Cloud Audit: App Cloud Readiness

Drop Application Cloud-Readiness Audit

MC Task: #1443 **Date:** 2026-02-19 **Auditor:** software-arch (CloudForge team) **Application:** Drop Fintech Payment App (Next.js 15 + SQLite/PostgreSQL dual-driver)

> **NOTE (2026-03-03):** This audit was performed on 2026-02-19. ADR-014 (2026-03-03) removed SQLite and the dual-driver architecture. Drop now uses PostgreSQL 16 exclusively in all environments. SQLite concerns noted in this audit are resolved. The `better-sqlite3` dependency has been removed.

---

1. Twelve-Factor Compliance

I. Codebase — PASS

- **Evidence:** Single Git repository at `/Users/makinja/ALAI/products/Drop/`
- `.github/workflows/ci.yml` triggers on `main` and `develop` branches
- One codebase tracked in revision control, multiple deploys (staging via Fly.io, production via Docker Compose)

II. Dependencies — PASS

- **Evidence:** `package.json:1-55` declares all dependencies explicitly
- `npm ci` used in CI (`ci.yml:36`) and Dockerfile (`Dockerfile:6`) for deterministic installs
- `package-lock.json` referenced in Dockerfile COPY (`Dockerfile:5`) and CI cache (`ci.yml:32`)
- Native modules (better-sqlite3) handled via `apk add python3 make g++` in Dockerfile

III. Config — PASS

- **Evidence:** `.env.example:1-87` documents all env vars with clear groupings
- `env.ts:1-45` validates critical vars at startup, crashes if missing in production
- `fly.toml:16-20` injects env vars at runtime
- `docker-compose.production.yml:7-8` uses `${JWT_SECRET:?}` required substitution
- `db.ts:9` — database driver selected via `DATABASE_URL` env var
- `db.ts:26-30` — SQLite path varies by environment (Vercel `/tmp`, Docker `/app/data`, local `./data`)
- Feature flags externalized as `NEXT_PUBLIC_FF_*` env vars (`Dockerfile:19-26`)
- **Minor concern:** `NEXT_PUBLIC_*` vars are baked into the build at compile time (Next.js limitation), requiring rebuild for changes. This is inherent to Next.js, not a code deficiency.

IV. Backing Services — PASS

- **Evidence:** `db.ts:9-22` — database treated as attached resource via `DATABASE_URL`
- PostgreSQL connection string is a single env var; switching databases requires zero code changes
- `docker-compose.production.yml:17-35` — PostgreSQL is a separate service with its own health check
- BankID, PISP, AISP, Stripe, Sumsub — all configured via env vars (`.env.example:19-53`)

V. Build, Release, Run — PASS

- **Evidence:** Dockerfile uses 3-stage build (deps → builder → runner)
- `Dockerfile:1-6` — Stage 1: dependency installation
- `Dockerfile:9-37` — Stage 2: application build with `next build`
- `Dockerfile:39-64` — Stage 3: minimal production runner
- `next.config.ts:8` — `output: "standalone"` generates self-contained deployment
- CI builds Docker image tagged with commit SHA (`ci.yml:63`)
- Build-time vs runtime config cleanly separated (ARG for build, ENV for runtime)

VI. Processes — PARTIAL

- **Evidence:** Application runs as a single `node server.js` process (`Dockerfile:64`)
- **SQLite concern:** When running with SQLite (no `DATABASE_URL`), the process is stateful — data lives on local filesystem at `/app/data/drop.db`. This works on Fly.io with mounted volumes (`fly.toml:36-38`) but violates share-nothing for horizontal scaling.
- **PostgreSQL mode:** Fully stateless — `pg.Pool` connects to external database (`db.ts:17-22`). Multiple processes can run concurrently.
- **Rate limiting:** `rate_limits` table in the database (`middleware.ts:15-43`), which works for single-instance but has race conditions under horizontal scale with SQLite.

- **Assessment:** PARTIAL because SQLite mode is actively used (Fly.io staging). In PostgreSQL mode this would be PASS.

