Migration Analysis Report

Next.js | tRPC | Drizzle ORM | Lucia Auth

Technical Feasibility Analysis

Ticket System Project

February 2026

Prepared for Technical Review

CONFIDENTIAL

Table of Contents

1. Current Architecture Summary

Technology Stack

Layer	Current Technology	Version
Frontend	React (SPA) + Vite	React 18.3, Vite 5.2
Routing (FE)	react-router-dom	v6
Server State	TanStack React Query	v5
Backend	Express.js	v4.18
ORM	Prisma	v5.20
Database	PostgreSQL	-
Auth	Clerk (external SaaS)	@clerk/express 1.0
Styling	Tailwind CSS	v3.4
Email	SendGrid	@sendgrid/mail 8.1
AI	Anthropic Claude SDK	v0.71
File Uploads	Multer (local disk)	v1.4
Jobs	node-cron	v4.2

Codebase Size

Metric	Count
Backend TypeScript files	39
Backend lines of code	~13,870
Frontend TypeScript/TSX files	44
Frontend lines of code	~21,500
Total lines of code	~35,370
Prisma models	27
Prisma migrations	17
Express route files	26
API endpoint groups	26+ modules

Key Features in Scope

- Ticket CRUD with filtering, pagination, sorting, merging, bulk operations
- Role-based access control (USER / AGENT / ADMIN) with 2FA enforcement
- Rich text comments with @mentions and HTML sanitization
- Email inbound/outbound via SendGrid webhooks + customizable templates

- AI-powered chat widget, ticket summaries, solution suggestions (Claude)
- Dynamic form builder with reusable field library
- File attachments with upload, download, and inline preview
- Agent session tracking, time tracking, analytics with charts
- Notification system (in-app bell + browser push notifications)
- External REST API with key management and per-key form restrictions
- Database import/export and Zendesk import tool
- Cron-based automation (auto-close, auto-solve, reminders, attachment cleanup)
- Customer feedback/satisfaction system with email requests
- Macros (agent reply templates) with categories and ordering
- Full dark mode support throughout the application

2. Proposed Stack Analysis

2.1 Next.js (replacing Express.js + React SPA + Vite)

Benefits

- Unified framework: frontend + backend in one codebase
- Server-side rendering (SSR), static generation (SSG), React Server Components
- File-based routing (replaces react-router-dom configuration)
- API routes / Route Handlers (replaces Express route definitions)
- Built-in middleware system, image optimization, font optimization

Migration Impact — HIGH

Area	Effort	Notes
44 frontend files	HIGH	Must be refactored to Next.js app directory. All react-router-dom navigation must change.
Vite config	LOW	Replaced by next.config.js. Path aliases, proxy, allowed hosts reconfigured.
SPA behavior	HIGH	Fundamentally changes rendering model from client-side to server-side.
26 Express route files	HIGH	Must become Route Handlers or tRPC procedures. Different API surface.
Middleware	HIGH	Next.js middleware runs at edge. Helmet, Morgan, rate limiting, Multer incompatible.
File uploads (Multer)	MEDIUM	Multer is Express-specific. Must use formidable, busboy, or server actions.
node-cron jobs	HIGH	Cannot run inside Next.js. Need external job runner (separate process or BullMQ).
SendGrid webhooks	MEDIUM	Need raw body parsing. Different from Express approach.
Static file serving	MEDIUM	Express serves /uploads dynamically. Next.js public/ is build-time only.

CRITICAL: The cron jobs (auto-close, auto-solve, pending reminders, attachment cleanup, backlog snapshots, AI cache refresh) run in the Express process. Next.js does not support long-running background processes. A separate worker process is required regardless.

2.2 tRPC (replacing REST API + Axios + manual types)

Benefits

- End-to-end type safety: procedures callable from frontend with full TypeScript types
- Eliminates manual API client code (replaces the 670+ line api.ts file)
- No need for fetch/axios — tRPC generates the client automatically
- Input validation via Zod schemas
- Integrates with TanStack React Query (already in use)

Migration Impact — MEDIUM-HIGH

Area	Effort	Notes
26 route files !' tRPC routers	**HIGH**	~200+ endpoints to rewrite as tRPC procedures.
api.ts (670+ lines)	POSITIVE	Entirely replaced by tRPC client. Net code reduction.
types/index.ts (314 lines)	POSITIVE	Partially replaced by inferred types from backend.
All frontend components	**MEDIUM**	Every useQuery/useMutation call signature changes.
File uploads	**HIGH**	tRPC does not handle multipart natively. Needs REST fallback.
External API (/api/v1)	N/A	Must remain REST. Third parties cannot use tRPC.
Webhooks (Clerk, SendGrid)	N/A	Must remain REST. Webhooks are external HTTP POSTs.
Public endpoints	N/A	Must remain REST. No tRPC client for unauthenticated pages.

tRPC cannot replace REST for: webhooks, external APIs, file uploads, and public endpoints. The result is a hybrid tRPC+REST architecture, which adds complexity rather than reducing it.

