TL;DR: Why vibe-coded projects break

- **Early progress is real; finish and maintain are hard.** The last 20% and ongoing ownership drive most of the pain.
- **Twelve patterns, one table.** Comprehension debt, last-mile stalls, context loss, regression churn, security gaps, and agent loops show up most often across nine source families—not as a failure rate, but as recurring mechanisms.
- **Root cause is often missing decisions.** Data model, permissions, tests, and definition of done deferred until the codebase is already tangled.
- **What helps matches classic engineering.** Small changes, version control, explicit constraints, and separating planning from generation (Spec-Driven Development goes deeper on workflow).

To understand where AI-built projects run into trouble, we examined 202 research items collected through ten research passes across nine source categories, including Reddit communities, Hacker News, GitHub issues, developer post-mortems, platform forums, social posts, industry research, and documented security incidents.

The review identified 12 recurring failure patterns, from context loss and regression loops to security gaps, abandoned projects, and codebases that become difficult to maintain.

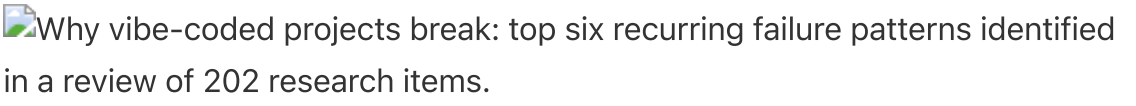
Failure pattern	Source families reporting it
Comprehension debt / unmaintainable code	9 of 9
Last-mile development stalls	8 of 9
Compiles and passes tests, but isn't ready	8 of 9
Context loss	7 of 9

Failure pattern	Source families reporting it
Regression churn	7 of 9
Security and configuration omissions	6 of 9
Agent loops and credit consumption	6 of 9
Project rewrites	6 of 9
Abandoned projects	5 of 9
Maintainer overload	4 of 9
Destructive agent actions	4 of 9
Commercial failure / no users	2 of 9

Source-family coverage indicates how many of the nine research categories contained reports of each failure pattern. These figures do not represent the percentage of projects affected or the frequency of failures across all vibe-coded applications. Multiple sources within a category may describe the same incident.

A note on methodology: This is a purposive review of publicly reported experiences, not a random sample of AI-built projects. The findings describe recurring problems and the stages where they appear, not how frequently vibe-coded projects fail overall. The accompanying research appendix documents the 12 patterns, their distribution across source families, and 95 linked references from the underlying research. The remaining sources are not individually catalogued in that appendix.

The sections below group those twelve patterns into themes builders describe most often (maintenance and comprehension debt, the 80% wall, regressions, security, trust in tooling, and experience). Patterns such as credit burn, abandoned builds, maintainer overload, and zero-traction launches appear in the table and in the Research Appendix at the end of this article even when they do not have their own heading.



Six of the most widely reported failure patterns across nine source categories

The app works. The builder can't change it.

The most troubling failure isn't a broken app. It's an app that works today but is too tangled to touch tomorrow—the comprehension-debt pattern at the top of the table (unmaintainable code you can no longer safely change). Several builders describe this trap. The initial version functions beautifully, users might even be happy, but a refactor or new feature touches code nobody fully understands. Small edits trigger unrelated regressions and a project that felt frictionless last week suddenly feels impossible to own.

The problem accumulates quietly. Each generation session may be reasonable on its own, while the codebase gradually loses consistency. Naming conventions start to drift, similar logic gets implemented twice, and dependencies are introduced without a clear reason. Eventually, a routine change demands far more investigation than it should.

That's a brutal outcome for someone who used AI specifically to avoid traditional software management. The app is live, but the builder has no reliable way to maintain it.

The final 20% isn't another 20%

The early stages of AI-assisted development are unusually rewarding. The prototype builds flawlessly, but then come the details: permissions, payments, data migrations, error handling, deployment, recovery, and security testing. Many prototypes don't need all of these, but real production applications used by real businesses do.

In the accounts we reviewed, builders repeatedly hit a wall right before launch. Sometimes the remaining work is highly technical. Sometimes they just don't know what counts as "ready." Often it's both.

The issue isn't that Claude or Cursor can't write authentication logic. The issue is that major architectural decisions got deferred until the app already had a hundred moving parts. Bolting them on late requires unwinding assumptions baked deep into the project.

