System Design Mini Project

Food-Delivery Aggregator (Orders, Dispatch, Live Tracking)

B Tech (CSE-GEN(AI))

Submitted To

Lovely Professional University

Food Delivery Aggregator – System Design Document

Table of Contents / Index

Food Delivery Aggregator – System Design Document

This document presents a comprehensive design for a **Food-Delivery Aggregator** system. We follow a structured architectural report style, combining high-level and detailed elements. It will include context and use-case diagrams, component designs, data models, APIs, caching strategies, rate-limiting, and SLO considerations. We assume a realistic scenario (e.g. a large metro area with millions of users) to ground our scale and performance targets, yet keep the design abstract enough to be generally applicable. Example API contracts and JSON schemas are provided. Throughout, we cite authoritative sources on large-scale food delivery architecturesozdemirtim.medium.comdev.to.

Overview and Context

The system is a **three-sided marketplace** connecting **customers**, **restaurants**, and **couriers/drivers**dev.to. Customers browse and order food, restaurants manage menus and prepare orders, and drivers deliver orders. In addition, operators/ops teams monitor the system. We integrate third-party services for **maps/routing** and **payments**, and provide user notifications. A high-level **context diagram** (Figure 1) depicts these external actors and services.

Figure 1: System Context – users interact via apps; integrations include mapping (for routing/ETAs), payment gateways, and notification services. Actors include Customers, Restaurants, Drivers, and Ops users.

Core goals include: **24/7 availability**, **real-time performance** (e.g. live location/ETA updates p95 < 1s), **high scalability** (handling tens of thousands of concurrent deliveries), and **maintainability** (clear domain/service boundaries). As Ozdemir notes, key design considerations are location-based search, CQRS (separating reads/writes), concurrency control (one-driver per order), real-time streaming, and route optimizationozdemirtim.medium.com. Our design leverages microservices, event-driven patterns, and partitioning by geography (hot regions) to meet these needs.

Stakeholders and Requirements

Stakeholders: Customers (end-users placing orders), Restaurants (vendors managing menus/orders), Couriers/Drivers (delivery agents), and Ops/Support (monitoring system health)dev.to.

Functional Requirements: The system must allow:

- **Search & Discovery:** Customers find nearby restaurants (by cuisine or location)ozdemirtim.medium.com.
- **Menu Browsing:** Fetch restaurant menus (cached for low latency)ozdemirtim.medium.comozdemirtim.medium.com.
- **Cart & Ordering:** Place orders (with item list) and pay via integrated payment gatewayozdemirtim.medium.com.
- **Order Tracking:** Real-time status updates (accepted, picked up, enroute, delivered) and live map trackingozdemirtim.medium.comdev.to.
- **Dispatching:** Automatically assign available drivers to new orders (with driver accept/reject flow)ozdemirtim.medium.comdev.to.
- **Notifications:** Push order/status updates to customers (e.g. via WebSocket or push) and to drivers.
- **Restaurant App:** Interface for restaurants to receive new orders, accept/reject, and update order status.
- **Driver App:** Interface for drivers to see assignments, get routing, update location, and mark delivery status.

Non-Functional Requirements:

- **Availability:** 24/7 uptime, with multi-region failover.
- **Performance:** Real-time location/ETA updates – target **p95 < 1s** delivery of location updates to customers.
- **Scalability:** Support **>10k concurrent deliveries**, millions of users, and hundreds of thousands of restaurants. Use geo-sharding to localize traffic.
- **Consistency:** Mixed consistency. Critical actions (order placement/pay) require transactional integrity, while tracking/ETA updates can be **eventually consistent**medium.com.
- **Maintainability:** Clear service boundaries (e.g. separate Order, Dispatch, Tracking services), favor microservices or bounded contexts.
- **Security:** OAuth for users, tokenization for payments, and protection against DDoS (e.g. API gateways, rate limits).

These requirements align with best practices: for instance, Ozdemir's list includes efficient search, high concurrency control, and real-time streamingozdemirtim.medium.com. Gregory

Chris's overview emphasizes optimizing delivery time, driver utilization, and restaurant workflowdev.to.

High-Level Architecture

We adopt a **microservices architecture** with stateless frontend APIs and specialized backend services. Figure 2 (schematic) shows the core service blocks:

```
Frontend (Apps/Web UI) —> [ API Gateway / LB ] —> Order Service

                                         |

                                         +--> Dispatch Service

                                         +--> Geospatial Service (routing, traffic)

                                         +--> Tracking Service (streaming)

                                         +--> Notification Service

                                         +--> (Others: Pricing, Analytics)

Back-end Services —> Databases (SQL for orders/payments; NoSQL for tracking/events)
```

- **API Gateway / Load Balancer:** Routes requests to microservices and handles SSL termination, auth, rate limiting, etc.
- **Order Management Service:** Handles cart checkout, payment processing, and order persistence. Likely a relational DB behind (for transactions).
- **Dispatch Service:** Matches orders to drivers (explained below).
- **Geospatial/Routing Service:** Calls third-party maps (e.g. Google Maps API) or on-prem routines to calculate routes and ETAs.
- **Tracking Service:** Aggregates real-time GPS from drivers and pushes updates (via WebSockets/SSE) to usersdev.todev.to.
- **Notification Service:** Sends SMS/push/email updates (through Twilio, etc.) when key events occur.
- **Other Services:** Dynamic pricing, demand forecasting, analytics pipelines, etc., as needed.

Data infrastructure: We use a mix of databases: a relational DB (e.g. PostgreSQL) for transactional data (orders, menus, user accounts)medium.com, and NoSQL stores (Redis, Cassandra) for fast reads/analytics (e.g. caching menu data, tracking events)medium.com. Kafka (or other message bus) carries events between services (e.g. order events to dispatch, GPS events to tracking)medium.com.

This high-level design echoes published architectures: for example, Gregory Chris illustrates a similar stack of Order Service, Geospatial/Dispatch, Tracking (Kafka/WebSockets)dev.todev.to. Bugfree.ai's design also highlights caching menus, replicating data, and using WebSockets for trackingmedium.commedium.com.

Domain Model & Data Schema

Key **entities** include:

- **User:** (Customer or Courier) – Profile, address, login.
- **Restaurant:** Name, location, availability, etc.
- **Menu/MenuItem:** Each restaurant's menu items (name, price, etc.).
- **Order:** Contains user_id, restaurant_id, list of (menu_item_id, quantity), total price, timestamp, status.
- **Delivery:** Ties an Order to a Driver; includes status, current location, ETA, and planned route.
- **Driver/Courier:** Profile of delivery agents (current status: available/occupied; current location).

This aligns with Ozdemir's domain list: User, Restaurant, Menu, MenuItem, Order, Delivery, DeliveryAgentozdemirtim.medium.com. We design database tables (or documents) accordingly. For example, an orders table has columns (order_id, user_id, restaurant_id, total_amount, status, created_at, etc.), and an order_items table listing the items. The deliveries table has (delivery_id, order_id, driver_id, route_polyline, status, updated_at). **Routes** (GPS paths) can be stored as encoded polylines or in a spatial data store; frequently they are fetched on-the-fly from map APIs but we may persist them for rerouting.

We may also use a **NoSQL store** (e.g. Redis or Cassandra) for *fast reads and analytics*: e.g. caching menus (as JSON)medium.com, and time-series of driver locations (for tracking). An example data model (simplified) is:

Users(user_id, name, role, auth_token, ...)

Restaurants(rest_id, name, location, hours, ...)

Menus(rest_id, item_id, name, price, availability, ...)

