> TOPNOTCH.TECH

TECHNICAL DUE DILIGENCE REPORT

SYSTEM AUDIT

Purpose: To determine the [REDACTED] application's viability for commercial deployment by evaluating its structural scalability, security posture, and operational readiness.

Last Updated: April 2026

TECHNICAL DUE DILIGENCE REPORT

Target	[REDACTED] (CRM, Real Estate Vertical)
Audit Date	November 2025
Scope	System Architecture & Product Strategy
Outcome	**UNVIABLE, REBUILD REQUIRED**

1. Executive Summary

This report provides a technical assessment of [REDACTED]. The evaluation is designed to determine the application's viability for commercial deployment, its structural scalability, and the operational readiness of its primary features.

The front-end user interface of [REDACTED] presents as a completed, high-fidelity real estate CRM, built for real estate agents and similar B2C professionals. However, detailed analysis reveals a systemic divergence between the platform's visual appearance and its functional implementation. The application was initially built using [REDACTED_AI_TOOL], an AI-powered rapid prototyping tool designed for generating interactive applications. While [REDACTED_AI_TOOL] can be used to generate MVPs, and is claimed to enable production grade software, [REDACTED] hasn't demonstrated either level of maturation. Under the hood, the backend logic, data persistence, and core third-party integrations (specifically Google Calendar and [REDACTED_INTEGRATION_PARTNER]) are non-functional, mocked, or implemented via transient front-end scripts that bypass fundamental operations or database integration.

Consequently, the existing codebase is structurally unviable as a production-grade software asset. Even with the back-end limitations remedied, releasing this application to live users would result in silent data corruption, authentication failures, and service degradation. To establish a viable MVP capable of scaling to a pilot group of real estate professionals, the platform requires a comprehensive rebuild using an industry-standard framework.

2. Background

[REDACTED] was conceived as a vertical SaaS platform tailored specifically to the real estate sector, focusing on lead ingestion, automated tracking, and calendar synchronization for agents. Market validation was strong; the product generated

immediate, enthusiastic demand, with numerous real estate agents actively queuing to use the software.

Based on our interviews and review of internal communications, two non-technical executives became fully convinced the application was sufficiently functional and production-ready to accommodate day to day use. This belief was reinforced by internal user acceptance testing (UAT) that appeared to succeed. However, this testing was executed in isolated, happy-path environments, and not rigorously conducted.

The asset was not a superficial, low-effort prototype; the team invested time, capital, and cycles into its development. However, technical execution was left entirely to [REDACTED] who lacked the architectural experience to identify the platform's structural deficits. They also appear not to have the authority to provide politically insensitive feedback and disagree with the leadership's aggressive "iterate fast, fail fast, break things" mindset.

Our investigation revealed that the platform was engineered entirely "front-end down." Rather than beginning with a technical design document, database schema/models, or architectural oversight; development started with the user interface, leveraging the AI-assisted prototyping tool [REDACTED_AI_TOOL]. The visual shell was iteratively expanded, while critical backend, database, and integration requirements were either mocked or bypassed completely due to a fundamental misunderstanding of how full stack applications work.

Consistent with standard *lean validation* doctrine, UI-first prototyping is a risk-mitigation technique designed to test a high volume of unproven concepts. Constructing a visual shell does not reduce the final cash outlay required to engineer a complete system. It strictly protects capital by allowing operators to kill unviable ideas before funding the underlying server architecture and database mechanics.

Ultimately, in the case of [REDACTED], this top-down visual design strategy successfully simulated an operating application to non-technical stakeholders while producing an empty technical shell under the hood.

Note: Interviews with [REDACTED_A] and [REDACTED_B] can be found in Appendix A.

3. Scoring Matrix

The target asset was evaluated across seven technical risk vectors as an initial exploration template. Based on the findings of this analysis, subsequent evaluations were deemed unnecessary for this audit. (Highly unusual, but will become clear below.)

Scores are assigned on a 0–100 scale based on industry standards, security parameters, and codebase scalability.

