

From chats to a multi-agent system

How, in ten weeks, working with AI became a managed system.



Role: architect and operator of the system all my work now runs on.

Period: 2026, ten weeks from the first chat to a working system.

Scale: 13 projects on one standard, agents with roles, memory across conversations, my own rules of behaviour for AI.

Context: a personal lab where I test what AI can really do at work. Report: report.rostyslavdanylenko.com.

The AI replies: "Document read." I ask about section three. A pause, then a confident answer that isn't in the document. It hadn't read it: it skimmed the surface and filled the rest with a guess.

Moments like this happened every day, and every time it turned out that skill had nothing to do with it. The AI lacked memory, rules that would hold, and a way to check itself.

It all started with a simple task for a client: show what AI can do in a design process. To answer honestly, I had to test it myself. Over ten weeks the testing grew into a lab, and the lab into a system that all my work now runs on. It stands on three principles, and this case follows them:

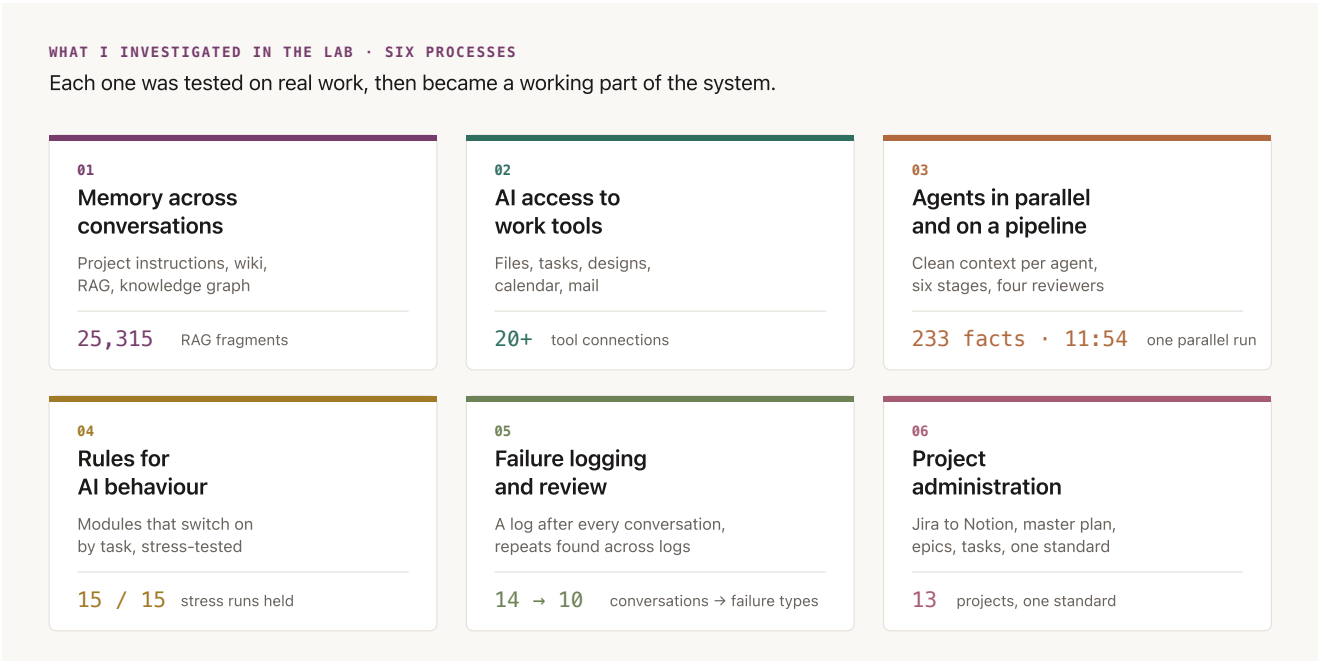
- every claim needs a source, otherwise it's a guess;
- a rule has to be a mechanism, otherwise it isn't followed;
- an AI error has to become data, otherwise it will repeat.

