College Pool Microservices Dispatch Guide

A step-by-step build plan for a deadline-aware, pooled ridesharing dispatch system (filter, shortlist, refine, commit) implemented as three core microservices: Dispatch, Routing/ETA, and Vehicle State.

Version: 2026-02-14

Audience: engineers building a rideshare back end from scratch.

Scope: the dispatch hot path only (matching and schedule commits). Pricing, payments, messaging, and analytics are intentionally out of scope.

Design principles

- Per-trip deadlines: riders provide an arrive-by time when they have a commitment; otherwise the system derives a deadline using a detour factor.
- Hard correctness invariants in one place: Vehicle State is the single writer for schedules; Dispatch never writes schedules directly.
- Batch routing calls: route planning and insertion evaluation must use matrix or batched APIs to stay fast.
- Graceful degradation: if routing is slow or load spikes, reduce shortlist size (kappa) or fall back to a cheap heuristic ranker.

Reference: Liu et al., "Towards Efficient Ridesharing via Order-Vehicle Pre-Matching Using Attention Mechanism", IEEE ICDM 2024 (conceptual basis: filter-and-refine + top-k candidate refinement).

Contents

1. System overview and build order

You are building an Uber Pool style matcher for college riders. Trips do not share a global bell time, so each rider request carries its own time constraints. The dispatch hot path is split into three services. Build them in the order below because each later service depends on the earlier ones.

Core services

- **Vehicle State**: authoritative vehicle schedules, schedule versioning, geo index for candidate search, and atomic schedule commits (CAS).
- **Routing/ETA**: batch travel time queries and cost matrices with aggressive caching.
- **Dispatch**: matching logic (filter -> shortlist -> exact insertion evaluation -> commit).

Recommended build order

- Define contracts (protobuf + event schemas).
- Stand up local infra (Kafka/Redpanda, Redis, Postgres, routing engine).
- Implement Vehicle State (telemetry ingestion, geo index, CAS schedule commits).
- Implement Routing/ETA (BatchTravelTime + CostMatrix + caching).
- Implement Dispatch (candidate search, heuristic top-k shortlist, exact insertion feasibility, commit via Vehicle State).
- Add a replay/simulator harness and full observability before building UI.

2. Contracts first: protobuf and event schemas

Start by freezing the contracts. This prevents later refactors across services and allows parallel work. Use gRPC + protobuf for service-to-service calls, and Kafka events for state propagation.

Event topics

- **orders.v1**: OrderCreated, OrderCancelled
- **vehicles.telemetry.v1**: VehicleLocationUpdated
- **vehicles.state.v1**: VehicleScheduleCommitted, VehicleAvailabilityUpdated
- **dispatch.decisions.v1**: TripAssigned, OrderRejected

Partitioning keys

- orders.v1: by pickup geocell (or order_id if you want uniformity).
- vehicles.*: by vehicle_id (keeps per-vehicle ordering).
- dispatch.decisions.v1: by order_id.

RPC surface area (minimum)

- VehicleState.GeoQuery(cells, required_availability, min_seats) -> vehicle summaries
- VehicleState.GetVehicle(vehicle_id) -> full schedule
- VehicleState.TryCommitSchedule(vehicle_id, expected_version, new_schedule, idempotency_key) -> success/failure
- Routing.BatchTravelTime(legs[]) -> durations
- Routing.CostMatrix(points[]) -> NxN durations (preferred for insertion evaluation)

Example: CAS schedule commit (Vehicle State)

```
TryCommitSchedule(
  vehicle_id,
  expected_schedule_version,
  new_schedule[],
  idempotency_key
) -> { success, new_schedule_version, failure_reason }
```

3. Local infrastructure (docker-compose)

Bring up the smallest local environment that exercises the hot path. Use docker-compose to keep onboarding simple.

Containers

- Kafka (or Redpanda) for events
- Redis for hot vehicle state and geo index
- Postgres for durable logs and audits (optional for MVP but recommended)
- Routing engine (OSRM/Valhalla/GraphHopper) plus your Routing/ETA wrapper service
- Optional: Prometheus + Grafana + Jaeger/Tempo (tracing)

Deliverable

One command starts everything, and a simple script can publish a VehicleLocationUpdated event and observe the resulting Vehicle State update.

4. Vehicle State service

Vehicle State is the single writer for schedules and schedule_version. Dispatch proposes a schedule; Vehicle State accepts it only if the version matches. This prevents double-assignments when multiple orders compete for the same vehicle.

