Backend, Database & System Architecture Research

September 25, 2025

Contents

1 Context and Objectives

This project aims to design and implement the backend of a **Parking Barrier Control System** that must reliably decide whether to open, deny, or charge for entry based on license plate recognition and subscription status. Beyond this critical control path, the system must support online subscriptions and payments, and provide operator dashboards with near real-time visibility.

The solution should be reproducible for development and deployment, secure in how it handles credentials and roles, extensible as new features emerge, and cloud-ready to run on common hosting platforms.

The resulting architecture needs to balance three sometimes competing priorities: (1) robust, auditable decisions at the barrier with low latency; (2) modern, responsive operator UX; and (3) straightforward integration with third-party payment providers.

2 Candidate Technology Stack

On the **application layer**, **FastAPI (Python)** was selected as the primary framework due to its asynchronous I/O support, first-class request/response validation through **Pydantic**, and an ecosystem well-suited to building APIs rapidly without sacrificing structure. Alternative frameworks such as **Flask (Python)** or **Express/Nest (Node.js/TypeScript)** were considered. Flask's minimalism is attractive for prototypes but requires more manual scaffolding as complexity grows; Node.js options, while strong, would add language/runtime heterogeneity without clear advantage for this domain.

For **persistence**, **PostgreSQL** was chosen for its ACID guarantees, mature query planner, partial indexes, and strong support for analytical SQL (CTEs, window functions). **SQLAlchemy** provides a flexible ORM layer, while **Alembic** manages schema evolution with versioned migrations—an essential capability for a system that will iterate on data models (sessions, payments, subscriptions) over time.

To ensure reproducibility and dev/prod parity, the system is containerized with **Docker** and orchestrated with **Docker Compose** in development. This allows one-command startup of the API and database, clean environment separation, and straightforward deployment paths to containers-on-cloud in later stages.

3 Architectural Alternatives

3.1 Postgres-Only (Relational Core)

In the Postgres-only approach, all data—vehicles, sessions, subscriptions, payments, blacklists—resides in a single relational database. The API (FastAPI) exposes endpoints for cameras, operator tools, and dashboards, and every read/write is served from the single store.

Strengths:

- Transactional integrity and strong consistency.
- Auditable state changes are straightforward.
- Complex analytical queries are easily expressed in SQL.
- Schema evolution is controlled with Alembic.

Trade-offs:

- Realtime UX requires additional mechanisms (polling, WebSockets, or SSE).
- Payment integration requires careful orchestration to avoid blocking or overloading the primary database.

Summary: Postgres-only maximizes correctness and auditability but leaves realtime dashboards and native payments UX as additional engineering work.

3.2 Firebase-Only (Realtime NoSQL)

A Firebase-only design relies on Firebase Authentication, Firestore/Realtime Database, Cloud Functions, and Hosting for the entirety of the system. Payments (Stripe/PayPal) are integrated via Cloud Functions that directly update Firestore, and dashboards achieve realtime behavior out of the box by subscribing to document/collection changes.

Strengths:

- Realtime updates natively supported.
- Built-in authentication and hosting simplify development.
- Excellent developer velocity and user experience for dashboards.

Trade-offs:

- Eventual consistency and limited transactional integrity.
- No Alembic-style migration system.

- Denormalization complicates correctness and increases cost under heavy load.

Summary: Firebase-only excels at realtime dashboards and developer ergonomics but is poorly matched to low-latency, strongly consistent gate decisions with rich relational constraints.

3.3 Hybrid (Relational Core + Event-Driven Firebase)

The Hybrid design purposefully splits responsibilities:

- **Relational Core (PostgreSQL + FastAPI + SQLAlchemy/Alembic):** System of record for vehicles, sessions, subscriptions, payments, blacklists, and audit logs. The control path (license plate events and ALLOW/DENY/PAY decisions) runs exclusively against this core.
- **Event-Driven Firebase Periphery (Auth, Dashboards, Payments Relay):** Firebase Authentication handles operator logins; dashboards read a denormalized mirror in Firestore/RTDB. Payment provider webhooks land in Cloud Functions, which normalize and forward to a JWT-protected internal webhook on the API; the API performs idempotent upserts in Postgres and updates audits.

