[7]

a[7]t · PLAYBOOK

Three-Body Agent Playbook

Blueprints for autonomous frontier LLM pipelines

Leonardo Cardoso

V0.1 (2026-05-18)

a7t.ai

Contents

CHAPTER 01

Why three bodies

Before there was a pipeline, there was IGNIO.

IGNIO was a personal-finance assistant that lived inside WhatsApp. You sent it a photo of a receipt, a voice note, a forwarded invoice; it parsed the thing, categorized the expense, and let you ask questions about your money in plain language. Under that single chat thread sat a monorepo: a Fastify API, a queue of background workers running OCR and speech recognition, a Postgres database with fuzzy-search extensions, a multi-agent core that classified intent and routed it, an i18n layer, a marketing site. I designed it, built it, deployed it, and answered the support messages. Every layer was mine.

The problem with running every layer yourself is not that the work is hard. It is that there is exactly one of you. IGNIO never stalled on ideas; the backlog was full of them. It stalled on throughput. A week holds only so many evenings, and every feature wanted design, implementation, review, and a release before it counted for anything. The move was obvious: delegate. The hard part was deciding what to delegate it to. Hiring meant onboarding a stranger into a codebase only I understood, then watching my conventions, my naming, the hundred small decisions that made IGNIO feel like one coherent thing get sanded flat by a second set of hands. I did not want more hands. I wanted more of mine.

So I delegated to a model. That pipeline became Three-Body Agent. In a week, it took roughly 240 issues from open ticket to merged pull request at about a 95% success rate. It runs against my iOS apps now; IGNIO itself is no longer online, because products end for reasons that have nothing to do with the tools that built them. The architecture the IGNIO problem forced into existence outlived the product, and it is the subject of this book. It did not begin as three bodies; it began, the way these things do, as one.

The first attempt was one agent

The first version is the one everyone writes: a single agent, and a prompt that says *here is the ticket, here is the repository, open a pull request*. It worked often enough to be dangerous: the code compiled, and the tests, where they existed, passed. But reading the diffs back, two things were wrong every time.

The first was quieter than a bug. The code was not mine. It solved each ticket the way an average of the public internet would solve it: a helper extracted where I would have inlined, an abstraction added for a case that would never arrive, error handling bolted onto a path that could not fail. Nothing a reviewer could circle and call incorrect. Just the slow laundering of the one thing I had refused to hire away: the taste, the house style, the essence. Delegation had not preserved my standards. It had quietly averaged them out.

The second failure was structural. When one agent writes a change and that same agent, in that same context, is then asked whether the change is good, it is not reviewing anything. It is agreeing with itself. A model that has just argued its way into a design will not turn around and take the design apart; the inference that produced the mistake is the inference now grading it. The loop had no seam. A single wrong call at any point rode the whole way to the main branch, because nothing downstream of it had either the independence or the mandate to stop it.

Isolation, not a better prompt

Here is the operating principle the rest of this book is built on: one bad inference, in one role, cannot be allowed to take the loop down.

An autonomous pipeline does not fail because the model is sometimes wrong. The model is *always* sometimes wrong; that is a constant, not a defect you can prompt away. The pipeline fails when one wrong inference has nothing standing between it and main. The answer is not a sharper prompt. It is isolation: split the loop into roles with bounded

jobs, so the blast radius of any single mistake is one role, and so every role's output must survive a different role that did not produce it.

I should have arrived here faster, because IGNIO already worked this way. Its core was never one model told to *handle the user*. It was an orchestrator that classified intent and a set of specialist agents under it, each with its own narrow toolset, precisely because a single model holding the entire job turned vague and unreliable exactly where precision mattered. I had already accepted that a hard task splits better across narrow, isolated agents than it survives in one. It simply took me too long to point that same conviction at the pipeline building IGNIO rather than at IGNIO itself.

Why exactly three

Three is not a slogan; it is the smallest split that holds. The loop has three irreducible jobs, and each one becomes a body.

The **Implementer** writes the change, and writes it in my style: it reads the spec and the codebase, plans in writing before it touches anything, and commits once per acceptance criterion. It is bounded to a single ticket, and it never reviews its own work.

The **Fixer** arrives at the change cold. It takes the findings from review, isolates the root cause instead of patching the symptom, and repairs it. The fresh context is the entire point: the Fixer is deliberately not the role that wrote the code.

The **Merger** decides one thing. It reads the verdict rather than the diff and answers MERGE or SKIP. It is the only role permitted to ship, and it holds the bar steady so that no earlier optimism can lower it.

Two roles is the tempting cut: implement, then review. But a review that no one acts on is just a comment, and an implementer that acts on its own review is back to grading itself. You need a hand to act on the findings that is not the hand that wrote the code, and a decision to ship that is invested in neither. Collapse any of the three and the self-judgment problem grows straight back. Add a fourth, a fifth, a sixth, and

you are mostly buying context handoffs: every role boundary is a place where state has to be re-serialized and re-explained, at a cost. Three is the floor that still gives you independent implementation, independent repair, and an independent decision to ship.

> *The external reviewer that feeds the Fixer is a fourth party, and a deliberate one, but it is not a body. Chapter 4 covers why the review that gates the loop is kept outside the three roles, not folded into them.*

From here the book takes one body per chapter, then the machinery that moves work between them. If you read only one more chapter, read the next one: the Implementer is where the essence is kept or lost, and every other role in the system runs downstream of that.

CHAPTER 02

The Implementer

Of the three bodies, only the Implementer originates a change. The Fixer touches code that review or CI has already flagged; the Merger writes nothing at all. Which is why Chapter 1 ended where it did: the Implementer is where the essence is kept or lost. Conventions, structure, the small decisions that make a codebase coherent: if they are going to survive delegation, they survive in this role, because this is the role that writes the lines.

The Implementer is governed by one file, `implementer.md`, public in the repo. It opens without ceremony: *"You are an autonomous implementer agent. Follow these steps IN ORDER. Do not skip steps. Do not ask for human input."* Six numbered steps follow. This chapter walks the disciplines they encode, and why each one is load-bearing rather than decoration.

Read before you write

Step 0 is not "write code." It is *"Read ALL documentation for full context BEFORE writing any code"*: the stack, the patterns, the naming, the structure, and then everything in the linked issue. Context-gathering happens once, comprehensively, before any tool call that changes a file.

This is the cheapest discipline in the pipeline, and it has the largest effect on whether a pull request is mergeable. An agent that skips it does not fail loudly; it invents. It extracts a helper where the repo would inline, picks a name three degrees off the local convention, reaches for a pattern the codebase already rejected. None of that is a bug, and none of it fails CI. It is the essence-laundering from Chapter 1, arriving one small decision at a time. Reading first is the anchor that keeps an autonomous PR feeling native instead of feeling like a bot dropped it in.