Responsibilities

- Ingest telemetry and maintain current_location.
- Maintain geo index (geocell -> set of vehicle_ids) for fast candidate retrieval.
- Store authoritative schedule and schedule_version for each vehicle.
- Expose CAS schedule commit API and emit VehicleScheduleCommitted on success.
- Enforce idempotency on commits (same idempotency_key returns the same outcome).

Suggested storage

- Redis: veh:{vehicle_id} (hot JSON), cell:{geocell} (set).
- Postgres (or event log): vehicle_schedule_log for audits and training data.

Implementation steps

- **4.1** Implement telemetry consumer: update veh:{id}.current_location and move between cell sets when the geocell changes.
- **4.2** Implement GeoQuery: union vehicles across cells, batch fetch vehicle summaries, filter by availability and seats.
- **4.3** Implement TryCommitSchedule with an atomic CAS (Redis WATCH/MULTI/EXEC or Lua).
- **4.4** Emit VehicleScheduleCommitted events and (optionally) persist to Postgres asynchronously.

CAS invariants (must hold)

- A schedule commit must be atomic and conditional on expected_schedule_version.
- schedule_version must increase monotonically.
- Schedule updates must be idempotent by idempotency_key.

5. Routing/ETA service

Routing calls can dominate latency if they are not batched and cached. The Routing/ETA service exists so Dispatch never calls OSRM (or similar) directly, and so you can standardize caching and observability.

Responsibilities

- Provide BatchTravelTime for many origin-destination legs.
- Provide CostMatrix for small point sets (ideal for insertion evaluation).
- Implement multi-level caching (in-process LRU + Redis).
- Expose routing latency and cache-hit metrics.

CostMatrix is the workhorse

For a vehicle schedule of length m and one new rider (two points), you can build a point set of size (m + 2 + 1 for current position). One matrix call returns all pairwise costs; Dispatch can evaluate many insertion positions without additional routing calls.

Implementation steps

- **5.1** Wrap your routing engine table/matrix API behind CostMatrix(points).
- **5.2** Add caching keyed by snapped node ids (preferred) or rounded lat/lng pairs.
- **5.3** Add BatchTravelTime for small, latency-sensitive checks (e.g., pickup ETA during filtering).

6. Dispatch service

Dispatch consumes OrderCreated events and attempts to assign each order to a vehicle by proposing an updated schedule and committing it through Vehicle State.

Dispatch pipeline

- **Filter**: get initial candidate vehicles near the pickup and with enough seats.
- **Shortlist**: rank candidates using a cheap heuristic (top-kappa).
- **Refine**: run exact insertion feasibility checks and compute objective for the shortlisted vehicles.
- **Commit**: CAS schedule commit via Vehicle State; retry on version mismatch.

Step-by-step implementation

- **6.1** Order consumer: subscribe to orders.v1 and enqueue work items.
- **6.2** Candidate search: compute pickup geocell and neighbor rings; call VehicleState.GeoQuery.
- **6.3** Cheap pre-checks: compute pickup ETA and reject candidates that cannot meet latest_pickup_time.
- **6.4** Shortlist to top-kappa: compute a fast score (pickup ETA + slack estimate + rough incremental time) and keep the best kappa vehicles.
- **6.5** Exact refinement: for each shortlisted vehicle, enumerate insertion positions for pickup and dropoff, compute ETAs via a single CostMatrix, and apply feasibility checks.
- **6.6** Commit: call VehicleState.TryCommitSchedule with expected_schedule_version and idempotency_key.
- **6.7** Emit decision: TripAssigned on success; OrderRejected with reason on failure.

Pseudo-code (high level)

```
on OrderCreated(r):
  Cr = GeoQuery(near pickup)
  Cr = fast_reject(Cr)
  Ck = shortlist_top_kappa(Cr)
  best = argmin_{w in Ck} objective(best_feasible_insertion(r, w))
  if best exists:
    if TryCommitSchedule(best.vehicle, best.expected_version, best.new_schedule):
      emit TripAssigned
    else:
      retry or pick next best
  else:
    emit OrderRejected
```

7. Feasibility checks (exact rules)

These checks determine whether inserting a new rider into a vehicle schedule is allowed. Treat them as hard constraints.

Inputs

- Order r: pickup, dropoff, request_time, latest_pickup_time, latest_arrival_time, party_size, optional detour factor rho.
- Vehicle w: current_location, capacity, current schedule (ordered stops), schedule_version, existing rider constraints, optional driver constraints.