Orders(order_id, user_id, rest_id, total, status, created_at)

OrderItems(order_id, item_id, quantity)

Drivers(driver_id, name, current_status, current_location)

Deliveries(delivery_id, order_id, driver_id, status, route_polyline, eta)

We ensure **database partitioning** by geography for scale – e.g. sharding restaurant and order data by city or regionmedium.comblog.bytebytego.com. Uber's H3 hex grid is an example: each store's delivery zones are mapped to hexagons, and queries target only relevant shardsblog.bytebytego.comblog.bytebytego.com. We will similarly shard by location to localize traffic.

Functional Components & Flows

Below are key components with their responsibilities:

- **Order Service:** Receives POST /orders (with cart and payment info), validates stock and payment, and creates an Order recordozdemirtim.medium.com. It then emits an "Order Created" event. It may follow a saga pattern to coordinate payment and order persistence (ensuring atomic commit across services).
- **Dispatch Service:** On receiving a new order, it **matches** an available driver. Matching algorithm (detailed below) considers proximity, ETA, and driver status. It locks the candidate driver (e.g. via a fencing token or distributed lockozdemirtim.medium.com to avoid double-assign). It updates the Delivery record and notifies the Driver app of the assignment. Drivers have an endpoint (e.g. POST /delivery/{id}/accept) to accept or reject.
- **Geospatial Service:** Provides routing and ETA calculations. For each delivery assignment, it calls a maps API to get the best route from the restaurant to the customer, estimating travel time with traffic. It can also be queried by the Order Service to show estimated time on checkout. Third-party maps (e.g. Google Maps) are used, with caching of common queries to reduce cost. Because external map data can change, ETA can be treated eventually consistent (updates as new traffic data arrives)medium.com.
- **Real-Time Tracking Service:** Drivers' mobile apps push GPS locations (e.g. every second) to this service via WebSocket or HTTP streaming. These updates are published on a message bus (e.g. Kafka). The Tracking Service consumes the stream, updates the Delivery's current location in cache/DB, and pushes updates to the customer's app via WebSocket/SSEdev.to. For low latency, we use a WebSocket server with per-client channels. The delivered ETAs are continuously refined based on actual location and speeddev.to.
- **Notification Service:** Listens to events (order confirmed, driver assigned, enroute, delivered) and sends messages (push, SMS or email) to stakeholders. We store user device tokens for push notifications and use reliable delivery with retries.
- **Restaurant & Driver Apps:** Lightweight clients. Restaurants can GET incoming orders and PUT status updates. Drivers see assigned delivery details and can PUT location updates. Both apps connect to backend via HTTP or WebSockets.

Dispatcher Matching Algorithm

For *driver dispatch*, we implement a **priority matching algorithm**dev.to. A new order triggers a search among nearby available drivers. We compute a score for each candidate based on: distance from restaurant, current predicted ETA to dropoff, driver's current load and acceptance likelihood, and fairness (round-robin or assignment history). We use a priority queue to rank drivers and select the top one. If the driver rejects (via the app), we repeat with the next-best. This is similar to Uber's and DoorDash's matchingdev.to.

To scale, we **partition drivers geographically** (e.g. by city blocks or H3 zones) so each dispatch query only considers local driversdev.toblog.bytebytego.com. This keeps the search fast. We run the dispatch logic in a stateless microservice that can scale out. We also ensure concurrency control: once a driver is chosen, we use a distributed lock or token (fencing) to mark the driver busy, guaranteeing *exactly one* assignment per orderozdemirtim.medium.com.

API Contracts

We define RESTful APIs (with JSON) for key interactions. Examples:

- **Browse Restaurants:**
  GET /restaurants?lat={lat}&lon={lon}&radius={r} – returns nearby restaurants (via geo-index).
- **Get Menu:**
  GET /restaurants/{id}/menu – returns the restaurant's menu (cached in CDN/Redis).
- **Place Order:**
  POST /orders with body {"userId":..., "restaurantId":..., "items":[{"id":..,"qty":..}], "paymentMethod": "card", ...}.
  Response: {"orderId":..., "status":"CREATED", "estimatedTime":...}. The system then processes payment and finalizes.
- **Order Status:**
  GET /orders/{orderId} – returns current status and details.
- **Driver Location Stream (WebSocket):** The client opens wss://.../ws/track?deliveryId=...&token=.... Server pushes messages like {"lat":..., "lon":..., "eta": 10} as the driver moves.
- **Driver Actions:**
  POST /deliveries/{deliveryId}/accept – driver accepts assignment.
  POST /deliveries/{deliveryId}/complete – driver marks delivery done.

We can specify these in OpenAPI style if needed. All APIs are idempotent or support idempotency keys for retries. For example, order creation should use an idempotency token to avoid double charges.

Each service also has internal APIs or message endpoints. For instance, the Dispatch Service might expose a gRPC or REST endpoint /dispatch/assign that the Order Service calls when a new order is ready to be routed. Internally, events (Kafka topics) carry order-created, driver-updated, etc., to decouple services.

Caching and CDN

To meet performance targets, we heavily cache **static or semi-static data**. In particular, restaurant menus change infrequently, so we serve them via a CDN or an edge cache (Redis/Cache layer)medium.com. When a restaurant updates its menu, we invalidate the cache or push a refresh. Similarly, static assets (logos, images) go through a CDNmedium.com.

We also use in-memory caching (Redis/Memcached) for hot data like user sessions, driver availability lists, and partially computed routes. For example, popular routes or recently computed ETAs can be cached for a minute. This reduces load on the mapping API and databases. We apply appropriate TTLs (e.g. 30s for GPS route data, 5 min for menus).

For **real-time location data**, we do **not cache on clients** but stream live updates. However, within the server, we may maintain a sliding-window cache of the most recent location of each driver (with, say, 5-second expiration) to answer queries quickly.

Scalability and Sharding

To handle large scale, we design for **horizontal scaling**medium.com. Each stateless service can be replicated behind a load balancer. Kubernetes (or similar) can auto-scale pods based on CPU or custom metrics (requests/sec). Databases are sharded and replicated:

- **User/Orders DB:** A primary RDBMS is partitioned by region (e.g. city or first letter of zip)medium.com. During peaks, we autoscale read replicas and queue writes if needed.
- **Cache Shards:** Redis clusters use consistent hashing to distribute sessions, or sharding by region for location data.
- **Geo-partitioning:** As noted, we use geo-sharding for location-based queriesblog.bytebytego.com. Each user/driver query only hits its local shard/region, minimizing cross-datacenter traffic. This also helps "hot region" scenarios (e.g. downtown at lunch) by isolating load.

We ensure **multi-region deployment** for HA. For example, a system serving "City A" might have two data centers; if one fails, the other can takeover (with an allowable RTO that fits our error budget). Data is asynchronously replicated cross-region.

Performance Optimization

To meet the **1-second p95** update goal, we minimize latency at every layer: edge caching of content (menus, profile data), use WebSockets for push updatesdev.to, and prefer binary protocols (gRPC) for internal calls. We also monitor and optimize key latencies (see SLO section).

Eventual consistency for ETA is acceptable; we might recalc ETA every 30s or so and push corrections, rather than insist on a single canonical source. This permits us to relax locks on the mapping API and use cached traffic data.

For search and discovery, we maintain a **geospatial index** (e.g. ElasticSearch with geo capabilities) updated as restaurants open/close. This handles "find nearby restaurants" queries in <100ms. In line with Uber Eats' approach, we limit search radius and early terminate queries to bound latencyblog.bytebytego.comblog.bytebytego.com.

Rate Limiting and Backoff

We implement **rate limiting** at API gateways and per-user basis to prevent abuse. For example, each user session might be limited to X restaurant searches or menu fetches per minute. If a client exceeds limits, we return HTTP 429. Clients are expected to use **exponential backoff** on 429s: e.g. wait 2s, then 4s, 8s, etc. before retryingsubstack.thewebscraping.club. This gives the server time to recover and prevents thundering-herd retriessubstack.thewebscraping.club.