Audit Vector	Score (0-100)	Risk Classification	Primary Finding
1. Codebase Quality & Structure	**15 / 100**	Critical Risk	Repetitive UI components, lack of separation of concerns, high technical debt.
2. Data Architecture & State Management	**5 / 100**	Critical Risk	No relational schema, absence of database migrations, inaccessible database, transient state storage.
3. Third-Party Integrations	**8 / 100**	Critical Risk	Mocked for [REDACTED_INTEGRATION_PARTNER] and Google Calendar; non-compliant OAuth requiring "developer account" hacks for end users to achieve rudimentary functionality.
4. Security & Threat Modeling	**0 / 100**	Hazardous Risk	Nonexistant security posture; **ACTIVE HAZARDS**.
5. Licensing & Provenance	**35 / 100**	Moderate Risk	Murky IP provenance due to generative AI usage, while low proprietary value reduces this risk.
6. Operational Cloud Spend & Scale	**20 / 100**	High Risk	Unoptimized API queries; lacking data architecture; possible high compute costs.
7. Quality Assurance (Functional Mapping)	**5 / 100**	Critical Risk	85%+ backend-dependent elements are cosmetic placeholders or broken.
OVERALL SCORE	**13 / 100**	**CRITICAL RISK**	**Asset is structurally unviable; it requires rebuild.**

4. Analysis per Risk Vector

Note: Comprehensive notes (i.e. captured network payloads, data logs, console error traces, static code analysis reports, relevant code snippet reviews, and interface screenshots) cataloged in Appendices B through F.

4.1. Codebase Quality

The [REDACTED] codebase is built using LLM-generated React components originating from the [REDACTED_AI_TOOL].

4.1.a Component Duplication: The application fails to utilize standard modular, or object-oriented coding practices. For example, the onboarding interface and the settings interface replicate identical visual elements (such as profile cards and toggle switches) as completely separate code blocks. Instead of importing a unified component with adjustable variables, the parameters, headings, and copy are hard-coded in multiple locations, introducing a high probability of desynchronization. Unofirmity across the UI is an illusion generated by LLM's tendency to produce the most repeated code across the internet.

4.1.b Feature Hiding vs. Removal: Non-functional modules (i.e. pipeline analytics, lead-tracking dashboards, and team calendar views) are handled on the front-end by DOM manipulation or routing users. There are no associated server-side features.

Investigation Methodology: Manual source code review and visual inspection of the interface; comparing UI elements and logic flows. The duplication becomes obvious when code is inspected.

4.2. Data Architecture

The application lacks a structured database schema capable of handling concurrent, multi-tenant relational operations. The data architecture was created entirely as a side effect of AI prompts.

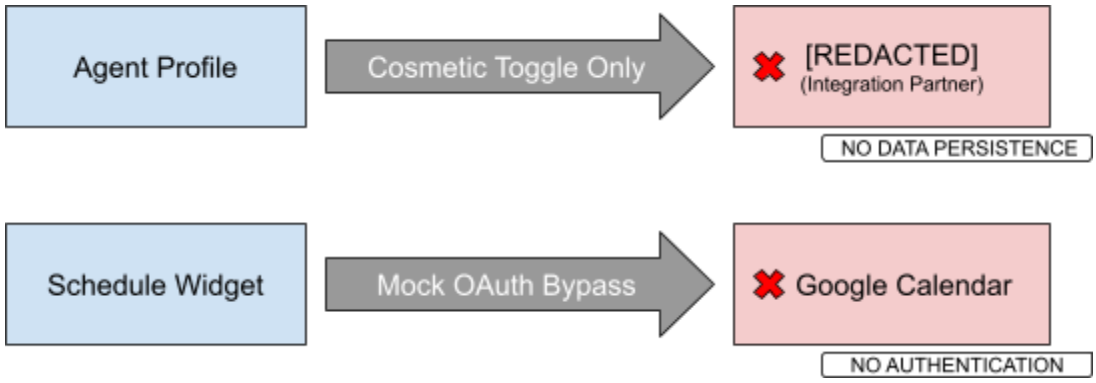
4.2.a Transient State Management: Changes do not persist reliably across user sessions; logging out and logging back in resets or corrupts local state.

4.2.b Data Validation: There are no backend schemas to validate structural inputs (e.g., verifying email syntax or timezone offsets). This introduces immediate security risks and database write-failures when malformed strings are passed.

Investigation Methodology: Manual session testing including account creation, profile modifications, and logout/login cycles to observe state retention. Code inspection.

