Orchestrating AI Agents

BUILDING A VENDOR-NEUTRAL CONTRAL PLANE
FOR MULTI-AGENT SOFTWARE DEVELOPMENT PIPELINES

by YOHAN J. RODRÍGUEZ

Preface

"The future is already here — it's just not evenly distributed."
William Gibson

A year ago, "using an AI coding agent" meant opening one tool. Today most developers have several: Claude Code, Codex, Antigravity, an editor extension, a local model or two, and a pile of MCP servers behind them. Each is good at something. None of them remembers anything between runs, schedules itself, shares what it learned, or decides which tool should handle a given task. The hard problem has quietly shifted from *which agent* to *how do I operate all of them together*.

This book is a practical, vendor-neutral guide to that operating layer. It is written for engineers, platform builders, and technical founders who already run more than one agent and want to coordinate them deliberately, by routing each unit of work to the right tool, giving the system a memory that outlives any single session, and letting it run unattended without losing control. You do not need to commit to one vendor. You do need to be comfortable on a command line and willing to think in systems. It is not a tutorial for any single agent.

The organizing idea is simple and recurs throughout. The place an agent runs its commands is just a shell, and every serious coding agent offers a non-interactive mode. So a persistent control plane can invoke any command-line agent as a tool, and once it can, the interesting work becomes routing, memory, skills, and guardrails rather than raw code generation. The book's answer to the temptation to build one super-agent is to orchestrate rather than replace. The control plane should be a modest coder and an excellent operator.

The book has four parts. **Part I** states the problem and the thesis. **Part II** introduces the pieces: a dated survey of the execution agents, read on a small set of axes that matter for orchestration; a concrete control plane, Hermes, used as the worked example; and the delegation primitive that lets the control plane drive a headless agent, with briefs, validation, and write-back. **Part III** covers the mechanics: a routing engine built on four axes, three outcomes, and one absolute rule; layered memory shared across every tool; skills as a capability registry; MCP as the shared tool layer; and local-first routing behind a model gateway. **Part IV** is about operating the result:

sandboxing and guardrails, scheduling and a multi-agent work queue, a full worked orchestration, budgets and spend governance, a catalogue of failure modes with the discipline of escalation, and a closing maturity arc that ends at an honest ceiling.

You can read the book straight through, and the order matters, because each mechanism leans on the ones before it. Chapter 2 ends with the smallest possible orchestration: three short shell scripts that delegate a task to a headless agent, record the outcome, and read the record back. Every later mechanism is a way of making one of those seeds sturdier. If you would rather start from your own situation, the closing chapter locates a setup on a four-phase arc and shows how the path differs for a solo operator, a startup, a consultancy, and an enterprise.

Hermes appears throughout because a worked example teaches better than an abstraction, but the point is never the product. Where a detail is specific to it, the text says so, and the patterns are written so that they survive whichever tools you actually use. The same caution applies to everything time-sensitive. Tools change every few months, so the book dates its claims, its survey of the agents separates what is official from what was observed and what is inferred, and flags, models, and vendor behavior are examples to confirm against current documentation. The listings are short shell, YAML, SQL, and Python sketches that show a mechanism, and many are labeled illustrative. Each chapter ends with key takeaways, and most also list the mistakes to avoid and include a hands-on exercise.

One conviction shapes the second half more than any other: autonomy is earned, not assumed. The book argues that high-stakes work always reaches a human, that agents may prepare while people authorize, and that production, money, and security stay above a permanent line. That is not a limit of today's models. It is where accountability sits. The goal is not to remove the human from the loop but to move the human up to the decisions that need one.

Yohan J. Rodriguez

Contents

Preface i

1 The Multi-Tool Problem **1**

1.1 The Year Everyone Got a Fleet 1
1.2 A Tuesday With Four Agents 2
1.3 Five Things Nothing Owns 3
1.4 Why "Which Agent Is Best?" Is the Wrong Question 5
1.5 Why the Problem Stayed Invisible 6
1.6 What an Operating Layer Would Do 6
1.7 Hands-On: Audit Your Own Agent Stack 7
1.8 The Operator's Mindset 7
1.9 What This Book Builds 9

1 | The Multi-Tool Problem

A year ago, "using an AI coding agent" meant opening one tool and talking to it. That era is already over. Open a terminal on a working developer's machine today and you are likely to find Claude Code, Codex, perhaps Antigravity, an editor extension or two, a local model served by Ollama, and a handful of MCP servers wired in behind them. Each of these is genuinely good at something. None of them, on their own, add up to a system.

