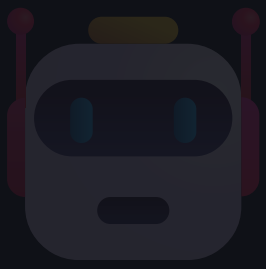

A I

Claude Code Agents: Advanced Orchestration

The Agent SDK & Programmatic Agents

Parallel/Sequential Orchestration, Background & Autonomous Agents

Specialized Agents, Handoffs, Debugging & Cost Tradeoffs

Capstone: A Production-Style Multi-Agent Pipeline for website-content

Philip Osztromok

Generated with Claude

Table of Contents

Claude Code Agents: Advanced Orchestration — 10 Chapters

Claude Code Agents: Fundamentals covered the mental model behind a single agent — what it is, how it's defined, how to brief it well — and applied that model to seven distinct agent types, one at a time. This course assumes all of that is comfortable working knowledge, and goes further into what happens once agents stop being used one at a time: building them programmatically, chaining them together, and running them at real production scale.

A Quick Recap of What Fundamentals Covered

- **The core mental model** — subagents vs. the default assistant, tool-permission isolation, agent-definition anatomy (Chapters 1–2)
- **Communicating with an agent** — writing a self-contained brief, foreground vs. background invocation, reading a final report critically (Chapter 3)
- **Seven individual agent types** — teaching, coding, code-review, testing, refactoring, documentation, and research (Chapters 4–10)
- **A capstone** applying all seven to one real project, one at a time, by hand (Chapter 11)

If any of that feels shaky rather than second nature, it's worth revisiting those chapters directly — this course builds on each of them without re-teaching them.

What This Course Adds

- **Building agents programmatically with the Claude Agent SDK** — moving beyond hand-written agent definitions (Chapter 2)
- **Parallel vs. sequential orchestration** — running several agents at once versus one after another, and when each is right (Chapter 3)
- **Background & autonomous agents** — long-running and self-scheduling work (Chapter 4)
- **Specialized agent types** — security/compliance review agents, and planning/architecture agents (Chapters 5–6)
- **Agent-to-agent handoffs & shared state** — passing results and context directly between agents (Chapter 7)
- **Debugging and evaluating agent behavior** — what to do when an agent gets something wrong (Chapter 8)
- **Cost, context & efficiency tradeoffs** — when orchestration is actually worth its own overhead (Chapter 9)
- **Capstone** — a production-style multi-agent pipeline built for the `website-content` project itself, formalizing Chapter 11's own capstone into a real, chained pipeline (Chapter 10)

CLAUDE CODE AGENTS: FUNDAMENTALS	CLAUDE CODE AGENTS: ADVANCED ORCHESTRATION
One agent, hand-written, used at a time	**Multiple agents, chained and coordinated together**
Agents defined by hand, one definition file each	**Agents built and configured programmatically via the SDK**
You launch each step yourself	**A pipeline hands work from one agent to the next automatically**
"Does this one agent work correctly?"	**"Does this whole chain of agents behave correctly together?"**

ORCHESTRATION IS A DIFFERENT SKILL, NOT JUST "MORE AGENTS"

Designing one good agent well (Fundamentals' own entire focus) is necessary but not sufficient for orchestrating several together. New failure modes only appear once agents interact — Chapter 1's own "an agent starts cold" problem, for instance, doesn't just apply once; in a chain of agents, each handoff is a fresh place for context to get lost unless it's deliberately carried forward. This course is about exactly those chain-level concerns, not a repeat of single-agent design.

WHAT THIS COURSE DELIBERATELY DOES NOT COVER

Three things are out of scope on purpose. First, this course does not re-teach the Claude API from first principles — Chapter 2's SDK material assumes either *Advanced AI Workflows & the Claude API* or equivalent existing familiarity with API basics like authentication and requests. Second, it does not cover fine-tuning a model's own weights to change its behavior — every technique here works with models as they already are, through prompting, tools, and orchestration, not retraining. Third, it does not attempt to build a general-purpose agent framework from scratch outside Claude Code's own agent system — everything here works within the tools and patterns Claude Code itself already provides.

Who This Course Is For

Anyone comfortable with Claude Code Agents: Fundamentals' own eleven chapters — the mental model, the seven agent types, and briefing an agent well. No new software is required beyond what Fundamentals already assumed; this course goes deeper into coordination and scale, not into new prerequisite tools.

Hands-On Exercises

EXERCISE 1

A colleague who finished Claude Code Agents: Fundamentals asks why orchestration needed a whole second course rather than one more chapter added to the first. Using this chapter's own tip box, explain the reasoning.

View solution

EXERCISE 2

Explain why Chapter 1's "an agent starts cold" problem is described in this chapter as getting worse, not just repeated, once several agents are chained together.

View solution

EXERCISE 3

A learner asks whether this course will teach them to fine-tune a model so it behaves better as part of an agent pipeline. Using this chapter's own warning box, explain why that falls outside this course's scope, and what it teaches instead for improving agent behavior.

View solution

CHAPTER 1 QUICK REFERENCE

- This course assumes **Fundamentals**' eleven chapters (the mental model, seven agent types, the capstone) as comfortable working knowledge
- Nine chapters follow: the Agent SDK, parallel/sequential orchestration, background/autonomous agents, two specialized agent types, handoffs/shared state, debugging agent behavior, cost/efficiency tradeoffs, and a production-pipeline capstone
- Orchestration is a genuinely different skill from single-agent design — new failure modes emerge only once agents interact
- **Out of scope, on purpose:** re-teaching the Claude API from scratch, fine-tuning models, and building a general agent framework outside Claude Code

Advanced AI Workflows & the Claude API introduced multi-agent patterns at the raw API level — tool use, streaming, and building an AI-powered app from scratch. This chapter picks up specifically where that course's own multi-agent chapter left off: using the Claude Agent SDK to define and run agents *programmatically*, in code, rather than as the hand-written definition files Fundamentals covered.

