PART 2 OF 3 · WORKING TOGETHER

How AI Agents Work Together

Clausey

Part 1 built one agent. The next idea is small but powerful: one of an agent's tools can be another agent. That's all a team really is.

Why one agent isn't enough · The subagent · Splitting the work · Keeping it reliable

Clausey

Part 1 built one agent.

Part 1's whole machine — but it runs one task at a time, inside one context window.

THE WHOLE AGENT — PART 1

Model

reasoning — reads the array, decides the next move

Tools

actuation — the model's reach into the real world

Loop

control flow — send, decide, execute, append, repeat

Message array

memory — the one growing object your code builds

Compaction + caching

economics — what makes long runs survivable

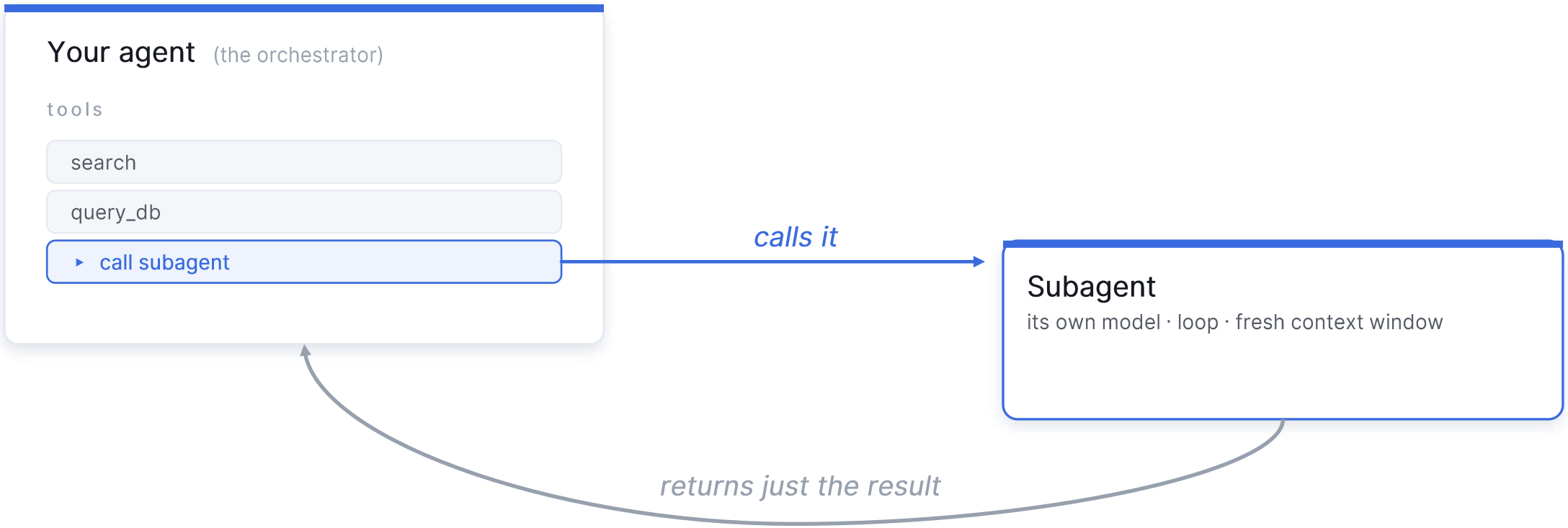
LIMIT 1 — A job too big, or too many at once, won't fit. This is the wall Part 2 climbs.

LIMIT 2 — Lives in one run's memory. Saved for Part 3.

One agent, one context window, one task at a time. That's the wall.

A tool can be another agent.

An agent already calls tools. Make one of them a second agent — its own loop, its own fresh context.



A subagent is just a tool whose job is to run another agent.

Clausey

SECTION 01 / 04

Why one agent isn't enough

What a single agent can't do on its own

01

1 · Why one agent

2 · The subagent

3 · Splitting work

4 · Reliability

Clausey

One agent hits two walls.