This chapter is about the gap that fragmentation opens, and why closing it — rather than hunting for a single perfect tool — is the real work of the next few years. It is the problem the rest of the book sets out to solve.

1.1 The Year Everyone Got a Fleet

The proliferation was not planned. It accreted. A developer adopts Claude Code for hard multi-file reasoning, reaches for Codex when an OpenAI-ecosystem task or a long autonomous run fits better, keeps a local model around for the routine or the private work, and installs an MCP server the first time an agent needs to touch a real system. Each decision is locally sensible. The sum is a drawer full of sharp, single-purpose tools that have never been introduced to one another. Figure 1.1 frames the fragmentation that appears when each useful tool owns its own context, memory, workflow, and outcome.

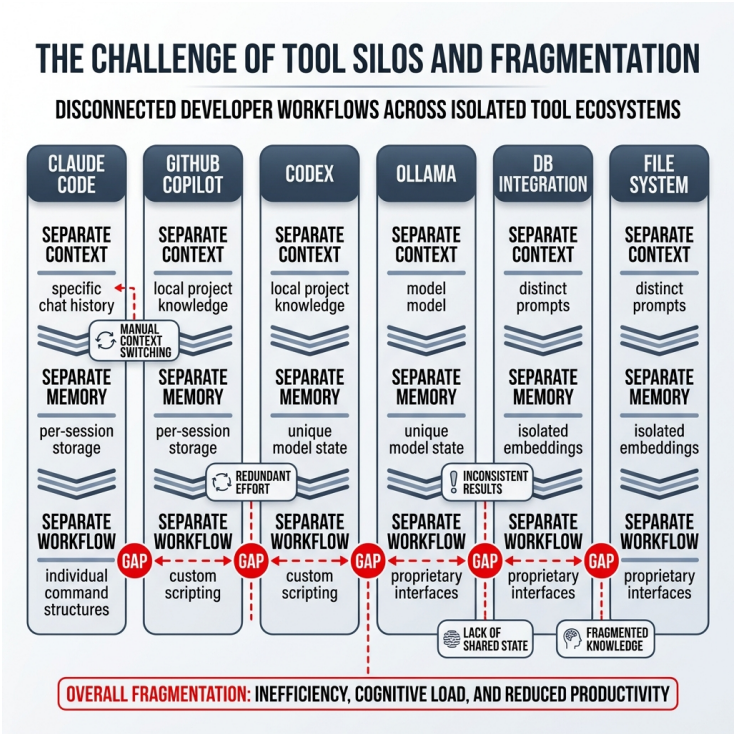

Figure 1.1: Modern AI developer tools are powerful, but each one often carries its own isolated context, memory, workflow, and execution path.

The fragmentation in Figure 1.1 becomes an operational problem whenever work crosses a tool boundary. Context must be copied by hand, assumptions must be restated, and evidence from one execution environment can disappear before it reaches the next.

The tools share a deeper shape than their branding suggests. Almost all of them are *per-invocation*: you start one, it does a unit of work, it exits, and everything it learned evaporates. They are *single-vendor*: Claude Code speaks to Anthropic, Codex to OpenAI, Antigravity to Google, and none of them will route to the others. And they are *shell-bound*: they run their commands in whatever terminal you launched them in, with no notion of where else that work could run or who else might want the result.

Hold that last property in mind. It looks like a limitation. By Chapter 2 it will turn out to be the opening through which the whole solution arrives.

1.2 A Tuesday With Four Agents

The abstract claim that fragmentation costs something becomes vivid when you watch an ordinary day. Consider a developer — call her a competent, well-equipped engineer with every modern tool installed — working through a Tuesday.

