ADR-005: Monolith First

ADR-005: Monolith-First Architecture

Status: Accepted **Date:** 2026-02-21 **Deciders:** John (AI Director), Alem (CEO) **Category:** Architecture

---

Context

Drop needs to go from concept to demo-ready product as quickly as possible. The team is small (1 human CEO + AI agents), and the initial scope is well-defined: 10 screens, ~24 API endpoints, single SQLite database.

Three architecture patterns were considered:

Pattern	Deployment Complexity	Dev Speed	Scaling	Team Size Fit
Microservices	High (orchestration, service mesh, API gateway)	Slow (interface contracts, distributed testing)	Excellent	Large teams
Modular monolith	Low (single deploy unit)	Fast (shared code, simple debugging)	Good (extract later)	Small teams
Serverless functions	Medium (cold starts, state management)	Medium	Good (per-function)	Any

The current scope (7 core pages + API routes) does not justify the operational overhead of microservices. The codebase is ~24 API route files and ~19 database tables -- well within what a single deployment can handle.

```
graph TB
    subgraph monolith["Drop Monolith (Current)"]
        nextjs["Next.js 15<br/>App Router"]
        nextjs --> pages["Frontend Pages<br/>(10 screens, React 19)"]
```

```
        nextjs --> api["API Routes<br/>(24 endpoints)"]
        api --> db["SQLite / PostgreSQL<br/>(19 tables)"]
    end

    subgraph future["Future Extraction (If Needed)"]
        web["Web Frontend<br/>(Next.js)"]
        mobile_api["Mobile API<br/>(Hono v4)"]
        payment_svc["Payment Service<br/>(PISP orchestration)"]
        banking_svc["Banking Service<br/>(AISP integration)"]
        shared_db["PostgreSQL<br/>(shared or per-service)"]
    end

    monolith -->|"Extract when:<br/>- Team grows to 5+ devs<br/>- 10K+ concurrent users<br/>- Independent deploy needed"| future

    classDef current fill:#C8E6C9,stroke:#2E7D32
    classDef future_style fill:#E3F2FD,stroke:#1565C0

    class nextjs,pages,api,db current
    class web,mobile_api,payment_svc,banking_svc,shared_db future_style
```

Decision

Start as a modular monolith. Extract microservices only when scaling demands it.

The monolith is structured with clear module boundaries to make future extraction feasible:

Module	Responsibility	Files
`auth/`	Authentication (BankID OIDC, JWT, sessions)	`api/auth/*`, `lib/auth.ts`
`transactions/`	Remittance and QR payment processing	`api/transactions/*`
`merchants/`	Merchant registration and dashboard	`api/merchants/*`
`cards/`	Card management (feature-flagged)	`api/cards/*`
`compliance/`	GDPR, AML, complaints	`api/consents/*`, `api/complaints/*`, `api/user/*`
`lib/`	Shared: middleware, DB, validation, feature flags	`lib/*`

Extraction triggers (when to consider splitting):

- Team grows beyond 5 developers working on different modules simultaneously
- Single-digit millisecond response time required for specific endpoints
- Independent deployment cadence needed (e.g., payment processing updated hourly, auth monthly)
- Database contention from concurrent writes exceeds SQLite/single-PostgreSQL capacity

Consequences

Positive

- Fastest path from concept to working demo
- Simple deployment: single Docker container on AWS App Runner
- No distributed system complexity (no service discovery, circuit breakers, distributed tracing)
- Easy debugging: single process, single log stream, single database
- Shared code: middleware, validation, and DB access used consistently across all routes
- Low operational cost at current scale

Negative

- All modules must deploy together (no independent deployment)
- Single point of failure (if the monolith crashes, everything is down)
- Scaling is all-or-nothing (cannot scale payment processing independently)
- Module boundaries are convention-based, not enforced by process isolation

Risks

- **Boundary erosion:** Without process isolation, module boundaries may erode over time. Mitigation: clear file organization, code review, and this ADR as a reminder.
- **Scaling ceiling:** Monolith will hit throughput limits at high concurrency. Mitigation: PostgreSQL handles concurrent writes; App Runner auto-scales horizontally.

References

- Architecture Document -- Section 1.2: Architecture Style
- System Context (C4 Level 1) -- High-level system view
- Container Diagram (C4 Level 2) -- Internal container structure
- ADR-012: AWS App Runner -- Deployment target for monolith
- Martin Fowler, "MonolithFirst" (2015)

Revision #5
Created 2026-02-21 05:58:58 UTC by John
Updated 2026-05-23 10:56:40 UTC by John