April 22, 2026 | Chris Lema

Your AI agents have one job each. Most builders give them four.

Most multi-agent systems fail for the same architectural reason: LLMs get wired into four positions when they only belong in three. Here's the rule that fixes it.

AI · Product Work · Agents · For Engineers

Here's what I know about AI agents: the ones that work have sharp boundaries. The ones that don't have LLMs doing everything.

That's the whole pattern. Once you see it, you can't unsee it. Most multi-agent systems that people are shipping right now fail for the same reason, and it's not a reason anyone's putting on a pitch deck. It's architectural. Builders are wiring LLMs into positions where LLMs don't belong, then acting surprised when the system is slow, expensive, flaky, and impossible to debug.

I want to show you a better pattern. I'm calling it the Three-Jobs Ru
It's not fancy. That's the point.

The four-position mistake

Walk through any multi-agent system that's giving its builders grief, and you'll find the same setup. There's a generator agent. There's a judge agent. There's a router agent. There's an executor agent. Every one of them is an LLM. They talk to each other in natural language. The whole system is one long chain of probabilistic calls, each one capable of misreading the last one's output or hallucinating something that wasn't there.

This is what I mean by the four positions. Builders put an LLM in each of them:

Generator. Creates something (text, code, a plan, a decision).

Judge. Evaluates something (was that good, is this correct, does it meet the bar).

Router. Decides where things go next (which tool, which branch, which agent).

Executor. Actually does the work (calls the API, writes to the database, sends the email).

Wire LLMs into all four and you have what looks like a sophisticated multi-agent system. You also have a system where every hop compounds error, every decision is non-reproducible, and every test you try to write dies in your hands because the router is a 70B parameter model having a mood.

Here's the uncomfortable part. You don't need an LLM for most of that. In fact, for most of it, using an LLM is strictly worse than writing the code.

The Three-Jobs Rule

There are exactly three places where LLMs earn their keep. Put them in those three places. Put deterministic code everywhere else. That's the rule.

Job 1: Generation in an open space. Drafting, ideating, transforming unstructured input into something new. The output space is unbounded, there's no single correct answer, and the value is in producing something coherent from nothing. LLM territory.

Job 2: Judgment over fuzzy criteria. Scoring a draft against a rubric, detecting tone, classifying intent when the categories have soft edges. Not "does this JSON validate" (that's code). "Does this reply sound warm and on-brand" (that's an LLM).

Job 3: Extraction from unstructured input. Pulling structure out of a messy email, an interview transcript, a user message. Turning human mess into clean inputs the rest of the pipeline can work with. This is the one people most often miss, because they think the LLM is doing the work when really it's the adapter between a messy human world and a deterministic pipeline. The pipeline is still doing the work.

That's it. Three jobs. Everything else (routing, state machines, policy evaluation, payment calls, escalation logic, retries, budget enforcement, logging) is code. Not because an LLM can't do those

things. An LLM can do almost anything, badly. It's because you can already write the rule, and the moment you can write the rule, laundering it through a probabilistic layer buys you nothing except latency, cost, and non-reproducibility. This is the same instinct behind separating expensive design-time thinking from cheap runtime execution: spend intelligence only where it earns its keep.

Let me show you what the rule looks like in practice.

A system built the right way

Imagine you're building an agent that handles customer refund requests for an e-commerce company. A customer emails in, and the system has to read the email, figure out what they want, check whether they qualify, issue the refund (or escalate, or decline), and write back. Sounds like a job for a multi-agent system. Five agents, all chatting, all powered by GPT or Claude.

That's the trap. Let's build it correctly instead.

Stage 1: Extraction. LLM. The email comes in messy. "hey so I ordered the blue one last tuesday i think? and it came but the handle is busted and also honestly it's smaller than i thought, can i just send it back." An LLM is the right tool here because the input space is unbounded. Humans write however they want. The LLM's job is narrow: extract a structured object. Order reference, stated reason, customer sentiment, requested resolution. Output is JSON. That's it. It's not deciding anything. It's translating.

Stage 2: Resolution. Deterministic. Now the structured object hits code. SQL lookup: does the order exist, when was it placed, what's the return window, what's the item category, has this customer requested refunds before, what's their lifetime value. None of this needs an LLM. It's a database query and a policy table. The policy table says things like "electronics, 30 day window, damage covered, buyer's remorse not covered, auto-approve under $200." That's a spreadsheet. A spreadsheet is infinitely more debuggable than a prompt.

Stage 3: Judgment. LLM, narrow. Here's where it gets interesting. The customer said the handle was "busted." Is that a damage claim (covered) or a quality complaint (not covered)? The policy can't enumerate every word humans use for "broken." So you call an LLM with a tight prompt: "Given this description, classify the damage claim as physical_damage, manufacturing_defect, cosmetic_complaint, or size_fit_issue." Four buckets. The LLM does one thing, fuzzy classification against a closed set, and returns a label. The label goes back into deterministic code.