2.3 Drizzle ORM (replacing Prisma)

Benefits

- Lighter weight, faster cold starts (no binary query engine)
- SQL-like query syntax (closer to raw SQL, more predictable)
- Schema defined in TypeScript (vs Prisma's custom DSL)
- Better suited for edge runtimes and serverless deployments

Migration Impact — HIGH

Area	Effort	Notes
schema.prisma (710 lines, 27 models)	**HIGH**	Full rewrite to Drizzle TypeScript schema. Every model, relation, enum, index.
17 existing migrations	**HIGH**	Cannot be reused. Drizzle uses own migration format. New baseline needed.
All DB queries (26 routes + 3 services)	**VERY HIGH**	Every prisma.* call must be rewritten. Fundamentally different syntax.
Complex queries (filter, paginate, include)	**HIGH**	Prisma include/select/where API does not translate 1:1.
Relation loading	**MEDIUM**	Prisma include: { requester: true } !' Drizzle with or manual joins.
prisma migrate workflow	**MEDIUM**	Replaced by drizzle-kit. Different commands and tooling.
Prisma Studio	**LOW**	Lost. Drizzle Studio exists but differs in functionality.

The current schema has 27 models with complex relations (tickets !' comments !' attachments !' mentions, forms !' fields, email threads, etc.). Rewriting all queries is the single largest piece of work. Every bug introduced here is a production data integrity risk.

2.4 Lucia Auth (replacing Clerk)

Benefits

- Self-hosted authentication (no SaaS dependency or vendor lock-in)
- No per-user pricing (Clerk charges per MAU after free tier)
- Full control over authentication flow and user data
- Session-based auth with database storage

Migration Impact — VERY HIGH

Area	Effort	Notes
Clerk webhooks (user sync)	HIGH	Eliminated, but user registration must be built from scratch.
2FA system	VERY HIGH	Must build TOTP from scratch: enrollment, verification, recovery codes, grace periods.
Auth middleware (267 lines)	HIGH	requireAuth, requireRole, requireTwoFactor — all Clerk-specific, full rewrite.
Frontend auth components	HIGH	Clerk SignIn, SignUp, UserButton, SignedIn/Out replaced with custom forms.
TwoFactorGuard component	HIGH	Must be rebuilt to query custom 2FA status.
SecuritySettings component	HIGH	Currently uses Clerk 2FA management UI.
Token management	HIGH	From Clerk JWT to Lucia session cookies. Different paradigm.
Password reset / email verify	MEDIUM	Must be built. Clerk handles these automatically today.

CRITICAL: Lucia Auth has been deprecated/archived by its maintainer as of 2025. The author recommends implementing session concepts directly rather than using the library. There is NO actively maintained library to migrate to — you would be adopting an unmaintained dependency.

Replacing Clerk with Lucia/custom auth means building the following from scratch:

- Registration form with email verification flow
- Login form with password hashing (bcrypt/argon2)
- Session management (create, validate, refresh, revoke)
- TOTP 2FA enrollment, verification, and recovery codes
- Password reset flow with secure email tokens
- CSRF protection for all state-changing requests
- Account lockout and brute force protection

3. Effort Estimation

Based on thorough analysis of the codebase, the following estimates assume one experienced full-stack developer working full-time. For a team of 2, reduce by approximately 40%.

Component	Estimated Effort	Risk Level
Next.js migration (Express + React SPA !' Next.js)	3–4 weeks	HIGH
tRPC migration (REST API + Axios !' tRPC)	2–3 weeks	MEDIUM
Drizzle migration (Prisma !' Drizzle)	3–4 weeks	HIGH
Auth migration (Clerk !' Lucia/custom)	4–6 weeks	VERY HIGH
Testing & QA (full regression of all features)	2–3 weeks	HIGH
Data migration (existing production database)	1 week	MEDIUM
Cron job separation (background workers)	1 week	MEDIUM
Total estimated	16–22 weeks (4–5.5 months)	HIGH

4. Risk Assessment

Critical Risks

& Production downtime during migration
The system is live and handling support tickets. A full rewrite requires a cutover plan with potential data loss if not handled carefully.

& Authentication regression
Authentication bugs are the highest-severity class of bug. Building custom auth from scratch introduces risk of session hijacking, broken 2FA, or access control bypass.

& Data integrity
Rewriting 27 models and all queries means every data path must be re-validated. One wrong JOIN or missing WHERE

clause could expose data across users.