VII. Port Binding — PASS

- **Evidence:** `Dockerfile:61-62` — `EXPOSE 3000`, `ENV PORT=3000`, `ENV HOSTNAME="0.0.0.0"`
- `fly.toml:23` — `internal_port = 3000`
- `docker-compose.production.yml:5` — `ports: "3000:3000"`
- Self-contained via Next.js standalone server, no external HTTP server dependency.

VIII. Concurrency — PARTIAL

- **Evidence:** Node.js single-threaded event loop handles concurrent requests via async I/O
- `db.ts:16-22` — PostgreSQL connection pool (pg.Pool) supports concurrent queries
- `fly.toml:25-27` — `auto_stop_machines`/`auto_start_machines` enables horizontal scaling
- **Limitation:** No explicit worker process types. Background work (e.g., exchange rate refresh) runs inline. No separate queue workers. For a fintech app, transaction processing should eventually be separated into dedicated worker processes.
- **Limitation:** SQLite mode limits to single process (WAL mode allows concurrent reads but single writer).

IX. Disposability — PASS

- **Evidence:** Process starts fast — Next.js standalone is ~500ms cold start
- `db.ts:719-789` — `initDb()` is idempotent with `_initialized` guard; safe for restarts
- Schema uses `CREATE TABLE IF NOT EXISTS` — safe for repeated initialization
- `fly.toml:25-27` — machines auto-stop/start, confirming disposability design
- Graceful shutdown handled by Node.js default SIGTERM behavior
- PostgreSQL pool (`pg.Pool`) handles connection cleanup on process exit

X. Dev/Prod Parity — PASS

- **Evidence:** `db.ts:9-13` — dual-driver architecture (SQLite for dev, PostgreSQL for prod) with unified async API
- `db.ts:47-63` — SQL compatibility layer translates SQLite idioms to PostgreSQL (placeholder conversion, `INSERT OR IGNORE` → `ON CONFLICT DO NOTHING`, `datetime('now')` → `CURRENT_TIMESTAMP`)
- `db.ts:204-460` (SQLITE_SCHEMA) and `db.ts:462-690` (PG_SCHEMA) — parallel schemas maintained in sync
- `migrations/0001_initial-schema.ts` — node-pg-migrate for PostgreSQL schema versioning
- Docker Compose production config (`docker-compose.production.yml`) mirrors production topology locally

- **Minor gap:** SQLite schema is maintained inline in `db.ts` while PostgreSQL uses proper migrations (`node-pg-migrate`). Schema drift is possible if one is updated without the other.

XI. Logs — PARTIAL

- **Evidence:** Health endpoint uses `createLogger()` (`health/route.ts:16`)
- `middleware.ts:82-84` — error tracking via `trackError()` and Sentry integration
- `.env.example:62-74` — Sentry DSN configurable via env vars
- **Concern:** No structured logging to stdout visible in the codebase. Next.js default logging goes to stdout which is good for containers, but there's no consistent structured logging format (JSON lines) that cloud log aggregators can parse efficiently. `console.error` is used in places (`middleware.ts:83`).

XII. Admin Processes — PASS

- **Evidence:** `package.json:12-14` — migration scripts: `migrate:up`, `migrate:down`, `migrate:create` via `node-pg-migrate`
- `db.ts:735-774` — programmatic ALTER TABLE migrations for schema evolution
- Seed data controlled by `SEED_DEMO` env var and `isDemoMode()` check — admin data seeding decoupled from main app
- No one-off scripts embedded in application startup (seeding only runs when database is empty)

---

2. Containerization Quality

Multi-Stage Build — EXCELLENT

- **3-stage Dockerfile** (`Dockerfile:1-64`):
  - Stage 1 (`deps`): `node:22-alpine`, installs native build tools, runs `npm ci`
  - Stage 2 (`builder`): Copies deps, builds Next.js app
  - Stage 3 (`runner`): Minimal alpine, copies only standalone output + static assets

Image Size

- Base: `node:22-alpine` (minimal, ~180MB base)
- **Issue:** Stage 3 installs `python3 make g++` (`Dockerfile:42`) for better-sqlite3 native module rebuild. This adds ~200MB to the production image unnecessarily if running in PostgreSQL mode. These build tools are a security and size concern in production.
- **Recommendation:** Either pre-build better-sqlite3 in stage 2 and copy the binary, or conditionally exclude it when PostgreSQL is the target.