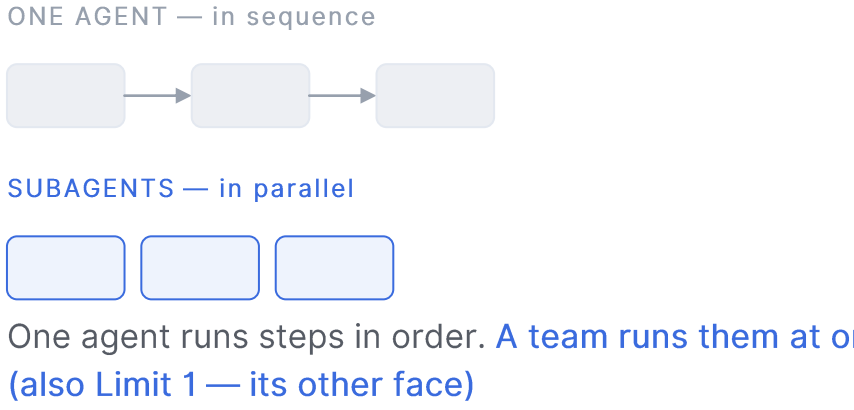
It's not about being slow. There are two hard limits.

Wall 1 · The context fills up

no room left to think

A big enough job overflows one window. Once it's full, quality falls off a cliff. (Limit 1 from Part 1)

Wall 2 · One thing at a time



One agent runs steps in order. A team runs them at once. (also Limit 1 — its other face)

Wall 2 is Part 1's intra-turn parallelism scaled up — the same concurrent tool calls, only now each tool is a whole agent.

Some jobs won't fit. Some shouldn't wait in line.

Clausey

More agents, bigger bill.

Every subagent is a full agent — its own model calls and tokens. Anthropic clocked a multi-agent run at roughly 15× the tokens of a single chat.

Start with one agent

Don't reach for subagents until a single agent truly can't keep up — because the context overflows or steps must run in parallel. Not to look sophisticated.

Reach for a second agent only when one genuinely can't cope.

Clausey

SECTION 02 / 04

A subagent is a tool

An agent whose caller treats it as just another tool.

