

White paper

We let a swarm of AI agents rewrite our own app

and it changed how we think about iso-functional rewrites

A field report from an experiment: rewriting one of our own applications with a governed, multi-agent pipeline. What worked, what didn't, and the numbers behind it.

Gilles Stragier, co-founder 5th floor



Introduction

For years, our default answer to "should we rewrite this application as-is, feature-for-feature?" was no. Iso-functional rewrites, rebuilding a system so its observable behaviour is identical, screen for screen, rule for rule, looked like the worst trade in software: all of the risk of touching a load-bearing system, little of the value, because at the end you have exactly what you started with.

This is a report about an experiment that made us less sure of that reflex.

We're careful with the word experiment. What follows is one project, not a proof. But it moved our thinking enough that we want to write it down, including the parts that didn't work. The short version: done correctly, an iso-functional rewrite might not be a dead end but a first step that de-risks everything after it. You rebuild the behaviour you already trust, on a foundation you now control, and then you modernise in small increments. Big-bang modernisation is still a bad idea. But framed this way, iso-functional stops being a cost and becomes the ground you stand on, and AI may be what finally tips its risk/value balance.

There's a second, more technical lesson underneath the first, and it's the one we're most confident about: **when you hand a rewrite to AI, the hard part isn't generating code. It's stopping the model from inventing.**

The starting point

We ran the experiment on ourselves, on one of our own internal applications, whose interface was a Streamlit app we wanted to move to Angular. Using our own tool made it a safe guinea pig: real behaviour to preserve, and no client exposed while we learned.

Here's the nuance that matters. The constraint was never "be iso-functional." The constraint was to get there, to actually ship a working Angular replacement. Iso-functional was the method we chose. And the reason that method works with AI turned out to be the whole point: the existing code already contains enough information to frame the model. An AI hallucinates in the empty spaces; a faithful rewrite of a working system leaves very few of them. You're not asking the model to imagine what the software should do, you're asking it to restate what the software already, provably, does.

The target we set ourselves was 10 working days. The real question wasn't "can AI write Angular?", it can, but "can a swarm of agents carry a rewrite end-to-end, under governance, on their own?"

The real loop: code → text → human → AI → code

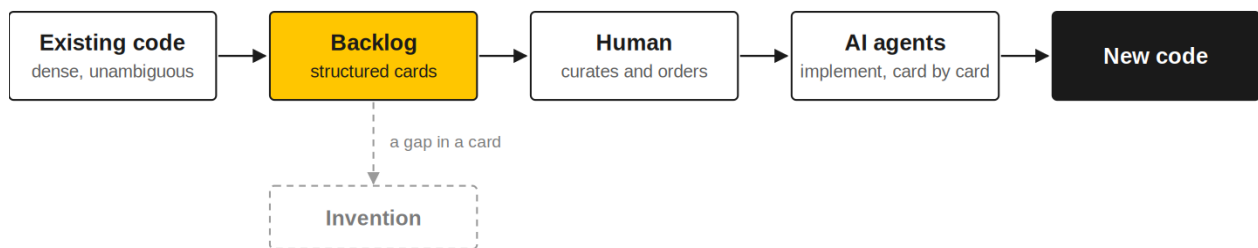
The mechanism that made it work is a loop, and naming it precisely matters: code → text → human → AI → code. The existing code is distilled into text, a backlog of implementation plans. A human curates and orders that text. The AI turns each piece back into code. The existing system is the source of truth at one end, the new system is the deliverable at the other, and the backlog in the middle is where all the judgement lives.

Which is why the structure of the backlog, not merely its existence, is the hard, decisive part. Each card had to be a self-sufficient plan, written as if for an AI with no business context and no access to the source repo:

An incomplete card is worse than no card, it generates invention.

Where a card was complete, the agent produced faithful code. Where it had a gap, the model didn't stop and ask, it filled the gap, plausibly and wrongly. So a card became a small spec: the objective, the source behaviour cited down to file:line, the API contract, mechanically checkable acceptance criteria, and Playwright scenarios. One unit, a component, a service, a route, per card.