Role and environment

I'm the architect and operator of this environment: I choose the tools, write the rules of behaviour for AI, build the memory, and own quality control. Design is the first proving ground here: every mechanism first

worked on a real design-system task, then on every other project. There's no human team in the lab; agents are part of the method. Everything is built for one person, but the mechanisms carry over to a team.

The work is split into projects, and each has an instruction the AI reads on its own at the start: whose project it is, where its files live, which rules are binding. While the AI lived in a chat window, files, mockups, tasks, and the calendar sat behind glass, and I carried every result across by hand. Now more than 20 connections let it reach the work itself.



Six processes I investigated in the lab. Each became a working part of the system.

More in the report: [Stack: what exists, what does not](#)

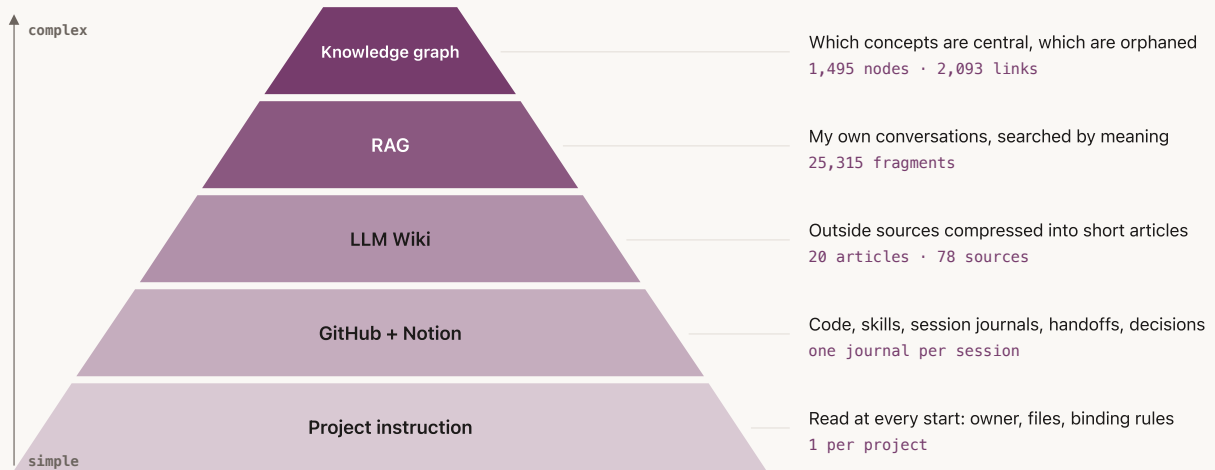
1. Every claim needs a source

You can check an answer when you can see where it came from. But at the start the AI didn't even have memory. Every conversation was the first one: I explained again and again what the project was and what had already been decided. Once there were dozens of conversations, they started contradicting each other.

A project instruction gives rules, but it doesn't remember what happened yesterday. So on top of it I built memory in layers, and each next layer keeps what the one below can't hold.

MEMORY IN FIVE LAYERS · SIMPLE AT THE BASE, COMPLEX AT THE TOP

Each layer remembers something the one below it cannot.



Personal content is filtered out before it reaches the RAG index, not after.

Scale figures are counts from build logs and graph snapshots, not quality measures.

Now a new conversation finds what was decided a month ago and shows how it knows. The answer arrives with its source, so it can be checked.

The next limit was volume. A large task didn't fit into one conversation: by the time the AI read the end, it had forgotten the beginning. I tested the way out in a study: four agents read the archive in parallel, each in its own clean context, and in 11 minutes 54 seconds returned 233 facts, each with a source. The first run that same morning produced zero usable files: the agents lacked access, and instead of stopping they quietly took a different source.

Since then, a figure without a source next to it counts as unmeasured for me. The material for this case was gathered the same way: seven agents read everything behind it, and I brought their findings together.

The same principle applies to me. One of the connections started hanging, and the easiest thing was to blame the tool. I investigated the cause: the computer was short on RAM and was running on slow swap. Now a hang doesn't mean the action didn't happen: I check first.

For a team, this answers the question of how to give it AI whose results can be checked.