& Feature parity gap
During migration, new feature development freezes. Any bugs found in production must be fixed in both old and new codebases simultaneously.

& Lucia Auth is deprecated
Adopting a deprecated library is a non-starter. Building fully custom session auth requires significant security expertise.

Moderate Risks

%¶ Cron jobs architecture
Next.js cannot run persistent background processes. Need a separate worker service, which partially negates the "unified codebase" benefit.

%¶ Hybrid API surface
Webhooks, external API, file uploads, and public endpoints must remain REST, creating a tRPC+REST split that adds complexity.

%¶ Team knowledge ramp-up
If the current team is proficient in Express/ Prisma/Clerk, switching to entirely new tooling has a significant learning curve.

5. What Problems Does This Migration Actually Solve?

Before investing 4–5 months of development time, it is important to ask what concrete pain points exist today and whether the proposed migration actually addresses them:

Claimed Benefit	Reality Check
"Express is old"	Express is stable, battle-tested, and the most used Node.js framework. v5 is now released. It is mature, not legacy.
"Next.js is modern"	True, but it optimizes for SSR/SSG use cases. A ticket system is an authenticated dashboard — SSR adds complexity without clear UX benefit.
"tRPC gives type safety"	Real benefit. But the current codebase already has typed API functions in api.ts and types/index.ts. The gap is not massive.
"Drizzle is lighter"	True for cold starts. But the app runs on a PM2 persistent process — cold start time is irrelevant. Prisma's DX is superior for this project size.
"Self-hosted auth saves money"	Clerk free tier covers 10K MAUs. A support system likely has far fewer users. Custom auth costs more in dev time than Clerk pricing.
"Unified codebase"	Partially true, but you lose independent backend scaling and still need a separate worker process for cron jobs.

6. Recommendation

RECOMMENDATION: Do NOT proceed with a full migration.

The current stack (Express + React + Prisma + Clerk) is:

- Not outdated — all dependencies are recent versions (2024–2025)
- Well-architected — clean separation of routes, services, middleware
- Feature-rich — 27 models, 26 route files, AI integration, email threading, 2FA
- Working in production — with real users, real data, real email flows

A full rewrite of ~35,000 lines of working code carries extreme risk for marginal architectural benefit.

Recommended Alternative: Targeted Improvements

If specific pain points exist, they should be addressed surgically:

Pain Point	Targeted Fix	Effort
Want type safety between FE/ BE	Add shared types package or OpenAPI spec generation from Express routes	1–2 weeks
Want to reduce Clerk cost	Evaluate Clerk pricing vs actual MAU count. If expensive, consider Auth.js with Express	1–2 weeks eval
Want better DX	Upgrade to Express v5, add Zod validation to route handlers	1 week
Want SSR for public pages	Add Next.js as frontend only (keep Express backend as API)	2–3 weeks

Pain Point	Targeted Fix	Effort
Want to modernize frontend	Migrate react-router v6 to TanStack Router, or upgrade to React 19	1–2 weeks
Want to try tRPC	Add tRPC alongside existing REST routes incrementally (Express adapter exists)	1–2 weeks pilot

If Migration Is Mandated

If the decision to migrate is non-negotiable (e.g., organizational mandate), then:

1. Do NOT migrate auth away from Clerk. The risk/reward is terrible. Clerk works natively with Next.js via @clerk/nextjs. Keep it.

2. Do NOT switch to Lucia. It is deprecated. If self-hosted auth is required, use Better Auth or Auth.js v5 instead.

3. Migrate in phases, not as a big-bang rewrite:
Phase 1: Move frontend to Next.js (keep Express backend as API)
Phase 2: Introduce tRPC for new endpoints (hybrid with existing REST)
Phase 3: Gradually migrate REST endpoints to tRPC
Phase 4: Evaluate Drizzle migration only if Prisma causes real problems

4. Keep the worker process separate for cron jobs regardless of framework choice.

7. Summary

Question	Answer
Is the current stack outdated?	No. Express 4.18, React 18, Prisma 5, Vite 5 are all current and actively maintained.
Would the new stack be better?	Marginally for DX (tRPC type safety). Worse for auth (losing Clerk). Neutral for ORM switch.
Is the migration worth the cost?	No. 4–5 months of rewrite work for a working system with no critical architectural flaws.
What is the biggest risk?	Auth migration (Clerk !' custom) and data integrity (Prisma !' Drizzle query rewrite).
Is Lucia Auth viable?	No. Deprecated as of 2025. Use Better Auth or Auth.js if self-hosted auth is truly needed.
Recommended path?	Keep current stack. Make targeted improvements. Only migrate frontend to Next.js if SSR is truly needed.

This report was generated based on a comprehensive analysis of the ticket system codebase.
All estimates are approximate and based on one experienced full-stack developer working full-time.