This is the infamous *80% wall*, when progress stops tracking with effort.

Why one fix breaks something else

AI coding assistants operate with limited context. They inspect files and retrieve snippets, but they don't hold a complete, accurate map of every dependency or architectural decision you made last Tuesday. As apps scale, this limitation manifests as hallucinated imports, inconsistent patterns, and edits that work locally but break production. Context compaction makes this worse.

The practical cost is familiar to anyone who has built with these tools. Fix a bug, find two new ones, ask the agent to fix the regression, watch it overwrite your previous fix. Eventually, you spend more time verifying what the AI did than building the next feature.

This isn't inevitable. But it's highly probable if changes are broad, dependencies are undocumented, and tests don't exist.

Security mistakes are harder to spot

A broken button is annoying. An insecure endpoint exposes your customers. Take CVE-2025-48757, a vulnerability involving missing database Row Level Security (RLS) in Lovable-generated apps. A researcher found inadequate RLS across 170 of 1,645 apps they examined. Lovable rightly pointed out that developers own their data protection policies, but it highlights exactly what happens when access controls aren't explicitly defined.

Similarly, an UpGuard investigation found over 16,000 Supabase-backed sites exposing readable tables. Not all were AI-built, but it shows how common and severe database misconfigurations are.

In addition, there's the Replit database deletion incident, where an agent made destructive changes because there wasn't adequate separation between dev and prod environments.

The takeaway? A working UI doesn't mean your app has safe authorization rules, protected secrets, or sensible agent guardrails.

Developers use AI. They don't necessarily trust it.

Adoption does not equal confidence. The 2025 Stack Overflow Developer Survey showed 84% of respondents using or planning to use AI tools. Yet 46% actively distrusted the accuracy of the output—a significant jump from the previous year. In a separate Harness survey, 67% of developers reported spending more time debugging AI code than they anticipated. This distrust creates a second shift of work. You still have to read the code, check the edge cases, verify the dependencies, and audit the security. Writing code faster doesn't make reading it faster.

Research on delivery performance reflects this drag. Google's 2024 DORA report correlated a 25% increase in AI adoption with an estimated 1.5% drop in delivery throughput and a 7.2% drop in stability. Meanwhile, GitClear analyzed 211 million lines of code and found a sharp rise in "copied code" at the expense of refactoring.

Perhaps most striking is METR's 2025 controlled study, in which 16 experienced open-source developers tackle 246 tasks in familiar repos. Using early-2025 AI tools, they took 19% *longer* on average, despite expecting a speed boost. (Newer tools may perform better, per their February 2026 update, but the baseline remains critical).

Speeding up generation is not the same as speeding up reliable delivery.

Experience still matters

Lovable, Bolt, and v0 sell a beautiful promise: anyone can turn an idea into an app. For prototypes, it's absolutely true, but when the app needs to grow up, the friction hits.

A 2025 Fastly survey found that senior developers (10+ years) were more than twice as likely as juniors to let AI generate over half their shipped code. Why? Because an experienced engineer knows how to spot a leaky abstraction, flag a dangerous permission model, and redirect an agent when it's solving the wrong problem.

A non-technical founder sees a working screen and assumes the underlying foundation is solid. This doesn't mean non-technical founders shouldn't build. It means they need a system to enforce the architectural decisions a senior engineer would normally make. If you don't make those decisions, the AI makes them for you.

What actually works

The builders succeeding with these tools are essentially adapting classic software engineering habits for AI workflows—the same direction as Spec-Driven Development, with lighter upfront structure. Simon Willison calls it "Vibe Engineering", and the rules are remarkably consistent:

- **Define the constraints upfront.** Map out the data model, permissions, and service dependencies before you write your first prompt.
- **Keep changes microscopic.** Ask for one bounded edit at a time.
- **Commit religiously.** Version control is your undo button when the AI hallucinates.
- **Test behavior, not pixels.** Verify failure states and edge cases, don't just check if the UI looks nice.
- **Separate planning from building.** Don't use the exact same continuous chat loop to brainstorm, code, and review.
- **Lock down production.** Sandbox your agents. Require human review before touching live data.

None of this guarantees success. But it reduces the cost of mistakes and makes a project easier to understand when something does go wrong.

