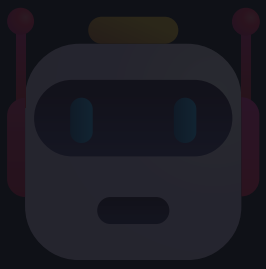

A I

Claude Code Agents: Fundamentals

Subagents, Tool Permissions & Isolation

Teaching, Coding, Review, Testing, Refactoring, Documentation & Research Agents

Capstone: Designing an Agent Team for website-content

Philip Osztromok

Generated with Claude

Table of Contents

Claude Code Agents: Fundamentals — 11 Chapters

Claude in VS Code: Pair Programming already covered the assistant you're used to talking to directly — the one that explains code, refactors it, writes tests, and debugs alongside you inside your own conversation. This course is about something different: **subagents** — separate, purpose-built agent instances you can delegate specific work to, each with its own context and its own restricted set of tools. Understanding exactly how a subagent differs from the assistant you already know is the foundation everything else in this course builds on.

The Default Assistant You're Already Using

Every message you send in a normal Claude Code session goes to the same assistant, in the same ongoing conversation. It remembers everything said earlier in that conversation, has access to whichever tools your setup grants it, and handles your request directly — reading files, editing code, running commands, all within that one continuous thread. This is exactly what Claude in VS Code: Pair Programming taught, end to end, and this course assumes it's already comfortable ground.

What a Subagent Actually Is

A subagent is a genuinely separate agent instance, launched to handle one specific task, rather than a continuation of your own conversation. It starts with its own independent context — it does not automatically inherit everything said earlier in your conversation with the default assistant — and it works through the task it's given, then reports back a result to whoever launched it. Some subagents run in the background, continuing their work while you keep talking to the default assistant about something else; others run synchronously, pausing the conversation until they finish.

Tool Permissions: Not Every Agent Gets Everything

A crucial structural fact about subagents: each agent *type* is defined with its own specific set of tools it's allowed to use — not necessarily the full set the default assistant has access to. A fast, read-only search agent built for exploring a codebase might be explicitly denied the ability to edit or write files at all, restricted to reading and searching only. A narrowly-scoped configuration agent might be granted only the two or three tools its one job actually requires, and nothing else.

Why Restrict Tool Access? This Is "Isolation"

Restricting an agent's tools isn't a limitation to work around — it's a deliberate design choice, and it's what this course means by **isolation**. An agent built purely to search and report back has no legitimate reason to modify a single file, so denying it write access isn't overly cautious — it's a guarantee: however that agent behaves, it structurally cannot alter your codebase, only describe it. The same logic applies to any narrowly-scoped agent — the tighter its allowed toolset matches what its actual job requires, the less that agent can do wrong, by construction rather than by hoping it behaves.

ASPECT	DEFAULT ASSISTANT	SUBAGENT
Context	Full ongoing conversation history	Starts fresh — no memory of your prior conversation unless briefed
Tool access	Whatever your setup grants	Whatever that specific agent type is defined with — often a restricted subset
Lifespan	Continues for the whole conversation	Runs for one task, then reports a result and ends
Best suited for	Ongoing, context-rich collaborative work	A well-defined, self-contained task, especially one you want isolated or run in parallel

A Concrete Example: A Search Agent vs. Doing It Yourself

Suppose you ask where user authentication is handled in a large, unfamiliar codebase. The default assistant could search directly itself — running several greps, reading a handful of candidate files, gradually narrowing in — but every one of those intermediate searches and file reads becomes part of your own conversation's context, whether or not any of it turned out to be useful. Delegating the same task to a dedicated search agent instead means all of that exploratory noise happens inside the agent's own separate context; only its final, distilled report — "authentication is handled in these three files, here's how" — comes back into your conversation. The search still happens; where the mess of getting there lives is what changes.