Hand-Written Definitions vs. Programmatic Agents

Fundamentals' Chapter 2 covered an agent definition as a markdown file — frontmatter fields (name, description, tools, model) plus a written system-prompt body. That works well for a fixed, well-understood role used repeatedly. Defining the same fields programmatically — as a data structure built in code, rather than a static file — opens up something a static file can't: generating or adjusting an agent's own configuration dynamically at runtime. A tool allowlist could be built conditionally based on the current user's permission level; a system prompt could be templated with variables filled in from the actual task at hand, rather than written once and left fixed.

The Agentic Loop

Running an agent that uses tools involves a repeating cycle: send a message, the model requests a tool call, your code executes that tool, the result is sent back, and the cycle repeats until the model produces a final answer with no further tool calls needed. The SDK's **Tool Runner** handles this entire cycle automatically — you register the available tools and hand off the conversation, and the loop runs to completion without you writing the repeat-until-done logic yourself.

Manual Tool-Use Loops

The alternative to the Tool Runner is writing that same loop by hand — genuinely more code, but full control over what happens at each step: custom logging between every tool call, injecting extra validation logic before a tool actually runs, or stopping the loop early under a specific condition the automatic runner wouldn't know to check for. This is a real trade-off between convenience and control, not a strictly better or worse option in either direction — worth choosing deliberately based on whether that extra control is actually needed.

```
# Illustrative pattern — actual SDK syntax may vary by version
agent_config = {
    "name": "changelog-writer",
    "description": "Writes changelog entries for finished work",
    "tools": [read_file_tool, edit_file_tool],
    "model": "claude-haiku-4-5",
}

# The Tool Runner handles the repeat-until-done loop for you
result = client.beta.messages.tool_runner(
    **agent_config,
    messages=[{"role": "user", "content": task_brief}]
)
```

Managed Agents

A managed agent runs server-hosted, inside infrastructure that persists independently of your own local process — useful for handing off a genuinely long-running task without needing your own machine to stay running and connected the whole time. This trades some direct control for not having to manage that infrastructure yourself.

ASPECT	HAND-WRITTEN DEFINITION (FUNDAMENTALS)	PROGRAMMATIC (SDK) AGENT
Where it's defined	A static markdown file	Built as data/code at runtime
Can it vary at runtime?	No — fixed once written	Yes — tools, prompt, model can depend on conditions
Best suited for	A stable, well-understood, repeated role	An agent whose configuration genuinely needs to change per run

DEFAULT TO HAND-WRITTEN UNLESS DYNAMIC BEHAVIOR IS ACTUALLY NEEDED

A hand-written definition is simpler to read, review, and reason about than a programmatically generated one. Reach for the SDK's programmatic approach specifically when an agent's configuration genuinely needs to vary at runtime — not as a default, more "advanced-feeling" way to define every agent.

PROGRAMMATIC DEFINITIONS ADD REAL COMPLEXITY

Building an agent's configuration in code means more code to maintain, and more that can go subtly wrong in how that configuration gets assembled — a conditional that doesn't cover every case, a templated prompt with a variable substituted incorrectly. This is worth the added complexity specifically when the dynamic behavior it enables is genuinely needed — the same "orchestration has a real cost" theme this course opened with in Chapter 1, applied here to configuration itself rather than to chaining agents together.

Hands-On Exercises

EXERCISE 1

Explain a concrete scenario where an agent's tool allowlist genuinely needs to be built at runtime rather than fixed in a hand-written definition file.

View solution

EXERCISE 2

Explain the trade-off between using the Tool Runner and writing a manual tool-use loop, and give an example of when the manual loop's extra control would genuinely be worth its extra code.

View solution

EXERCISE 3

A developer converts every agent in their project to a programmatically-defined SDK agent, even ones whose configuration never changes between runs. Using this chapter's own warning box, explain what's questionable about this decision.

View solution

CHAPTER 2 QUICK REFERENCE

- A programmatic agent's configuration is built **in code**, letting it vary at runtime — unlike a fixed, hand-written definition file
- The **Tool Runner** automates the repeat-until-done agentic loop; a **manual loop** trades that convenience for full control
- A **managed agent** runs server-hosted, independent of your own local process
- Default to hand-written definitions unless an agent's configuration genuinely needs to vary at runtime
- Programmatic definitions add real maintenance complexity — worth it only when the dynamic behavior is actually needed

Once more than one agent is involved in a task, a basic structural question arises: should they run at the same time, or one after another? This chapter covers both patterns, how to tell which one actually fits, and a hidden risk specific to running agents in parallel that's easy to overlook.

Parallel Orchestration

Launching several agents at once makes sense when their tasks are genuinely independent — neither needs the other's result to do its own job. Several research agents each investigating a different area of an unfamiliar codebase simultaneously, or a code-review agent and a documentation agent working on two entirely separate, unrelated modules at the same time, are both good fits. The benefit is straightforward: total wall-clock time is much closer to the slowest single agent's own time, rather than the sum of every agent's time added together.

Sequential Orchestration

Running agents one after another is necessary specifically when a later agent's task genuinely depends on an earlier agent's actual result. A research agent's findings feeding into a planning agent's proposal, which then feeds into a coding agent's implementation, can't be parallelized — each step needs the previous step's real output to even begin working, not just a guess at what that output might turn out to be.

How to Tell Which Pattern Fits

The deciding question for any two steps: **does the second step need the first step's actual result before it can start?** If yes, sequential is the only option that makes sense. If the tasks are genuinely independent, parallel is available and usually faster. A common mistake in each direction: running independent tasks sequentially "just to be safe," wasting time for no real benefit; or parallelizing tasks that turn out to have a dependency nobody noticed, leaving one agent working from incomplete or missing information because the agent it actually depended on hadn't finished yet.

Mixed Patterns: Fan-Out and Fan-In

Most real workflows aren't purely one pattern or the other. A common mixed shape: three independent research agents run in parallel (a **fan-out**), and once all three finish, their combined findings feed sequentially into a single planning agent (a **fan-in**) that couldn't reasonably start until every one of those three results was actually available.