The problem starts before the code

These failures aren't isolated; they compound. Vague requirements lead to messy implementations, messy implementations make debugging harder, weak tests let regressions slip through, and missing security constraints sit like landmines.

It's important to note that not every bug is a planning failure. Models hallucinate and platforms go down, but many of the problems described in these reports trace back to decisions that were never explicitly made: the data model, feature boundaries, security constraints, and definition of done.

AI has made it effortless to generate code. It hasn't removed the responsibility for what that code actually does.

FAQ

Why is it so easy to start but so hard to finish an AI-generated app? Because prototypes and production apps require completely different things. AI tools excel at wiring up user interfaces and basic flows—the first 80%. But the final 20% involves data migrations, permissions, edge-case error handling, and deployment. If you defer those architectural decisions until the end, you have to unwind a deeply entangled codebase that you didn't actually write.

Are AI-generated apps inherently insecure? Not inherently, but they often are by default. A working UI doesn't mean your database is secure. As the CVE reports and database exposure incidents in the research show, if you don't explicitly define Row Level Security (RLS) and access controls, the AI usually won't do it for you. It builds for functionality, not defense.

Does developer experience still matter if the AI is writing the code? It matters more than ever, just differently. The data shows senior engineers actually ship *more* AI-generated code to production than junior engineers. An experienced dev knows how to spot a leaky abstraction, flag a dangerous permission model, and redirect the agent when it goes off track. If you don't have that experience, you need a system to make those architectural decisions for you.

Why does fixing one bug constantly create two new ones? Context loss, incomplete dependency awareness, and inadequate regression testing can all

contribute. As your application grows, the AI agent may lose track of architectural decisions made earlier in the project. When you request a broad change, it can modify code without accounting for dependencies elsewhere, introduce duplicate logic, or overwrite previous fixes. Without proper tests and clear architectural boundaries, fixing one issue can easily introduce another.

How do I stop my codebase from becoming a tangled mess? Stop treating continuous chat loops as your architecture plan. You have to define the constraints upfront. Map out your data models, service dependencies, and permissions before you write a single prompt, and keep your agent's tasks narrow and scoped. (This missing layer of upfront planning is exactly why I'm building Zalcro).

Related:

- Spec-Driven Development: A Practical Start-to-Ship Workflow
- Vibe Coding Hasn't Won. The Industry Is Misreading Its Own Data

Works Cited

- [Anthropic, 2025] — Auto-compaction loses context and creates loops, Claude Code issue tracker
- [DORA, 2024] — Accelerate State of DevOps Report 2024
- [Fast Company, 2025] — What really happened when an AI agent wiped a production database
- [Fastly, 2025] — Senior developers ship more AI code
- [GitClear, 2025] — AI Copilot Code Quality: 2025 Research
- [Harness, 2025] — State of Software Delivery
- [METR, 2025] — Measuring the impact of early-2025 AI on experienced open-source developer productivity
- [METR, 2026] — Uplift update, February 2026
- [NVD, 2025] — CVE-2025-48757: Insufficient Row Level Security in Lovable-generated applications
- [Reddit r/vibecoding, 2025] — Vibe coding and hitting the 80/20 wall

- [Reddit r/vibecoding, 2026] — Vibe coded for 6 months, my codebase is a disaster
- [Stack Overflow Developer Survey, 2025] — AI section, 2025 Developer Survey
- [UpGuard, 2026] — Everything everywhere: systemic data exposure in Supabase apps
- [Willison, 2025] — Vibe engineering, simonwillison.net

Appendix: Research Methodology and Supporting Evidence

Research scope

This paper draws on ten research passes covering 202 research items across nine source families. The items include developer discussions, post-mortems, platform issues, security incidents, and industry studies. Some sources discuss the same underlying incidents.

The review focused on identifying recurring problems in AI-assisted software development, particularly where projects stalled, required substantial rework, became difficult to maintain, or encountered problems after deployment.

The research covered nine source families:

1. Reddit vibe-coding communities, including r/vibecoding and r/ChatGPTCoding.
2. Reddit development and founder communities, including r/SideProject, r/webdev, r/ExperiencedDevs, r/SaaS, and r/indiehackers.
3. AI-tool communities and vendor forums, including r/ClaudeAI, r/cursor, r/ClaudeCode, and the Cursor and Replit forums.
4. Hacker News discussions and firsthand accounts.
5. GitHub issues, discussions, repositories, and maintainer policies.
6. Developer post-mortems published on Dev.to, Medium, Substack, and personal blogs.

