SaaS Marketplace — Full Project Audit

Project:[client URL redacted]
Audit Date:Q1 2026
Stage: Proof of Concept

---

1. Project Overview

The Client Platform is an independent SaaS application marketplace, built as a full-stack Next.js application. The project includes:

- **Storefront** for buyers
- **Vendor Portal** for publishing and managing applications
- **Back-Office Panel** for moderation, finance, and certification
- **Quality Review System** with security/code/UX scoring
- **Payment System** via payment processor Connect with settlement payouts

Stack: Next.js + React + TypeScript + managed PostgreSQL + Stripe + UI framework

Codebase Size: ~12,700 lines of TypeScript/TSX, 16 test files (~4,200 lines of tests)

---

2. Code Quality — Score: 7/10

Strengths

- **Good architecture** — clean separation into route groups: `(storefront)`, `(portal)`, `(admin)`, `(buyer)`
- **Separation of concerns** — database clients (client/server/admin), Stripe, Email, job queue service extracted into `lib/`
- **Consistent error handling** — all API routes wrapped in try-catch with proper HTTP status codes (400, 401, 403, 404, 500)
- **TypeScript strict mode** enabled
- **No TODO/FIXME/HACK** — codebase is clean of technical debt markers
- **Good test coverage** — 16 files covering models, API, features, and e2e workflows
- **Loading/Error states in UI** — skeletons, disabled buttons, error messages

Issues

Issue	Severity	Details
37+ `as any` type casts	Medium	database queries cast to `any`, undermining type safety
ESLint `no-explicit-any (linter rule)` disabled	Medium	`linter config` — rule turned off globally
Code duplication	Low	`formatPrice()` duplicated in `app/[slug]/page.tsx` and `app-card.tsx`; category icon mapping duplicated
No Prettier	Low	No code formatting configuration
No component tests	Medium	No unit tests for React components
No E2E UI tests	Medium	No Cypress/Playwright for UI scenarios
No `error boundary files` files	Low	Missing error boundaries in App Router

---

3. Hosting — Score: 6/10

Current State

- **Hosting: cloud platform** (based on `version control config` with `.platform/`, README linking to hosting platform)
- **Database: managed PostgreSQL** — managed PostgreSQL with Auth, RLS, real-time
- **Payments: payment processor** — webhooks via serverless functions
- **Email: transactional email service** — transactional emails
- **Background Jobs: job queue service** — serverless job queue

Hosting Issues

Issue	Impact
No monitoring	No Sentry, DataDog, or any error tracking. Errors are lost in `console.error()`

No logging	23 `console.log/error` calls without structured logging. platform logs rotate quickly in production
Empty `framework config file`	No security headers, image domains, or redirects configured
No CDN for static assets	No image optimization setup (logos, screenshots)
No staging environment	CI deploys only to main, no preview environments in pipeline
Webhook reliability	payment webhooks processed in serverless functions without a retry queue. Payment processing may fail silently

---

4. CI/CD — Score: 5/10

Current Pipeline (`CI pipeline config`)

Push/PR to main → dependency install → lint → type-check step → build step

What's Missing

Gap	Criticality
Tests are NOT run in CI	**Critical** — 16 test files exist but `test runner` is not called in the pipeline
No `dependency vulnerability scan`	High — vulnerabilities are not checked automatically
No secret scanning	High — no `gitleaks` or equivalent
No CD (deployment)	Medium — deployment is likely via platform auto-deploy, but undocumented
No version matrix	Low — tests run only on Node 20, ubuntu-latest
Placeholder env vars	Medium — build uses `sk_test_placeholder`, hiding runtime errors

---

5. Security — Score: 5/10

CRITICAL Vulnerabilities

5.1 Open Redirect in Auth Callback

File: `auth/callback handler`

```
const redirect = searchParams.get("redirect") || "/";
return NextResponse.redirect(`${origin}${redirect}`);
```

An attacker can pass `?redirect=//evil.com` — the user will be redirected to a malicious site after authentication. **This is a classic phishing attack vector.**

5.2 Open Redirect in Sign In

File: `auth/signin page`

```
const redirect = searchParams.get("redirect") || "/";
router.push(redirect); // No validation
```

Same issue on the client side.

5.3 Vulnerable Dependencies

4 HIGH severity vulnerabilities:
- next 14.2.35 → DoS via Image Optimizer, HTTP request smuggling, unbounded disk cache
- glob 10.x → Command injection

HIGH Vulnerabilities

5.4 XSS in Email Templates

File: `api/applications endpoint`

```
html: `<h2>Thanks for applying, ${body.applicant_name}!</h2>
      <p>application for <strong>${body.product_name}</strong></p>`
```

User input (`applicant_name`, `product_name`) is inserted into HTML **without escaping**. An attacker can inject `<script>` tags or phishing links into the email.