For **WebSocket streams**, we apply **backpressure**: the server limits the rate of messages per connection (e.g. max 10 location updates/sec) and drops or batches older updates if the client is slowdev.to. The server monitors socket buffers (see [34†L372-L380]) and pauses reading upstream data if the client is overwhelmed. This ensures a misbehaving or slow client does not crash the server.

All client apps implement jittered exponential backoff on transient errors. We also add rate-limit headers (Retry-After) in API responses so clients can adapt.

Reliability & SLOs

We define strict **SLIs/SLOs** for critical flows. Example SLOs might be:

- **Location Update Latency:** p95 < 1s, p99 < 3s from driver to customer.
- **Order Placement API:** p95 < 200ms.
- **System Uptime:** 99.9% (allowing ~43 minutes downtime per month).

These translate to error budgets: e.g. 0.1% downtime or request failures. We continuously monitor metrics like **API latency (p50/p95)**, **error rates**, and **throughput**. Tools and dashboards track backend p95 latencies and client-side performance (page load times)ozdemirtim.medium.com. We also log queue lengths (load balancer, Kafka) and resource usageozdemirtim.medium.com to detect surges.

For **alerts and auto-scaling**, we use thresholds (e.g. CPU 70%, queue len high) to provision more instances. If SLOs are breached, we escalate (pager, or automated rollback if a deploy caused latency spike). An error budget policy drives decision-making: as long as we are within budget, we can perform upgrades; otherwise we focus on stabilization.

Observability and Error Handling

Extensive logging and tracing (e.g. via OpenTelemetry) are used so that we can debug issues end-to-end. Each service logs structured events (order_id, delivery_id, etc.) and spans RPC calls. We instrument metrics on key operations (cache hit rates, DB query times, etc.).

Graceful degradation is planned: e.g. if the real-time tracking system is overloaded, we may temporarily reduce update frequency (send every 2–5 seconds instead of 1s) to preserve overall availability. If a service fails (e.g. Map API), we have fallback flows (use last-known ETA, or a simple distance heuristic).

Error budgets guide us here: for instance, allowing some percentage of location updates to fail under extreme load, rather than crashing the service entirely. We aim to keep core functionality (ordering, dispatching) fully available even if auxiliary features (e.g. real-time ETA accuracy) degrade slightly.

API Example and Data Schema (Mock)

Below is a sample OpenAPI-style schema for the **Place Order** endpoint (for illustration):

```
paths:
 /orders:
  post:
   summary: Place a new order
   requestBody:
    content:
     application/json:
      schema:
       type: object
       properties:
        userId: { type: string }
        restaurantId: { type: string }
        items:
          type: array
          items:
            type: object
            properties:
              menuItemId: { type: string }
              quantity: { type: integer }
        paymentMethod: { type: string, enum: [credit_card, paypal, apple_pay] }
      required: [userId, restaurantId, items]
  responses:
   '201':
     description: Order created
     content:
       application/json:
         schema:
           type: object
           properties:
             orderId: { type: string }
             status: { type: string, example: "CONFIRMED" }
             estimatedDeliveryTime: { type: integer, example: 27 } # minutes
```

And a simplified **Order** table schema (SQL) might be:

```
CREATE TABLE Orders (
  order_id UUID PRIMARY KEY,
  user_id UUID NOT NULL,
  restaurant_id UUID NOT NULL,
  total_amount DECIMAL NOT NULL,
  status VARCHAR(20) NOT NULL,
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
```

);

With an associated OrderItems(order_id, menu_item_id, quantity) table.

Use-Case Diagram

Key use-cases (actors in parentheses):

- **Place Order** (Customer → System).
- **Browse Menu** (Customer → System).
- **Track Delivery** (Customer ← System).
- **Manage Order** (Restaurant ↔ System).
- **Accept Delivery** (Driver ↔ System).
- **Update Location** (Driver → System).
- **Assign Driver** (System internal).
- **Notify User** (System → Customer).

These map to user stories like “As a customer, I can browse restaurants, place an order, and track it live” and “As a restaurant, I receive orders and confirm them.” A use-case diagram (e.g. in UML) would show actors Customer, Restaurant, Driver, and external services (Maps, Payment) linked to these use cases.

Dispatcher Design (Detail)

The **Dispatch Service** will typically use a micro-batch or event-driven approach: it subscribes to “OrderCreated” events. For each order, it identifies candidate drivers by geospatial query (within radius, or next free driver). It then evaluates each by computing the travel time from that driver’s current location to the restaurant+customer. This may use the Geospatial Service’s API (or just Haversine distance + average speed). It scores each driver as in [9†L361-L366] and selects the optimal. If no driver is free, it can queue the order until one becomes available (and alert ops).

We ensure that **only one driver is assigned** per order via concurrency control. For example, when picking a driver, we write a lock record or use a “fencing token” so that if two dispatch threads race, only one succeedsozdemirtim.medium.com. If a driver declines, we remove that candidate and retry. All this should happen within a short time budget (e.g. <500ms) to start delivery promptly.

Scalability Scenario (Example)

As a concrete example, consider a city with **1,000,000 daily active users**, **300,000 restaurants**, and **200,000 orders/day**ozdemirtim.medium.com. If 80% of orders come in during a 4-hour dinner peak, that’s ~40,000 orders/hour (~11 orders/sec)ozdemirtim.medium.com. With each driver sending ~1 update/sec for 20 minutes, that's ~720 location updates/sec in steady state, spiking up to ~14,400 updates/sec during a 20× surgeozdemirtim.medium.com. To handle this, our Tracking Service uses Kafka and stream processors (e.g. Flink) to aggregate and forward updatesdev.to. We shard drivers by neighborhood so each service instance handles only local streams, and we geo-replicate servers for HAdev.to.

The Order Service sees ~~0.2k orders/sec on average, requiring a DB pool that can handle ~14,400 orders/min burstsozdemirtim.medium.com. We use batching (e.g. database write queues) if needed to smooth spikes, and autoscale pods to meet demand.

Maintainability and Domain Boundaries

By splitting the system into focused services (Order, Dispatch, Tracking, etc.) with well-defined APIs, we achieve clear domain boundaries. This eases development and maintenance: each team can own one service. We also adopt **domain-driven design** concepts: e.g. a Delivery domain containing driver assignment and route, distinct from the Order domain. Shared libraries (for auth, models) reduce duplication.

Using CQRS (as Ozdemir suggests)ozdemirtim.medium.com, we separate read queries (e.g. "get order status") from writes (place order, update status) so they can scale independently. Read models (e.g. denormalized order view) are updated via events.

Security and Operations

We require **authentication** (OAuth/JWT) for all client requests. Sensitive data (payments, user info) is stored encrypted. We use HTTPS everywhere. We protect against DDoS by rate-limiting and having scalable frontends. For payments, we never store raw card data – we use tokenization and PCI-compliant gateways.

Monitoring/Logging: All services emit metrics (Prometheus) and logs (structured JSON). We track errors, latencies, and business metrics (#orders, #deliveries).

Error Handling: For user-facing flows (ordering), we use transactional techniques (database 2PC or reliable saga) so that failures trigger compensating actions (e.g. refund if order creation fails mid-way). APIs are idempotent wherever possible (duplicate requests have no extra effect).

Conclusion

In summary, the proposed design addresses the functional and non-functional requirements through a microservices-based, event-driven architecture. By leveraging caching/CDN for menusmedium.com, partitioning data geographicallyblog.bytebytego.com, and using WebSockets/Kafka for real-time streamsdev.todev.to, we meet the goals of 24/7 availability, real-time tracking, and massive concurrency. Rate limits and backoff protect system stabilitydev.tosubstack.thewebscraping.club. Clear APIs and modular services ensure maintainability. We define strong SLOs (e.g. 99% of location updates <1s) and continuously monitor error budgets to keep the system healthy. The references above illustrate that these design patterns are standard in production food-delivery platformsozdemirtim.medium.comdev.tomedium.com.