7. Public X posts and articles.
8. Developer surveys and empirical research, including DORA, Stack Overflow, METR, and GitClear.
9. Documented security incidents, platform limitations, and vendor disclosures.

How the findings were organized

The research identified 12 recurring failure patterns. These were organized by the nature of the problem, the development stage where it typically became apparent, and the source families in which it appeared.

Source-family coverage was used to describe how widely a pattern appeared across different types of evidence. For example, a pattern appearing in nine of nine source families was reported across all nine categories, not in every project examined.

The patterns were not mutually exclusive. A single project could experience context loss, regression loops, and maintenance difficulties. Likewise, several sources could discuss the same incident.

Evidence and limitations

This was a purposive review of the public record, not a representative survey of AI-built projects.

Several limitations affect how the findings should be interpreted:

- **Selection bias:** Developers are more likely to publicly describe frustrating experiences, failures, or unusual successes than routine development work.
- **Self-reported experiences:** Many accounts come from individual developers and cannot be independently reproduced.
- **Overlapping incidents:** Different sources sometimes discuss the same underlying project or failure.
- **Different forms of evidence:** Developer anecdotes, controlled studies, vendor documentation, and security disclosures do not carry equal evidentiary weight.

- **No project failure rate:** The number of sources reporting a problem cannot establish how frequently that problem occurs across all AI-assisted projects.

The findings therefore describe recurring failure mechanisms and where they tend to surface. They should not be interpreted as estimates of the percentage of vibe-coded projects that fail.

Supporting references

The main article cites 14 sources directly. The underlying 12-pattern research report contains a broader bibliography of 95 linked sources, including developer accounts, GitHub discussions, platform documentation, industry studies, and incident reports.

The 95-reference bibliography documents part of the reported 202-source corpus. The remaining sources are not individually listed in the available research report, so the full corpus is not independently auditable from this appendix.

▼ Supporting bibliography (95 sources)

Linked references documented in the underlying research report (main article cites a subset).

1. Vibe coding and hitting the 80/20 wall: r/vibecoding
2. 99% of my projects be like: r/vibecoding
3. How I stopped abandoning Lovable projects at 80%: r/lovable
4. Aaron Gooch on the final mile of vibe coding: X
5. Shane Jordan on the gap between vibe coding tools and production: X
6. How do you deal with finishing your project: r/vibecoding
7. How I keep Claude from losing context on bigger projects: r/vibecoding
8. How do you handle context loss between Claude sessions: r/ClaudeAI
9. Solutions for the dead loop problem in Cursor vs Code: r/ChatGPTCoding
10. Applying changes to a large file fails: Cursor Community Forum
11. Tokens and context limits: Bolt documentation

12. Auto-compaction loses context and loops: claude-code issue 10727
13. Composer continuously breaking existing code functionality: Cursor Community Forum
14. Built this app with vibecoding, now I am stuck: r/ClaudeAI
15. An AI agent renamed a field and broke three services: Hacker News
16. New agent is such a nightmare: r/replit
17. Building Syntaqlite with AI, and why I threw it away: Lalit Maganti
18. Nobody tells you the real work starts after the AI says done: Chandler Nguyen
19. Holding an eight-agent API rewrite before cutover: X
20. The code compiles, passes tests, so you stop reading it: Hacker News
21. State of Software Delivery: Harness
22. Composer stops writing files: Cursor Community Forum
23. Vibe coded for 6 months, my codebase is a disaster: r/vibecoding
24. How many of you are already employed in AI code cleanup: r/ExperiencedDevs
25. Has anyone actually maintained a vibe coded project: r/webdev
26. After two years of vibecoding, I am writing code by hand: Atmoio
27. My hot take: the code produced by Claude Code is not good: r/ClaudeAI
28. Statement on CVE-2025-48757: Matt Palmer
29. GHSA-773x-pxjg-gxgx: insufficient Row Level Security in Lovable generated sites
30. Everything everywhere: systemic data exposure in Supabase apps: UpGuard
31. Some Supabase customers are publicly exposing reams of people's data: TechCrunch
32. AI-pinned vulnerable dependency led to a cryptominer: Hacker News
33. Are AI app builders secure? We tested Lovable, Base44 and Bolt: OX Security
34. Vibe coders at my company did not pay attention to security: r/webdev