ASPECT	PARALLEL	SEQUENTIAL
Requires a dependency?	No — tasks are independent	Yes — each step needs the prior step's real result
Total time	Roughly the slowest single agent's own time	The sum of every step's own time
Coordination risk	Hidden dependencies causing conflicts	A stalled early step blocks everything after it

ASK THE DEPENDENCY QUESTION FOR EVERY ADJACENT PAIR EXPLICITLY

Before choosing a structure, walk through each pair of steps in the workflow and ask directly: does this step need the previous step's actual output? This simple, explicit check is what actually determines the right structure — defaulting to sequential "to be safe" or parallel "to be fast" out of habit, without asking this question, is how the two common mistakes above actually happen.

APPARENT INDEPENDENCE ISN'T THE SAME AS TRUE INDEPENDENCE

Two agents can look conceptually unrelated while still sharing a hidden, real dependency — for instance, both editing files inside the same shared module or directory, even though their actual tasks seem entirely separate on the surface. Running them in parallel in that case risks a race condition: one agent's changes conflicting with or overwriting the other's, a failure mode sequential execution simply doesn't have, since only one agent is ever touching anything at a time. Verify tasks are truly independent — not just conceptually different — before parallelizing them.

Hands-On Exercises

EXERCISE 1

A workflow has three steps: research a bug, write a fix, then write a changelog entry describing the fix. Explain which of these steps can run in parallel and which must run sequentially, using this chapter's own deciding question.

View solution

EXERCISE 2

Two agents are launched in parallel to update documentation for two different features. Both features happen to be documented in the same single README file. Explain what could go wrong here, using this chapter's own warning box.

View solution

EXERCISE 3

Describe a fan-out/fan-in workflow of your own (different from this chapter's own research-agent example), naming which steps fan out and which single step they fan into, and why that structure is appropriate.

View solution

CHAPTER 3 QUICK REFERENCE

- **Parallel**: independent tasks, run at once, total time is roughly the slowest single agent
- **Sequential**: dependent tasks, run one after another, each step needs the previous step's real result
- The deciding question for any two steps: does the second genuinely need the first's actual output?
- **Fan-out/fan-in**: several parallel agents whose combined results feed into one later sequential step
- Apparent independence isn't the same as true independence — shared files/state can create a hidden race-condition risk unique to parallel execution

Chapter 3's orchestration patterns both assumed agents run to completion within a bounded task, whether alone or as part of a chain. This chapter covers a different dimension entirely: agents that run over an extended period, and agents that deliberately repeat their own task on a schedule rather than finishing once and stopping.

Background Agents, Revisited

Fundamentals' Chapter 3 introduced background invocation: an agent continues independently while you keep working, with its result arriving later. This chapter extends that same idea to genuinely longer-running work — not "finishes in a minute or two while you do something else," but work that may legitimately continue for a long stretch, or work that needs to check back in repeatedly over time rather than running once straight through.

Scheduled Wake-Ups

Rather than either blocking synchronously or ending outright, an agent can pause and schedule itself to resume later — waking up again after a specific delay to check on something and decide what to do next. This matters for two genuinely different situations, and the right delay differs between them:

- **Polling something external you don't control** — checking whether a long-running external process has finished. The delay should match how fast that external thing actually changes: a process that typically takes about eight minutes deserves one check around that mark, not eight separate one-minute checks.
- **A long fallback heartbeat** — when something else is the real signal that work is ready to continue (a notification, an event), the wake-up is just a safety net in case that signal never arrives. This should be scheduled generously — not frequently "just to stay responsive," since a signal that's actually going to arrive will arrive regardless of how often you separately check.

Looping/Autonomous Agents

Rather than being relaunched by hand each time, an agent can be set up to repeatedly re-enter its own task on a schedule, continuing across many cycles until some genuine stopping point is reached. A meaningful distinction worth drawing: **dynamic pacing**, where the agent itself decides how long to wait before its next cycle based on what it's actually waiting on (matching the polling-vs-heartbeat distinction above), versus a **fixed schedule**, where the same interval applies every time regardless of what's actually happening.

Why Not Just Poll Constantly?

Checking back every few seconds "just to be safe" feels cautious, but it wastes real resources for no genuine benefit if nothing relevant could plausibly have changed that quickly. In any system where each check carries a real cost, unnecessary frequent wake-ups are an avoidable, ongoing expense — matching the wake-up cadence to what's actually being waited on, per the polling-vs-heartbeat distinction above, is not a minor optimization but a real, practical discipline.

MATCH THE DELAY TO WHAT YOU'RE ACTUALLY WAITING FOR, NOT A SINGLE DEFAULT

There is no one correct wake-up interval — the right delay depends entirely on what's being waited on. Fast-changing external state deserves a short, matched delay; a fallback heartbeat behind some other real signal deserves a long one. Always ask "what am I actually waiting for, and how fast does it realistically change" before picking a number.

ASPECT	ONE-SHOT BACKGROUND AGENT (FUNDAMENTALS CH.3)	AUTONOMOUS/LOOPING AGENT
Lifespan	Runs once, reports, ends	Repeats across many cycles until a real stop condition
Resuming	Relaunched by hand if more work is needed	Re-enters its own task automatically, on a schedule
Best suited for	A single bounded task	Ongoing monitoring or multi-cycle work with no fixed end time known upfront

A LOOP NEEDS A GENUINE STOP CONDITION, DECIDED IN ADVANCE

An autonomous loop with no real stopping point simply keeps consuming resources indefinitely without ever concluding anything — a real, avoidable waste, not a hypothetical concern. Before starting a loop, decide explicitly what actually ends it: a task genuinely being finished, an explicit request to stop, or some other concrete condition — not "it'll probably become clear when to stop."

Hands-On Exercises

EXERCISE 1

An agent is waiting on a deployment pipeline that typically takes about six minutes to finish. Explain what wake-up delay would be appropriate, and why checking every 15 seconds would be the wrong choice.