Stage 4: Decision. Deterministic. Code now has everything it needs. Order exists. Within window. Damage claim classified as physical_damage. Under auto-approve threshold. Customer in good standing. The decision tree is a dozen lines of code. Approve. Issue refund via Stripe API. Log the transaction. No LLM involved, because there's nothing to decide that a human engineer hasn't already thought through.

Stage 5: Generation. LLM. Write the reply email. This is unbounded creative output. There's no "correct" email, just better or worse ones. LLM territory. It gets the decision, the customer's original tone, the resolution details, and drafts a reply. Maybe a second LLM call scores

the draft against a rubric (warm, clear, on-brand, under 120 words) and loops if it fails.

Stage 6: Escalation. Deterministic with narrow LLM extraction. If any stage hits an ambiguous case (order not found, claim over threshold, customer previously flagged), deterministic code routes to a human queue with a summary. The summary itself might be LLM-generated. The routing decision is a rule.

Now count the LLM calls. Extraction, classification, generation, maybe rubric scoring. Four narrow jobs. The spine of the system (the lookup, the policy check, the decision tree, the payment call, the escalation routing, the logging) is all code. If a refund goes wrong, you can point to the exact line. You can write unit tests. You can change the refund threshold without re-prompting anything.

Now imagine the naive multi-agent version. "Refund Agent" talks to "Policy Agent" talks to "Payment Agent" talks to "Communication Agent." All LLM-powered. All chatting in natural language. All occasionally hallucinating policy details or issuing refunds to the wrong order because one agent misread another agent's message. Same problem. Ten times the cost. A hundred times the failure modes. Zero ability to audit.

The restaurant kitchen

Here's the metaphor that I want you to keep.

A working restaurant kitchen already solved this problem a century ago. Escoffier's brigade, more or less. Tickets come in on a rail. The

expeditor reads the ticket and calls it out. Each station knows exactly what to do when its item is called. The salad goes out with the entrée, not before. Plates leave the pass when the expeditor says they leave. None of this is creative. None of this requires judgment. It's choreography, and the choreography is the entire reason the kitchen can serve two hundred covers a night without collapsing.

Where does creativity live? Two places. First, in the recipe itself. Someone, at some point, decided what goes on the plate and how it's composed. That's unbounded creative work. Second, in the cooking moments that require judgment. Is this steak medium-rare yet. Does this sauce need more acid. Is this fish fresh. Those are fuzzy calls that require a trained palate.

Everything else (who does what, in what order, when the plate leaves, how the bill is calculated, which table it goes to, what happens when an allergy is flagged) is protocol. It's deterministic. And the reason is simple: if the expeditor had to improvise every ticket, service would collapse in ten minutes.

Now imagine a kitchen run the way people build multi-agent systems. Every line cook is a genius improv artist. The expeditor is also a genius, but they're having a freeform conversation with the sauté station about whether this ticket should be fired now or in two minutes. The garde manger is debating with the grill about plating philosophy. The pastry chef is weighing in on the entrée. Every decision is a committee meeting. Every ticket is a new negotiation.

That kitchen cannot serve food. It will serve some food, occasionally brilliantly, but the throughput will be a tenth of a normal kitchen and the failure rate will be catastrophic.

The LLM is the chef's palate and the chef's creativity. Use it for tasting and for composing. The kitchen system (the rail, the stations, the timing, the pass) is deterministic code. You don't ask a palate to route tickets. You don't ask a rail system to taste a sauce. Each tool does what only it can do, and the system works because the boundaries are crisp.

This is why the brigade outlives technical frameworks. The technology will change. The brigade won't.

What to do tomorrow

Here's the test. Take whatever multi-agent system you're building right now and walk through it, component by component. For each component, ask one question: does this component do generation in an open space, judgment over fuzzy criteria, or extraction from unstructured input?

If yes, keep the LLM. Tighten the prompt. Narrow the job.

If no, it shouldn't be an LLM. Rewrite it as code. It doesn't matter how "intelligent" the component feels. If the space is enumerable, enumerate it. If the routing rule is knowable, write the rule. If the policy fits in a table, put it in a table. A policy table is always going to be more debuggable than a prompt that restates the policy in English.

One more thing worth saying, because people over-correct. Deterministic doesn't only mean hand-coded if-statements. It includes traditional ML classifiers, embeddings plus cosine similarity for routing, regex, SQL, and plain old code. It's a wide tent. The

question isn't "can an LLM do this." An LLM can. The question is, "is this a problem where I can enumerate the space?" If yes, don't reach for the LLM.

Build it like a kitchen. Three jobs for the palate. Everything else on the rail.

That's the rule.

ABOUT THE AUTHOR

Chris Lema has spent twenty-five years in tech leadership, product development, and coaching. He builds AI-powered tools that help experts package what they know, build authority, and create programs people pay for. He writes about AI, leadership, and motivation.

AI IS MOVING FAST. YOU DON'T HAVE TO FIGURE IT OUT ALONE.

I help business leaders cut through the hype and put AI to work where it actually matters.

Let's Talk AI

CHRIS LEMA

Making Experts Dangerous

Tools

Your Voice Profile

Your Audience Segments

Your Content Agent

Your Platform Profile

Your Lead Magnet Agent

Content

Connect