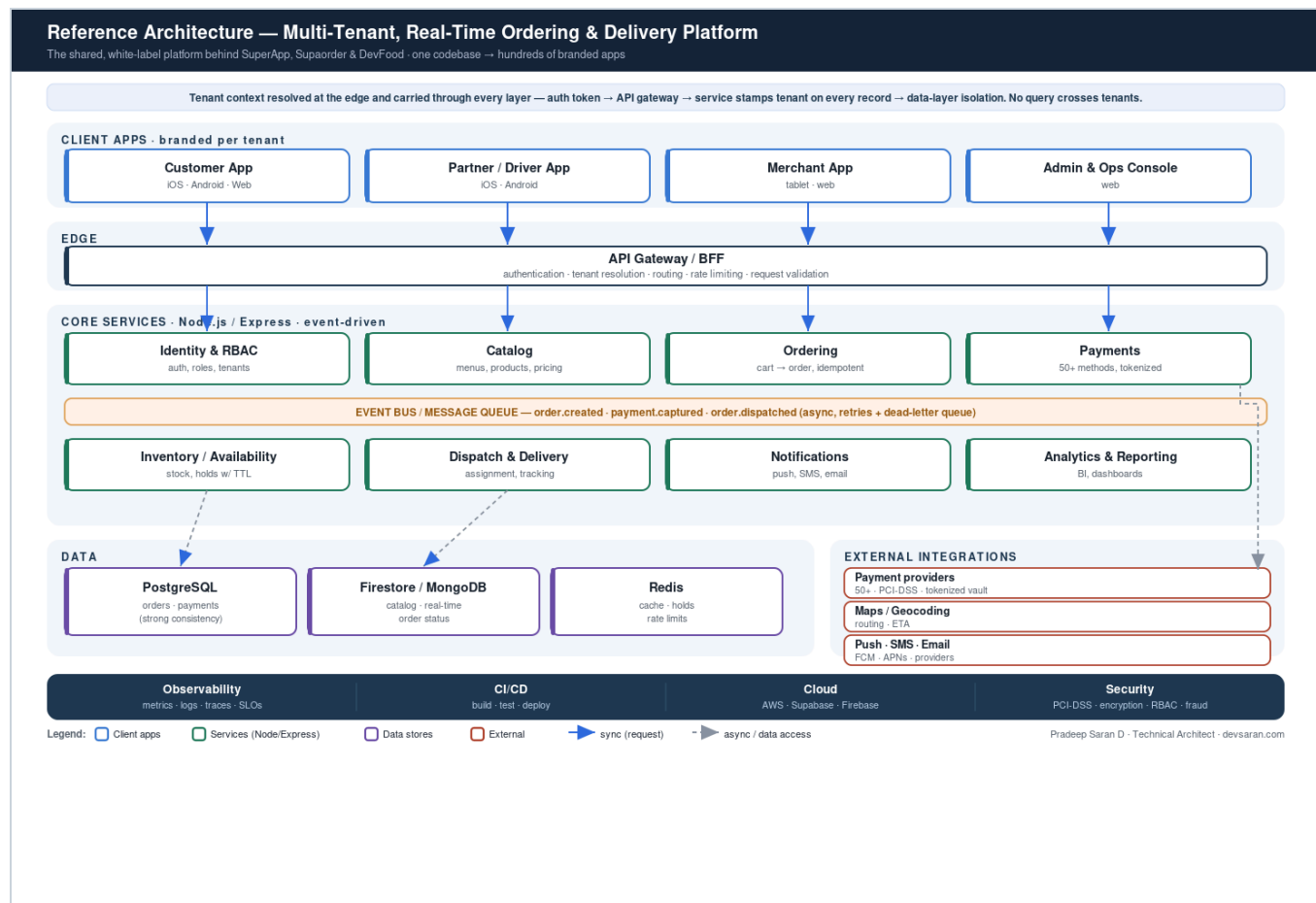


# Architecture Case Studies

Pradeep Saran D · Technical Architect · Multi-Tenant SaaS · Distributed Systems · Cloud · devssaran.com · saran@devssaran.com

Three production platforms I personally architected and operated end to end — white-label, multi-tenant, real-time ordering and delivery systems. Each case study states the context, the hard problems, the key architectural decisions and the trade-offs I weighed, and the outcome. Diagrams are C4-style container views.

## Reference Architecture — the shared platform



SuperApp, Supaorder and DevFood run on one shared, multi-tenant platform — a single backend and codebase that launches hundreds of independently branded apps.

Tenant context is resolved at the API gateway from the auth token; every service stamps the tenant on each record and isolation is enforced at the **data layer**, so no request can read or write across tenants. Services are **Node.js / Express**, communicating synchronously on the request path and asynchronously over an **event bus** for the order pipeline. Persistence is split by need: **PostgreSQL** for transactional data (orders, payments), a document/real-time store for catalog and live order status, and **Redis** for cache, inventory holds, and rate limits. The same auth, payment, observability and deployment rails are shared across all tenants and products.