Security

- Non-root user: `nextjs:nodejs` (UID/GID 1001) created and used (`Dockerfile:48-49, 58`)
- `NEXT_TELEMETRY_DISABLED=1` set (`Dockerfile:14, 46`)
- Data directory owned by non-root user (`Dockerfile:56`)
- CI runs Trivy vulnerability scanner on built image (`ci.yml:67-73`) with HIGH/CRITICAL severity gate
- SARIF results uploaded to GitHub Security tab (`ci.yml:85-89`)

Layer Caching

- Dependencies cached in separate stage (`Dockerfile:5-6` — `COPY package.json package-lock.json*` before source)
- Source code copy happens in stage 2 after deps, enabling Docker layer cache for unchanged dependencies
- **Good practice:** Build args for feature flags allow cache invalidation only when flags change

Missing

- No `.dockerignore` verified (could copy unnecessary files like `.git`, `node_modules` into build context)
- No image tagging strategy beyond CI SHA tag

---

3. Database Portability

Dual-Driver Architecture — STRONG

- **Implementation:** `db.ts:9-13` — Runtime driver selection via `DATABASE_URL` presence
- **Unified API:** `query()`, `getOne()`, `run()`, `transaction()` — all async, both drivers (`db.ts:67-199`)
- **Type exports:** `DbClient` interface (`db.ts:136-140`) for transaction context

SQL Translation Layer

SQLite Idiom	PostgreSQL Translation	Location
`?` placeholders	`$1, $2, ...`	`db.ts:47-50`
`INSERT OR IGNORE INTO`	`INSERT INTO ... ON CONFLICT DO NOTHING`	`db.ts:56, 104-118`

SQLite Idiom	PostgreSQL Translation	Location
`INSERT OR REPLACE INTO`	`INSERT INTO ... ON CONFLICT (col) DO UPDATE SET`	`db.ts:58, 120-134`
`datetime('now')`	`CURRENT_TIMESTAMP`	`db.ts:60`
`INTEGER PRIMARY KEY AUTOINCREMENT`	`SERIAL PRIMARY KEY`	`db.ts:278 vs 530`
`hex(randomblob(32))`	`encode(gen_random_bytes(32), 'hex')`	`db.ts:248 vs 504`

Transaction Support

- PostgreSQL: `BEGIN/COMMIT/ROLLBACK` with `pgClient.connect()` and proper release in `finally` block (`db.ts:142-173`)
- SQLite: `db.exec("BEGIN/COMMIT/ROLLBACK")` wrapper (`db.ts:174-198`)
- Error handling: Both paths catch and rollback on failure

Migrations

- **node-pg-migrate** for PostgreSQL (`package.json:12-14`, `migrations/0001_initial-schema.ts`)
- Proper `up()` and `down()` functions with ordered table creation/deletion
- SQLite uses inline schema with `CREATE TABLE IF NOT EXISTS` + ALTER TABLE try/catch migrations (`db.ts:756-774`)
- **Risk:** Two parallel schema definitions (SQLITE_SCHEMA and PG_SCHEMA in `db.ts` + node-pg-migrate files) could drift. No automated parity check exists.

