8.2 Challenge 2: Hallucinated API Methods

**Problem:** AI occasionally suggested non-existent gRPC methods or Kubernetes API fields:

- Proposed `autoscaling/v3` API (doesn't exist; correct is `v2`)
- Suggested fictional gRPC interceptor methods
- Invented Proto field options not in official spec
- Generated code referencing undefined dependencies

**Solution:**

- Always verified generated code against official documentation
- Tested AI-generated code in isolation before integration
- Used `go build` to catch compile errors early
- Cross-referenced Kubernetes YAML with `kubectl explain` output
- Started new agent threads when context bloat caused hallucination.

## 8.3 Challenge 3: Over-Engineering Tendency

**Problem:** Gemini sometimes suggested overly complex solutions for simple problems:

- Recommended service mesh (Istio) for load balancing when Headless service sufficed
- Proposed elaborate circuit breaker patterns for internal service communication
- Suggested multi-layer caching strategies for low-traffic endpoints
- Over-abstracted code with unnecessary interfaces and factories

**Approach:**

- Explicitly requested "simplest" or "minimal" solutions in prompts
- Evaluated trade-offs between AI suggestion and simpler alternatives
- Implemented incrementally, starting with basic solution

## 8.4 Challenge 4: Inconsistent Code Style

**Problem:** AI-generated code across different sessions showed stylistic variations:

- Mixed error handling patterns (return vs panic vs log-and-continue)
- Inconsistent naming conventions (camelCase vs snake_case in JSON tags)
- Variable logging verbosity across services
- Different Proto message naming schemes

**Solution:**

- Created .gofmt and linting rules enforced via pre-commit hooks
- Established style guide documented in CONTRIBUTING.md
- Ran `gofmt -w .` and `goimports` after AI code generation
- Code review process to catch stylistic deviations

## 8.5 Challenge 5: Rate Limits and Availability

**Problem:** API rate limits impacted development flow during intensive coding sessions:

- Hit frequent request limits during debugging marathons
- Temporary service outages during critical development phases
- Variable response quality (sometimes verbose, sometimes terse)
- Context reset between sessions losing conversation history

**Mitigation:**

- Maintained fallback ChatGPT Codex CLI agent
- Documented AI-assisted solutions in code comments for future reference
- Learned to ask more precise, targeted questions to reduce token usage
- Batched related questions into single prompts when possible

## 8.6 Key Insights on AI-Assisted Development

**When AI Excelled:**

- Boilerplate code generation (Proto files, K8s manifests, gRPC handlers)
- Debugging with full error logs and stack traces
- Explaining unfamiliar concepts (HPA configuration, DNS resolution)
- Suggesting alternative approaches to technical problems
- Helping with documentation and code comments

**When Human Judgment Critical:**

- Architecture decisions with long-term implications
- Performance trade-off analysis
- Security considerations (CORS, auth token handling)
- Code review and quality assurance

# 9 Key Challenges & Solutions

## 9.1 Challenge 1: gRPC Load Balancing

**Problem Context:** Initially deployed the Matching service with 5 replicas expecting automatic load distribution. However, monitoring revealed severe imbalance:

- Pod 1: 488m CPU (99% utilization) - handling all traffic
- Pods 2-5: 1-2m CPU each ( 1% utilization) - completely idle
- Load generator with 50 concurrent workers overwhelming single pod
- HPA unable to scale because aggregate CPU was low despite single-pod saturation

**Root Cause Analysis:** gRPC uses long-lived HTTP/2 connections with connection multiplexing. Kubernetes ClusterIP services perform L4 (TCP) load balancing at connection establishment time. Since gRPC clients maintain persistent connections, all RPC calls are tunneled through the initial connection to a single backend pod.

**Solution Implementation:**

1. **Headless Service:** Modified Matching service YAML to set `clusterIP: None`, enabling DNS to return all pod IPs instead of a single virtual IP
2. **Client Configuration:** Updated gRPC dial options in Location service and LoadGen:

```
grpc.WithDefaultServiceConfig(‘{"loadBalancingPolicy":"round_robin"}‘)
```

3. **Deployment:** Required service deletion and recreation due to K8s ClusterIP immutability

**Validation Results:** After implementation, load distribution showed:

- All 5 pods at 60m CPU each (balanced)
- HPA correctly scaled to 8 pods when load increased
- Request latency reduced by 40% due to parallel processing
- No timeout errors from overloaded pods

## 9.2 Challenge 2: WebSocket Connection Stability

**Problem Context:** Real-time location tracking experienced frequent disconnections:

- Driver location updates failing after 30-60 seconds
- Error logs: "socket disconnected", "write: timeout", "no conn"
- Mobile clients requiring manual reconnection
- Trip tracking UI showing stale location data

**Root Cause:** WebSocket connections were idle during periods without location updates, causing intermediate proxies and firewalls to close connections. Additionally, Gateway write buffer was undersized for concurrent location broadcasts.