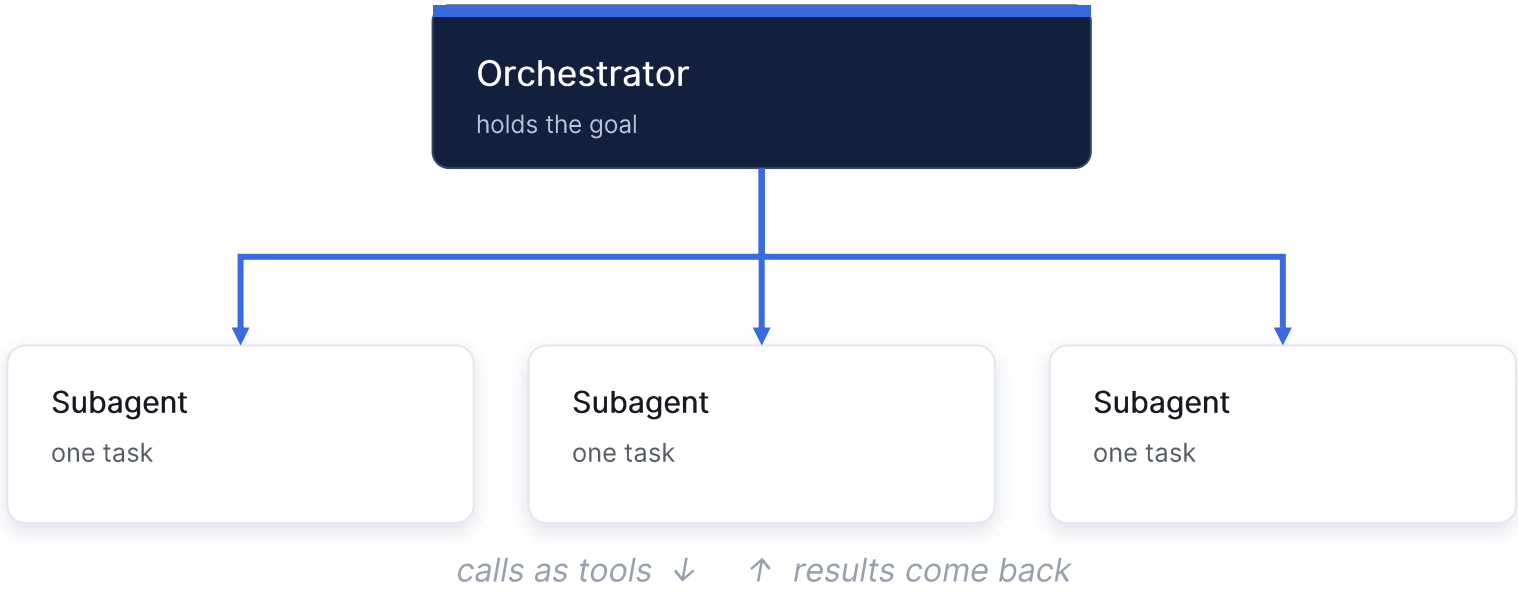
02

1 · Why one agent | 2 · The subagent | 3 · Splitting work | 4 · Reliability

Clausey

One agent coordinates the rest.

One agent holds the overall goal, hands each subagent its own goal, calls them as tools, then composes the results. That’s the orchestrator.



Just an agent

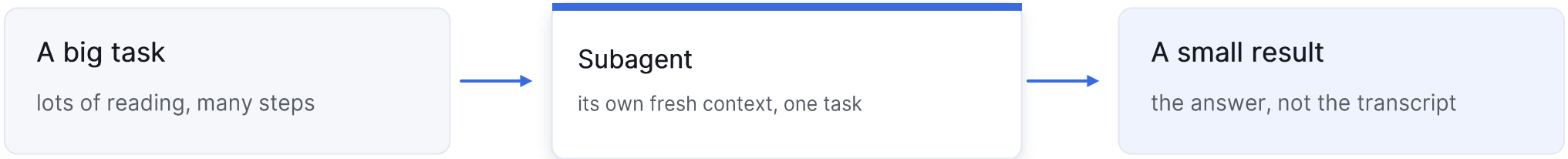
The orchestrator isn't special — it's the same machine from Part 1, given the job of coordinating.

One agent in charge, calling the others — that’s the orchestrator.

Clausey

A subagent has its own context.

A subagent does the heavy work in its own context and returns just the answer — the orchestrator's stays small.



Like handing off research.

You ask a colleague a question; they read fifty pages and come back with one paragraph. You never see the fifty pages — and your own desk stays clear.

The flip side: the orchestrator never sees those fifty pages, so it can't course-correct mid-run, and when the answer is wrong the reasoning is gone — debugging is harder.

Heavy work happens in the subagent's context — the orchestrator only sees the result.

Clausey

The subagent, in the array.

A subagent call is the same tool_use / tool_result you saw in Part 1 — the tool just happens to be a whole agent.

1 · Orchestrator asks

USER	the goal
ASSISTANT	tool_use: run_subagent

Emits a tool call like any other — names the subagent, hands it a task.

2 · Subagent runs · its own array

USER	the task
ASSISTANT	tool_use: query_db
TOOL	result: rows
ASSISTANT	one-paragraph answer

Its own loop, fresh context. The orchestrator never sees these turns.

3 · Back in the orchestrator

USER	the goal
ASSISTANT	tool_use: run_subagent
TOOL	tool_result: paragraph
ASSISTANT	final answer

The answer rides back as a tool_result — exactly Part 1's shape.

A whole subagent collapses to one tool_use and one tool_result — the team idea rides entirely on Part 1's array.

Clausey

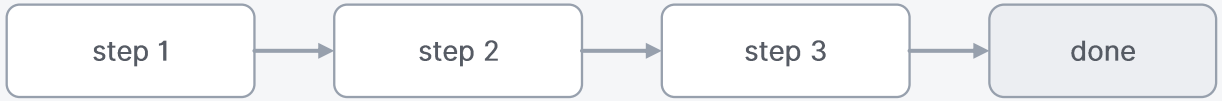
Who decides what runs next?

You can fix the steps in code — or let the model choose them as it runs.