Summary: Hybrid combines relational guarantees and realtime UX but introduces more moving parts.

4 Comparative Evaluation

What the table compares? It compares three architectural options—Postgres-only, Firebase-only, and a Hybrid approach—under criteria that matter for a barrier control system: decision reliability and latency at the gate, realtime operator UX, payment handling, migration/analytics, and operational complexity.

How to read the columns?

- **Barrier reliability & integrity** — how well each option guarantees correct, auditable decisions (ACID vs. eventual consistency).
- **Decision latency** — typical p95 latency for access checks assuming proper indexing/caching.
- **Payments integration** — fit for webhooks, idempotency, and provider event normalization.
- **Realtime dashboards** — effort to provide live counters/tiles without overloading the primary store.

- **Schema evolution / Analytics** — migrations (Alembic) and ability to run rich SQL reporting.
- **Operational complexity** — number of moving parts and day-2 operations burden.

Criteria	**Postgres-Only**	**Firebase-Only**	**Hybrid (Chosen)**
Barrier reliability & integrity	High (ACID, transactions)	Medium (eventual consistency)	High (decisions on Postgres)
Decision latency	Predictable; <150 ms (indexed)	Variable; modeling overhead	Predictable; <150 ms (Postgres + cache)
Payments integration	Custom relay + idempotency	Native Cloud Functions	Functions normalize $\rightarrow$ upsert in Postgres
Realtime dashboards	WS/SSE/polling (extra infra)	Native (RTDB/Firestore)	Mirrored read-model in Firebase
Schema evolution	Alembic migrations	No native migrations	Alembic in core; Firebase for read-model
Analytics & reporting	Full SQL	BigQuery export recommended	Full SQL in core; Firebase UI
Operational complexity	Low	Low–Medium (data modeling)	Medium (two systems)

Table 1: Comparison of architectural alternatives.

Takeaways. Postgres-only and Hybrid both satisfy the reliability and latency needs of gate control. Firebase-only excels at dashboards but weakens the critical control path due to eventual consistency. Hybrid offers the best operator UX, but its added integration complexity is not essential for an MVP.

5 Final Architecture

For the MVP, a **PostgreSQL-only layered architecture** is adopted to minimize risk and complexity while meeting all functional requirements.

- **Core (authoritative):** PostgreSQL with SQLAlchemy and Alembic. All entities—vehicles, sessions, subscriptions, payments, blacklist, audit logs—reside in the relational database. Unique constraints and idempotent upserts ensure reliable handling of external events.
- **FastAPI application:** Layered structure (Routers → Services → Repositories) for maintainability and clear separation of concerns.
- **Payments integration:** Stripe/PayPal webhooks connect to FastAPI, verified and stored transactionally in Postgres with idempotent handling (provider IDs).
- **Realtime dashboards:** Implemented via SSE or WebSockets; events are emitted post-commit to keep operator UIs live without overloading the core.
- **Security & roles:** Managed in Postgres; RBAC enforced at the API layer.
- **Resilience:** Barrier decisions depend only on API + Postgres; explicit fail-mode policies and reconciliation jobs maintain correctness under upstream outages.

6 Conclusion and Rationale

The research compared Postgres-only, Firebase-only, and Hybrid designs. While Hybrid enhances realtime UX, it introduces unnecessary complexity for the MVP. Given the system's core responsibility—reliable, auditable, low-latency gate control—the **PostgreSQL-only layered architecture** with FastAPI is optimal. It ensures:

- Reliable and auditable barrier decisions,
- Straightforward schema evolution (Alembic),
- Safe, idempotent payment integration,
- Realtime dashboards via SSE/WebSockets without overloading the core.

This approach delivers a robust, maintainable MVP and provides a clear path to future enhancements (e.g., mirroring a read-model to Firebase) without compromising the integrity of the control path.