**Solution:**

- Implemented ping/pong keep-alive frames every 30 seconds
- Added automatic reconnection with exponential backoff (1s, 2s, 4s, max 30s)
- Increased WebSocket write buffer from 4KB to 64KB
- Added connection pooling to reuse healthy connections
- Implemented graceful degradation: buffered updates during disconnection

**Outcome:** Stable connections maintained for 10+ minute sessions with 99.5% uptime.

## 9.3 Challenge 3: Kubernetes Service Immutability

**Problem:** Attempted to convert existing ClusterIP service to Headless via `kubectl apply`, resulting in error:

```
The Service "matching" is invalid:
spec.clusterIPs[0]: Invalid value: ["None"]:
may not change once set
```

**Root Cause:** Kubernetes marks certain service spec fields as immutable post-creation to prevent breaking existing client connections and DNS entries.

**Solution Workflow:**

1. Delete existing service: `kubectl delete svc matching -n lastmile`
2. Verify DNS record cleared: `nslookup matching.lastmile.svc.cluster.local`
3. Apply updated manifest with `clusterIP: None`
4. Roll out pods to pick up new DNS configuration
5. Wait for DNS propagation ( 30 seconds)

**Lesson:** Plan service types during initial design; converting between ClusterIP/-Headless/LoadBalancer requires downtime.

## 9.4 Challenge 4: Mobile Authentication Flow

**Problem:** iOS app login consistently failed with "Network request failed" despite backend services running:

- Android app successfully authenticated
- Browser-based web app worked correctly
- iOS-specific network security policies blocking local IPs
- Minikube Gateway ClusterIP not externally accessible

**Solution Architecture:**

1. **Local Dev:** Exposed Gateway via Minikube tunnel: `minikube service gateway -n lastmile`
2. **iOS Testing:** Configured ngrok HTTPS tunnel for external access with valid SSL certificate
3. **CORS Fix:** Updated Gateway to allow `Origin: *` for development
4. **Mobile Config:** Set `EXPO_PUBLIC_API_URL` to ngrok URL in `.env`

**Additional Issues Resolved:**

- iOS App Transport Security (ATS) requiring HTTPS
- Certificate validation bypassed for dev environment
- Network reachability checks added before API calls

### 9.5 Challenge 5: Real-time Room State Synchronization

**Problem:** Driver action buttons ("Arrived at Pickup", "Fast-Forward") not triggering backend state changes. Frontend showed button clicks but trip state remained unchanged.

**Root Cause:** Event handlers in React Native components not properly bound to Gateway WebSocket room API. State updates were local-only without server propagation.

**Solution:**

- Debugged event flow with Gemini's assistance analyzing component hierarchy
- Fixed API endpoint routing in Gateway for room state mutations
- Implemented bidirectional state sync: UI $\rightarrow$ WebSocket $\rightarrow$ Gateway $\rightarrow$ Room State $\rightarrow$ Broadcast
- Added optimistic UI updates with rollback on server rejection

## 10 Conclusion

LastMile demonstrates a production-grade microservices architecture with:

- 8 microservices with gRPC communication
- Horizontal autoscaling (1-8 replicas)
- Client-side load balancing for gRPC
- Real-time WebSocket tracking
- Kubernetes deployment with fault tolerance
- AI-accelerated development

**Key Lessons:**

1. gRPC requires client-side load balancing in Kubernetes
2. Headless services enable pod-level DNS resolution
3. HPA requires careful resource limit configuration
4. AI assistants excel at debugging distributed systems

## References

[1] gRPC Authors, *gRPC Load Balancing*, https://grpc.io/blog/grpc-load-balancing/

[2] Kubernetes Authors, *Horizontal Pod Autoscaling*, https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[3] Kubernetes Authors, *Headless Services*, https://kubernetes.io/docs/concepts/services-networking/service/#headless-services

[4] Google DeepMind, *Google Gemini*, https://deepmind.google/technologies/gemini/