Indexes

- 22 indexes defined for both drivers (identical set)
- Partial indexes supported: `idx_users_national_id WHERE national_id_hash IS NOT NULL`, `idx_tx_idempotency WHERE idempotency_key IS NOT NULL`

---

4. Config Externalization

Environment Variables

Category	Variables	Source
Core	`JWT_SECRET`, `JWT_EXPIRY`, `NODE_ENV`	`.env.example:12-14`
Database	`DATABASE_URL`	`db.ts:9`

Category	Variables	Source
Service Mode	`NEXT_PUBLIC_SERVICE_MODE`, `DROP_MODE`	`.env.example:8`
Auth (BankID)	`BANKID_CLIENT_ID/SECRET/URLS`, `BANKID_MOCK`	`.env.example:19-29`
Payments	`PISP_API_URL/KEY`, `AISP_API_URL/KEY`	`.env.example:32-40`
Cards	`STRIPE_SECRET_KEY`, `STRIPE_PUBLISHABLE_KEY`	`.env.example:43-47`
KYC	`SUMSUB_APP_TOKEN`, `SUMSUB_SECRET_KEY`	`.env.example:50-52`
Monitoring	`SENTRY_DSN`, `SENTRY_TRACES_SAMPLE_RATE`	`.env.example:63-74`
Feature Flags	8x `NEXT_PUBLIC_FF_*`	`.env.example:77-87`
Exchange	`EXCHANGE_RATE_API_KEY/URL`	`.env.example:55-59`

Secrets Management

- `env.ts:14-45` validates critical vars at production startup
- `Dockerfile:15` — `JWT_SECRET=build-phase-placeholder` (safe build-time placeholder)
- `env.ts:21-25` — Skip validation during build phase (detects `NEXT_PHASE` or placeholder)
- `env.ts:36-38` — Rejects known dev placeholder in production runtime
- `docker-compose.production.yml:7` — `${JWT_SECRET:?}` required substitution (fails if missing)
- **No hardcoded secrets** found in source code

Feature Flags

- 8 client-side feature flags via `NEXT_PUBLIC_FF_*` env vars
- Defaults to `false` (safe) for all card-related features
- `NEXT_PUBLIC_FF_NOTIFICATIONS=true` and `NEXT_PUBLIC_FF_MERCHANT_DASHBOARD=true` as defaults
- Build-time injection for client code (`Dockerfile:19-35`), runtime for server code

---

5. CI/CD Quality

Pipeline Structure (`ci.yml`)

```
lint-test (parallel)          docker-scan (sequential, needs lint-test)
  -- npm ci                     -- docker build
  -- eslint                     -- Trivy scan (table, exit-code=1 on HIGH/CRITICAL)
  -- tsc --noEmit               -- Trivy SARIF -> GitHub Security
```

```
-- vitest run
-- npm audit (production)
```

Reproducibility

- Pinned Node.js version: `NODE_VERSION: "22"` (`ci.yml:15`)
- `npm ci` for deterministic installs (`ci.yml:36`)
- Dependency caching via `actions/setup-node` with `cache-dependency-path` (`ci.yml:30-32`)
- Docker image tagged with commit SHA (`ci.yml:63`)

Security Scanning

- **npm audit:** Production dependencies, HIGH level, continue-on-error (`ci.yml:48-49`)
- **Trivy:** Container vulnerability scan, blocks on HIGH/CRITICAL unfixed vulns (`ci.yml:67-73`)
- **SARIF:** Results uploaded to GitHub Security tab (`ci.yml:85-89`)
- **Permissions:** Minimal — `contents: read`, `security-events: write` (`ci.yml:11-12`)

Testing

- `vitest run` in CI (`ci.yml:44`)
- Unit test framework configured (`package.json:10-11`)
- Coverage tool available: `@vitest/coverage-v8` (`package.json:43`)
- **Missing:** No coverage threshold enforcement in CI
- **Missing:** No E2E/integration tests in CI pipeline (Playwright is in devDependencies but not wired into CI)

Deployment

- **Fly.io staging** configured (`fly.toml`) with health checks, auto-scaling, volume mounts
- **Docker Compose production** (`docker-compose.production.yml`) for self-hosted deployments
- **Missing:** No automated deployment step in CI (manual `fly deploy` or similar)
- **Missing:** No environment promotion pipeline (develop -> staging -> production)

---

6. Overall Score and Top 5 Improvements

Overall Cloud-Readiness Score: 7.5 / 10

The application demonstrates strong cloud-native fundamentals:

- Excellent dual-driver database abstraction
- Proper multi-stage Dockerfile with security hardening
- Configuration fully externalized via environment variables
- Comprehensive CI with security scanning (Trivy + npm audit)
- Health endpoint with real database connectivity check

Top 5 Improvements (Priority Order)

1. Eliminate Build Tools from Production Image (HIGH)

- **File:** `Dockerfile:42`
- **Issue:** `python3 make g++` in production stage adds ~200MB and attack surface
- **Fix:** Pre-compile better-sqlite3 in builder stage, copy only the `.node` binary. Or use a conditional build that excludes better-sqlite3 entirely when targeting PostgreSQL.