And the genuinely hard structural problem was dependencies between cards: card B can't be built before card A, and a swarm working in parallel will happily start B anyway. Getting the order right, and keeping a few cards in flight at a time, not thirty, is not trivial, but it is tractable with a backlog tool that models blockers and states properly.

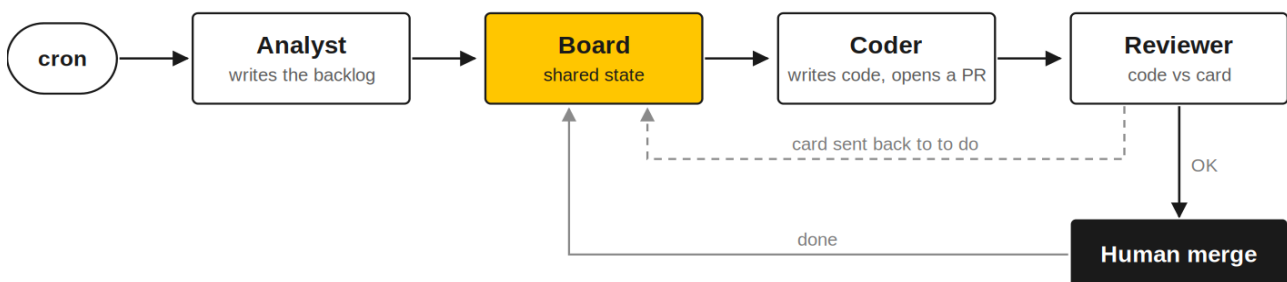


Existing code becomes a structured backlog, curated by a human and implemented by AI into new code; a gap in a card leads to invention

A swarm coordinated by state, not by chat

The rewrite was driven by a small fleet of agents, running on a self-hosted, open-source agent gateway. Three roles mattered:

- The analyst reads the existing app and writes the backlog, the implementation plans, the priorities, the parallelism.
- The coder picks up a card, spawns sub-agents with deliberately reduced context, writes the Angular code, runs the tests, and opens a pull request.
- The reviewer checks the CI is green, resolves merge conflicts, and, crucially, checks that the produced code actually matches the card. If not, it sends the card back.



The agents coordinate through a shared board and pull requests, not by talking to each other; the reviewer sends failing cards back and a human merges

The design decision we'd repeat: the agents don't talk to each other. They coordinate entirely through state, card statuses and pull requests. A cron job wakes them on a schedule; each reads the board, does its job, and writes a fact back. Coordination is boring, deterministic plumbing; only the work inside each card is agentic. (A principle we've since sharpened for the successor system: deterministic where you need safety, agentic where you need intelligence.)

The safety net that kept the AI honest

The pipeline around the agents mattered as much as the agents. Every card became one pull request, and every pull request got its own ephemeral environment, a full instance, on its own URL, one-click deployed and destroyed on merge, with an end-to-end test suite on top.

Let's be precise about what this did, because it's easy to overclaim. We did not prove that the old and new applications pass the same tests, that's a stronger statement than we earned. What the pipeline gave us was regression safety during development: as the swarm changed the app, card after card, the end-to-end tests stopped it from breaking what already worked. That alone was worth a great deal, it is what let us move fast without silently rotting the parts already done.

The real bugs showed up in manual testing, and they were almost always the same shape: not bad code, but bad specification, a gap in a card that the model had papered over. After the rewrite was finished, we found roughly ten bugs over a few months of real use, a modest number for a system rebuilt this way.

And none of it is as tidy as a diagram makes it look. You begin a rewrite without fully knowing the system you're rebuilding, that is the whole reason it is hard. What stops it from becoming the familiar "six months that quietly turn into five years" is one unglamorous rule: no functional changes during the rewrite. Freeze the behaviour. With the running system as the reference, there is no moving target and no fresh round of business arbitration to stall on, the delivery team can simply execute. That discipline is doing as much of the work here as the AI.

One honest caveat on the pipeline itself: a PR flow that goes this far, a live, isolated instance per pull request, asks for real maturity in how you operate your target environment. The polished result is only the tip of the iceberg; underneath sits a good deal of continuous-delivery discipline. It is a prerequisite, not a detail.