REACH FOR A SUBAGENT WHEN A TASK WOULD FLOOD YOUR OWN CONTEXT

Any task that involves a lot of exploratory searching, wide file reading, or otherwise-disposable intermediate output is a strong candidate for delegating to a subagent — it keeps your own conversation focused on the actual decision or result, rather than cluttered with search noise you'd never want to scroll back through anyway.

A SUBAGENT STARTS COLD — IT DOESN'T KNOW WHAT YOU KNOW

Because a subagent doesn't automatically inherit your prior conversation, launching one with a vague or under-explained task produces a shallow or wrong result — not because the agent is incapable, but because it genuinely doesn't have the context a human colleague reading your whole conversation would have. Writing a self-contained, well-briefed task is the actual skill this requires, and it's substantial enough to be Chapter 3's own entire topic.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, the difference between "the default assistant" and "a subagent," specifically in terms of context and lifespan.

View solution

EXERCISE 2

A search-only agent is deliberately denied the ability to edit or write files. Explain why this restriction is described in this chapter as a guarantee rather than a limitation.

View solution

EXERCISE 3

A colleague launches a subagent with the brief "fix the bug we were just discussing," expecting it to already understand which bug that refers to. Explain why this will likely fail, using this chapter's own warning box.

View solution

CHAPTER 1 QUICK REFERENCE

- The **default assistant** continues your ongoing conversation, with full access to its history
- A **subagent** is a separate instance launched for one task, starting with its own fresh context
- Each agent **type** is defined with its own specific set of allowed tools — often a restricted subset, not the full set
- **Isolation**: restricting an agent's tools to only what its job needs guarantees what it structurally can and can't do
- Delegate to a subagent when a task would otherwise flood your own conversation with disposable search/exploration noise
- A subagent must be briefed with enough context to work — it doesn't inherit your prior conversation automatically

Chapter 1 established that every agent type has its own restricted set of tools, and explained why that restriction matters. This chapter covers exactly where that restriction — and everything else about how an agent behaves — actually gets defined: a small, structured file with a specific set of fields, each controlling one aspect of the agent's identity and behavior.

Where an Agent Is Defined

A custom agent is typically defined as a markdown file with a block of structured metadata — **frontmatter** — at the top, followed by ordinary written instructions in the body. The frontmatter is machine-readable configuration; the body is the agent's own system prompt, written in plain language exactly the way you'd brief a new colleague. Both parts matter, and they do genuinely different jobs.

The Frontmatter Fields

- **name** — a short, unique identifier for the agent.
- **description** — a summary of what the agent does and, critically, *when* it should be used. This field is what a routing decision (or a person choosing between several available agents) actually reads to decide whether this agent fits a given task — a vague description makes the agent hard to select correctly, even if the agent itself is well built.
- **tools** — an explicit allowlist of which tools this agent may use. Omitting a tool from this list means the agent cannot use it at all, regardless of what its instructions say — this is where Chapter 1's isolation principle is actually enforced.
- **model** (optional) — pins the agent to a specific model rather than inheriting whatever model the parent conversation is using, letting a simple, high-volume task use a smaller/faster model while a genuinely difficult one uses the most capable model available.

The System Prompt Body

Below the frontmatter, the body is the agent's actual briefing — written instructions describing its role, how it should approach a task, and any conventions it should follow. Because Chapter 1 established that an agent starts with no memory of anything outside what it's given, the system prompt has to stand on its own: it should read like documentation for someone who's never seen this project before, not like a note assuming shared context that only exists in your own head.

```
---
name: changelog-writer
description: Use this agent after a feature or fix is finished and ready to record in
CHANGELOG.md. Do not use it for in-progress or exploratory work.
tools: Read, Edit, Grep
model: haiku
---
# Changelog Writer

You write concise, user-facing changelog entries. Read the relevant
diff or summary provided to you, then add one entry to CHANGELOG.md
under the "Unreleased" heading.

Rules:
- One line per change, present tense ("Add", not "Added")
- No implementation detail — describe the user-visible effect only
- Never remove or reorder existing entries
```

Tool Allowlists in Practice

The effect of the `tools` field is concrete, not theoretical. A dedicated read-only search agent might be granted every tool *except* the ones that edit or create files, making it structurally unable to change anything no matter what it's asked. A narrowly-scoped configuration agent, built for one specific settings task, might be granted only two tools — enough to read a file and edit it — and nothing else. Neither restriction is arbitrary: each one is sized to exactly what that agent's own job actually requires.

