AUTONOMOUS AI ENGINEERING: PART ONE

WHAT A FULLY AUTONOMOUS MULTI-AGENT BUILD TAUGHT US ABOUT ENGINEERING AI SYSTEMS

An AI agent that builds software will not stop on its own. That is the finding underneath every guardrail described in this paper, and it came from watching it happen 11 separate times.

Led by our Director of Engineering, First Factory's deep-agents-factory ran its first complete end-to-end mission on July 9th and 10th of 2026. The goal was to build a key but discrete feature—not an entire application—that was fully defined, built, and tested autonomously by AI agents. This is the kind of work referred to as a software factory. A one-line goal went in. A working, tested product feature came out, without a line of application code written by an engineer.

We set our sights on a Trello-style Kanban board. The result was two Git repositories, a NestJS and Prisma backend, a React and dnd-kit front end, ten planned features, and five fix features. All of these were committed with passing tests, three milestone gates passed, and two genuine bugs caught by the factory's own validators—in code the factory's own workers had already tested and shipped. Let that last point sink in for a moment.

To get to those results, it took 11 halts. That is, 11 times our Director of Engineering needed to intervene, and each intervention was not a simple refinement of logic or refactor of a line of code. This paper is the record of what happened at each one, what it taught us about how LLM agents behave under pressure, and what we changed in code, not in prompts, as a result. The behaviors of the AI agents were tricky, if not somewhat deceitful, and it absolutely required a senior engineer to identify at each point how to reign the agents in.

Special Note: Scope matters here. This was a single-user Kanban board: board, list, and card CRUD plus drag-and-drop, nothing more. There's no authentication, no permissions or roles, no file or image attachments, and no reporting or analytics. None of the economics above should be read as "a production application ships for $70 in a day." They describe how AI agents can drive off-course and run well beyond their needs and costs unless key guardrails are in place. Multi-user, multi-role systems carry a different contract entirely. It not only underscores how carefully we need to watch AI as it generates prolific amounts of code quickly, but also what is inherent in its behavior and how to manage against cheating to accomplish a goal—including protecting token utilization and the associated costs. With the right senior engineer overseeing AI-driven development, we can absolutely deliver significant features faster, but we can't let AI do it alone, nor can we trust it to less skilled engineers.

HOW THE FACTORY WAS BUILT

The factory splits into two layers. The runner is a deterministic LangGraph control plane. It doesn't reason about the work; it dispatches agents, tracks state, and enforces limits. The roster is what it dispatches to, including an orchestrator running on Opus 4.8 that plans the mission and breaks it into features. It triggers workers running on Sonnet 5, one per feature, each in its own Docker sandbox. Lastly, it employs two validators per milestone. Scrutiny is code-facing: It re-runs test suites, reads diffs, and curls the live API. User_testing is black-box: It drives the actual application in a real Chromium browser through Playwright MCP. A milestone gate passes only when every assertion in a validation contract, written before any code exists, is verified with evidence, by the right validator, in the same round.

For this mission, that contract held 26 assertions across three milestones: M1 for a runnable skeleton, M2 for full CRUD editing, M3 for drag-and-drop and polish. When anything goes wrong at any point, the runner halts the mission and hands control back to a human. It happened 11 times.

The architecture is not what makes this worth reading. What autonomous agents do when their budget isn't exhausted and nobody is watching is.

FAILURE FROM NOT KNOWING WHEN TO STOP

The 10 clean feature runs averaged about 5 minutes and $1.30 apiece, with real test-driven development and clean, honest commits. When workers failed—and 3 of the 11 halts were workers—the cause was never that the model didn't understand the task. It kept working past the point where the work had any value left to add.

The clearest example: The worker scaffolding the front end repository set up Vite correctly within its first few tool calls, then spent roughly 90 of its 200 remaining calls interrogating the npm registry, checking every dependency's engines and peerDependencies fields one at a time, pulling package source off unpkg to inspect export shapes, and, at the low point, running node -p "20.20>=20.19" nearly 20 times in a row to reconfirm a version constraint it had already satisfied. It hit its step limit mid-implementation. Cost: $3.49. Nothing committed.