Sources: We have drawn on industry examples and expert writings on food-delivery architecturesozdemirtim.medium.comdev.todev.toblog.bytebytego.comozdemirtim.medium.com. All design choices above are grounded in these references and established best practices.

Food Delivery Aggregator: System Design

Introduction & Context. A modern food-delivery marketplace is a *three-sided marketplace* connecting **customers**, **restaurants**, and **drivers/couriers**dev.to. Customers browse menus and place orders (with payment), restaurants confirm and prepare meals, and couriers pick up and deliver the food. All actors must stay in sync: customers need live order/ETA updates, restaurants get timely order notifications, and drivers get dispatch info. A context-level flow might be drawn with customers→system→restaurants and payment gateways, plus drivers notified for dispatchdev.to. For example, "Customer places orders and makes payments. Restaurant receives orders and confirms. Delivery Partner receives dispatch information to deliver food. Payment Gateway handles payments"dev.to. This high-level flow underpins our system.

Requirements

- **Functional Requirements:**
    - *Menu ingestion & browsing:* The system must ingest/upsert restaurant menus and serve them (e.g. via CDN/caching). Customers browse items.
    - *Cart & Order:* Customers add items to a cart and place orders with payment. Restaurants receive the order (and accept/reject).
    - *Dispatch & Delivery:* The system assigns an available courier to each confirmed order (dispatch), optimizing for location and timing. Drivers receive dispatch info and update status.
    - *Real-time tracking:* Continuously track courier location and order status; stream live updates (order received, ready, picked-up, en-route, delivered) to customers and restaurants via push (WebSocket/SSE) connections.
- **Non-functional Requirements:**
    - *Availability:* The service is 24/7 with high uptime (e.g. ≥99.9% availability). Shards/databases are replicated across regions for fault-tolerance.

    - *Performance:* Real-time responsiveness – e.g. 95% of location/status updates should reach clients within 1 second (latency SLI)uber.com.
    - *Scalability:* Support large scale – e.g. tens of thousands of concurrent deliveries. We assume peak loads (e.g. lunch/dinner rush) require handling ~10k+ live orders and driver-tracking sessions simultaneouslymiracuves.comaerospike.com. Horizontal scaling via sharding and microservices is essential.
    - *Maintainability:* Clear domain boundaries (microservices per domain) to allow independent development and deploymentmiracuves.com. Use standard protocols (REST/gRPC) and schemas for all APIs.
- **Technical Requirements:**
    - Use push-based streaming for real-time updates (WebSockets or gRPC bidirectional streaming) rather than pollinguber.commiracuves.com.
    - Integrate third-party services: maps/routing API (e.g. Google Maps/Mapbox) for geocoding and routing, and payment gateways (e.g. Stripe).
    - Use content delivery (CDN) for static assets (menu images, etc.) and caching for menu data to offload backends.
    - Enforce rate limits (per-user/API keys) and client-side backoff on failures to prevent overload.

Stakeholders & Use Cases

Stakeholders:

- **Customers:** Place orders, track delivery, view history, pay via app.
- **Restaurants:** Manage menus, receive/confirm orders, update preparation status.
- **Couriers/Drivers:** Receive dispatch notifications, view route, update delivery status (picked up, delivered), report real-time location.
- **Operations/Admin:** Monitor system health, onboard restaurants/couriers, manage content (promotions, featured items).

Key Use Cases: (context diagram)

- *Order Placement:* Customer searches menu, adds items, checks out (calls Order API).
- *Order Confirmation:* Restaurant views incoming order, confirms or cancels.
- *Dispatching:* Upon confirmation, dispatch service matches order with a courier (based on proximity, predicted ETA, etc.).

- *Real-time Tracking:* Customer (and restaurant) watch courier's progress on a map.
- *Payment & Notification:* Payment is processed securely, and notifications (email/SMS/push) are sent at key events (order accepted, courier arriving, delivered).

High-Level Architecture

We adopt a **microservices, event-driven architecture**miracuves.comdev.to. Core services and data stores include:

- **Frontend Clients:** Mobile/web apps for customers, restaurants, drivers.
- **API Gateway:** Routes requests to backend services, enforces auth and rate limits.
- **Order Service:** Handles order creation, updates status. Persists orders in a database.
- **Menu/Restaurant Service:** Manages restaurant profiles and menus (ingestion and updates). Menu data is read-heavy and cached/CDN-distributed.
- **Dispatch Service:** Matches orders to drivers. Implements the dispatch algorithm (greedy or global optimization with predicted timesinfoq.com).
- **Driver Service:** Manages driver status (available/busy) and location updates.
- **Notification Service:** Sends out push notifications or SMS (order status, ETA).
- **Real-Time Messaging (Event Bus):** E.g. Kafka or Pub/Sub layer. All services publish and subscribe to events (orders placed, payment confirmed, order accepted, driver assigned, driver location).
- **Payment Service:** Integrates with payment gateway to process transactions (via asynchronous events).
- **Data Infrastructure:** Mix of relational databases and NoSQL/caches. For example, use a SQL DB for ACID transactions (orders, payments), and a fast NoSQL or time-series DB for high-throughput data (e.g. driver locations).

The following illustrates an event-driven design:

Figure: Event-driven microservices architecture for a food delivery system. User actions (order placement) publish events to Kafka topics; multiple services (Order, Payment, Inventory, Restaurant, Dispatch) consume the same event in paralleldev.to. Delivery updates feed back into Kafka ("delivery-events") and drive real-time notifications to the user.

Here, placing an order creates an "order-events" message in Kafka. Order, Payment, Inventory, etc., all consume it independently to handle validation, payment, stock, and notifying the restaurantdev.to. The Dispatch service also consumes the order event to assign a courier. When the driver reaches the restaurant and picks up the food, Dispatch publishes a "delivery-events" message (including updated ETA). The Notification service consumes these events to push updates (via WebSocket/SSE) to the customer's appdev.to. This decoupling (using message queues) allows each service to scale independently and survive failures. Uber Eats and others use similar event-driven, microservices setupsdev.todev.to.

Data Model

Key entities and their core fields (illustrative):

- **User (Customer/Courier):** id, name, contact info, type (customer/driver), profile details (e.g. vehicle info for courier).
- **Restaurant:** id, name, location, menu reference.
- **Menu:** (often versioned per restaurant) list of items. Each **MenuItem** has id, name, description, price, image_url, availability (stock or time windows). The Menu/Restaurant Service might push updates into a CDN or in-memory cache for fast reads.
- **Order:** id, customer_id, restaurant_id, list of (item_id, qty), status, total_price, payment_id, assigned_driver_id, timestamps (placed_at, confirmed_at, picked_at, delivered_at), ETA.
- **Driver/Delivery:** id, current location (latitude/longitude), status (idle, en-route to pickup, delivering, offline), vehicle type, rating.
- **Payment:** id, order_id, amount, status. (Often offloaded to a PCI-compliant external gateway with only tokenized details stored internally.)
- **DeliveryRoute:** (optional) planned route (polyline points) for mapping.

Datastores:

- **Relational DB:** Used for orders, payments, user profiles (strong consistency for transactions, joins).
- **NoSQL/Key-Value/Time-Series DB:** For dynamic data like driver locations (millions of writes/sec). For example, a geospatial index database (or Redis) can store each driver's last lat/long for fast nearest-driver queries.
- **Cache Layer (Redis/Memcached):** For hot data (restaurant menus, user session tokens). E.g., restaurant menus can be cached on read to achieve low-latency menu browsing.
- **CDN:** Serve static assets (menu images, etc.) and possibly full menu JSON (since menus change infrequently) to reduce origin load.