Model Selection: Matching Capability to the Task

Not every agent needs the most capable model available. A changelog-writer agent (as in the example above) is doing a small, well-defined formatting task — a faster, lighter model handles it perfectly well, at lower cost and higher speed. A code-review agent reasoning carefully about subtle correctness issues across an unfamiliar codebase genuinely benefits from the most capable model available. Choosing a model isn't just a cost setting — it's part of matching the agent's design to the actual difficulty of its job.

AGENT PROFILE	TYPICAL TOOLS	TYPICAL MODEL CHOICE
Read-only search agent	Read, Grep, Glob (no Edit/Write)	Fast/lightweight — task is retrieval, not deep reasoning
Code-review agent	Read, Grep, Glob (no Edit/Write — reviews, doesn't fix)	Most capable available — subtle correctness judgment
Narrow config agent	Read, Edit only	Fast/lightweight — task is small and well-defined

WRITE THE DESCRIPTION FIELD AROUND "WHEN," NOT JUST "WHAT"

A description like "reviews code" is far less useful than "use this agent after implementation is finished and before merging, to check for bugs and style issues — not for architecture decisions." The second version tells whoever's choosing an agent exactly which situation it fits, which is what the description field actually needs to communicate.

MORE TOOLS ISN'T AUTOMATICALLY MORE USEFUL

Granting an agent every available tool "just in case" doesn't make it more capable at its actual job — it only widens what could go wrong if it misinterprets an instruction or the task drifts. The discipline worth building here is the same one Chapter 1 introduced: match the tool allowlist tightly to what the agent's job genuinely requires, not to the maximum available, even when it would be more convenient to just grant everything upfront.

Hands-On Exercises

EXERCISE 1

Explain the difference in purpose between an agent's frontmatter and its system-prompt body, using this chapter's own terms.

View solution

EXERCISE 2

An agent's description field reads simply "helps with code." Explain why this is a poorly written description, and rewrite it to be genuinely useful, following this chapter's own tip box.

View solution

EXERCISE 3

A colleague grants a new documentation-formatting agent every available tool, including Edit and Bash, reasoning "it's simpler to just give it everything." Using this chapter's own warning box, explain why this reasoning is flawed.

View solution

CHAPTER 2 QUICK REFERENCE

- An agent definition has two parts: **frontmatter** (structured config) and a **system-prompt body** (written instructions)
- Frontmatter fields: **name**, **description** (what AND when to use it), **tools** (the allowlist), **model** (optional)
- Omitting a tool from the allowlist means the agent structurally cannot use it — this is where isolation (Ch.1) is actually enforced
- The system-prompt body should stand alone, like documentation for someone with no shared context
- Model choice should match task difficulty — a lightweight model for simple, well-defined work; the most capable model for subtle judgment
- Granting more tools than a job needs doesn't add useful capability — it only widens the blast radius of a mistake

Chapter 1 warned that a subagent starts cold, with no memory of your prior conversation, and promised this chapter would cover the actual skill of briefing one well. This chapter delivers on that — how to write a brief an agent can actually act on, how to choose between running it in the foreground or the background, and how to read what it hands back without over-trusting it.

What a Good Brief Actually Contains

A brief that only states the goal ("fix the bug") gives an agent starting from nothing to work with. A brief that actually succeeds includes:

- **What you're trying to accomplish**, stated plainly, and why it matters
- **What you already know** — file paths, specific line numbers, what you've already tried or ruled out — so the agent doesn't waste its own effort rediscovering ground you've already covered
- **Constraints** — what NOT to do, any approach already rejected, anything that must stay unchanged
- **What form the result should take** — a short summary, a specific list, a completed edit — so you get back something you can actually use directly

A useful habit: write the brief as though you're handing the task to a genuinely capable colleague who has never seen this codebase before — because that's functionally exactly what's happening.