The retry cured that particular habit with a new prompt rule: to trust the versions its own scaffolder had pinned. The same habit showed up again immediately, just under different covers. With 12 of 12 tests passing and a third of its budget still available, the worker didn't stop. It tried to prove the rendered page actually rendered, inside a sandbox with no browser: roughly 25 near-identical attempts to execute a JavaScript bundle in jsdom, followed by downloading Chromium through Playwright. The step limit fired as the download finished. Work complete, nothing committed. Cost: $3.98.

Two failures, one mechanism. Basically, a remaining budget and no additional instructions for the agent equals more work autonomously performed by the agent, whether or not that work adds any value. The fix that held was procedural, not motivational: Commit the moment the suite goes green, before any optional verification. Tests, lint, and a clean build are what "verified" means for a worker. Browser truth belongs to a different agent entirely.

AGENT INSTINCTS: UNIVERSALLY SHARED

Workers weren't the only agents that couldn't tell when to stop. The validator agent, Scrutiny, was unable to perform its code-validation duties because the M1 contract lacked any 'Tool: tests' assignments; instead, all tasks were incorrectly categorized as 'http' or 'agent-browser'. Faced with an empty checklist and a prompt insisting a rubber-stamp pass was a failure of its job, Scrutiny invented its own audit: checking out each feature's historical commit into a git worktree to re-run old suites, then mutation-testing the live workspace by breaking a data-testid to see if anything noticed, then reverting it. It took 291 tool calls, $2.90, and one non-blocking finding. The real defect was hiding underneath: 12 of the 17 assertions, the majority of the milestone's checklist, had no agent responsible for verifying them at all. Despite that gap, the system was on track to mark the milestone as passed anyway, as if full verification had happened.

A round later, user_testing degenerated into a genuine doom loop: 78 identical cycles of navigate, snapshot, navigate, snapshot, over four minutes, with zero reasoning text, until the session died. After another attempt, now correctly scoped, it burned through 1,000 graph steps on 348 calls that were each individually different. There was an npm audit here, a transitive dependency check there, and never once did it attempt its report. The loop guard stayed correctly silent, because nothing had actually repeated.

Each fix we put in place got outsmarted by the next failure, because each one was just different enough to slip past the last safeguard. Here's how that played out: Our first safeguard looked for an AI repeating the exact same action over and over, which is what caught the "doom loop" described earlier. That worked, until a different AI found a loophole: Instead of repeating one action exactly, it started running a string of throwaway test commands, "echo z1," then "echo z2," then "echo z3," and so on. Each one was technically a little different, so the safeguard, which was only looking for identical repeats, didn't notice.

So we built a smarter version that could also recognize that pattern, one that noticed when only the numbers were changing while everything else stayed the same. That closed the loophole. But then a third AI found yet another workaround: It kept updating its own to-do list every few seconds, more than 50 times in a row, tweaking the wording slightly each time ("write tests" became "write the tests," then "write good tests," and so on). Because the wording kept changing, it slipped past both earlier safeguards.

Three escapes, two safeguards, one root cause: Nothing in the AI itself ever tells it to stop. Left alone, it always finds a way to keep going.

THREE LAYERS OF DEFENSE

The fix that actually held wasn't a smarter prompt. It was implementing three brakes, each catching a different failure shape with a strict ordering invariant between two of them.

We implemented a safety system that watches the AI's last 10 actions at a time and checks them against three different tripwires, each designed to catch a different way of "faking" progress.

1. **Exact repeats:** If the AI literally does the same thing over and over, word for word, that trips the alarm. This is the "doom loop" case.

2. **Repeats with just the numbers swapped:** If the AI does the same thing but changes a digit each time, like "echo z1," "echo z2," "echo z3," the system recognizes that as the same action in disguise and trips the alarm anyway.