35. Vibe code IRL left Stripe API keys public: r/webdev
36. False confirmations and wasted credits: Cursor Community Forum
37. Have the error loops gone away yet: Replit Community Forum
38. Maximizing token efficiency: Bolt documentation
39. Why do v0 agents keep failing and consuming my credits: v0 FAQ
40. 300 credits in 2 days with basic prompts: r/lovable
41. After 6 months of vibe coding, my codebase is a mess: Medium
42. Jason Lemkin on day-8 technical debt: X
43. The reality of building an app with AI: r/SideProject
44. Daniela Petruzalek on rebuilding a fragile vibe-coded backend: X
45. Looking for opinions on rewriting my project: r/vibecoding
46. Anyone else have a graveyard of half-built projects: r/vibecoding
47. Of projects abandoned: r/vibecoding
48. You're vibe coding wrong and it's not your fault: Sterling Chin
49. vibe-coded-web-server: archived repository
50. ai-store: archived vibe-coded e-commerce demo
51. Vibe coding lessons nobody talks about: Dev.to
52. Large influx of extremely low-quality pull requests: google/go-github
53. Policy on vibecoded contributions: rust-lang leadership council
54. Automatically closing pull requests from external contributors: tldraw
55. Restricting new pull requests to collaborators: pydicom
56. AI policy: Mastodon
57. mem0 maintainer on the vibe coding contribution phase: X
58. We should review PRs on their merits: OpenZFS
59. Jason Lemkin on Replit deleting a production database: X
60. Amjad Masad acknowledging the production data deletion: X
61. Replit agent deleted a production database: AI Incident Database
62. Broke my Lovable project beyond restoration: r/vibecoding
63. Cursor context retrieval degraded: Cursor Community Forum

64. I vibecoded 4 apps in 6 weeks, all had 0 users: r/SideProject
65. Built an AI app in 2 weeks: 140K views, 0 revenue: r/SideProject
66. I built 4 apps on ideas that AI told me were great: r/SaaS
67. AI: 2025 Stack Overflow Developer Survey
68. Stack Overflow 2025 Developer Survey press release
69. Announcing the 2024 DORA report: Google Cloud
70. Announcing the 2025 DORA report: Google Cloud
71. Measuring the impact of early-2025 AI on experienced open-source developer productivity: METR
72. AI Copilot code quality: 2025 research: GitClear
73. AI for developer productivity: Uplevel
74. Quantifying GitHub Copilot's impact on developer productivity and happiness: GitHub
75. The state of developer ecosystem 2024: JetBrains
76. Octoverse: GitHub
77. Update on the adblocker I vibe coded last week: r/vibecoding
78. Vibe engineering: Simon Willison
79. How are you keeping AI-generated slop out of your codebase: r/webdev
80. Anyone else feel like vibe coding hits a wall: r/vibecoding
81. How I got Claude to stop going in circles: r/ClaudeAI
82. The hottest new vibe coding startup Lovable is a sitting duck for hackers: Semafor
83. Need help with the iOS build: r/boltnew
84. Rewriting whole apps with AI: Tom Beckenham
85. I spent 2 months building an AI app and almost nobody wants it: r/SideProject
86. I built 4 apps in 30 days, 3 are dead: r/SideProject
87. Porting a major project by prompt alone produced 4,500 compilation errors: Hacker News
88. Cursor and Claude failed a multi-file Angular refactor: Hacker News

89. Hitting a wall: r/ChatGPTCoding
90. My project became so big that Claude cannot handle it: r/ChatGPTCoding
91. Summary of Replit issues: Replit Community Forum
92. Tom on complexity turning building into fighting your own codebase: X
93. Do coding agents create technical debt? Yes, and surprisingly no: Eric Broda
94. Is debugging AI-generated code a nightmare: r/indiehackers
95. Two AI-started projects abandoned: Hacker News comment

P.S. If you are building software with AI and want to get past the part where it breaks, Zalcro was built for this. David is the founder; it defines architecture, data models, and dependency-aware implementation plans before development begins.