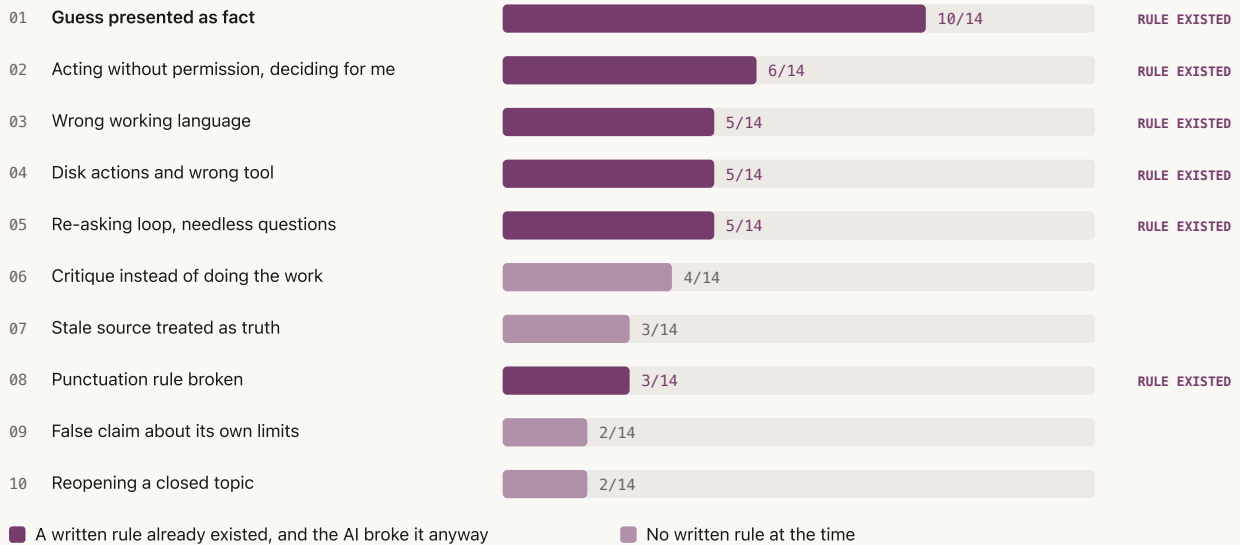
More in the report: [Memory](#) · [Execution](#)

2. A rule has to be a mechanism

The strangest finding was waiting in the logs. After every working conversation I wrote down what went wrong, and then I brought together the logs from 14 conversations. The most frequent failure turned out to be that same guess presented as fact: 10 conversations out of 14. A rule against it had already been written, as it had against most of the other failure types. The AI knew the rules and broke them anyway.

14 CONVERSATIONS · 10 FAILURE TYPES · 6 ALREADY HAD A WRITTEN RULE

How many of the 14 conversations each failure type showed up in.



The finding: the problem was not missing knowledge. The fix is a procedure that stops a rule from slipping, not another rule.

From the failure review of 14 conversation logs, 02.08.2026. Each conversation wrote its own log.

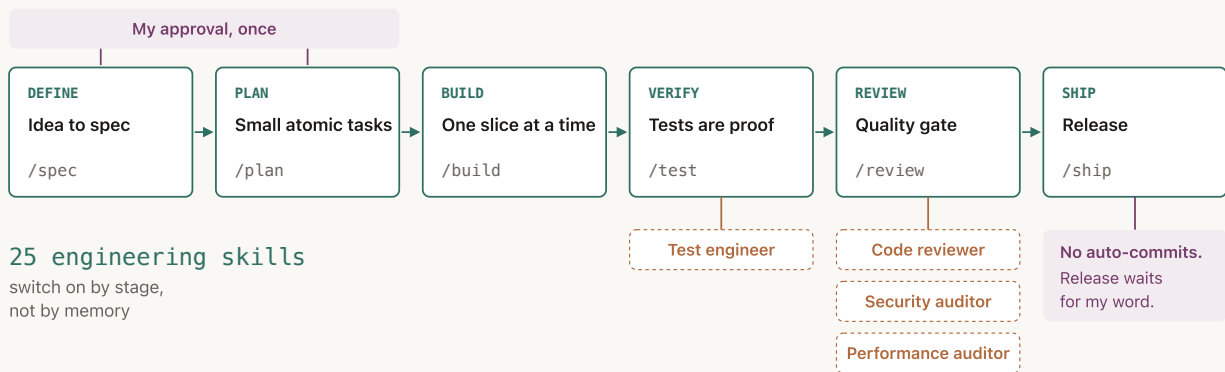
So the problem is where a rule lives. Written in the chat, it exists until that chat ends, and in a long conversation it gets lost even earlier. By the sixth hour the AI is working from a retelling of its own morning, the rules quietly slip, and when it starts telling you you're a genius, it's already adjusting to you, and the task takes a back seat. That's why for me one task equals one conversation.