NON-AGENTIC

Workflow

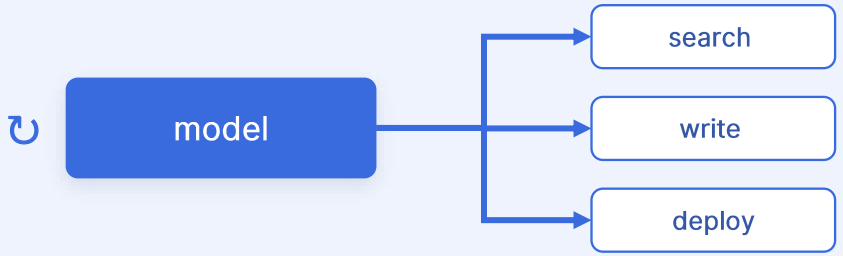
you set the path



AGENTIC

Agent

the model picks the path



...the model decides, step by step

Most real systems blend both — fixed scaffolding, agentic steps inside.

Fixed path = workflow. Model picks the path = agent.

Clausey

Three ways to wire agents.

How agents connect. Most real systems use the middle one.

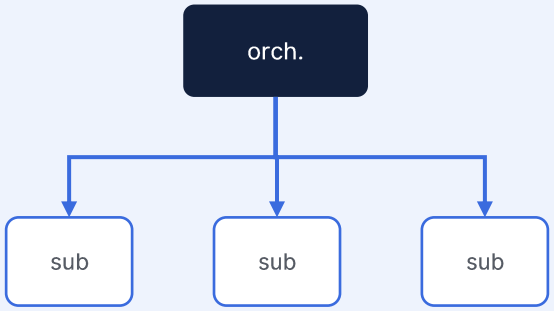
1 Pipeline

(prompt chaining)

each agent's result feeds the next

2 Orchestrator

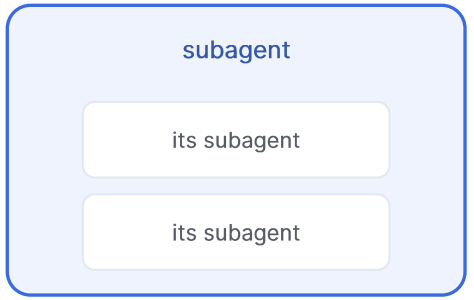
(orchestrator-workers)



one agent calls several subagents

3 Nested

(hierarchical)



a subagent with its own subagents

Most real work fans out from one orchestrator. Save nesting for last.

Clausey

Composing results is the hard part.

Calling subagents is easy. Combining what they return, without flooding the orchestrator, is the hard part.

Orchestrator

If results pile in faster than it can stitch them together, its context fills with raw output — the exact Limit 1 we set out to dodge, back again.

The takeaway

Keep what each subagent returns small — the result, not its whole transcript.

Calling agents is easy. Combining what they return is where it jams.

Clausey

SECTION 03 / 04

Splitting the work

How to carve a goal into tasks a subagent can own.

03

1 · Why one agent | 2 · The subagent | 3 · Splitting work | 4 · Reliability

Clausey

Three ways to split the work.

Cut the goal by task, by role, or by data. All three are fine — they just trade off differently.

By task

One subagent owns a whole task end to end — "research the competitor." (split by area)

By role

One subagent does one kind of step for everything — "fact-check every claim." (split by skill)

By data

Same task, split across slices — one subagent per region or document. (split by partition)

Like staffing a project.

Give one person a whole feature; give one specialist — all the testing — across every feature; or split the same review across ten files. Same project, three ways to divide it.

By task, by role, or by data — pick the cut with the fewest handoffs.

Clausey

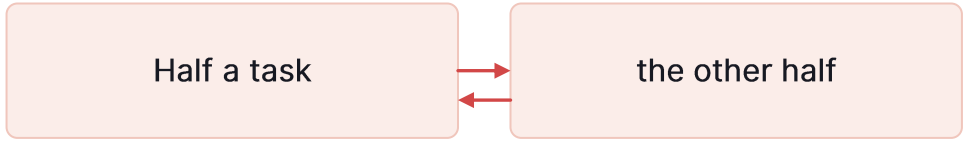
Cut where it comes apart cleanly.

Give a subagent a whole task — clear input, clear output — not half a job another has to finish.

✓ Clean cut

Each subagent owns a whole task. Nothing crosses the line.