In the morning she opens Claude Code to work through a gnarly refactor in the billing service. It spends forty minutes building up an understanding of the module: which functions call which,

where the tax logic lives, why a particular abstraction exists. The refactor lands, the tests pass, she closes the session. That forty minutes of hard-won understanding is now gone; the next time anyone touches that module, an agent will rebuild it from scratch.

Before lunch she switches to a different repository — the marketing site — and reaches for Codex, because that is the tool she happens to have wired into that project. It hits a bug that is, in essence, the same class of bug she solved in the billing service three weeks ago. Neither agent knows that, because what one learned in one repository is invisible to the other. She solves it again, by hand, slightly differently.

After lunch she means to run the weekly dependency audit across her services. She does not, because running it requires her to be present, to start it, to watch it, and the afternoon fills with other things. The audit is not hard; it is simply something no tool will do unless a human initiates it, and the human had a meeting. The audit slips, as it slipped last week.

In the late afternoon she needs to update a WordPress site through an agent that can reach it over a tool server, and she spends ten minutes remembering which of her several agents has that integration configured and what the incantation was. The capability exists. It is just not catalogued anywhere she can ask.

None of these is a failure of any individual tool. Claude Code refactored well; Codex fixed the bug; the audit script works; the integration is solid. Every tool did its job. And yet the day leaked value at every seam between them — knowledge that evaporated, work repeated, maintenance skipped, capability mislaid. That leakage is the multi-tool problem, and it is not visible if you evaluate the tools one at a time. It is only visible across the day, which is exactly the altitude at which no tool operates.

1.3 Five Things Nothing Owns

The quickest way to feel the gap is to name the jobs that no tool in the drawer is responsible for. There are five, and they recur throughout this book.

Persistence. Every session starts from nothing. The model that spent an hour learning the shape of your codebase yesterday begins today as a stranger. Context files such as `CLAUDE.md` and `AGENTS.md` help, but the procedural knowledge — how this project builds, why that webhook is idempotent — dies when the process exits.

Scheduling. Nothing runs unless you start it. A nightly documentation audit, a weekly dependency scan, an overnight refactor: all of them require a human to be awake and paying attention at the moment they begin.

Cross-project memory. What an agent learns in one repository does not travel to the next. The lesson learned debugging the payments service is not available to the session working in the marketing site, even though the same person is driving both.

Routing. The decision of which tool should handle a given task — frontier model or local one, supervised or unattended, this vendor or that — is made silently, in the operator's head, with no cost model and no record.

A capability registry. The skills, the MCP servers, and the agents themselves are not catalogued anywhere as a callable inventory. There is no table the system can consult to answer "what can I do, and with what?"

These are not exotic requirements. They are the ordinary services an operating system provides to the programs that run on it: persistence, scheduling, shared memory, a scheduler's routing, a syscall table. The agent ecosystem has the programs. It is missing the operating system.

It is worth dwelling on *why* these gaps are so easy to ignore, because the answer explains why the problem persisted long enough to need a book. Each gap levies a tax that is real but invisible — paid in small, individually unremarkable increments, never itemized on any bill.

The missing persistence is a *re-derivation* tax: every session that rebuilds context a previous session already had is paying, in tokens and in wall-clock minutes, for knowledge you already purchased once. On a single run it is a rounding error. Across a year of daily work it is one of the largest line items in your whole AI spend, and it buys you nothing you did not already have.

The missing scheduling is an *idle-capacity* tax. You own tools capable of doing useful work unattended — audits, scans, summaries, routine refactors — and they sit idle every night because nothing is there to start them. The capacity exists and is simply never used, the way a factory that runs one shift leaves two-thirds of its capital asleep.

The missing cross-project memory is a *relearning* tax, the one our Tuesday developer paid when she solved the same bug twice. Every lesson that does not travel between repositories is a lesson you will pay to learn again, in full, the next time it comes up in a different corner of your work.

The missing routing is a *wrong-default* tax. When the choice of tool is made by habit rather than by fit, you systematically reach for the expensive frontier model on tasks a local one would have handled for free, and — more dangerously — occasionally reach for the fast cheap option on work that needed the careful one. Without a routing decision you are not choosing; you are defaulting, and defaults are rarely optimal and never measured.