I moved the rules out of the chat into separate modules that switch on by themselves when a task matches their description. Eight modules carry the work from start to finish. Each goes through a check that resembles a usability test: a clean conversation, identical prompts, and a provocation that deliberately pushes toward a violation. One of them held 15 attempts out of 15. The approach has a limit: a module fires only on its own task. A rule that must always apply has to live where it is always read.

Where the cost of an error is high, a rule isn't enough: an irreversible action needs my permission. I'm now rolling out an engineering pipeline where the rules are built into the process itself. It has six stages, each switches on its own modules, and at the checkpoints the result is reviewed by four separate reviewer agents. I approve the spec and the plan once; after that the agents carry the task themselves. There are no automatic commits: release waits for my word.

AGENTS ON AN AGREED PIPELINE · SIX STAGES · FOUR REVIEWING ROLES

I approve the spec and the plan once. After that the agents carry the task through every stage themselves.



25 engineering skills

switch on by stage,
not by memory

WHERE IT STARTED

A study on 12.08: four agents read in parallel, each in a clean context, and returned 233 sourced facts in 11 min 54 s.

The first run that morning returned zero usable files: agents without access did not stop, they quietly swapped the source.

Pipeline from the agent-skills package now being integrated. The 12.08 study sits underneath as the first proof of the principle.

For a team, this is a way to keep its rules from slipping in long work.

More in the report: [What went wrong](#) · [Control](#)

3. An AI error has to become data

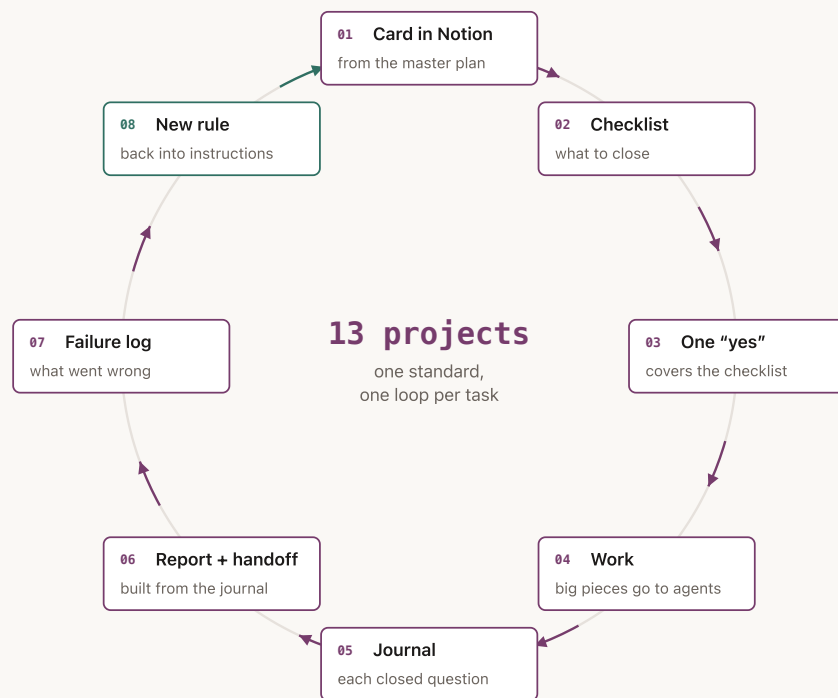
I kept the failure log after every conversation: it's what shows the repeats you can't see from inside a single chat. And once there were six projects, I started losing track: where the plan is, what's in progress, what was decided last week. Every new conversation asked about it all over again, and I was the one answering.