× Tangled cut

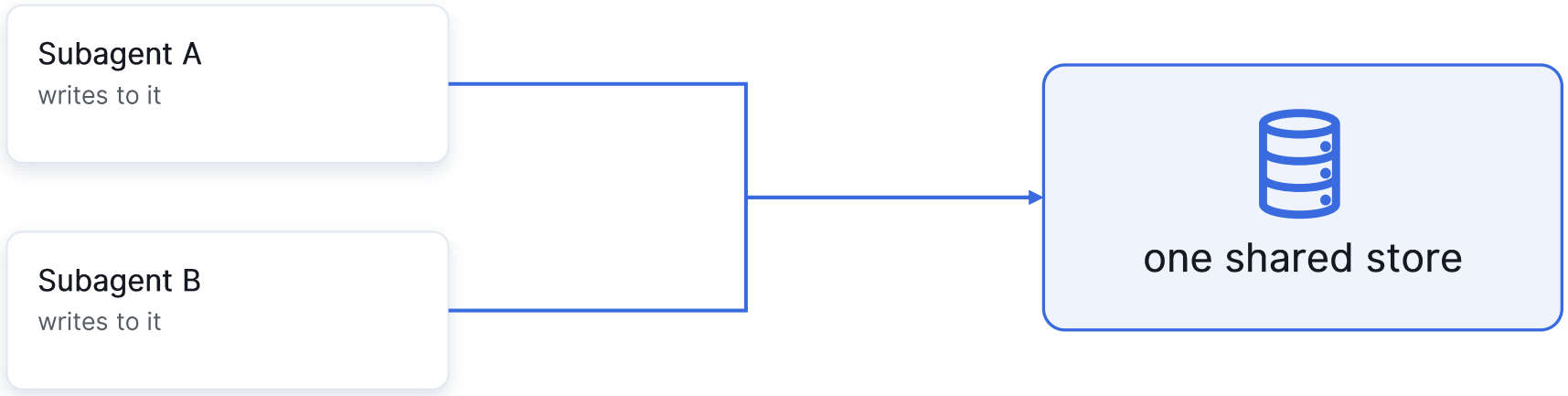


Two subagents finishing each other's work — constant back-and-forth, dropped results.

The fewer the handoffs, the less that slips through.

Mind the shared state.

Sometimes two subagents need the same file or record. Anything they both touch is where trouble starts.



Anything two subagents share is where the next section begins.

SECTION 04 / 04

Keeping it reliable

Two kinds of safety: keeping shared state correct, and designing each subagent so it can’t do harm.

04

1 · Why one agent
2 · The subagent
3 · Splitting work
4 · Reliability

Clausey

Two subagents, one store.

Two subagents write the same record at once — and you can't predict which one lands last.

What goes wrong

The final value depends on who writes last — and that changes run to run.

A race condition

Two workers, one piece of state, no set order. The outcome is a coin flip.

Two writers, one record, no order of operations — that's a race condition.

Clausey

One resource, one owner.

Give every shared resource exactly one subagent that may write it. Everyone else asks that one.

The order record

owner: Subagent A

ask the owner — don't write it directly

The shared doc

owner: Subagent B

ask the owner — don't write it directly

The job queue

owner: Subagent C

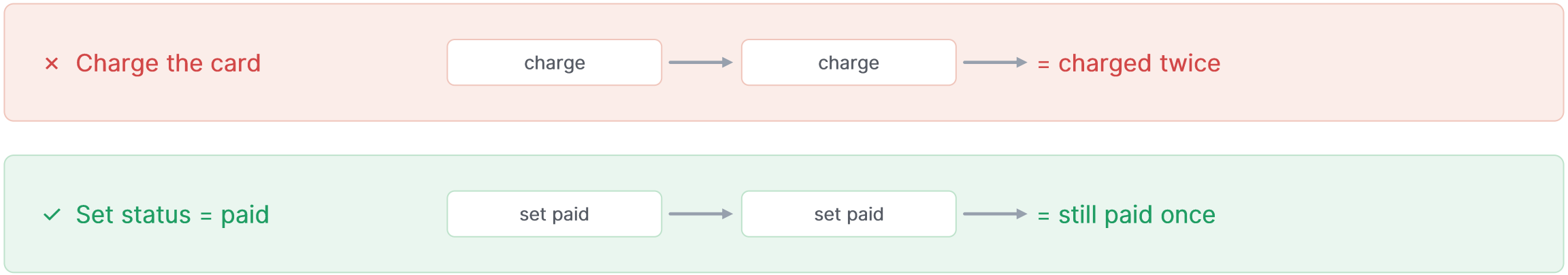
ask the owner — don't write it directly

Give every shared resource a single owner.