View solution

EXERCISE 2

Explain the difference between dynamic pacing and a fixed schedule for a looping agent, and describe a scenario where a fixed schedule would actually be the wrong fit.

View solution

EXERCISE 3

A developer sets up an autonomous agent to "keep monitoring the system and improving things," with no specific end condition defined. Using this chapter's own warning box, explain what's wrong with this setup.

View solution

CHAPTER 4 QUICK REFERENCE

- A scheduled wake-up should match either how fast an external process actually changes, or serve as a generous fallback behind some other real signal
- **Dynamic pacing** lets the agent decide its own next delay based on what it's actually waiting on; a **fixed schedule** uses the same interval regardless
- Polling constantly "just to be safe" wastes real resources without adding genuine benefit
- Every autonomous loop needs a genuine, decided-in-advance stop condition — not an assumption it'll become clear later

Fundamentals' Chapter 6 covered a general code-review agent: read-only, reporting structured, severity-ranked findings without fixing anything itself. A security & compliance review agent is a specialized version of that exact same pattern, narrowed onto a specific, higher-stakes category of issue — and it's worth being just as honest about what it can't catch as about what it can.

What Makes Security Review Different from General Review

A general code-review agent looks broadly at correctness, style, and quality. A security & compliance review agent narrows that focus deliberately: injection vulnerabilities, authentication and authorization gaps, exposed secrets or credentials, insecure use of dependencies, and data handling that violates a specific compliance requirement (like mishandling personal data). This narrower scope requires deeper, more specialized knowledge of actual attack patterns than a general-purpose review agent's system prompt would typically need to cover.

Designing a Security Review Agent's Definition

"Use this agent to check code changes for security vulnerabilities — injection risks, authentication/authorization flaws, exposed secrets — and relevant compliance requirements, before merging. This is not a substitute for a full security audit or a real penetration test." Tool access follows the same restricted pattern as the general review agent — `Read`, `Grep`, `Glob`, no `Edit` — and the isolation principle matters even more here: a security reviewer should never itself be the agent making changes to security-sensitive code. Model choice should favor the most capable model available, since genuinely subtle security issues require real judgment, not pattern-matching against a fixed checklist.

Writing the System Prompt for a Security Review Agent

A security review agent's system prompt should instruct it to check specifically for:

- Hardcoded secrets or credentials committed directly into code
- Injection risks — user-controlled input reaching a database query, a shell command, or similar, without proper handling
- Missing authentication or authorization checks on sensitive operations or endpoints
- Compliance-relevant patterns specific to the project — for instance, personal data being logged in plaintext where it shouldn't be

It should use the same structured finding format from Fundamentals' Chapter 6 (file, line, summary, failure scenario, verdict) — and a concrete failure scenario matters even more here than for a general finding, since a vague "this could be a security risk" gives no way to judge how serious or exploitable an issue actually is.

```
// Example structured security finding
file: src/api/search.ts
line: 18
summary: User-supplied search term is concatenated directly into a SQL query string
failure_scenario: A request with a search term like `' OR '1'='1` would alter the
query's logic, potentially returning records the requester shouldn't have access to —
the input is never parameterized or escaped before being inserted into the query
verdict: CONFIRMED
```

ASPECT	GENERAL CODE-REVIEW AGENT (FUNDAMENTALS CH.6)	SECURITY & COMPLIANCE REVIEW AGENT
Scope	Bugs, style, general quality	Vulnerabilities and compliance requirements specifically
Required expertise	General code understanding	Specific knowledge of attack patterns and applicable regulations
Tools	Read, Grep, Glob — no Edit	Same — no Edit, arguably even more important here

STATE THE SPECIFIC COMPLIANCE REQUIREMENTS THAT ACTUALLY APPLY

"Check for security issues" is far less useful than "this feature handles EU user data, so data-minimization and consent requirements apply — flag anything storing or logging more personal data than the feature genuinely needs." Compliance requirements vary by project and jurisdiction, and a generic brief has no way to know which specific ones are actually relevant.

THIS IS NOT A SUBSTITUTE FOR A REAL SECURITY AUDIT

A security review agent catching known, common vulnerability patterns is genuinely useful, but it is not equivalent to a comprehensive security audit or a real penetration test. Sophisticated or novel attack vectors, and issues in infrastructure or deployment configuration entirely outside the code itself, are genuinely outside what a code-focused review agent can catch. Treat its findings as one useful layer, not as a substitute for genuine, dedicated security review by the people or processes actually responsible for that.

Hands-On Exercises

EXERCISE 1

Explain why a security review agent is denied Edit access with even more emphasis placed on this than for a general code-review agent, using this chapter's own reasoning.

View solution

EXERCISE 2

A team briefs their security review agent with just "check for security issues" for a feature that stores health-related user data. Using this chapter's own tip box, explain what's missing from this brief and rewrite it.

View solution

EXERCISE 3

A team runs their security review agent, receives zero findings, and decides no further security review is necessary before their product launch. Using this chapter's own warning box, explain why this decision is risky.

View solution

CHAPTER 5 QUICK REFERENCE

- A security & compliance review agent specializes Fundamentals' general review agent onto vulnerabilities and compliance requirements specifically
- Same restricted tools (`Read` / `Grep` / `Glob`, no `Edit`) — isolation matters even more here
- Checks for hardcoded secrets, injection risks, missing auth checks, and project-specific compliance patterns
- A concrete failure scenario matters even more for a security finding than a general one
- State the specific compliance requirements that actually apply — a generic brief can't know which ones do
- This is one useful layer, not a substitute for a real security audit or penetration test

Chapter 5 specialized code review toward security. This chapter covers a different kind of specialization entirely: an agent whose job is to think through *how* something should be built — before any implementation begins — and produce a reviewable plan rather than working code.