That's when the orchestrator appeared: a separate project that administers all the others. It sets the standards, templates, and rules, and writes neither code nor text itself. Every project starts from a master plan, epics grow out of it, tasks out of the epics, and every task in Notion goes the same way from "not started" to "done".

Inside a task there are three key points. The conversation takes a card from Notion and breaks it into questions to close, and I approve the list with a single "yes"; from there the AI works without asking again. Every closed question goes straight into the journal, so the report at the end is assembled from the journal. And the failure log goes to the orchestrator: it turns repeats into new rules and puts them into the instructions every next conversation reads at the start.

ONE TASK, ONE LOOP · THE SAME FOR ALL 13 PROJECTS

Like a sprint cycle: what the orchestrator learns from one task goes into the instructions for the next.



Step 8 closes the loop: the new rule lands in project instructions, which every next conversation reads at the start.

Now I don't have to keep the state of 13 projects in my head. It lives in the databases and journals, and any new conversation reads it from there.

For a team, this is a way to turn errors into a process instead of hallway talk, and to run a dozen parallel streams without losing context.

More in the report: [Control](#)

What changed

I now check "Document read" with one question about a specific place in the text: it shows whether the AI read it or skimmed the surface. A separate module handles sources, errors go into the log, and repeats come back as new rules.

The client received a full study of how designers can optimise a department's work with AI. I gained invaluable experience building a business system with a heavy project load in an agentic environment, on a single standard that can be carried over to any organisation with many parallel processes in the shortest possible time. A method proven in practice that delivers simple value:

- speed up project work with agents that run in parallel and, after a single approval, carry the task themselves;
- cut company costs by automating manual work: 13 parallel projects are run by one person with agents;
- standardise production processes, because every task in every project follows the same path, and the rules switch on by themselves and hold even through long AI work.

And one last lesson: the feeling that the system finally works perfectly is an alarm. Most often it means something has broken, and you just don't see it yet.

The full report

This case tells the story of the system in brief. The full report is open: it has the method, the data with sources, and the sections that didn't make it in here.

ORCHESTRATOR

From chats to a multi-agent system

INTRODUCTION

What this report is about

Structure of the report

Method of work

Stack: what exists, what does not

PART A · COMPONENTS

Control

Memory

Execution

Production

PART A · SUMMARY

Role tree

What went wrong

Ten weeks in numbers

PART B · PROJECTION

Where new ideas come from

Publishing outward

INTRODUCTION

Ten weeks: from ordinary chats to a system that runs itself

30 May ~ 11 August 2026. One developer, one computer, no budget for tools. This is what came out of it, what it cost, and where it still does not work.

THESIS

A rule is not followed until it becomes a mechanism.

Six classes of error had a rule already written down, and recurred anyway. The same rule, turned into a skill, fired 15 times out of 15. A figure with no source was caught by a check, not by good intentions. That is what the client can take from this: a way of turning policy into mechanism.

PART A · EVIDENCE

What already works

Every figure carries its source: a command run on disk, or a quote from the archive with a date and a conversation name. No source means the document says "not measured".

Four groups of components, a role tree, three post-mortems of my own failures, and a summary of ten weeks.

PART B · PROJECTION

What could be built from it

There is no evidence here and there will be none. Every scenario comes with a precondition and a way to check whether it worked.

The intake loop, the publication pipeline, six directions for the department, a ladder of levels, a pilot design, and the limits.

HOW TO READ THIS

SECTIONWHAT IS IN ITWHO IT IS FOR

What this report is about

From chats to a multi-agent system · full report: report.rostyslavydanylenko.com

Sections not covered in this case:

- **Method of work:** how the report was assembled, and why every figure in it has a source.
- **Production:** the design system, the path from design to code, and accessibility: real client work where the mechanisms were tested.
- **Role tree:** 22 roles in one scope of work: what a person covers, what agents cover, and what nobody covers yet.
- **Ten weeks in numbers:** ten weeks in figures.
- **Where new ideas come from:** how new ideas enter the system.
- **Publishing outward:** how the results of the work go out.
- **What this means for the client:** what a client can take from this system.