Clausey

Make a retry harmless.

Retries happen, so a step can run twice. “Charge the card” twice hurts; “set paid” twice doesn’t.



An operation you can run twice with the same result is idempotent — and you need it now: the moment you run subagents, steps get retried (one fails, you re-dispatch it), long before Part 3’s durable retries make it unavoidable.

Write each step so running it twice changes nothing.

Clausey

A focused subagent beats a do-everything one...

One subagent with a dozen tools and a vague job gets confused. Several focused ones win.

× One subagent, everything

search | write | email
deploy | bill | refund

overloaded — slow, and easy to confuse

✓ Three focused subagents

subagent research | subagent draft | subagent review

each one clear job — fast, and hard to confuse

Like hiring specialists.

You want the person who does one thing all day — not one generalist pretending to cover every role at once.

Many small, sharp subagents beat one doing it all.

Clausey

Only the tools the task needs.

A subagent that only summarizes shouldn’t be able to send email or delete records.

✓ A summarizer subagent

- ✓ read documents
- ✓ search (read-only)

Just the two tools the job needs.

× Every tool you have

- × delete records
- × send email
- × deploy code
- × move money
- × edit billing
- × admin API

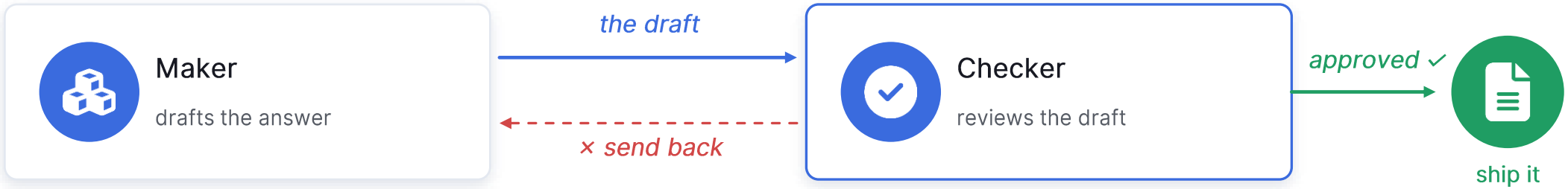
One confused step and a summarizer wires money.

Give each subagent only the tools its job needs — that’s least privilege.

Clausey

Have one agent check another.

A subagent can be confidently wrong. The fix that scales: a second agent reviews its work before you trust it.



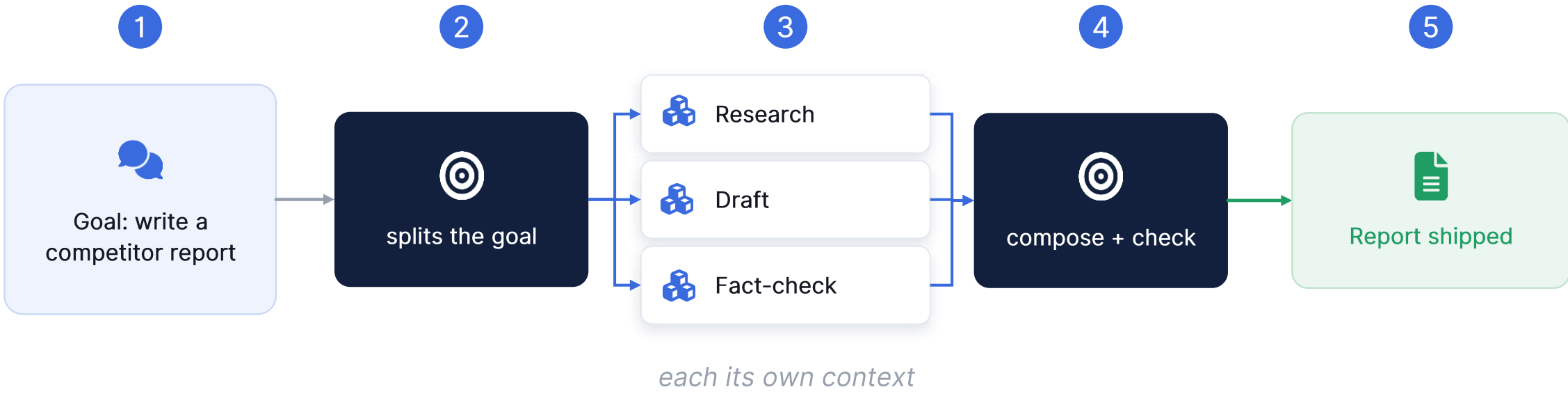
A checker catches confident mistakes — not crashes. A subagent that errors is just a tool that errored: the failure rides back and the orchestrator retries, exactly as in Part 1.