Scalability via Sharding: We partition both data and traffic geographically (geo-sharding). For example, customers in New York hit the New York region's servers and database shard, while Paris users hit a Paris shardaerospike.com. This lowers latency and fulfills data locality (and legal) requirementsaerospike.com. Hotspots (e.g. one city having 10× more traffic) must be monitored and additional capacity added to that shard to avoid overloadaerospike.com.

APIs and Data Contracts

We define RESTful (or gRPC/GraphQL) endpoints. Key examples include:

- GET /menus/{restaurantId} – fetch the current menu for a restaurant. (E.g. Uber Eats uses GET /eats/stores/{id}/menusdeveloper.uber.com.) This call should be cacheable/CDN-able as menus update relatively infrequently.
- POST /orders – place a new order. Request includes customer ID, restaurant ID, list of item IDs & quantities, address, payment info. Response returns an orderId and initial status.
- GET /orders/{orderId} – get current status and details of an order, including assigned driver and ETA.
- POST /orders/{orderId}/cancel – cancel an order (with appropriate rules/time-window).
- POST /payments – (or handled internally) to process payment for an order. On success/failure it triggers events for order update.
- **WebSocket Endpoint** /ws – Clients open a WebSocket (or SSE) connection after placing an order. The server pushes events like order-confirmed, driver-en-route, driver-location-update, delivered. These messages contain JSON with the updated state (e.g. {orderId, status, timestamp, driver:{lat,lng}, ETA}).

Data schemas: We design JSON schemas for each entity. For example, an Order JSON might be:

```
{
  "orderId": "A12345",
  "customerId": "C456",
  "restaurantId": "R789",
  "items": [{"itemId":"I1","qty":2}, {"itemId":"I2","qty":1}],
  "totalPrice": 37.50,
  "status": "CONFIRMED",
  "assignedDriverId": "D234",
  "etaSeconds": 900
```

}

APIs should follow versioning (e.g. /api/v1/...) and return clear HTTP status codes (200 OK, 400 Bad Request, 404 Not Found, 500 Server Error). For example, the Uber Eats **Get Menu** API is defined as:

GET https://api.uber.com/v2/eats/stores/{store_id}/menus

which returns the restaurant's menu JSONdeveloper.uber.com. We would use a similar pattern (GET /menus/{id}) internally.

Dispatch & Matching Algorithm

At order-confirmation time, the **Dispatch Service** must quickly match the order with an available courier. A simple **greedy** strategy matches each order to the nearest idle driver. However, to improve overall system efficiency, a **global matching** strategy can be used: periodically solve a global assignment problem matching all pending orders to available drivers, minimizing total delivery timeinfoq.com. Uber Eats found that switching from greedy to global matching (using predicted pickup times) reduced total travel timeinfoq.com. In practice, the system can do a rolling horizon: run a matching algorithm every few seconds. This requires fast estimation of *time-to-restaurant* for each driver (using real-time maps/traffic data) and *time-to-deliver*. We likely incorporate ML models to predict preparation and travel times.

Both strategies require location data: the system constantly ingests driver GPS updates (1–10Hz). This data feeds both dispatch and tracking. If scale is huge, the dispatch module may run geo-index queries (e.g. find K nearest drivers) in a spatial index.

Circuit-breakers: If no drivers are available in the area, the system can either queue the order (with user waiting or cancel option) or broaden search radius. Surge pricing or incentives might be triggered for future orders.

Real-Time Tracking & Notifications

Real-time updates are crucial. Mobile clients and restaurant dashboards must reflect order status and driver movement live. We **push** updates rather than polling. For example, Talabat's infrastructure uses **WebSockets** (and Redis Pub/Sub) to stream driver locations and order events to clientsmiracuves.com. Uber similarly eliminated aggressive polling (80% of their app's calls were polling requestsuber.com) by switching to a push-based system (gRPC/WebSocket).

We adopt a similar model: when status changes (restaurant confirms order, driver picks up food, etc.), services publish an event. A dedicated **WebSocket server** (or gRPC streaming endpoint) pushes the event to all subscribed clients (customer and restaurant)miracuves.com. Driver apps also maintain a live connection to receive "new dispatch" messages. Under the hood, we may use a scalable pub/sub layer (Kafka/ZMQ/Redis) to fan out messages.

For example, as reported in Talabat's tech stack, "WebSockets for instant notifications (order updates, rider location)… ensure that users see their order tracking screen updating live with every step"miracuves.com. This meets our p95 <1s update goal for most cases. Clients must handle reconnection/backoff if a socket drops (especially on mobile networks).

Scalability & Sharding

To handle massive scale, we shard by **geography** and **function**. Each region (city) can be served by its own cluster of services and database shardaerospike.com. This geo-sharding lowers latency (drivers/customers hit the nearest data center) and meets data-locality needsaerospike.com. However, one region may become a "hot" shard (e.g. a city with 10× users). We must monitor shard load: if a region becomes overloaded, we can add more instances or split further (e.g. multiple DB shards in that region)aerospike.com.

Within each region, services can scale horizontally (stateless services behind load balancers, autoscaled containers/VMs). Databases can be replicated (e.g. primary-replica) for high throughput. For the **Dispatch** component specifically, we use *microservices* so it can scale independentlydev.to. As one design note: partitioning drivers and orders by sub-region (like city neighborhoods) can reduce computation for matchingdev.to.

We also scale the real-time messaging layer: e.g. run Kafka clusters partitioned by region so event streams are local. Redis instances (for pub/sub) should be sharded. WebSocket servers can be sharded by user region or use sticky sessions.

Caching/CDN: We cache menus and static assets. A Content Delivery Network serves images, and possibly caches the JSON menu responses (as Uber notes, menu payloads can be largedeveloper.uber.com). Caching reduces load on the Menu Service. We also use in-memory caches (Redis) for hot lookup (e.g. restaurant search results, available drivers).

Rate Limiting & Backoff

All APIs pass through a gateway enforcing per-user or per-API rate limits (to prevent abuse or runaway clients). For example, limit each client to X requests/sec for menu or order APIs. If a client exceeds, return HTTP 429 and require exponential backoff. Similarly, internal calls to third-party APIs (like Maps) should be throttled (e.g. max X requests/minute) and implement retry with backoff on 429/5xx. This prevents our system from being throttled by external services.

SLOs and Error Budgets

We define **SLIs/SLOs** to ensure reliability. Example SLIs: request latency (e.g. API calls under 200ms), WebSocket message latency, error rate, and location-update latency. An SLO might be "95% of status updates delivered to clients within 1 second". This implies a 5% *error* *budget*docs.aws.amazon.com (5% of updates can be late or dropped). We track these metrics continuously (using tools like Prometheus/Grafana) and set alerts when burn rates are high. Adopting a Google-SRE-style error budget approachdocs.aws.amazon.com balances feature rollout vs stability.

For example, if our SLO is 99% availability per month, the error budget is 1% of time. If exceeded, feature deployments may halt until reliability is restoreddocs.aws.amazon.com. Similarly, we may have SLOs on database replication lag (e.g. <100ms) and on third-party API uptime. Regular chaos testing (e.g. simulate a region failure) helps validate availability goals.

Constraints & Assumptions

- **Third-party Maps:** We rely on Google/Mapbox for geocoding and routing. This means routing/ETA accuracy depends on their service; we may cache route results briefly. We assume eventual consistency for ETA updates: as new traffic data arrives, we can update ETAs gradually.
- **Eventual Consistency:** Order ETA and status are eventually consistent. For example, if a driver’s location report is delayed, the displayed ETA might be slightly outdated but will correct within seconds.
- **Device Connectivity:** We assume mobile clients have intermittent connectivity; they will automatically reconnect sockets and request missed updates if needed.
- **Load Patterns:** Peak demand is assumed (e.g. lunchtime spikes). Systems must handle sudden 5–10× load increases (autoscaling groups, overprovisioning, and CDNs help).
- **Geography:** We might simulate a real scenario (e.g. launching in two large metro areas with ~1000 restaurants, 2000 drivers, and scaling up to 10k concurrent deliveries within a year).