The missing registry is a *rediscovery* tax, paid in the ten minutes of every week spent remembering which tool can do the thing you know is possible. A capability you cannot find on demand is, operationally, a capability you do not reliably have.

The instructive thing about these five taxes is that not one of them shows up when you evaluate a tool. A benchmark measures how well Claude Code refactors, not how much you spend re-establishing context because it cannot remember the last refactor. The taxes live in the *spaces between* tools, which is precisely why a market organized around comparing individual

tools has been so slow to address them. They are nobody's product because they are nobody's tool.

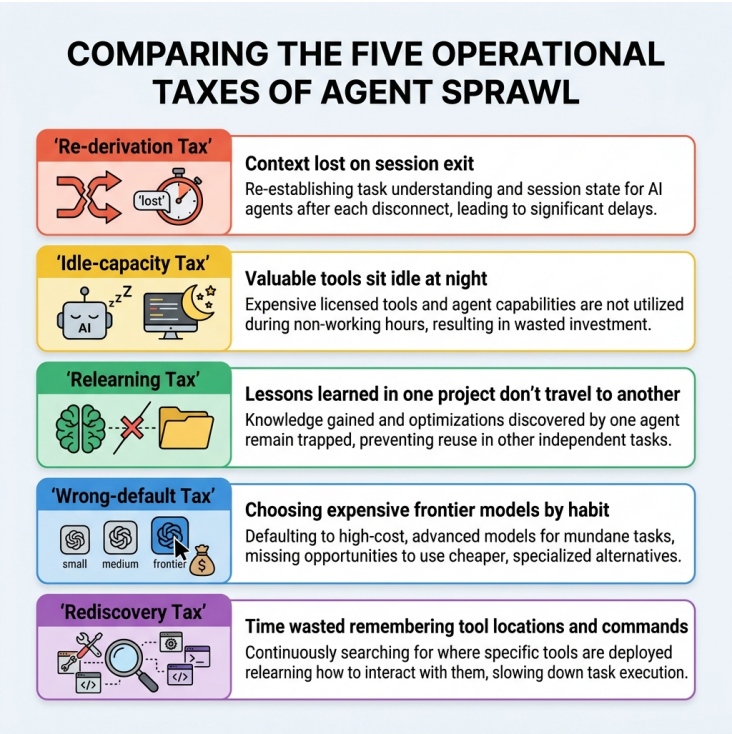

Figure 1.2: The five architectural gaps of agent sprawl levy real, recurring operational taxes on developer productivity.

1.4 Why "Which Agent Is Best?" Is the Wrong Question

Most writing about coding agents asks which one wins. It is the wrong question, for the same reason that asking whether a hammer beats a screwdriver is the wrong question. The frontier CLIs are converging on a similar level of raw capability for hard coding, and where they differ they differ by ecosystem fit, not by a gap a benchmark can settle for your particular work.

More to the point, "best at coding" is a property of the *execution* of a task, while the pain you actually feel is in the *coordination* of many tasks across many tools over time. A slightly better code generator does nothing for the four-o'clock realization that you have run the same context-gathering by hand for the fifth time today, or that the audit you meant to schedule never ran because no one was there to start it. Those are coordination failures, and no amount of single-tool excellence repairs them.

The useful question is therefore not "which agent is best?" but "what owns the layer above all of them?" — the layer that remembers, schedules, routes, and keeps the registry. Nothing in the drawer does. That vacancy is the subject of this book.

1.5 Why the Problem Stayed Invisible

If the coordination problem is so costly, why has so little been built to address it, and why does so much of the public conversation still revolve around which model tops which leaderboard? The answer is partly structural and worth understanding, because it tells you why the opportunity is still open.

Begin with incentives. A model vendor's product is a model; the natural axis of competition is capability on tasks you can score — coding benchmarks, reasoning evals, context length. Those are real and they have driven extraordinary progress. But they are all measures of *execution*, of how well a single tool performs a single bounded task, because that is what a benchmark can measure. Coordination across many tasks over time has no leaderboard. There is no benchmark for "how little context did you re-derive this quarter" or "how much idle capacity did you actually use," so there is no scoreboard pressure pushing anyone to improve them. The market optimizes what it can measure, and it cannot measure the spaces between tools.