What a Planning/Architecture Agent Actually Produces

A planning agent's output is a structured plan: the proposed approach, which files or areas of the codebase will be touched, the key architectural tradeoffs actually considered and the reasoning behind the recommended choice, and any genuinely open questions or risks. It does not produce the implementation itself — that's Fundamentals' Chapter 5's own coding agent's job, once a plan exists to actually implement.

Why Separate Planning From Implementation

Committing straight to an implementation approach without thinking it through first risks real wasted work if that approach turns out to be wrong partway through building it. Keeping planning as its own separate, reviewable step — proposing an approach, getting it reviewed and approved, *then* implementing — catches a flawed approach while it's still just a written plan, cheap to revise, rather than after a real implementation has already been built around it. This is the same underlying logic as reviewing code before merging it, applied one step earlier in the process.

Designing a Planning Agent's Definition

"Use this agent to design an implementation approach for a non-trivial feature or change before any code is written — identifying the files involved, the architectural approach, and the key tradeoffs. Not for writing the actual implementation." Tool access should stay read-only — `Read`, `Grep`, `Glob`, no `Edit` or `Write` — for the same reason a review agent stays read-only: keeping the plan itself as a discrete, reviewable proposal, rather than an agent that quietly starts implementing while it's supposed to be planning.

Writing the System Prompt for a Planning Agent

A planning agent's system prompt should instruct it to actually understand the existing codebase's own relevant conventions before proposing anything — a plan disconnected from how the codebase already works isn't genuinely useful — to name real tradeoffs explicitly rather than presenting one option as though it were the only possibility, and to flag genuine open questions or risks honestly rather than glossing over uncertainty to make the plan look more complete than it really is.

Briefing a Planning Agent Well

A good brief states the actual goal, any known constraints (must integrate with an existing system, can't change a specific public interface), and how much freedom the agent actually has — a genuinely open architectural question versus "given we've already decided on approach X, plan the specifics of implementing it."

A Worked Example

"We need to add real-time notifications to the app. Plan the approach — consider WebSockets, Server-Sent Events, and polling, and identify the tradeoffs between them for our specific situation: mostly one-way server-to-client updates, a moderate number of concurrent users. Don't implement anything yet — just produce the plan for review." This states the actual goal, names the specific options worth weighing, gives the real constraints that should inform the choice, and explicitly confirms this is a planning task, not an implementation one.

ASPECT	CODING AGENT (FUNDAMENTALS CH.5)	PLANNING/ARCHITECTURE AGENT
Output	Working, implemented code	A structured plan and its tradeoffs — no code
Tools	Read, Edit, Write, Grep, Bash	Read, Grep, Glob — no Edit/Write
When it runs	After the approach is decided	Before any implementation begins

ACTUALLY REVIEW AND APPROVE THE PLAN BEFORE IMPLEMENTING

The entire value of separating planning from implementation depends on the plan actually being reviewed before it's handed off — skipping straight from "the plan exists" to "implement it" without genuinely considering whether the proposed approach is right defeats the whole purpose of having a planning step at all.

A PLAN IS A SNAPSHOT, NOT A PERMANENT GUARANTEE

A plan is only as good as the planning agent's own understanding of the codebase and constraints at the moment it was written. If the codebase changes meaningfully between planning and implementation, or if a real constraint was missed during planning, the plan itself may need revisiting rather than followed blindly — it represents the best approach given what was known at the time, not an unquestionable, permanently correct answer.

Hands-On Exercises

EXERCISE 1

Explain why a planning agent is denied Edit access, using the same underlying reasoning this chapter connects to code review.

View solution

EXERCISE 2

A planning agent produces a plan that presents only one approach, with no mention of any alternative considered or rejected. Explain what's missing from this plan, using this chapter's own system-prompt guidance.

View solution

EXERCISE 3

A plan was approved two weeks ago, but before implementation began, an unrelated change significantly altered one of the systems the plan assumed would stay the same. A developer decides to implement the original plan exactly as written anyway. Using this chapter's own warning box, explain the risk here.

View solution

CHAPTER 6 QUICK REFERENCE

- A planning agent produces a **reviewable plan** — approach, files involved, tradeoffs, open risks — never the implementation itself
- Keeping planning separate from implementation catches a flawed approach while it's still cheap to revise
- No `Edit` / `Write` access — the same isolation logic as a review agent, keeping the plan a discrete proposal
- A good plan names real tradeoffs explicitly and flags genuine uncertainty rather than hiding it
- Actually reviewing and approving the plan is what makes the separation worthwhile — skipping that defeats the purpose
- A plan is a snapshot based on what was known when it was written — not a permanent guarantee if circumstances change

Chapter 1 of this course warned that Fundamentals' own "an agent starts cold" problem compounds across a chain — each handoff is a fresh opportunity for context to get lost. This chapter covers the actual mechanics of doing a handoff well, and why a long chain often needs more than just passing results from one step to the next.

What a Handoff Actually Requires

Fundamentals' Chapter 3 established what a good brief needs: the goal, known context, constraints, and the expected form of the result. A handoff between two agents is really the same thing, except the "known context" section is populated automatically from the prior agent's own final report, rather than written by a human from scratch. The job of designing a good handoff is making sure that automatic transfer actually carries forward everything the next agent genuinely needs — the same standard a well-written human brief would meet.

Passing a Full Report vs. a Distilled Summary

There's a real design choice in how much of a prior agent's output to hand to the next one. Passing the **full report** verbatim preserves everything, at the cost of length and noise — echoing Fundamentals' own point that unfiltered exploratory detail clutters a context without adding proportional value. Passing a **distilled summary** is cleaner, but introduces a real risk: whoever (or whatever) does the distilling might drop something the next agent actually needed, simply because it didn't look important at the time.

Shared State Beyond a Single Handoff

In a chain longer than two agents, a pure point-to-point handoff — where each agent only ever sees the immediately preceding agent's output — has a real limitation: something stated once at the very start of the chain (an original constraint, for instance) can silently disappear a couple of hops downstream if each successive summary only distills forward what the immediately prior step itself covered. A **shared state** — a running document or object carried alongside the chain, not just handed step to step — solves this by keeping information that matters for the whole chain visible throughout it, not just to whichever agent happened to receive it most recently.