5.5 Token in URL (Information Disclosure)

File: `api/token-validation endpoint`

```
const token = searchParams.get("token"); // GET parameter
```

Access token passed via query string → appears in server logs, browser history, and Referer headers.

5.6 No Rate Limiting

No rate limiting on any API endpoint:

- `/api/applications` — can spam applications
- `/api/token-validation` — can brute-force tokens
- `/api/vendor/apps` — can create unlimited apps

MEDIUM Vulnerabilities

5.7 Missing Security Headers

No CSP, X-Frame-Options, X-Content-Type-Options, or Referrer-Policy. The application is vulnerable to clickjacking and content sniffing.

5.8 No URL Validation

`demo_url`, `app_url`, `repo_url`, `video_url` are accepted without format validation — `javascript:` protocol or data URIs are possible.

5.9 Stripe Webhook — No Business Logic

File: `api/webhooks/payments endpoint`

All event handlers are stubs (`console.log`). Signature is verified, but **no payment is actually processed**:

```
case "payment.succeeded":
  console.log("Payment succeeded:", event.data.object.id); // ← That's it
  break;
```

5.10 Error Information Leakage

Webhook signature failure returns the error text to the client:

```
return NextResponse.json({ error: message }, { status: 400 });
// message from internal exception
```

What's Implemented WELL

- SSR auth middleware — correct route protection
- RBAC (Role-Based Access Control) — developer, admin, finance
- Ownership checks — developers can only modify their own apps
- Parameterized queries via ORM — no SQL injection
- Secrets are not hardcoded, `NEXT_PUBLIC_` only for public keys
- Payment webhook signature verification works
- Service Role Key used only server-side

---

6. Logic Gaps

Issue	Description
Application not linked to user	`/api/applications` uses `createAdminClient()` and does not bind the application to an auth user. Anyone can submit an application under someone else's name
Job Queue — stubs only	`job queue client module` creates a client, but all functions are placeholders. Background tasks (settlements, payouts) are non-functional
Email from `noreply@client-domain.com`	Domain `client-domain.com` vs `client-domain.net` — emails may not be delivered if the sending domain is not configured in the email service
No webhook idempotency	Stripe may retry webhooks, but there is no duplicate check. Potential double-processing of payments
No payment processor Connect account verification	`vendor_account.status_changed` is only logged. No logic to verify the developer's Connect account status

---

7. Recommendations for Production Readiness

Phase 1 — Critical Fixes (Week 1-2)

1. **Fix Open Redirect:**

```
// auth/callback/route.ts
const redirect = searchParams.get("redirect") || "/";
const safeRedirect = redirect.startsWith("/") && !redirect.startsWith("//")
  ? redirect
  : "/";
```

2. **Update Next.js** to 16.x (or at minimum 15.x with security patches)

3. **Escape HTML in email templates** — use React Email components (already in dependencies) instead of template literals

4. **Switch token-validation to POST** with the token in the request body

5. **Add `test runner` to CI** — tests are already written, just not being run

Phase 2 — Security Hardening (Week 2-3)

6. **Add Security Headers** in middleware or `framework config file`:

```
// framework config file
const nextConfig = {
  async headers() {
    return [{
      source: '/(.*)',
      headers: [
        { key: 'X-Frame-Options', value: 'DENY' },
        { key: 'X-Content-Type-Options', value: 'nosniff' },
        { key: 'Referrer-Policy', value: 'strict-origin-when-cross-origin' },
        { key: 'Content-Security-Policy', value: "default-src 'self'; ..." },
      ],
    }];
  },
};
```

7. **Add Rate Limiting** — via edge middleware or `edge rate limiter`

8. **URL Validation** — validate protocol (only `https://`)

9. **Add `dependency vulnerability scan` to CI**

10. **Set up Sentry** for error tracking

Phase 3 — Business Logic (Week 3-5)

11. **Implement payment webhook handlers** — currently stubs. Real processing needed:
    - `payment.succeeded` → create record in `transactions`
    - `subscription.payment_succeeded` → extend subscription
    - `subscription.cancelled` → deactivate access
    - `payment.disputed` → freeze funds
    - `vendor_account.status_changed` → update Connect status
12. **Implement job queue functions** — settlements, payouts, certification workflows
13. **Add idempotency** for webhook processing (check `payment_transaction_id` for duplicates)
14. **Link applications to auth user** or add CAPTCHA

Phase 4 — Production Infrastructure (Week 5-7)

15. **Structured logging** — replace `console.log` with Pino/Winston with log levels
16. **Monitoring and alerting** — Sentry + platform analytics + uptime monitoring
17. **Staging environment** — separate database instance + deployment preview
18. **Database backups** — configure Point-in-Time Recovery
19. **E2E tests** — add Playwright for critical user flows (signup → apply → checkout)
20. **Remove `as any`** — type managed PostgreSQL queries via generated types