> **NAME EXACT FILES AND LINE NUMBERS, NOT VAGUE DESCRIPTIONS**
>
> "The auth file" forces an agent to spend part of its own effort just locating what you meant. "`src/auth/login.ts`, around line 42" removes that ambiguity entirely, and lets the agent spend its effort on the actual task instead of on figuring out where to even start.

Foreground vs. Background Invocation

An agent can run in the **background** — continuing independently while you keep working or talking about something else, with its result arriving later as a notification — or in the **foreground**, which pauses the conversation until the agent finishes and returns its result immediately. Background is the natural default for anything exploratory or open-ended, since it doesn't block you from doing anything else in the meantime. Foreground makes sense specifically when you genuinely need the agent's result before you can take the very next step — a research question whose answer determines what you do next, for instance, rather than something you could just as easily review once it's ready.

ASPECT	BACKGROUND	FOREGROUND
While it runs	You can keep working on other things	The conversation waits for it to finish
When its result arrives	Later, as a notification	Immediately, before anything else continues
Best suited for	Exploratory or independent work	A result you need before your very next step

What Comes Back: The Agent's Final Report

You don't see an agent's own internal reasoning or every tool call it made along the way — only the final report it returns once its task is finished. That report is a summary written by the agent itself, describing what it did.

> **TRUST, BUT VERIFY — A SUMMARY DESCRIBES INTENT, NOT A GUARANTEE**
>
> An agent's own final report describes what it intended to do and believes it did — it is not automatically proof that the work is correct or complete. Before treating an agent's work as finished, check the actual result directly (the real diff, the real file, the real output) rather than accepting the summary at face value. This matters more, not less, the more consequential the task was.

Don't Race Ahead of a Background Agent

Once a background agent is launched, its results are genuinely unknown until its completion notification actually arrives — in a later turn, not the current one. Continuing to describe, predict, or act as though you already know what it will find is a mistake worth actively avoiding: if a background agent hasn't reported back yet, the honest answer is that it's still running, not a guess dressed up as a result.

Continuing a Conversation With an Agent

An agent that has already finished a task isn't necessarily gone for good — sending it a follow-up message resumes that same agent with its own full context from the earlier run intact, which is different from launching a brand-new agent of the same type, which starts cold again with no memory of what came before. Choosing to resume an existing agent versus starting fresh depends on whether the follow-up genuinely builds on what that agent already did, or is really a new, unrelated task.

Hands-On Exercises

EXERCISE 1

Rewrite the brief "fix the login bug" into a genuinely well-formed brief, following this chapter's own list of what a good brief contains.

View solution

EXERCISE 2

You need a research agent's findings before you can decide your very next step in a task. Explain whether you should launch it in the foreground or the background, and why, using this chapter's own comparison.

View solution

EXERCISE 3

An agent's final report says "successfully updated all references to the renamed function." Explain, using this chapter's own warning box, what you should do before accepting that this task is actually finished.

View solution

CHAPTER 3 QUICK REFERENCE

- A good brief states the **goal**, **known context** (exact files/lines), **constraints**, and the expected **form of the result**
- **Background** runs independently, result arrives later — the default for exploratory work
- **Foreground** blocks the conversation until finished — use only when you need the result before your very next step
- An agent's final report describes what it **believes** it did — verify the actual result before treating a task as complete
- Never predict or fabricate a background agent's results before its notification actually arrives
- A follow-up message can **resume** an existing agent with its own context intact — a new agent of the same type starts cold again

A teaching (or tutor) agent has a genuinely different purpose from most agents in this course: it exists to build *your own* understanding of a concept, not to produce a finished piece of work on your behalf. This chapter covers designing one, briefing it well, and — just as importantly — recognizing when it's the wrong tool for what you actually need.