Then there are the misconceptions that make the problem look already-solved, each of which is worth naming because you will be tempted by all of them.

The first is that an *IDE or an extension* solves it. It does not; it relocates it. A good editor integration makes one tool pleasant to use inside one project, but it does nothing about the four other agents, the work that should happen unattended, or the knowledge that should travel between repositories. It improves a single seat at the table without addressing the table.

The second is that a *bigger model* solves it. This is the most seductive, because models do keep getting better and it is easy to assume that a sufficiently capable one will simply not need coordinating. But persistence, scheduling, routing, and a capability registry are not deficits of intelligence; a perfect model invoked per-session still forgets on exit, still does not start itself at 3 a.m., still has no record of what the other tools did. These are architectural gaps, and you do not close an architectural gap by making the component at one corner of it smarter.

The third is that the problem is *temporary* — that the tools will converge and the mess will sort itself out. The opposite is more likely. As agents proliferate and specialize, the number of seams grows, not shrinks, and the coordination problem gets harder. Waiting it out is a strategy for paying the five taxes indefinitely.

Clear those three misconceptions away and the shape of the real work is exposed. It is not a better agent, a better editor, or a bigger model. It is the layer above all of them — and because the market's incentives point elsewhere, that layer is still, remarkably, yours to build.

1.6 What an Operating Layer Would Do

Imagine the missing layer as a single long-running process — a *control plane* — that never does the hard coding itself but decides, for every unit of work, who should. It holds the memory the

CLIs lack. It runs a scheduler so that work can happen while you sleep. It keeps a registry of skills and tools. And when a task genuinely needs a frontier coder, it hands that task to Claude Code or Codex, captures the result, and writes what happened back into its memory so the *system* gets smarter even though each CLI stays amnesiac.

This is not a thought experiment. A real, documented implementation of exactly this shape — Hermes — runs through the book as the worked example, and the patterns are deliberately vendor-neutral so they survive whichever tools you actually use. For now, the only claim is that the layer is conceivable and that nothing you already own provides it.

1.7 Hands-On: Audit Your Own Agent Stack

Before building a solution it helps to feel the problem on your own machine. The point of the audit in Listing 1.1 is not the inventory it prints; it is the realization that follows — several capable tools, side by side, none of them aware of the others or of what the others have learned.

Listing 1.1: A one-screen audit of the agents and runtimes on your machine.

```bash
#!/usr/bin/env bash
# Audit which coding agents and runtimes are installed on this machine.
# The point is not the list itself, but to make the fragmentation concrete:
# several tools, each good at one thing, none of them aware of the others.

set -euo pipefail

for tool in claude codex aider opencode gemini ollama litellm; do
  if command -v "$tool" >/dev/null 2>&1; then
    printf '%-10s present  (%s)\n' "$tool" "$(command -v "$tool")"
  else
    printf '%-10s absent\n' "$tool"
  fi
done
```

Run it, then ask the five questions from the previous section about your own setup. Which of these tools remembers anything between runs? Which can start itself on a schedule? Which knows what the others did this morning? If the honest answer is "none," you have located the gap this book fills.

1.8 The Operator's Mindset

The shift this book asks of you is smaller in code than it is in posture, and it is worth naming the posture directly, because everything practical follows from it.

A shopper evaluates tools. The shopper's questions are comparative and terminal: which agent is best, which model wins, which one should I buy. Those questions have answers, the answers change every few months, and chasing them is a treadmill. An operator does something different. An operator assumes a fleet of imperfect, changing tools and asks how to get reliable

work out of them as a system: what should run, where, under whose supervision, remembered how, at what cost. The operator's questions are not about the tools at all. They are about the work and the system that does it. Figure 1.3 makes that shift concrete: the point is not to pick a winner, but to add the layer that turns scattered tools into an operated system.

