The Working Application Fallacy: What Four and a Half Months of Silence Hid in a Vibe-Coded Codebase

A case study measuring the gap between "it works" and "it's sound."

An audit case study • 2026

Executive Summary

We audited a live, revenue-facing application before and after cleanup: a small company's marketing website with a contact form, built through "vibe coding." The operator prompted an artificial intelligence (AI) coding tool and shipped what it produced, with no human review pass. The site ran in production for roughly four and a half months with no one opening the source. Each new feature got clicked through and confirmed working. Every signal available to a non-technical operator said the site was healthy.

Then the operator noticed the contact form seemed to have stopped delivering leads, opened the codebase for the first time, and commissioned an audit. The audit found:

- **A contact form that had never delivered a lead.** The "stopped working" form that triggered the audit had never been wired to send anything. Submissions accumulated in server memory and vanished on every restart.
- **Two-thirds of the source code doing nothing.** Hand-maintained source fell from 6,736 lines to 2,405 in a cleanup that added and removed no features — a 64% reduction that was almost entirely subtraction. The single shipped page referenced 10 of its 47 user interface (UI) components.
- **A form that turned three rapid clicks into nine submissions**, with no guard against duplicates.
- **Mobile navigation that did not navigate.** Tapping a menu link on a phone closed the menu and left the page where it was.
- **Pinch-to-zoom disabled sitewide**, with no recorded reason, on a site aimed at an older small-business demographic.
- **Error text invisible in dark mode:** 2.15:1 contrast against a 4.5:1 accessibility floor.
- **No typecheck gate, no concurrency protection, and a runtime that reached end-of-life eleven days after the last commit.** None of this shows up to anyone testing the site by hand.

- **Zero lines of process or architectural documentation** in the repository.

The build succeeded every time. The console stayed clean. Clicking through the site showed nothing wrong. A clean console and a working demo told the operator nothing about whether the application was sound, because AI-generated code degrades in ways that hide from exactly the testing a non-engineer, or a satisfied engineer who stops looking, can do.

This paper also reports, with numbers, where the cleanup made things worse and where a plausible fix failed once we measured it. An audit that cites only its favorable results deserves no trust, and we wrote this one accordingly.

1. The Setup

The application is a single-page marketing site, a company homepage with a contact form, generated through an AI coding assistant over a series of prompted iterations. The operator checked each iteration by hand: does the page load, does the new section look right, does the form appear to submit. On that basis they judged the site working and left it alone.

It stayed frozen for **four and a half months**. In that window exactly one file changed on the server: a favicon swap. That rules out the usual explanation for a codebase in poor shape, years of rushed changes by many hands. This codebase had one author (an AI assistant, prompted by one person), a short build period, and then silence. Whatever the audit found had been there from the start, or nearly so, and no one had looked.

2. Methodology

We measured the findings in this paper rather than recalling or estimating them. We checked out the pre-audit code into its own environment, built it with its original toolchain, and served the production build on its own port. We built and served the post-cleanup state the same way on an adjacent port. We then drove both with identical scripts in a real browser and captured the differences: form submission counts, scroll positions, computed contrast ratios, bundle sizes, and dependency counts.

A cleanup effort can describe itself as fixing things without ever confirming the "before" was broken as described, or that the "after" is fixed. Several findings in this audit exist only because we held to that discipline, including a fix that read as correct but failed under test (§4) and a completed piece of work we reverted because a measurement came back wrong (§6.3).

Every number in this paper comes from that process. Where we repeat something from the project's own history instead of a direct measurement, we say so.

3. Finding One: Most of the Code Was Never Used

The largest category of findings, by volume, was code that did nothing. Measured across everything hand-maintained — client, server, shared code, and scripts — the cleanup removed 64% of the source, from 6,736 lines to 2,405, without adding or removing a single user-visible feature. Most of that dead weight sat in one place.

The application shipped with **47 UI components totaling 4,131 lines**. The site, one scrolling page with a contact form, used **10** of them. The other 37 included a full sidebar system, a carousel, a data table, a command palette, a calendar picker, and a pagination control, on a site with two anchors and no logged-in state of any kind.

This is the fingerprint of a scaffold. A code generator produces the full component library it knows how to produce, because that is cheaper than reasoning about which parts a specific page will need. Nobody prunes it, because nobody who isn't reading the code has a reason to.

The consequences compounded outward from there:

- **Dependencies.** The cleanup cut declared dependencies from 87 to 31, a 64% reduction, and resolved packages in the lockfile from 592 to 404. Among the removals: eight backend packages, including a session store, an authentication library, and a Postgres driver, implying a login system and a database that were never built. The application's actual storage is an in-memory list. Nothing these packages did was reachable, yet a supply-chain audit or a Common Vulnerabilities and Exposures (CVE) scanner still had to account for every one of them.
- **Shipped styles.** Removing the unused components removed 466 Cascading Style Sheets (CSS) selectors from the stylesheet sent to every visitor, roughly 4 kB gzipped of styling for a sidebar, a carousel, and a chart that do not exist on the page.
- **Ongoing cost, not just historical waste.** During an unrelated framework upgrade, an automated migration tool introduced a genuine bug into one of the unused components, a file with zero importers. We spent time finding and correcting that bug before deleting the file. Dead code cost real engineering time even though nothing ever ran it.