Designing a Teaching Agent's Definition

The description field matters especially here, since it's easy to confuse a teaching agent with a coding agent if the description doesn't draw a clear line: "*Use this agent when you want a concept or piece of unfamiliar code explained so you understand it yourself. Do not use it when you want code written, fixed, or changed — use a coding agent for that.*" Tool access can stay minimal — typically just enough to read the specific code you're asking about (`Read`, `Grep`), with no need for `Edit` or `Write` at all, since this agent's whole job is explanation, not modification. Model choice should favor a genuinely capable reasoning model — a good explanation requires real understanding of the concept and the judgment to adapt it to what you already know, not just retrieving a definition.

Writing the System Prompt for a Tutor

A tutor agent's system prompt should instruct it to:

- Ask what the learner already knows before launching into a full explanation, rather than assuming a fixed starting level
- Ground new concepts in analogies to things the learner has already confirmed they understand
- Favor checking understanding along the way over delivering one long, uninterrupted explanation
- Define any jargon the moment it's introduced, rather than assuming it's already familiar

Briefing a Teaching Agent Well

Chapter 3's brief-writing principles apply directly here, with one addition specific to teaching: state your **current level of understanding** explicitly, not just the topic. "Explain decorators" gives a tutor agent nothing to calibrate against; "explain decorators — I'm comfortable with regular functions and passing functions as arguments, but I've never used the `@` syntax" gives it a genuine starting point to build from.

A Worked Example

Given the brief above, a well-designed tutor agent wouldn't open with a formal definition of decorators. It would more likely start from what's already confirmed known — "since you're already comfortable with passing a function as an argument, a decorator is really just a function that takes your function as an argument, does something around it, and gives back a new function to use instead" — before introducing the `@` syntax as a shorthand for exactly that pattern already just explained, rather than as a new, separate thing to learn from scratch.

> **TELL IT WHAT YOU ALREADY KNOW, NOT JUST WHAT YOU DON'T**
>
> A tutor agent has no way to guess your starting point on its own — per Chapter 1, it starts cold. Stating explicitly what you're already comfortable with is often more useful than stating what confuses you, since it gives the agent solid ground to build the explanation on top of.

When Not to Use a Teaching Agent

If what you actually want is code written or fixed quickly, routing that request through a teaching agent instead of a coding agent wastes time on an explanation you didn't need — you'll get understanding when what you wanted was a finished result. The two are genuinely different goals, and picking the right agent for which one you actually want in the moment matters more than it might seem.

> **A GOOD EXPLANATION ISN'T THE SAME AS A CORRECTNESS GUARANTEE**
>
> A teaching agent explaining why a piece of code works (or how a concept generally functions) is not the same as verifying that your own specific code is actually correct or bug-free. Don't mistake a clear, confident-sounding explanation for a review of your actual implementation — that's a code-review agent's job (Chapter 6), not this one's.

Hands-On Exercises

EXERCISE 1

Write a description field for a teaching agent that clearly distinguishes it from a coding agent, following this chapter's own example.

View solution

EXERCISE 2

Rewrite the brief "explain recursion" into a genuinely well-formed brief for a teaching agent, stating a specific starting level of understanding.

View solution

EXERCISE 3

A developer asks a teaching agent to "explain why this function is broken and what's wrong with it," expecting the code to also get fixed as part of the answer. Explain what's likely to go wrong with this expectation, using this chapter's own material.

View solution

CHAPTER 4 QUICK REFERENCE

- A teaching agent builds **your own understanding** — it doesn't produce finished work on your behalf
- Description should explicitly rule out coding tasks; tools can stay minimal (`Read` / `Grep`, no `Edit` / `Write`)
- Favor a capable reasoning model — good teaching requires real understanding and adaptation, not retrieval
- Brief it with your **current level of understanding**, not just the topic you want explained
- Don't route "just fix it" requests through a teaching agent — use a coding agent instead
- A clear explanation is not a correctness guarantee — that's a code-review agent's job

Where Chapter 4's teaching agent explains without acting, a **coding agent** is the opposite: a general-purpose delegate for actual implementation work — writing new code, modifying existing code, and verifying that what it produced actually works. This chapter covers designing one, briefing it with a genuinely usable spec, and reviewing what it hands back before trusting it's done.