Pair a maker with a checker — confident mistakes get caught.

Clausey

One job, start to finish.

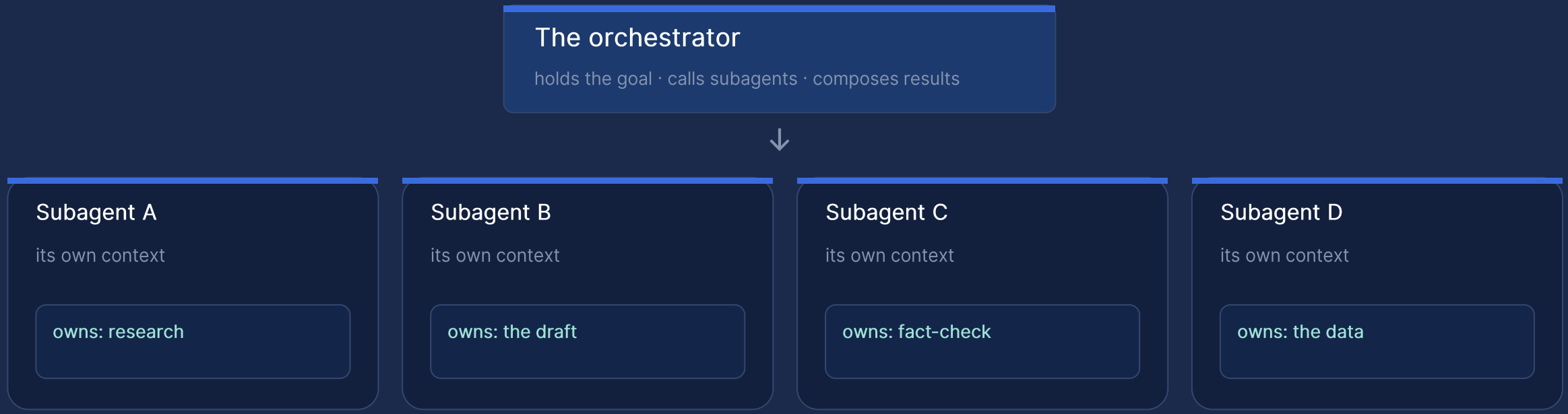
A competitor report, end to end — every idea from this part, in a single run.



One orchestrator, a few focused subagents, each checked — that's a team.

Clausey

The whole system, working as one.



Each subagent is a full agent · its own context · one owner per resource · called like a tool.

One orchestrator, a few focused subagents, one owner each — that's a team.

ONE LIMIT SOLVED, ONE MADE WORSE

You solved one limit. You made the other worse.

Clausey

Spreading work across subagents — each with its own context — solved Part 1's first wall. But running many agents quietly made the second one worse.

SOLVED · LIMIT 1

No single context holds it all.

A job too big for one window now spreads across subagents, each with its own. The first wall is gone.

WORSE · LIMIT 2

Now more things can crash.

The orchestrator, every subagent, and the record of who's-doing-what can each fail — and it all lives in memory that vanishes on any restart.

→ PART 3 · DURABLE EXECUTION

We taught agents to work together. We never taught them to survive.

THE SERIES

Clausey

This was Part 2 of 3.

If you can now say that a subagent is just a tool that runs another agent, who the orchestrator is, and why one owner per resource stops the chaos — you've got the team.

01

What Exactly Is an AI Agent?

The anatomy of a single agent — down to the API call.

✓ done

02

How AI Agents Work Together

Subagents as tools, an orchestrator, one owner per resource.

✓ here

03

Why Most Agents Die in Production

Durable execution — keeping the whole team alive when any agent drops out mid-run.

next

Next: how to keep the whole team alive when any agent can drop out mid-run — and why the habits from this part are exactly what makes that possible.