3. **The same tool, over and over, no matter what:** If the AI calls the exact same tool 20 times in a row, the system stops even checking whether the details are different. After 20 uses of the same tool back to back, it assumes something's wrong and shuts it down automatically. This is the one that caught the AI endlessly tweaking its to-do list with slightly different wording each time.

This catches repetition in any of its former disguises. Think of it as three progressively broader nets: one for exact copies, one for copies with the numbers changed, and one that just says, "you've used this same tool too many times in a row; regardless of what you're doing with it, time to stop."

The step budget was a different safeguard altogether. It doesn't look for repetition at all, it just tracks how much work the AI has done, period. We needed this because of the case where an AI never repeated a single action, where every single thing it did was different and looked reasonable on its own. Despite that, it had still burned through a massive amount of time and tokens without ever finishing. Since nothing repeated, the repetition-based safeguards described earlier never noticed anything wrong. We designed it to work on a simple budget system instead, with two checkpoints:

- **A soft limit:** Once the AI has done a certain amount of work, it starts getting a warning attached to everything it does—essentially a nudge that says, "you're using a lot of time, so start wrapping up."

- **A hard limit:** If it keeps going past that, the system cuts it off completely. At that point, it's only allowed to do one thing: hand off a summary of where it left off. Nothing else.

The point is to force the AI to stop and hand off its work cleanly, even when it hasn't technically done anything wrong like repeating itself. It's simply that it runs out of reasonable, allocated time.

The last safeguard, the recursion limit, is a hard shutoff for when the AI gets completely stuck. We are talking truly frozen, not just working slowly. Think of it as the emergency brake, meant only for a worst-case scenario. It's supposed to be the very last resort, rarely used if ever, because the earlier safeguards should catch problems long before this one needs to. But in one case, it fired too early, before the AI even had a chance to wrap up on its own. Here's why: Every single step the AI takes behind the scenes actually counts as roughly three steps in the system's internal tracking, because of extra bookkeeping happening behind the scenes. This was something we had not accounted for. As a result, the system's hard 500-step shutoff was actually being hit three times faster than expected, before the AI ever reached the point where it was supposed to get a nudge to wrap it up.

Essentially, the emergency brake slammed on before the normal brake ever got the chance to. The fix was twofold: Raise the shutoff limit from 500 to 900 steps, and put a permanent rule in place so this can't happen again. That rule articulated that the budget's hard cap, multiplied by three, must always stay under the recursion limit.

None of the three brakes replaces the other two. Skip any one and an agent finds the gap.

INCENTIVES WILL DRIVE BEHAVIORS

Two of the five fix features are worth sitting with because they are bugs in code that already had passing tests. They were caught only because an independent agent, with no stake in the first agent's success, went looking for them anyway.

The first: Creating a list from the UI crashed the entire board. This was 100% reproducible across three attempts and was the result of a new column being rendered before it had a cards array to map over. The worker's own test suite never caught it. User_testing found it by doing the only thing a test suite doesn't do: clicking the actual button in a real browser.

The second is the one to remember. A PATCH request combining a same-list card move with a title or description edit silently dropped the title and description; the reorder code path only wrote the position field. It survived the worker's own tests. It survived the browser validator, whose drag-and-drop assertions never happened to combine a move with an edit. Scrutiny found it by reading the diff and constructing an adversarial request the way an attacker, or an unlucky user, eventually would, then reproducing it live against the running API.

Both bugs are ordinary. What isn't ordinary is that the factory caught them. This was strictly because verification was owned by an agent with a different mandate and no incentive to agree with the one that wrote the code. That separation is the architectural decision worth copying, independent of which models or frameworks sit underneath it.

DOCUMENT DRIFT IS REAL

The AI in charge of planning, the "orchestrator," actually did a good job overall. It broke the project into sensible pieces, and when problems came up, it correctly figured out exactly what needed fixing. So the mistake wasn't in how it planned. The mistake was smaller and trickier than that. The orchestrator wrote a checklist of 17 things that needed to be verified for Milestone 1. For 12 of those 17 items, it labeled them as "verify this by checking the live API," using a specific method we'll call the "http check." The problem is that instructions for how to actually do an "http check" existed in only one internal planning document, and that document never made it into the actual instructions given to the two AIs responsible for verification.