Designing a Coding Agent's Definition

A coding agent's description should point at implementation work with a reasonably clear scope: *"Use this agent for a well-defined implementation task — adding a feature, fixing a specific bug, or making a scoped code change. Not for open-ended research or vague 'improve this' requests without a clear target."* Tool access is necessarily much broader than the teaching agent's — `Read`, `Edit`, `Write`, `Grep`, and typically `Bash` (so it can run the code or tests it writes and confirm the result works, rather than just assuming it does). Model choice should favor the most capable model available — correctness in real implementation work matters more than speed here.

Writing the System Prompt for a Coding Agent

A coding agent's system prompt should instruct it to:

- Follow the existing codebase's own conventions and style rather than introducing a different pattern
- Avoid adding abstractions, features, or refactoring beyond what the task actually asked for
- Verify its own change actually works — run the relevant tests or build, don't just assume the edit is correct
- Flag genuine ambiguity rather than silently guessing and proceeding on an assumption

Briefing a Coding Agent Well

Chapter 3's brief-writing principles apply here directly, with implementation work adding one more requirement: **acceptance criteria** — a concrete way to know the task was actually done correctly, not just attempted. A brief that says "add validation to the signup form" is vaguer than one that says exactly what should be validated and how a failure should be shown to the user.

A Worked Example

A well-scoped brief: "*Add input validation to the signup form in* `src/forms/signup.ts`*. Email must match a standard email pattern; password must be at least 8 characters. Show a clear, field-specific error message when either check fails. Don't change the form's overall structure, layout, or styling — only the validation logic and the error messages.*" This gives the agent a specific target file, exact validation rules, a defined success condition (clear per-field errors), and an explicit boundary (don't touch structure or styling) — everything it needs to implement the task correctly without guessing at scope.

> **STATE WHAT NOT TO TOUCH, NOT JUST WHAT TO BUILD**
>
> A coding agent with broad edit access can otherwise interpret an open-ended task as an invitation to "improve" things nearby that were never actually part of the request. Naming explicit boundaries in the brief — which files, patterns, or areas should stay untouched — prevents scope creep before it happens, rather than having to catch and undo it afterward.

Reviewing What a Coding Agent Hands Back

Chapter 3's warning about trusting a final report over the actual result applies with particular force here: implementation work is exactly the kind of consequential task that deserves real verification, not just acceptance of a confident-sounding summary. Review the actual diff, confirm it matches what was asked for, and check that anything outside the requested scope wasn't quietly touched along the way.

ASPECT	TEACHING AGENT (CH.4)	CODING AGENT
Purpose	Builds your understanding	Produces a finished code change
Typical tools	Read, Grep only	Read, Edit, Write, Grep, Bash
Output	An explanation	A code change, verified to work

AN AMBIGUOUS SPEC GETS FILLED IN, NOT LEFT BLANK

A coding agent given an incomplete or ambiguous spec generally doesn't stall and wait — it makes its own reasonable-seeming assumption and proceeds, the same way any capable collaborator might rather than blocking on every unclear detail. The fix for this isn't hoping the agent guesses correctly; it's the tighter, more complete brief Chapter 3 and this chapter both describe, written before the task is launched.

Hands-On Exercises

EXERCISE 1

Explain why a coding agent typically needs Bash access while a teaching agent typically doesn't, tying your answer to each agent's actual purpose.

View solution

EXERCISE 2

Rewrite the brief "clean up the payment module" into a well-scoped coding-agent brief, including acceptance criteria and an explicit boundary on what shouldn't change.

View solution

EXERCISE 3

A coding agent's final report says the requested feature is complete and working. Explain what you should still do before accepting that, and why this matters more for a coding agent than it might for some other agent types.