ETA computation

- Build the candidate stop sequence for a proposed insertion.
- Request one CostMatrix over {current_position, existing stop locations, r.pickup, r.dropoff}.
- Compute ETAs by accumulating leg durations and stop service durations.

Hard constraints

- **Pickup after request**: eta_pickup(r) >= request_time(r).
- **Pickup by latest pickup**: eta_pickup(r) <= latest_pickup_time(r).
- **Dropoff by latest arrival**: eta_dropoff(r) <= latest_arrival_time(r).
- **Existing riders remain feasible**: all their pickup and dropoff deadlines still satisfied.
- **Precedence**: for every rider, pickup occurs before dropoff in the stop sequence.
- **Capacity**: onboard count never exceeds capacity (accounting for party_size).
- **Driver constraints** (if enabled): driver must reach must_arrive_at by must_arrive_by and finish before shift_end.

Optional detour constraint (recommended)

If you want to enforce a detour factor, compute direct_time(q) and actual_ride_time(q) for each rider q and require actual_ride_time(q) <= direct_time(q) * rho_q.

8. Objective function: lateness risk vs detour

Even when all plans are feasible, you should prefer plans with more slack (lower risk of missing deadlines) while still limiting detours and waiting.

Definitions

- For each participant p (each rider and optionally the driver): slack_p = deadline_p - ETA_p.
- Incremental vehicle time: DeltaT = route_time(new_schedule) - route_time(old_schedule).
- New rider waiting: wait_r = ETA_pickup(r) - request_time(r).
- Per-rider detour: detour_q = max(0, actual_ride_time(q) - direct_time(q)).

Risk penalty

Choose risk_buffer_ms (example: 5 minutes). Penalize plans that leave very little slack.

```
late_penalty  = sum_p (max(0, -slack_p))^2
tight_penalty = sum_p (max(0, risk_buffer_ms - slack_p))^2
```

Recommended plan comparison (lexicographic)

- 1) minimize late_penalty (should be zero for feasible plans)
- 2) minimize tight_penalty (more slack is safer)
- 3) minimize DeltaT (system efficiency)
- 4) minimize wait_r (rider experience)
- 5) minimize sum(detour_q) (fairness and comfort)

Scalar alternative

```
J = 1e6*late_penalty + 1e3*tight_penalty + 1*DeltaT + 2*wait_r + 1*sum(detour_q)
```

9. End-to-end test harness and observability

Before building a UI, build a replay/simulator that produces realistic load and validates invariants. Observability is how you tune kappa, buffers, and routing caches.

Replay harness checklist

- Generate N vehicles near campus; publish telemetry pings every few seconds.
- Generate OrderCreated events with random pickup/dropoff around campus and random per-trip deadlines.
- Validate: no capacity violations, no precedence violations, no missed deadlines in committed schedules.
- Record: candidate counts, shortlist size, refine evaluations per order, and decision latency.

Metrics to ship on day one

- Dispatch: candidate_count, shortlist_kappa, refine_evaluations_per_order, decision_latency_ms (p50/p95), rejection reasons.
- Safety: min_slack_ms distribution, tight_penalty distribution, % of trips with slack < risk_buffer_ms.
- Routing: cost_matrix_latency_ms, cache_hit_rate (L1 and Redis), routing_errors.

10. Hardening and scaling checklist

Correctness and reliability

- Idempotency: enforce idempotency keys for schedule commits and order processing.
- Cancellation: define policy for OrderCancelled (remove future stops if not picked; or mark as canceled and handle refunds).
- Vehicle offline: stop assigning when availability != ONLINE; decide how to reassign affected riders.
- Backpressure: if routing degrades, reduce kappa or fall back to heuristic-only matching.

Scaling

- Partition orders by pickup geocell (Kafka partitions).
- Run Dispatch as a consumer group; keep Vehicle State as the single writer for schedules.
- Scale Routing horizontally behind a load balancer; caching is mandatory.
- Consider sharding Vehicle State by vehicle_id if Redis becomes a bottleneck.

Appendix: Implementation milestones (suggested)

Milestone	Deliverable
M0	Contracts: protos + event schemas frozen; codegen in CI
M1	Vehicle State: telemetry ingestion, geo index, CAS schedule commits
M2	Routing/ETA: BatchTravelTime + CostMatrix with caching
M3	Dispatch: filter/shortlist/refine/commit; emits TripAssigned/OrderRejected
M4	Replay harness: invariant checks + load profiles
M5	Observability: tracing + dashboards + alerting; tuning buffers and kappa