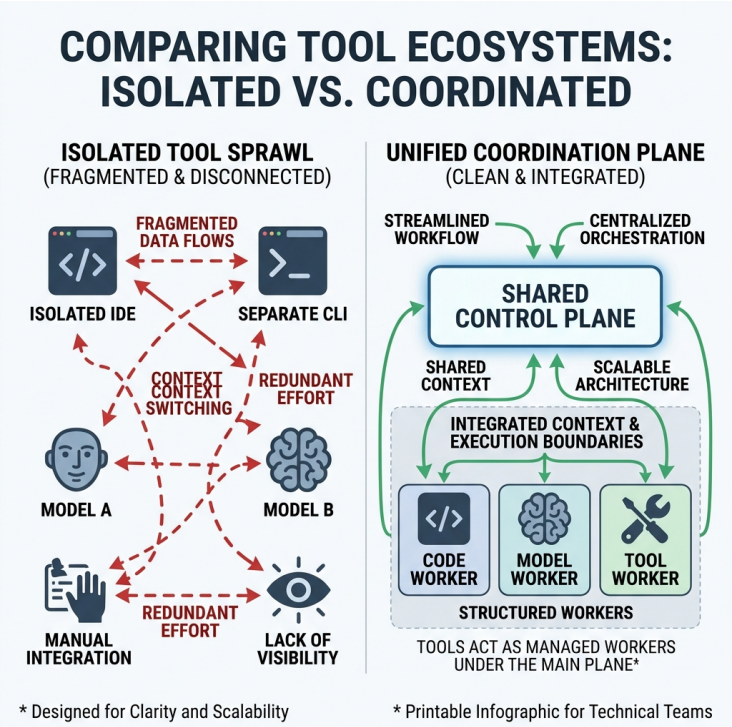

Figure 1.3: Orchestration adds a control plane that coordinates the relationship between user intent, model routing, memory, queues, and agents, turning isolated tools into a coordinated system.

This is the mindset that runs power plants, data centers, and kitchens — domains that long ago stopped asking "which is the best machine" and started asking "how do I run this collection of machines safely and predictably." The maturity of a field is often marked by exactly this transition, from fascination with the components to discipline about the system. The agent ecosystem is making that transition now, a little late and largely unremarked, and this book is a field guide to it.

Adopting the operator's mindset has immediate, concrete consequences for how you work, and they are worth previewing so you can feel the change coming. You stop asking "which agent should I switch to" and start asking "which agent should handle this particular task, and who decides." You stop treating an agent's forgetfulness as a quirk to work around and start treating memory as a thing the *system* must own. You stop starting routine work by hand and start asking why it is not scheduled. And — this one matters most as the system grows — you stop conflating "the tool can do this" with "it is safe to let the tool do this unattended," and you start drawing an explicit line between the work that may run on its own and the work that must wait for a human.

That line, more than any clever automation, is what makes an operated fleet trustworthy, and the whole of Part IV is in service of drawing it well.

None of this requires you to abandon the tools you like or to commit to a particular vendor. It requires only that you raise your gaze from the individual tool to the system above it — and once you have, the gaps from this chapter stop looking like annoyances and start looking like a specification for the thing you are about to build.

1.9 What This Book Builds

The rest of the book constructs the operating layer piece by piece. Part I finishes framing the problem and states the organizing thesis: that because an agent's commands run in a shell, a control plane can invoke any command-line agent as a tool. Part II introduces the pieces — the landscape of execution tools, the control plane itself, and the delegation primitive that lets it drive them. Part III is the mechanics: routing, memory, skills, MCP, and the local-versus-cloud decision. Part IV is about operating the result safely — sandboxing, scheduling, a worked end-to-end orchestration, budgets, failure, and the multi-year arc this all sits on.

You do not need to commit to a single vendor to follow along; in fact the whole point is that you should not have to. You do need to be willing to think about agents the way an operator thinks about a fleet, rather than the way a shopper thinks about a purchase.

Key Takeaways

- The hard problem has shifted from *which agent* to *how to operate all of them together*.
- Today's agents share three constraining properties: they are per-invocation, single-vendor, and shell-bound.
- Five services nothing currently owns: persistence, scheduling, cross-project memory, routing, and a capability registry — exactly what an operating system provides.
- "Which agent is best?" is the wrong question; coordination, not single-tool excellence, is where the pain lives.
- The missing layer is a control plane that delegates hard coding to the CLIs while owning memory, scheduling, routing, and the registry — the subject of the rest of this book.