A Worked Example

Consider a research agent, feeding a planning agent, feeding a coding agent — the same shape as this course's own Chapter 11 capstone. Suppose the original task states a hard constraint from Fundamentals' own Chapter 8 example: *the public API signatures must not change*. If that constraint is only ever passed forward as part of each step's own distilled summary, there's a real risk it gets quietly dropped by the second hop — the planning agent might mention it, but a summary of the planning agent's own output handed to the coding agent might not repeat it explicitly. Carrying that constraint in a shared state document, visible to every agent in the chain regardless of how many hops have happened, guarantees the coding agent still sees it at the very end, not just the step immediately after it was first stated.

ASPECT	POINT-TO-POINT HANDOFF	SHARED STATE
What persists	Only what the immediately prior step's own output included	Anything deliberately added, visible to every later step
Risk	Something stated early can silently drop after a few hops	Can grow unmanaged if everything gets added indiscriminately
Best suited for	Short chains, two or three steps	Longer chains where early constraints must stay visible throughout

CARRY THE ORIGINAL TASK'S HARD CONSTRAINTS EXPLICITLY, NOT JUST EACH STEP'S OUTPUT

The original task's own genuine constraints — things that must or must not happen, stated once at the start — are exactly the kind of information that silently drops out after a couple of hops in a pure point-to-point chain. Add them to shared state explicitly and keep them visible for the whole chain, rather than trusting each successive summary to keep repeating them on its own.

SHARED STATE CAN RECREATE THE SAME BLOAT PROBLEM, ONE LEVEL REMOVED

Dumping everything from every step into shared state indiscriminately just moves Fundamentals' own "exploratory noise clutters context" problem from a human's conversation into an agent chain's own shared state, rather than actually solving it. The real skill is deliberately choosing what's genuinely worth carrying forward — the original constraints, key decisions actually made — versus what was just one step's own disposable working detail that later steps don't actually need.

Hands-On Exercises

EXERCISE 1

Explain the trade-off between passing a prior agent's full report versus a distilled summary to the next agent in a chain.

View solution

EXERCISE 2

A four-agent chain relies purely on point-to-point handoffs, with no shared state. A constraint stated at the very start doesn't appear anywhere in the fourth agent's own brief. Explain why this happened, using this chapter's own material.

View solution

EXERCISE 3

A team, having learned about shared state, decides to add every single detail from every agent's output into it, reasoning "more shared context can only help." Using this chapter's own warning box, explain what's wrong with this approach.

View solution

CHAPTER 7 QUICK REFERENCE

- A handoff is the same brief-writing skill from Fundamentals Ch.3, populated automatically from the prior agent's own report
- **Full report**: nothing lost, but noisy. **Distilled summary**: cleaner, but risks dropping something needed
- **Shared state** keeps information relevant to the whole chain visible throughout — not just to the immediately next step
- Carry original hard constraints explicitly in shared state — they're exactly what silently drops after a few hops otherwise
- Indiscriminately dumping everything into shared state just recreates the same context-bloat problem one level removed

Fundamentals' Chapter 3 established "trust but verify" — an agent's report describes intent, not a guarantee. This chapter covers what happens once verification actually turns up something wrong: isolating where in a chain the mistake originated, telling a bad brief apart from a genuinely unreliable agent, and evaluating behavior systematically rather than reacting to one incident.

Isolating Where in a Chain Something Went Wrong

For a single agent, "trust but verify" means checking that one agent's own work. In a chain, a wrong final result could have originated at any step — the research agent found the wrong thing, the planning agent proposed a flawed approach despite correct research, or the coding agent implemented a good plan incorrectly. Debugging a chain means checking each step's own actual output against what it should have produced, rather than assuming the last agent in the chain is automatically the one at fault simply because its output is what surfaced the problem.

Distinguishing a Bad Input From a Bad Agent

An agent — even a genuinely well-designed one — given a poor brief, or working from an earlier step's own already-wrong output, will produce a wrong result. That's not the agent's own fault; it's a symptom of a problem that originated earlier in the chain and simply propagated forward, exactly as Chapter 7 described. Before concluding an agent itself is unreliable, confirm what it was actually working from was correct in the first place.

Evaluating Agent Behavior Systematically

Beyond debugging one failure, genuinely evaluating whether an agent — or a whole pipeline — is reliable over time benefits from a small set of representative test cases with known-correct expected outcomes, run repeatedly, rather than judging reliability from a handful of anecdotal results noticed in passing. This applies Fundamentals' Chapter 7's testing discipline to the agents themselves, not just to code.

When to Revise a Brief vs. When to Revise the Agent's Own Definition

If the same agent definition, given a new and better brief, now produces a correct result, the original problem was the brief — a per-task fix, not a sign anything is wrong with the agent itself. If the same agent, given an already-good brief, still produces the wrong kind of result repeatedly, the problem more likely lives in the agent's own definition — its system prompt, tool access, or model choice — and needs revising at that level, rather than re-briefed differently every single time it's used.

ASPECT	DEBUGGING A SINGLE AGENT	DEBUGGING A CHAIN
What to check	That one agent's own actual output	Every step's own output, working through the chain
Where the fault could be	The agent, or its brief	Any step, or the handoffs between them (Ch.7)
First move	Verify against the task's own requirements	Check the earliest step first, not just the last

CHECK THE EARLIEST STEP FIRST, NOT JUST THE LAST ONE

A wrong final result naturally draws attention to the last agent in the chain, since that's where the problem became visible — but the actual mistake may have happened several steps earlier and simply gone unnoticed until it reached the end. Working through the chain from the earliest step forward (or backward from the failure, methodically) finds the real origin faster than assuming the most visible step is automatically the culpable one.