A directory of duplicate binary assets, four stock photos and a logo present two or three times at identical checksums, accounted for another 1.1 MB of tracked storage.

Why this belongs in a paper about AI-generated code: none of it was a wrong decision. It is what a process produces when it generates the median useful scaffold for a *type* of project and never comes back to subtract what a *specific* project didn't need. The scaffold itself is not the AI-specific part — human developers install full component kits from templates constantly. What was missing here is the person who eventually reads the code and subtracts. In this project, no reviewing human ever existed in the loop, so nothing was ever pruned.

4. Finding Two: The Defects Were Real, User-Facing, and Silent

Dead code is a maintainability story. The defects in this section reached customers, and they are the strongest evidence in this paper because they are not code-quality opinions.

The contact form submitted duplicate leads. The form had no guard against a fast or double click. Three quick, real submissions produced **nine separate POST requests** to the server. The business's entire sales funnel runs through this one form, so in a working system, nine recorded leads for one prospect means a sales team following up several times on the same person, and a prospect who concludes the company doesn't have its act together. Here the harm was latent rather than felt — as §6.1 explains, the submissions were going nowhere at all, duplicates included. But latent is the dangerous kind: the day someone wired the form to actually deliver, this defect would have gone live with it, already installed.

The fix itself shows why testing a fix beats reasoning about one. The first proposed guard checked the form library's own "is this submitting" flag before allowing a new submit. It reads as correct and would pass a code review. In a browser it did nothing: that flag updates only after React finishes a render cycle, so two clicks in the same instant both read "not submitting" before either submission registers. We replaced it with a plain flag that flips the instant a click happens, not on the next render. The wrong fix would have shipped clean, read as fixed, and kept double-posting. We caught it by clicking the button twice and counting the network requests. One honest limit on the shipped fix: a client-side guard stops the double click it was measured against, not a retry, a second tab, or a refresh-and-resubmit. The durable version is server-side deduplication, which remains an open item.

Mobile navigation did not navigate. Tapping any link in the mobile menu closed the menu and left the page where it had been, a complete no-op, for the entire four and a half months the site was live. Mobile is where a prospective customer first meets a small-business site like this one, so the primary navigation path was broken for a large share of visitors while the desktop version worked and hinted at nothing. The root cause was a race between a scroll command and an animation library reading and restoring the page's scroll position mid-flight. A person clicking around the site on their own laptop, in a hurry, checking the last shipped feature, will not stumble into that interaction by accident.

Pinch-to-zoom was switched off across the entire site, a single viewport setting with no rationale attached anywhere in the project's history. The site sells a service to small-business owners, a demographic that skews older and relies on being able to zoom in. This may be the highest-impact, lowest-effort defect in the whole audit: one attribute made the site harder to read for exactly the visitors it needed to persuade.

Error messages were unreadable in dark mode. The text explaining why a form submission failed, the most important string on the page whenever it appears, measured 2.15:1 contrast against its background. The accessibility floor is 4.5:1. A visitor in dark mode whose form failed to send saw no visible explanation of why.

Every one of these hid from the two tests a non-technical operator can run: does it look right, and is the browser console clean. All four builds, before and after, produced **zero console errors and zero warnings** throughout. This application's problems never touched the console.

5. Finding Three: The Infrastructure Had No Safety Rails

Beyond the visible product, the delivery pipeline carried risks a working demo could never surface:

- **No typecheck gate on the deploy.** The only script that publishes to the live site never ran the project's one form of static verification. We confirmed this by planting a type error and watching the deploy path ship it.
- **No protection against overlapping deploys.** Two deploy runs firing close together read the same server-side state, both write back a diff, and the second write wins without a warning. A stale file can land on the live site with nothing in any log pointing at the cause, and the defect surfaces weeks later as an inconsistency nobody can trace.
- **The deploy step duplicated and stranded files on the server**, an artifact of a copy command redundant with what the build tool already did, written with `|| true` so that no failure could ever surface.
- **The production runtime reached end-of-life eleven days after the last commit** — and then served production traffic unsupported for the remaining four months of the frozen window. Nothing in the project would have raised it on its own.

The security pass added three more, none visible from a browser:

- **The server's global error handler echoed internal error text to the client.** A malformed request got the JSON parser's own message back instead of a fixed string.
- **The contact form had no spam protection of any kind** — no honeypot, no challenge, nothing between the open internet and the endpoint.
- **The lead data itself — names, contact details, whatever a prospect typed — lived only in process memory**, unpersisted and gone on every restart.