Security and Privacy

All communication is over TLS/HTTPS. User data (personal info, payment tokens) is encrypted at rest and in transit. We comply with payment security (PCI DSS) by not storing raw card data (using tokenization or external payment services). Authentication/authorization should use OAuth/JWT with scopes (e.g. customers vs restaurants vs drivers have different permissions). We audit actions (order changes, payment events) for integrity. Privacy laws (GDPR, CCPA) are respected by data minimization and user consent for location sharing.

Summary

This design emphasizes *availability, performance, scalability, and maintainability*. By using microservices and event-driven architecturemiracuves.comdev.to, we isolate domains (orders, dispatch, tracking) so each can scale or be upgraded independently. Geo-sharding and caching/CDN ensure low-latency operations worldwideaerospike.com. Real-time updates via WebSockets and Kafka enable live trackingmiracuves.comdev.to while keeping backend load manageable. We back our choices with known industry practices (Uber, DoorDash, Talabat) and enforce reliability through SLOsdocs.aws.amazon.com.

Overall, the system architecture blends proven patterns (pub/sub messaging, distributed databases, scalable APIs) to meet the demanding requirements of a 24/7 food-delivery platform serving tens of thousands of concurrent orders.

Sources: Industry blogs and tech articles on food-delivery and real-time systems

Diagrams:

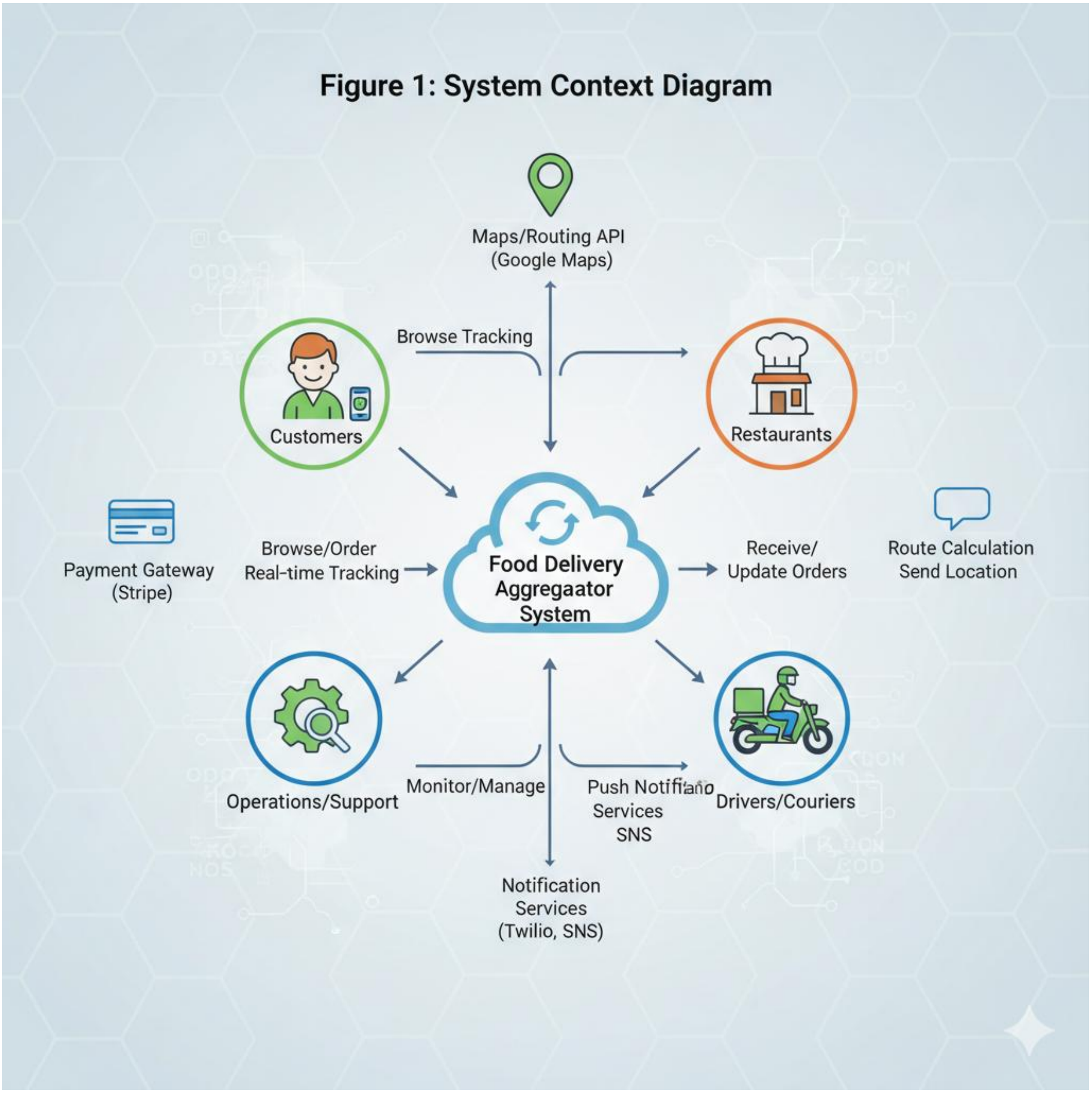
Figure 1: System Context Diagram
Maps/Routing API (Google Maps)
Browse Tracking
Customers
Restaurants
Payment Gateway (Stripe)
Browse/Order Real-time Tracking
Food Delivery Aggregaator System
Receive/ Update Orders
Route Calculation Send Location
Operations/Support
Monitor/Manage
Push Notifiano Services SNS
Drivers/Couriers
Notification Services (Twilio, SNS)

Figure 1: This diagram illustrates the high-level ecosystem of the food delivery aggregator. It shows the primary actors—Customers, Restaurants, Drivers/Couriers, and Operations—and their interactions with the central system, as well as the system's reliance on external services like Payment Gateways, Maps/Routing APIs, and Notification Services.