4.3. Third-Party Integrations

The core value proposition of the CRM relies on integration with external service providers. The current implementations of these integrations are non-functional:



4.3.a [REDACTED_INTEGRATION_PARTNER] Integration: The integration toggle within the settings panel is cosmetic. It does not perform handshake validations, establish webhooks, or parse incoming lead data. The scheduled lead capture via the schedule button uses static mock arrays.

4.3.b Google Calendar Synchronization: The integration has been modified to prevent client-side application crashes during demos rather than actually performing real calendar synchronization. It does not process recurring events, manage conflicting schedules, or handle timezone differentials. It does present the appearance of successful creation of calendar events for correctly configured accounts (see below.)

Investigation Methodology: Manual testing and scheduling flow inspection.

4.4. Security & Threat Modeling

The asset exhibits an entirely compromised security posture, introducing severe data liabilities, operational risks, and immediate compliance exposure.

4.4.a Identity & Privacy Breach: The platform completely lacks a production OAuth configuration. To simulate calendar synchronization, the application requires end-users to manually enable "developer mode" settings. Bypassing standard ecosystem safety boundaries constitutes a catastrophic breach of user privacy, leading to a likely blacklist by Google and introducing high legal risk.

4.4.b Exploitable Authentication: User sessions rely on client-side local storage, rendering accounts vulnerable to hijacking.

4.4.c Secret & Credential Exposure: Third-party API keys, system endpoints, and environment secrets are hardcoded in cleartext. Few are accessible via browser session Javascript.

4.4.d Multi-Factor Architecture: Two-Factor Authentication (2FA) infrastructure is non-existent, even though the UI claims to enable this functionality. (Misleading)

Investigation Methodology: Manual source code inspection and runtime validation.

4.5. Licensing & Provenance

(Note: TopNotch.tech provides technical audits, not legal counsel. This scope of work did not include independent legal review. The following technical findings regarding asset provenance must be evaluated by counsel for legal conclusions.)

No proprietary license or copyleft elements were found. Additionally, the technical proprietary value of this codebase is near zero. The repository consists primarily of unmodified output generated by a consumer-grade AI tool ([REDACTED_AI_TOOL]). Because the code is an automated artifact rather than bespoke engineering, it presents no technical moat; a competitor can replicate it by estimating prompts or even submitting screenshots. Furthermore, current legal frameworks regarding the enforceability and copyrightability of primarily AI-generated software remain highly ambiguous.

Investigation Methodology: Code audit, plus interview confirmation that the repository is almost exclusively composed of generative AI output.

4.6. Operational Cloud Spend

While current operational costs are low, this is purely a function of current usage load. Because the queries generated by the automated framework are un-indexed and lack relational constraints, loading user data (such as multiple agents with hundreds of recurring calendar slots) would result in significant database hits and CPU usage, causing a disproportionate spike in cloud billing and potential downtimes or bottlenecks. This risk is further compounded by the tech-stack provided by [REDACTED_AI_TOOL], which naturally leads towards a serverless execution model, exposing the operation to billing spikes. Additionally, using [REDACTED_AI_TOOL] causes vendor lock-in; despite the marketing statements presented on their website, extraction of the tech to another infrastructure vendor is prohibitively expensive and functionally amounts to a re-write.

Despite the above limitations, the operational load for this specific feature set is expected to remain relatively low. This is because the platform is currently designed as an asynchronous email relay and calendar application for real estate agents. It does not require real-time communication protocols. Expanding this current architecture to support real-time communications (i.e. texting, audio comms, video meetings, etc.) would require considerable investment. (Such expansions would not be simple feature additions, as was hinted by current leadership, but would require considerable re-engineering efforts.)

Investigation Methodology: Manual review of data-fetching patterns, [REDACTED_AI_TOOL] framework inspection, production environment tests.

4.7. Quality Assurance (Functionality Mapping)

An audit of individual interactive elements was conducted to map claims against technical reality:

Interface Element	Stated Function	Current Implementation Status
Embeddable Lead Widget	Lead ingestion form for agent websites.	**Failed.** Requires copying and pasting two un-minified script blocks instead of a single script payload. Theme toggles (Dark/Light) and color-picker customizations are client-side variables that do not compile into functional CSS on third-party sites.
Notification Engine	Instant email alerts to agents and leads.	**Mocked.** Uses client-side “mailto:” triggers. No background worker queue or transactional mail services exist.
SMS/Push Notifications	Real-time text and push alerts.	**Failed.** Static UI placeholders only; zero underlying API or gateway integrations.
Profile Management	Update user details.	**Buggy.** Always updates temporary frontend state memory but fails to write changes to a persistent database consistently.