2. Add Structured Logging (HIGH)

- **Files:** Throughout — `console.error` used in `middleware.ts:83`, health endpoint has `createLogger()` but no consistent format
- **Issue:** Cloud log aggregators (CloudWatch, Datadog, ELK) need structured JSON logs. Current mix of console.log/error and ad-hoc logger makes log parsing unreliable.
- **Fix:** Adopt `pino` or similar JSON logger, output to stdout in `{ level, message, timestamp, requestId }` format.

3. Add CI Coverage Enforcement and E2E Tests (MEDIUM)

- **File:** `ci.yml` — no coverage gate, no Playwright CI step
- **Issue:** `@vitest/coverage-v8` and `@playwright/test` are in devDeps but not enforced in CI
- **Fix:** Add `--coverage --coverage.thresholds.lines=80` to vitest. Add Playwright E2E job with containerized app.

4. Automate Schema Parity Check (MEDIUM)

- **File:** `db.ts:204-690` — two parallel schema definitions (SQLite + PostgreSQL)
- **Issue:** Manual sync between SQLITE_SCHEMA, PG_SCHEMA, and node-pg-migrate files. Drift will cause runtime errors that only surface in specific deployment targets.
- **Fix:** Write a CI check that extracts table/column definitions from both schemas and compares. Or generate both schemas from a single source of truth.

5. Add Deployment Pipeline and Environment Promotion (MEDIUM)

- **File:** `ci.yml` — CI only, no CD
- **Issue:** No automated deployment from CI. Fly.io deploy is manual. No staging -> production promotion gate.

- **Fix:** Add `fly deploy` step on `develop` push (staging) and manual approval gate for `main` (production). Add smoke test after deploy. Consider GitHub Environments for approval workflows.

Honorable Mentions

- SQLite mode limits horizontal scaling — document clearly when to switch to PostgreSQL
- Rate limiting via database has race conditions under concurrent writes (consider Redis for high-throughput)
- No readiness probe separate from liveness (health endpoint serves both)
- No graceful shutdown handler (SIGTERM -> drain connections -> exit)
- `playwright-core` in production dependencies (`package.json:27`) — should be devDependencies only

---

Appendix: File Reference

File	Purpose
`src/drop-app/src/lib/db.ts`	Dual-driver database abstraction (SQLite + PostgreSQL)
`src/drop-app/Dockerfile`	3-stage multi-stage build
`src/drop-app/.env.example`	Environment variable documentation (87 lines)
`src/drop-app/fly.toml`	Fly.io deployment config (Stockholm region)
`src/drop-app/docker-compose.production.yml`	Self-hosted production config
`src/drop-app/package.json`	Dependencies and scripts
`.github/workflows/ci.yml`	CI pipeline (lint, test, type-check, Trivy)
`src/drop-app/migrations/0001_initial-schema.ts`	PostgreSQL migration (node-pg-migrate)
`src/drop-app/next.config.ts`	Next.js config (standalone output, security headers)
`src/drop-app/src/middleware.ts`	Edge middleware (CSRF, CSP nonce)
`src/drop-app/src/lib/middleware.ts`	Server middleware (rate limiting, auth, validation, audit)
`src/drop-app/src/app/api/health/route.ts`	Health endpoint (real DB check)
`src/drop-app/src/lib/env.ts`	Environment validation at startup

Revision #8
Created 2026-02-23 11:28:57 UTC by John
Updated 2026-06-28 20:02:23 UTC by John