ONE INCIDENT, IN EITHER DIRECTION, DOESN'T PROVE RELIABILITY OR UNRELIABILITY

A single anecdotal failure doesn't necessarily mean an agent's own definition is broken — it could simply be one unusual edge case its brief or design never anticipated. Just as importantly, a single anecdotal success doesn't prove an agent is reliable either. Judging based on one incident, in either direction, risks the wrong conclusion — this is exactly why systematic evaluation over multiple representative cases matters more than a gut reaction to any one result.

Hands-On Exercises

EXERCISE 1

A three-agent chain (research, plan, implement) produces an incorrect implementation. Describe how you would go about finding which step actually caused the problem, using this chapter's own guidance.

View solution

EXERCISE 2

A coding agent produces incorrect code once, and a developer immediately concludes the agent's definition needs to be rewritten. Using this chapter's own material, explain what should be checked first before reaching that conclusion.

View solution

EXERCISE 3

Explain the difference between fixing a problem by rewriting a single brief and fixing a problem by revising an agent's own definition, and how you would tell which one a given failure actually calls for.

View solution

CHAPTER 8 QUICK REFERENCE

- A wrong result from a chain could originate at **any** step, not necessarily the last one — check the earliest step first
- An agent given a poor brief or bad earlier output isn't itself broken — confirm its actual input was correct before blaming the agent
- Evaluate reliability over a **set of representative test cases**, not a handful of anecdotal results
- A new brief fixing the result → the brief was the problem. The same good brief still failing → revise the agent's own definition
- One incident, in either direction, doesn't prove an agent is reliable or unreliable

Several chapters in this course have touched a recurring theme in passing — orchestration has a real cost, worth weighing against its benefit. This chapter makes that theme the explicit main topic: what orchestration actually costs, what it genuinely buys, and how to decide when the tradeoff is actually worth it.

What Orchestration Actually Costs

Running multiple agents instead of one means more total model calls, more overall tokens processed across the whole workflow (since each agent needs its own briefing and context), and — in a sequential chain (Chapter 3) — more total wall-clock time than a single agent doing the whole task itself. These are real, measurable costs, not abstract concerns to wave away.

Context Costs Specifically

Fundamentals' Chapter 1 named a genuine context-efficiency *benefit* of delegating to a subagent: large, disposable exploratory work stays out of your own main conversation. But this course's own Chapter 2 (programmatic definitions adding maintenance complexity) and Chapter 7 (shared state risking its own bloat) both showed real costs on the other side of the ledger. The honest picture is genuinely two-sided — not "agents are always more context-efficient," but a real tradeoff to weigh in each specific case.

When Orchestration Pays For Itself

Fundamentals' own capstone closed with a direct point worth repeating here: the real payoff of formalizing a full agent team or pipeline comes specifically from doing that work at real, repeated volume, where a fixed setup cost gets amortized across many uses. Building an elaborate pipeline for a task that will genuinely only happen once or twice rarely earns back its own setup cost.

When It Doesn't

For a genuinely simple, one-off task, doing the work directly — without launching any subagent at all — is often both faster and lower in total overhead than briefing and launching a separate agent for it. Delegation is a tool reserved for tasks that genuinely benefit from isolation, parallelism, or specialized focus — not a default reflex applied to every request regardless of whether it actually needs any of that.

Parallel Agents and Cost

Chapter 3 established that parallel execution reduces total wall-clock time — but that speed benefit doesn't come free. Running several agents at once roughly multiplies the total resource cost by however many agents are running simultaneously, even though the elapsed time is shorter. Parallel orchestration is a genuine time-vs-total-cost tradeoff, not an unambiguous win on every dimension at once.

APPROACH	SETUP COST	BEST SUITED FOR
Direct work, no agent	None	A genuinely simple, one-off task
A single agent	Low — one definition, one brief	A task benefiting from isolation or specialized focus, done occasionally
A multi-agent pipeline	High — several definitions, handoffs, shared state	The same workflow repeated at real, ongoing volume

ESTIMATE REAL REPETITION BEFORE BUILDING A PIPELINE

Before investing in a multi-agent pipeline, roughly estimate how many times this exact workflow will genuinely be repeated. If the honest answer is "probably once or twice," a direct approach or a single well-briefed agent is very likely more efficient overall than the fixed setup cost of a full pipeline.

MATCH THE MACHINERY TO THE ACTUAL NEED, NOT TO WHAT'S TECHNICALLY POSSIBLE

It's genuinely tempting to build an elaborate multi-agent orchestration because it's technically interesting, or feels more sophisticated, rather than because the actual task genuinely requires it. This is the same lesson Fundamentals' own capstone closed on, restated here as this chapter's own central point: orchestration should be reached for because a task's real characteristics call for it — genuine independence worth parallelizing, a genuine need for isolation, real repeated volume — not because building the more elaborate version is possible or appealing on its own terms.

Hands-On Exercises

EXERCISE 1

Explain why running five agents in parallel is faster in wall-clock time but not necessarily cheaper overall than running the same five tasks one at a time.

View solution

EXERCISE 2

A developer is deciding whether to build a full multi-agent pipeline for a task they expect to run roughly twice a year. Using this chapter's own tip box, explain what they should conclude and why.

View solution

EXERCISE 3

A developer builds an elaborate five-agent pipeline for a simple, one-off task, explaining "I wanted to try out the orchestration patterns from this course." Using this chapter's own warning box, explain what's questionable about this reasoning.

View solution

CHAPTER 9 QUICK REFERENCE

- Orchestration has real, measurable costs: more model calls, more total tokens, more setup and maintenance
- Delegating exploratory work out of your own context is a genuine benefit — but it's one side of a two-sided ledger
- A pipeline's fixed setup cost pays off at **real, repeated volume** — not for a one-off task
- Parallel execution trades faster wall-clock time for roughly multiplied total resource cost — not a pure win on every dimension
- Match the machinery to the actual need — not to what's technically possible or interesting to build

Fundamentals' own capstone mapped seven agent types to seven real steps in generating a course chapter for website-content, run one at a time, by hand. Its own tip box named this exact capstone as where that gets formalized into a real, chained pipeline. This capstone builds it — applying every chapter of this course to turn that manual workflow into an automated one.