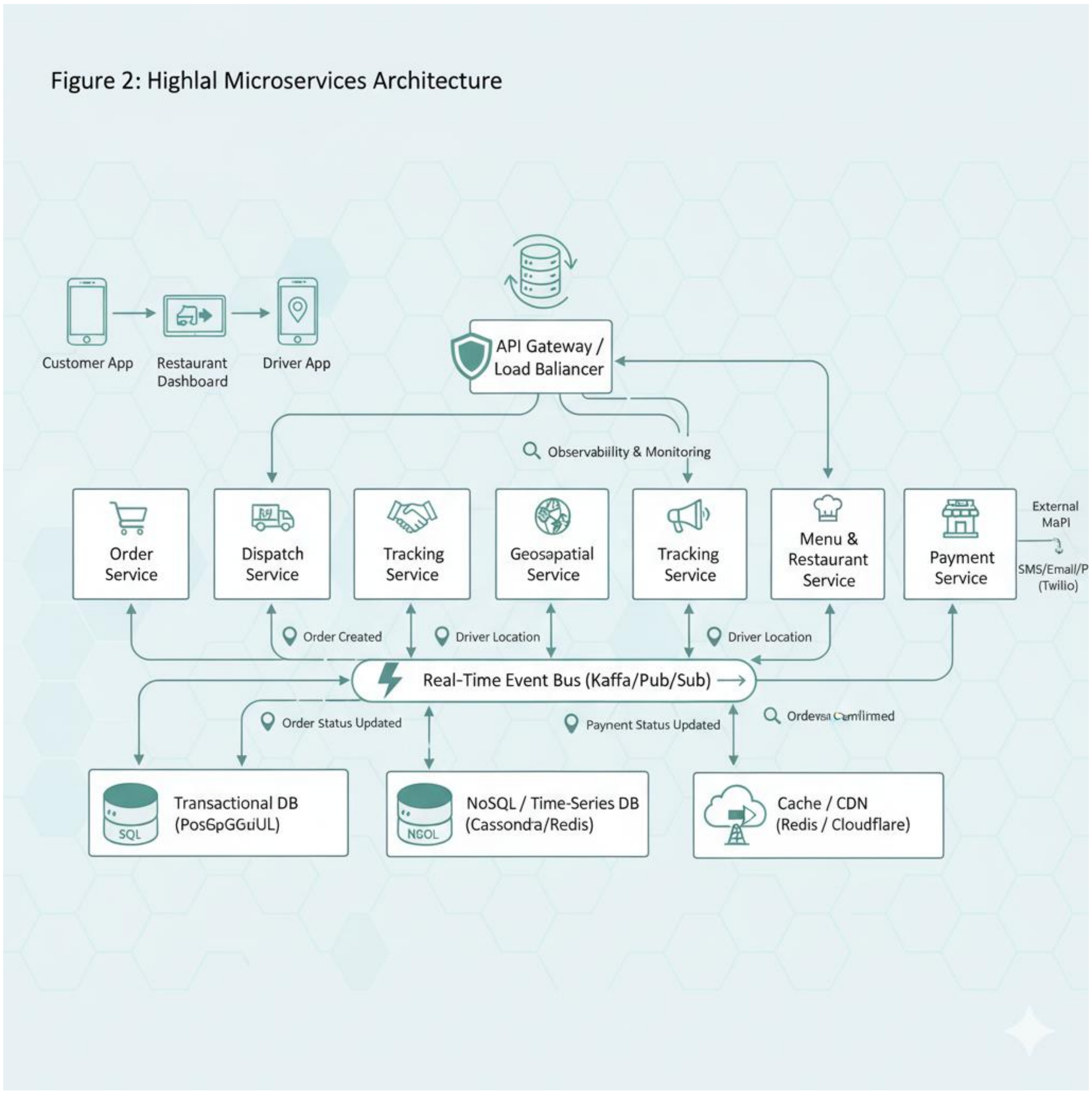

Figure 2: This image outlines the internal microservices architecture. It shows client applications (Customer, Restaurant, Driver) connecting via an API Gateway to a suite of specialized backend services (Order, Dispatch, Tracking, etc.). These services communicate through a real-time event bus (like Kafka) and utilize a polyglot persistence approach with SQL, NoSQL, and Cache/CDN databases.

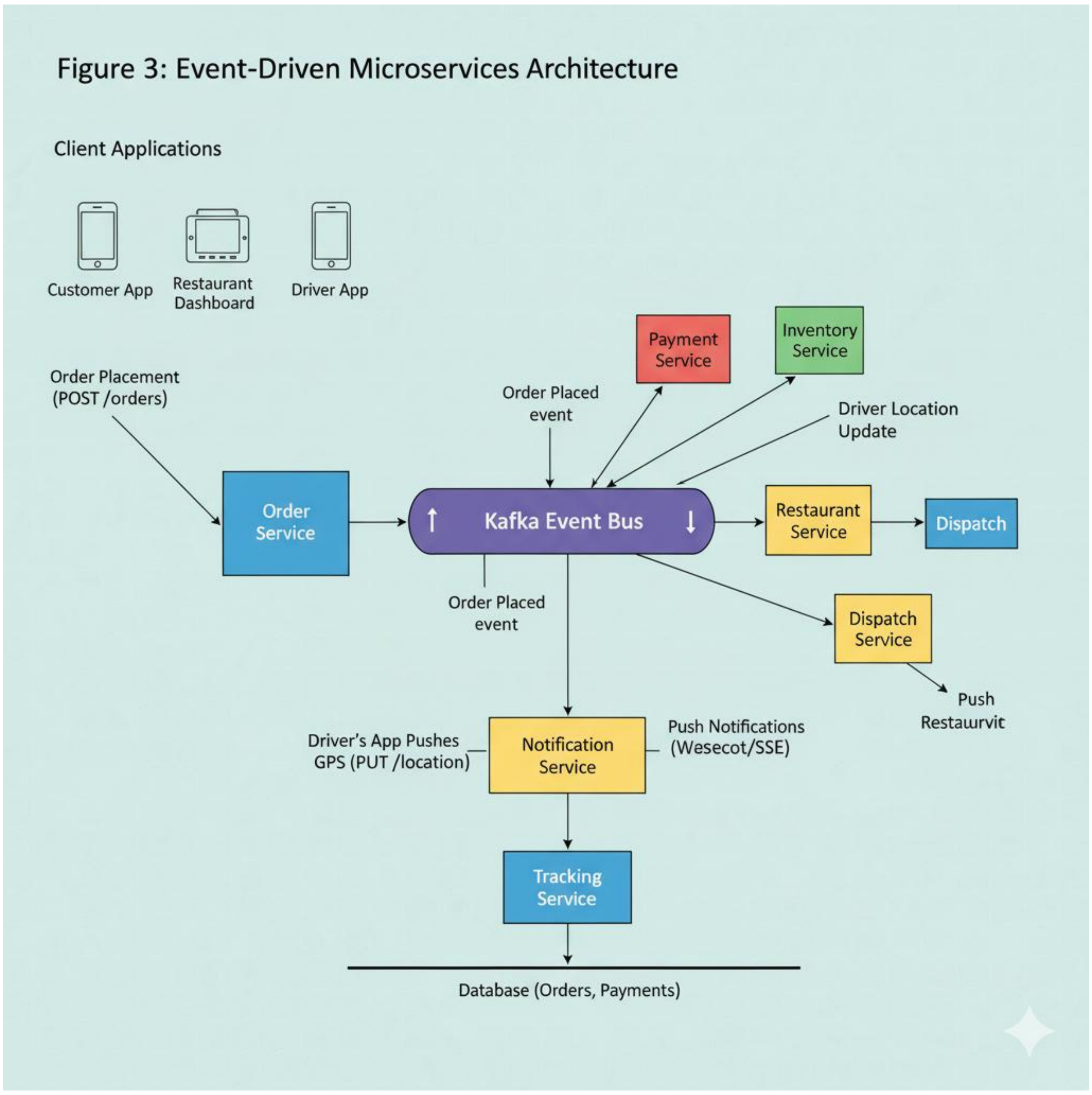

Figure 3: This diagram focuses on the event-driven flow, demonstrating how services are decoupled. When a customer places an order, the Order Service publishes an "Order Placed" event to a Kafka Event Bus. Multiple other services (Payment, Inventory, Restaurant, Dispatch) then subscribe to this event and react in parallel, enabling asynchronous and resilient processing