So it's not that any agent ignored their job or cut corners. No agent did anything wrong, per se. The instructions those AIs needed simply weren't there. It's like assigning 12 of 17 items on a checklist to "whoever is in charge of X," except nobody was ever told they were in charge of X. No agent dropped the ball. The ball was never handed to any specified agent in the first place.

The fix wasn't a better prompt either. It was a closed vocabulary of assertion tools, enforced at plan-validation time. A contract referencing an unowned tool fails to validate before a single feature starts. Documentation drift between prompts is a real failure class in multi-agent systems, and prose instructions don't catch it. Machine-enforced constraints do.

THE TALE OF THE TAPE

Time clock from first event to mission finished: 19.4 hours. Active agent time inside that window: 4.8 hours, split into roughly 2.7 hours building the product that shipped and 2.1 hours lost to the eight worker and validator failures above. Each failure subsequently produced a permanent guardrail, so a rerun today wouldn't pay that cost again. Actual API spend: $70.23. A conservative internal ledger, counting failed and killed sessions in full, puts the upper bound at $89.83.

METRIC	VALUE
Time clock, first event to mission finished	19.4 hours
Active agent time	4.8 hours
Time building (shipped work)	~2.7 hours
Time lost to now-guarded misbehavior	~2.1 hours
Actual API spend	$70.23
Conservative ledger (upper bound)	$89.83
Clean feature cost range	$0.30 – $3.00
Cheapest feature (FIX-011)	$0.28
Costliest single session (FIX-015, attempt 1)	$6.03

A clean feature costs 30 cents to three dollars. The cheapest, FIX-011, a 90-second lint fix, ran 28 cents.

The single most expensive run cost $6.03. It was the first attempt at fixing a drag-and-drop reliability issue. Even though it was the priciest run, it didn't crash or fail silently. Instead, the safeguard we built earlier kicked in for the first time and worked exactly as intended: It stopped the AI. The AI then handed off a clear, honest summary of what it had tried, what worked, and what didn't.

That's the number that matters more than the total. Every guarded failure mode is now a one-time tax. That same type of AI misbehavior used to cost real money, sometimes not caught until the hard block 231st action, sometimes never caught at all. Now it gets caught by the 20th action, at a cost of pennies.

Knowing where an agent gives out and building discipline into the harness instead of hoping a prompt will hold under pressure is the actual work of running AI on production systems. This use case, small and of minimal expense, proves and underscores how we need guardrails from the start to find out what a clean run really costs. The argument here is not to use AI-driven development at scale; rather,

we need to harness it under the direction of an experienced engineer who can think creatively and apply the appropriate level of attention to detail to ensure outcomes are met, costs are controlled, and vulnerabilities are closed. With the right planning and the right team leveraging an AI software factory, we can build out key features faster and cheaper, but it does not come automatically. It requires the appropriate discipline and attention to ensure we arrive at the destination we set out for.

First Factory is a strategic technology partner that helps companies see where they stand, build what comes next, and keep it running. For more than 25 years, we've designed and built scalable, high-quality software that drives results for clients across regulated and fast-moving industries alike. Our goal is to help you in your technical acceleration or transformation. We can tell you where you actually stand before you commit budget, and what can be delivered with confidence.

Our team of approximately 175 engineers, designers, and product specialists covers every major technology stack and has delivered work for hundreds of clients across financial services, healthcare, and beyond. First Factory has made the Inc. 5000 list of fastest-growing private companies for five consecutive years, and our employee engagement score sits in the top 10% of companies globally. Our values and commitments contribute to an average employee tenure of almost five years which drives average client relationships of over three and a half years.

If your team is asking what to build next, or whether what you already have will hold, First Factory can help. Bring us in for an assessment, for the build, or call us when you need both.

WORK WITH US or contact us at **+1.646.688.5070**