None of these is a coding-style complaint. Each is a specific, dated mechanism by which the site could have broken, served stale content, or become harder to patch for a real security issue, while looking exactly as healthy from a browser as it had the day before.

6. What the Cleanup Did *Not* Fix, or Made Worse

A paper built on this material earns its credibility by including this section in full.

6.1 The contact form still does not email anyone

This is the most consequential finding in the audit. The event that triggered the whole exercise, the impression that the contact form had "stopped sending emails," rested on a premise that was never true. Nobody ever wired the form to send email. Submissions land in an in-memory list on the server and vanish when the process restarts. The scaffolding for a database exists in the code; no one connected it. The cleanup neither introduced nor repaired this gap, and it remains the largest open item in the project. From day one the application had no path from "someone filled out the form" to "a human sees it," and the operator could not tell this apart from a working form until they went looking for emails that were never going to arrive.

6.2 The JavaScript bundle got 20% larger

Removing 56 unused dependencies did not shrink what ships to the browser, because code that nothing referenced never entered the bundle. Its cost was install size and audit surface, not delivery weight. Meanwhile, moving to current, supported major versions of the UI framework and its animation library added real weight: shipped JavaScript grew from 562.92 kB to 675.37 kB (176.04 kB to 208.95 kB gzipped). We accepted that cost to get off dependencies that were themselves about to become the next silent liability. A claim that this project came out "lighter" across the board would be false, so this paper does not make it.

6.3 We threw away completed work because the numbers didn't support it

One planned upgrade, a newer version of the validation library, was implemented, tested, and confirmed behaviorally identical to what it replaced. Then we measured the shipped bundle: the new version added roughly 200 kB of locale data the site would never use, with no way to strip that piece alone. We reverted the finished work. We include this case because it was uncomfortable: correct, complete engineering effort is not by itself a reason to ship. The measurement is.

6.4 A dramatic-looking number that isn't real

Partway through the frozen period, an unrelated commit bloated the site's favicon from about 1 KB to over 300 KB; the cleanup then shrank it back down. Read in isolation, "300 KB down to 1.7 KB" looks like a large win. Measured against the starting point four and a half months earlier, it nets out to **a 608-byte increase**, a wound the project inflicted and healed on itself inside the measurement window. We flag it as a caution: a headline number in a change log is not a verified before/after comparison, and this paper stays trustworthy only as long as it keeps that distinction.

7. What Generalizes

Four claims survive scrutiny of this material.

1. A working demo does not establish a sound application. Every serious defect found here, the duplicate-submitting form, the dead mobile navigation, the unreadable error text, the disabled zoom, produced zero console errors in both the broken and the fixed state. Vibe-coded software rarely fails by crashing. It fails by doing the wrong thing while looking, to any inspection short of deliberate testing, like it is doing the right thing.

2. Unreviewed scaffolding costs money, and the cost compounds. The unused components in this project did not sit idle. They pulled in unnecessary dependencies, shipped unnecessary CSS to every visitor, and later consumed real engineering time when an automated tool broke a file that had no reason to exist. Code nobody asked for is a standing liability that grows more expensive the longer it goes unexamined.

3. Plausible fixes need testing, not just review. More than once in this audit, a fix that read as correct, a "pending" flag to stop duplicate form submissions, a set of color values that looked right, turned out wrong

in ways that only showed up in a browser. Reading AI-generated code is necessary and was not enough here; the defects that mattered most only surfaced when we drove the real site and counted what happened.

4. Missing documentation was itself a root cause. At the start of this audit, the repository contained zero lines describing how the project was built, what its architecture assumed, or what had already been tried and found wrong. We rediscovered every correction in this paper from scratch, in a live production codebase, because no one had written anything down. No person had ever been in the loop to write it.

8. Recommendation

This paper does not argue against AI-assisted development. The site described here ran, stayed live, and presented the business credibly for four and a half months on the strength of prompted, unreviewed code. What it demonstrably did not do is convert: its one conversion mechanism delivered nothing the entire time (§6.1), and nobody could tell. The argument is narrower and, we believe, more useful: "it works" and "it is sound" are different properties, the gap between them is measurable, and the gap does not shrink on its own while nobody is hitting it.

A note on scale, because a reader will reasonably ask what an exercise like this costs: the entire cleanup — every fix, every measurement in this paper — ran roughly 14 hours of working time across ten branches. The gap between "it works" and "it is sound" was four and a half months wide and about two working days deep.

If you run production software that was substantially AI-generated without a review pass, the takeaway is not "stop using AI to write code." It is this: schedule the first real code review before a customer forces one on you, verify defects by driving the real application rather than reading the diff, and keep the resulting record of what was found, including what the plan got wrong along the way, as an asset rather than a document to discard once the branch merges.

This paper draws on a full internal audit comparing two dated commits of the same application, with every figure independently reproduced in a real browser against a checked-out copy of the original code. Detailed measurements, methodology, and the complete list of open items are available on request.