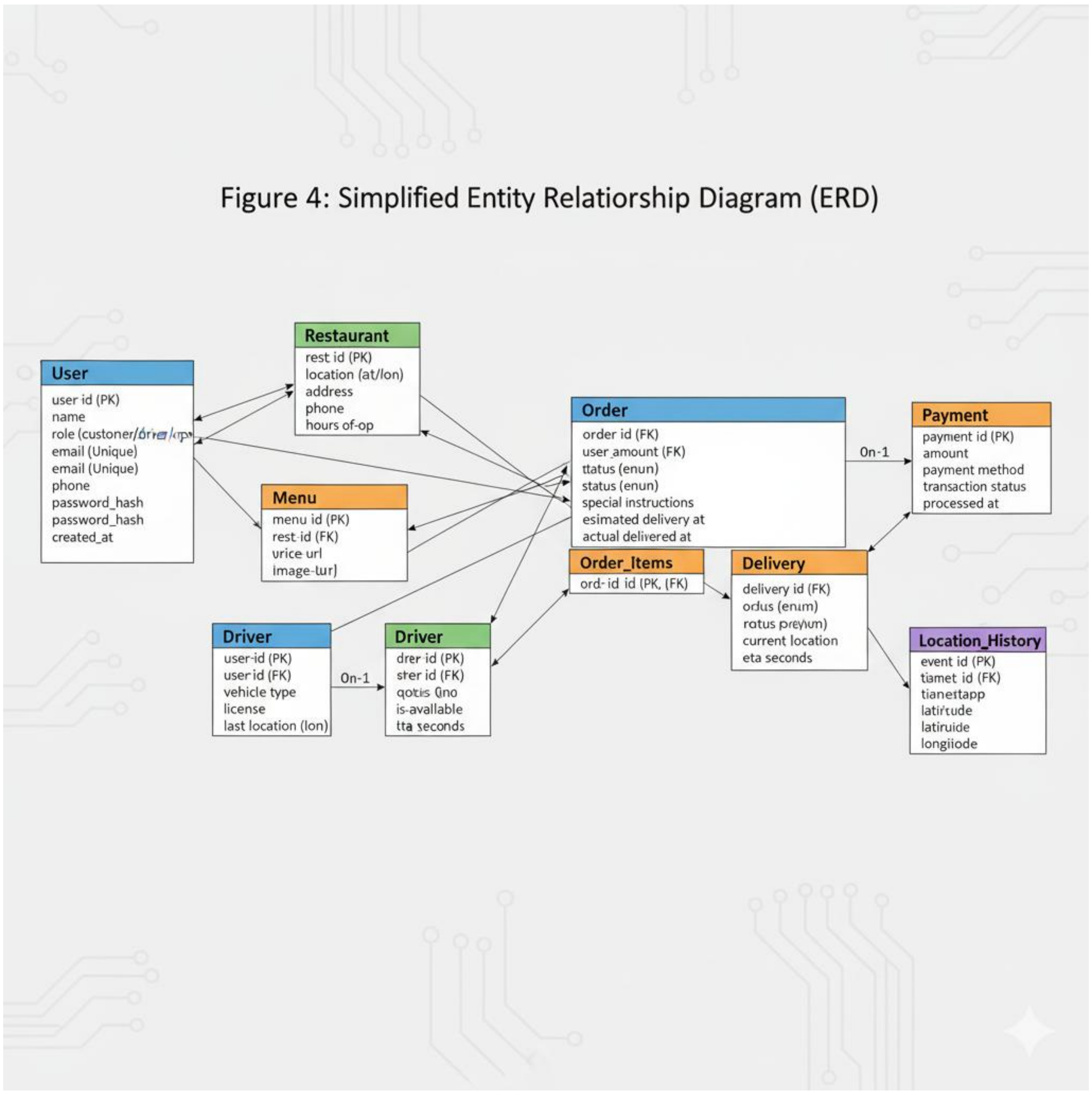

Figure 4: This ERD shows the core data model and the relationships between key database tables. It links entities like User, Restaurant, Menu, Order, Order_Items, Driver, and Delivery, providing a blueprint for the system's transactional and "source of truth" data.

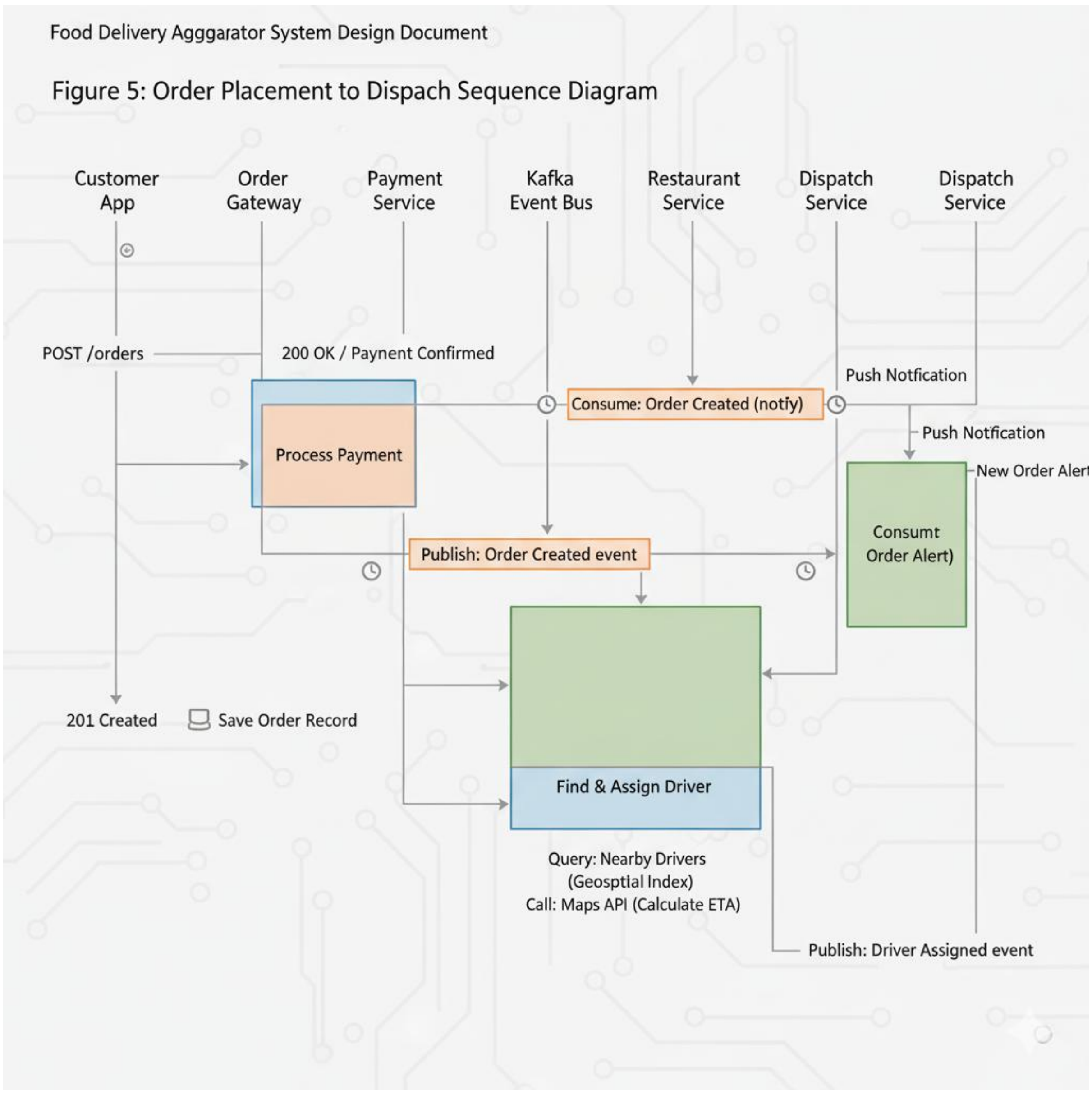
Food Delivery Agggarator System Design Document
Figure 5: Order Placement to Dispach Sequence Diagram
Customer App
Order Gateway
Payment Service
Kafka Event Bus
Restaurant Service
Dispatch Service
Dispatch Service
POST /orders
200 OK / Paynent Confirmed
Process Payment
Consume: Order Created (notify)
Push Notfication
Push Notfication
New Order Alert
Consumt Order Alert)
Publish: Order Created event
Find & Assign Driver
201 Created
Save Order Record
Query: Nearby Drivers (Geosptial Index)
Call: Maps API (Calculate ETA)
Publish: Driver Assigned event

Figure 5: This sequence diagram visualizes the step-by-step flow of a new order. It starts with the customer's API request, traces the process through the Order Gateway, Payment Service, and the publishing of an "Order Created" event. It concludes by showing how the Dispatch Service consumes this event to find and assign a nearby driver.