UI Layout Split (Main vs. Settings)	Separate onboarding workflows from active configurations.	**Mocked.** Broken or un-implemented modules (Pipeline, Analytics, Calendar, Leads) are hidden from the end user. Identical elements are hard-coded and duplicated across views due to [REDACTED_AI_TOOL] limits.
Cross-site integration	External lead capture and dashboard routing.	**Functional.** Toggles cleanly. Form captures lead info, maps from the [REDACTED_INTEGRATION_PARTNER], and sends a baseline notification email.
Google Calendar Sync	Bi-directional booking.	**Mocked.** Hardcoded bypass to avoid auth errors during pitches. Does not read calendar availability, block out busy times, or provide required OAuth privacy disclosures. Regardless, appears to be fully functional on initial inspection (excellent mocking of features.)
Alpha 1.1 Features	2FA, text reminders, and cross-agent lead portals.	**Missing.** Cosmetic buttons with no underlying microservices, database schemas, or infrastructure.

The "Automated Testing" Illusion

"100% Pass Rate" on automated tests merely confirm that elements exist on the page and that visual containers render without throwing fatal compilation errors. No value-add tests exist, with the current tests incapable of verifying data persistence, state mutation, or API integration validity. Creates a false metric of project completion that misled non-technical leadership.

Investigation Methodology: Code inspection, manual testing, functional codepath audit across the entire application.

5. Conclusion

The [REDACTED] CRM appears to be market-ready and hints at the possibility of intriguing commercial success, but this is ultimately a technical illusion. [REDACTED] team successfully validated market demand, demonstrating real estate agents actively wanted this feature set. However, the codebase being evaluated is not a functional software asset; it is a disposable, AI-generated prototype.

Because the frontend is highly polished, non-technical leadership was misled into believing the platform was production-ready. A standard visual inspection and basic manual review of the repository were insufficient to expose its critical flaws, as the LLM-generated voluminous output obscured the lack of foundational engineering behind seemingly complete syntax, project files, and seemingly object-oriented syntax. Static analysis and cursory technical reviews were equally ineffective, whereas an exhaustive line-by-line review of the entire codebase would have been prohibitively expensive. Consequently, effective due diligence required an interactive inspection of specific use cases executed concurrently with runtime codepath audits and transaction analysis (network payload capture, console log output, multi-stage testing.)

Ultimately, the current implementation cannot support live commercial activities. This is a high-fidelity interactive wireframe, not a viable software product. Moreover, the current codebase cannot be salvaged, patched, or launched. It must be entirely discarded and rebuilt from scratch guided by a technical design and using an industry-standard framework.

6. Recommendations

Given the immediate commercial reality that end users are already scheduled for software demonstrations and onboarding, abandoning the asset is commercially unviable.

We recommend a phased approach:

6.1. Establish a "Functional Demo": Immediately pause all LLM only development and patch the most obvious UI gaps. Place a hard guardrail stop on [REDACTED_AI_TOOL] prompts. The objective is to ensure the application does not crash upon first contact during scheduled prospect demonstrations. Institute QA process and a hard release cycle to prevent bug regression.

6.2. Extract Customer Feedback: Utilize the scheduled prospect meetings strictly as discovery and validation sessions. Identify which features can be safely discarded and which are absolute requirements for a successful launch. Remove discarded features from the codebase ASAP.

6.3. Core Re-Implementation: Once the feature set is defined and validated, re-implement using an established MVC framework. Recommend using [REDACTED], since [REDACTED_INTEGRATION_PARTNER] is already built using this framework. The estimated rebuild cycle is approximately 8-man-months, assuming all currently advertised features are to be implemented. The React calendar UI/UX components may be salvageable as a stand-alone module to be used in the rebuild; may be extracted using LLM prompts as long as they are inspected by a human engineer.

~ End of Document ~

Additional technical due diligence reports and architectural teardowns are published via The Backchannel. Access is subject to corporate domain verification.

https://topnotch.tech/backchannel