## Case Study 1 — SuperApp: white-label, multi-vertical marketplace

---

**Context.** An enterprise-grade, white-label marketplace platform spanning multiple verticals (food, retail, supermarkets). It launches branded customer, partner, and business apps for iOS, Android, and web from one multi-tenant backend, and is built for 100+ countries with multi-currency billing and fully translatable interfaces.

### THE HARD PROBLEMS

- **Multi-tenancy at scale** — hundreds of brands on shared infrastructure with strict isolation and per-tenant configuration (branding, currency, catalog, commission rules).
- **A multi-vertical domain model** — food, retail, and grocery have different catalog and fulfilment shapes behind one ordering core.
- **Cross-platform delivery** — every brand wants native apps on both stores, without a per-brand rewrite.

### KEY DECISIONS & TRADE-OFFS

- **Pool (shared) multi-tenancy over silo.** Far lower cost and single-deploy operations, at the price of designing isolation into the data layer from day one (tenant claim in token → data-layer rules → tenant stamped on every record). Chose pool because tenants share workload shape; a regulated or unusually heavy tenant can be promoted to a bridge model without re-architecting.
- **Config-driven branding + Angular / Ionic / Capacitor.** One codebase ships to iOS, Android, and web, so launching a brand is a build-pipeline step, not engineering. Trade-off: heavy native UI is harder than fully-native — an acceptable cost for commerce apps and a large multiplier across hundreds of brands.
- **A generalized catalog & fulfilment model** so adding a vertical is configuration, not a new service.

### OUTCOME

One platform powering branded marketplaces across verticals and regions; adding a new branded marketplace is configuration on the same backend, auth, and payment rails. The multi-vertical model is proven in production by Green Delivery Ibiza — food, grocery, pharmacy, drinks, and yacht provisioning in one branded super app.

## Case Study 2 — Supaorder: zero-commission, real-time ordering

---

**Context.** A zero-commission, white-label restaurant ordering platform — branded iOS/Android apps plus website, real-time order management, a multi-tier loyalty engine, membership, and local payment methods across 100+ countries through PCI DSS-compliant processors. Restaurants move off third-party marketplaces and keep 100% of their margin.

### THE HARD PROBLEMS

- **One tap = one order** — networks retry, users double-tap, and payment webhooks fire twice.
- **Real-time order capture and status** across customer, kitchen, and driver.
- **Accepting many local payment methods** across markets without ever holding card data.

### KEY DECISIONS & TRADE-OFFS

- **Idempotency everywhere it matters.** An idempotency key on every order-creating request, stored before the write, so a retry returns the original result instead of creating a second order; payment callbacks are processed once, keyed on the provider's event ID. Trade-off: an extra write and a key store — non-negotiable when money is involved.
- **Real-time store + event bus for live status** (placed → accepted → preparing → out for delivery) so every client updates without polling. Strong consistency is reserved for orders and payments (PostgreSQL); status is eventually consistent, which users tolerate.
- **Payment-provider abstraction with tokenization.** The provider vaults the card and returns a token; a single interface lets the next provider or method plug in behind it, and most systems stay out of PCI DSS scope.

### OUTCOME

Restaurant brands run their own branded ordering with reliable real-time updates and broad payment coverage, keeping the commission a marketplace would otherwise take — the same engine behind Menuro's branded restaurant apps and Marry Me Mochi's direct ordering (75,000+ customers).

## Case Study 3 — DevFood: enterprise aggregation & dispatch

---

**Context.** An enterprise food-aggregation and restaurant-management platform serving restaurants and aggregators across multiple countries — and the system whose patterns seeded SuperApp and Supaorder. One aggregator built on it, Ceger Dostava, grew to the #2 food aggregator in Bosnia at 80,000+ orders/month and was acquired by Korpa (2022).

### THE HARD PROBLEMS

- **Data consistency under transactional load** — orders, inventory, and payments must stay correct during peaks.
- **Dispatch and delivery** — assigning and tracking drivers in real time.
- **High availability** — an aggregator can't go dark at the dinner peak.

### KEY DECISIONS & TRADE-OFFS

- **Saga + transactional outbox** for order-and-payment atomicity across services, so an event and its database write commit together (no lost or phantom events); optimistic locking and inventory holds with a TTL prevent oversell. Trade-off: more moving parts than a single DB transaction — necessary once a flow spans services.
- **Event-driven order pipeline with retries-and-backoff and a dead-letter queue**, so a slow downstream (a kitchen printer, a courier API) degrades gracefully instead of dropping orders.
- **Horizontal, stateless services behind a load balancer**, with read replicas and Redis caching to hold availability under peak load.

### OUTCOME

A fault-tolerant aggregation platform proven in production — including a market-#2 aggregator at 80,000+ orders/month that ran through its acquisition without re-engineering — and the architectural foundation reused across the product line.

---

These are systems I personally architected and operated. Diagrams are C4-style container views; quantitative figures are as designed-for or observed in production (market ranking, peak order volume and the acquisition are my own operating knowledge from running the platform; Menuro's restaurant count is Menuro's published figure). I'm happy to walk through the specifics in detail. — October 2026