Phase 5 — Hardening (Week 7-8)

21. **WAF** — cloud WAF or Cloudflare
22. **Penetration testing** — before public launch
23. **GDPR/Privacy** — privacy policy, cookie consent
24. **Documentation** — API docs, incident runbook
25. **Disaster recovery plan** — recovery procedures

---

8. Production Readiness Summary

Category	Score	Status
Code	7/10	Good architecture, needs type improvements
Tests	6/10	Good coverage, but not run in CI and no UI tests
CI/CD	5/10	Basic pipeline, critically missing tests and audit
Security	5/10	Has auth/RBAC, but 2 critical vulnerabilities + missing headers
Hosting	6/10	Cloud hosting + managed DB — good choice, no monitoring
Business Logic	4/10	Payment webhooks and job queue are stubs, payments not processed
Overall Readiness	**~5.5/10**	**Proof of Concept — not production-ready**

Verdict: The project has a **solid architectural foundation**, but requires 6-8 weeks of work before production. The most critical items are: fixing security vulnerabilities (Open Redirect, XSS in email), implementing payment logic (payment webhooks), and setting up monitoring.

SaaS Marketplace — Scalability Review

Date: 2026-03-18 **Platform:** SaaS Application Marketplace **Stack:** Next.js + managed PostgreSQL + Payment processor + job queue

---

1. Database (PostgreSQL)

Issues:

- **Monolithic schema** — all 17 tables in a single database with no sharding. As `transactions`, `installs`, and `reviews` grow, query performance will degrade
- **Full-text search on PostgreSQL** — `pg_trgm` + GIN indexes work well up to ~1M records; beyond that, Elasticsearch / Meilisearch is needed

- **RLS on every query** — Row Level Security adds overhead to every SELECT. Under high load this becomes significant
- **No connection pooling** configured (platform defaults are limited)

Recommendations:

- Add table partitioning for `transactions` and `installs` (by date)
- Extract search into a dedicated service as the catalog grows
- Configure connection pooler / connection pooling

---

2. Application Architecture

Issues:

- **Next.js monolith** — storefront, vendor portal, admin, and buyer are all in a single deployment. Public storefront traffic impacts the back-office panel
- **No caching layer** — no Redis / Memcached detected. Every request hits the database directly
- **API routes in Next.js** — serverless functions suffer from cold starts and execution time limits
- **No rate limiting** on API endpoints

Recommendations:

- Add **Redis** for caching catalog data, profiles, and categories
- Configure **ISR / SSG** for catalog pages (they change infrequently)
- Add rate limiting (via middleware or edge functions)
- Long-term: split storefront and portal into separate deployments

---

3. Payment System

Issues:

- **payment webhook** — a single endpoint processes all events synchronously. During payment spikes, the queue backs up
- **Settlements** — monthly developer payouts run as batch jobs via the job queue, but there is no retry strategy for large batches

Recommendations:

- The job queue service already provides queuing — good. But a dead-letter queue should be added for failed webhooks
- Break settlement processing into smaller chunks (per developer)

---

4. Files and Media

Issues:

- Screenshots, logos, and icons are stored as URLs in the database — unclear if a CDN is in place
- No image optimization beyond what Next.js provides built-in

Recommendations:

- Use object storage + CDN for static assets
- Optimize images via `next/image` with an external loader

---

5. CI/CD and Deployment

Issues:

- CI only runs lint + type-check + build — **tests are not executed in CI**
- No staging environment
- No monitoring or alerting

Recommendations:

- Add tests to the CI pipeline
- Set up monitoring (Sentry, Datadog, or database observability)
- Add a staging environment

---

6. Horizontal Scaling

Component	Current State	Ceiling	Action Required
Frontend (Next.js)	serverless (cloud)	~10K RPS	CDN + edge caching

Database (PostgreSQL)	Single instance	~5K QPS	Read replicas, partitioning
Search	pg_trgm	~1M records	Elasticsearch / Meilisearch
Payments	Payment processor + job queue	High	Already scales well
Auth	Auth service	~10K DAU	Auth service scales, but needs monitoring

---

7. Priorities

Priority	Action	Impact
1	**Caching** (Redis + ISR)	Fastest performance win
2	**Monitoring** (Sentry / Datadog)	Cannot scale intentionally without metrics
3	**Database read replicas**	Reduces read load on primary
4	**Extract search** (Elasticsearch)	Required when catalog exceeds ~1M records
5	**Service separation**	Justified at significant team/traffic growth

---

Conclusion

The current architecture is **well-suited for MVP and early growth** (up to ~10K users, ~1K apps in the catalog). The immediate focus should be on caching and monitoring. Splitting into microservices is only justified when traffic and team size grow significantly.