Three attempts, honestly

It was not push-button. A teammate bootstrapped the project by hand, the Angular scaffolding, the Docker/nginx build, the design system, the test harness, and then piloted the swarm. It took us three passes.

Attempt 1, one agent, a coarse backlog. A functional app that was incomplete and had invented features. The backlog was too high-level; the model filled the blanks.

Attempt 2, multiple agents, plus a Figma design. Here we got greedy: we wired in a design and tried for something better than the original, not just equal to it. It backfired in the most instructive way. A Figma that wasn't quite exact, mixed into the coding loop, pulled the agents away from the existing behaviour, they followed the design and invented to fill the difference. The lesson: you can't do iso-functional and redesign in the same motion through an imprecise reference. (What makes it almost funny is that our own analysis already said it, in bold, "the existing app is the source of truth; the design is layout inspiration only; do not invent." Writing the rule down is not the same as being able to hold it.)

Attempt 3, design unplugged, backlog exhaustive, humans on the PRs. Back to strict iso-functionality, the most complete backlog we could write, and a human validation gate on every pull request. This is the one that produced an iso-functional Angular app, a few minor bugs, a cleaner design than the original, real test suites.

And the honest gap: we still believe code → exhaustive backlog → intervene on the backlog is the path to controlled, incremental improvement, the backlog becoming the surface where you inject the "better than before" changes without falling back into chaos. We believe it strongly. We did not demonstrate it here. We're leaving that flag planted where it is.

One more boundary, stated plainly: this was a frontend. A backend carrying heavier business logic may well be harder, we haven't tried it, and we won't pretend the result generalises for free.

The numbers

Planned effort	10 working days
Model cost	~\$1,200
Backlog	11 epics, ~90 cards (mostly done)
Existing Streamlit UI	~12,500 lines across 56 files
New Angular app	~23,000 lines of TypeScript, 58 components, 9 API areas reproduced
Tests	56 unit + 37 end-to-end specs · coverage bar ≥ 80%
Bugs found after completion	~10 over a few months

A word on speed, measured crudely and worth exactly that. Strip out the long pauses between the three attempts and the real work was two concentrated weeks. Over that burst the swarm touched code at roughly twice the daily rate of a comparable, human-built module of ours, and because it reworked far less of what it wrote, more of each day's output survived. Call it one-to-two times a team's pace: a soft signal, not a benchmark.

The flip side, the invention we did see in practice, was concrete: the AI offered 3 language options where the existing app had 8 (it guessed) and added a status badge to a list that never had one. Both were caught, by the reviewer agent and by humans, and pinned back to the source of truth. Everyone traced to a gap in a card. That loop, catching invention and pinning it to what the existing code actually does, is the whole game.

What worked, what didn't

Worked. The PR + ephemeral-environment + end-to-end pipeline. Code quality held, the agents followed the project's engineering guidelines. And the workflow ran almost autonomously, up to the pull request.

Didn't. Generating a genuinely good backlog is hard, and it is the real bottleneck. Mixing iso-functional with redesign cost us a whole attempt. And, honestly, collaborating with the AI, knowing when to trust it, when to re-scope, is a real skill, not a given.

Going further

The version we ran was a research rig, and we've since retired it. Its clearest lesson pointed straight at the successor we're designing let deterministic code own coordination and safety, let agents own the intelligence inside a well-specified task, and never let a model move the state. Once issue-to-PR is automated, the bottleneck simply moves, to human review, which is a much better problem to have.

So, would we do an iso-functional rewrite now?



Sometimes, and that's the shift. Not as an end in itself, but as the de-risking first step of a modernisation: rebuild the behaviour you trust, wrap it in tests that keep it honest, and earn the right to change it incrementally afterwards. The tooling to make that economical is arriving faster than the caution around rewrites has updated. We're still cautious. We're just no longer saying no by default.

And perhaps that's the quieter prize. The behaviour that used to live only in the running code, and in a few people's heads, now also lives in a backlog and a suite of tests the whole team can read. Modernising it is the next question; being able to hold it is the one we answered here.