The Scenario

website-content generates course chapters repeatedly and often — this pipeline exists specifically because that repeated volume (Chapter 9's own deciding question) genuinely justifies building it, unlike a one-off task. The goal: take a chapter-generation request in and produce a rules-compliant, cross-reference-accurate chapter out, with tracking files updated automatically, and a human reviewing only what genuinely needs judgment.

Stage 1 — Fan-Out Research (Chapter 3)

Three research agents launch in parallel: one checking `course_folder_mapping.md` for the correct folder and filename convention, one checking `course_name_index.md` for the full course name, and one checking `completed_courses.md` for any existing entry that might overlap. These three checks are genuinely independent of each other — a real fan-out, per Chapter 3's own deciding question.

Stage 2 — Fan-In Planning (Chapter 6)

Once all three research agents finish, a planning agent proposes the chapter's actual outline, informed by their combined findings — this step genuinely cannot start until every one of the three parallel results is available, the fan-in half of Chapter 3's pattern. The plan itself stays a reviewable proposal, not an implementation, per Chapter 6.

Carrying Constraints Through Shared State (Chapter 7)

The original request's own hard constraints — the required P2 banner format, the P16 code-block/copy-button convention, this course's established accent color and wrapper class — are added to shared state at the very start, not just handed forward step to step. This guarantees they're still visible at the final review stage, several hops downstream, rather than risking silently dropping out along the way.

Stage 3 — Generation (Fundamentals Chapter 5)

A coding agent, briefed with the approved plan and the shared-state constraints together, writes the actual chapter — matching this project's own established per-course template exactly, the same way every chapter in both courses of this track was actually produced.

Stage 4 — Rules-Compliance Review (Fundamentals Chapter 6, specialized per Chapter 5)

A review agent — read-only, no `Edit` — checks the generated chapter against the shared-state constraints directly: is the banner present and correctly formatted, does every code block use the copy-button markup, is the wrapper class and accent consistent. This is the same specialized-review-agent pattern this course's own Chapter 5 applied to security — here applied to rules compliance instead.

Handling a Failure at Review (Chapter 8)

If the review stage finds a violation — a missing banner field, say — debugging means checking whether the plan itself specified the right convention in the first place, or whether the generation stage simply failed to follow a plan that was actually correct, per Chapter 8's own "check the earliest step first" guidance, rather than assuming the generation agent is automatically at fault.

Was This Pipeline Worth Building? (Chapter 9)

Applying Chapter 9's own deciding question honestly: website-content generates chapters at real, ongoing volume — not once or twice, but repeatedly across many sessions. This is exactly the profile Chapter 9 names as justifying a pipeline's fixed setup cost, rather than a case where a direct approach would have been more efficient.

Running Autonomously (Chapter 4)

Once built, this pipeline doesn't need to be launched by hand each time — it can run as a background process, triggered whenever a new bucket-list entry is marked ready, checking back on its own progress at a cadence matched to how long generation and review actually take, per Chapter 4's own wake-up guidance, rather than polling constantly or requiring manual restarts.

PIPELINE STAGE	CHAPTER IT DRAWS FROM
Stage 1 — Fan-out research	Chapter 3
Stage 2 — Fan-in planning	Chapter 6
Carrying constraints forward	Chapter 7
Stage 3 — Generation	Fundamentals Chapter 5
Stage 4 — Compliance review	Fundamentals Chapter 6, specialized per Chapter 5
Handling a review failure	Chapter 8
Deciding it was worth building	Chapter 9
Running it autonomously	Chapter 4
Built and orchestrated programmatically	Chapter 2

THIS FORMALIZES EXACTLY WHAT THIS WHOLE TRACK TAUGHT, NOTHING MORE

Every stage of this pipeline is a technique already covered somewhere in this course or Fundamentals — nothing new was introduced here. The capstone's own job was showing how those individually-taught pieces combine into one real, working system, not teaching anything beyond what the prior twenty chapters already established.

AUTOMATION STILL DOESN'T REPLACE JUDGMENT AT THE EDGES

Exactly as Fundamentals' own capstone found — 58 files processed correctly, 2 caught and fixed by hand — this pipeline handles the repeatable majority of chapter generation reliably, while genuinely novel edge cases (a chapter that doesn't fit the established pattern, a rule that's ambiguous for this specific case) still need a human's own judgment. Building this pipeline doesn't remove that need; it narrows down what actually requires it.

Hands-On Exercises

EXERCISE 1

Explain why the three research checks in Stage 1 are genuinely independent of each other, making them a valid fan-out, using Chapter 3's own deciding question.

View solution

EXERCISE 2

Explain why the accent color and wrapper class constraints are placed in shared state rather than only stated once in Stage 1's own brief, using Chapter 7's own reasoning.

View solution

EXERCISE 3

A different, much smaller project generates a new course chapter only once a year. Using this chapter's own Stage "Was This Pipeline Worth Building?" and Chapter 9's material, explain whether building this same pipeline would make sense for that project.

View solution

COURSE COMPLETE

Claude Code Agents: Advanced Orchestration — 10 of 10 chapters complete. Combined with Claude Code Agents: Fundamentals' own 11 chapters, the full Claude Code Agents track — 21 chapters in total — is now complete.

CHAPTER 10 QUICK REFERENCE

- Every technique from this course and Fundamentals combines into one real, working pipeline — nothing new is introduced in this capstone
- Fan-out research (Ch.3) → fan-in planning (Ch.6) → shared-state constraints (Ch.7) → generation → compliance review → failure isolation (Ch.8)
- The pipeline's own setup cost is explicitly justified by website-content's real, repeated chapter-generation volume (Ch.9)
- Runs autonomously in the background (Ch.4), built and orchestrated programmatically (Ch.2)
- Automation narrows down what needs human judgment — it doesn't eliminate the need for it entirely