View solution

CHAPTER 5 QUICK REFERENCE

- A coding agent produces **a finished, working code change** — not just an explanation
- Needs broader tools than a teaching agent: `Read`, `Edit`, `Write`, `Grep`, typically `Bash` to verify its own work
- A good brief includes **acceptance criteria** — a concrete way to know the task was done correctly
- State explicit boundaries (what NOT to touch) to prevent unrequested scope creep
- Review the actual diff before trusting a "done and working" report — implementation work deserves real verification
- An ambiguous spec gets filled in with an assumption, not left blank — write a tighter brief instead of hoping it guesses right

A code-review agent examines code and reports what's wrong with it — it deliberately does not fix anything itself. Keeping "finding a problem" and "fixing a problem" as two separate steps, done by two different agents, is a deliberate design choice this chapter explains in full, along with how to structure what a review agent actually reports back.

Designing a Code-Review Agent's Definition

"*Use this agent after implementation is finished, before merging, to check for bugs, security issues, and style problems. Do not use it to make the fix — it reports findings only.*" Tool access is deliberately restricted to `Read`, `Grep`, and `Glob` — no `Edit` or `Write` at all. This is Chapter 1's isolation principle applied for a specific reason here: a reviewer that structurally cannot modify code keeps its findings honest and separate from the act of fixing, rather than quietly "fixing" something it noticed while reviewing and never actually reporting it as a finding. Model choice should favor the most capable model available — catching a genuinely subtle bug requires real judgment, not just pattern-matching against common mistakes.

Structuring Findings: Why Format Matters

An unstructured wall of prose — "I noticed a few issues, one around the loop and something with error handling" — is hard to act on and easy to skim past. A well-designed review agent reports each finding in a consistent shape: the specific file and line, a one-sentence summary of the problem, and a **concrete failure scenario** — the specific input or state that would actually trigger it, not a vague "this could be a problem in some cases."

Severity Levels

Findings should be ranked most-severe first, and it's worth distinguishing **confirmed** issues (verified as a real, reproducible problem) from **plausible** ones (looks likely but not fully certain without further testing). Treating every finding — a genuine security hole and a minor naming inconsistency — with identical urgency makes the truly serious ones harder to spot at a glance.

```
// Example structured finding
file: src/orders/discount.ts
line: 47
summary: Off-by-one error excludes the last item from the discount total
failure_scenario: A cart with 3 items only applies the discount to the first 2 — the
loop condition uses `< items.length - 1` instead of `< items.length`
verdict: CONFIRMED
```

Writing the System Prompt for a Review Agent

A review agent's system prompt should instruct it to surface genuine, concrete issues rather than padding out a list with stylistic nitpicks dressed up as bugs, to always state a real failure scenario per finding rather than a vague concern, to rank findings by severity, and — importantly — to treat an empty findings list as a perfectly valid, good outcome, not something to stretch for content to avoid appearing unhelpful.

ASPECT	CODING AGENT (CH.5)	CODE-REVIEW AGENT
Modifies code?	Yes	No — read-only, findings only
Typical tools	Read, Edit, Write, Grep, Bash	Read, Grep, Glob only
Output	A completed code change	A ranked, structured list of findings

GIVE IT THE ACTUAL DIFF, NOT THE WHOLE FILE

Reviewing an entire file when only a small section actually changed wastes the agent's attention on code nobody asked it to look at, and risks findings about pre-existing code unrelated to the current change. Provide the actual diff (or clearly state what changed) so the review stays focused on what's actually new.

ZERO FINDINGS ISN'T PROOF THE CODE IS BUG-FREE

A review agent reporting no issues means it didn't find anything within its own read of the code — not that the code has been proven correct. A review is a filter that catches what it's able to catch, not an exhaustive correctness guarantee; genuinely subtle bugs, especially ones depending on runtime data or external state, can still slip past any review, human or agent.

Hands-On Exercises

EXERCISE 1

Explain why a code-review agent is deliberately denied Edit access, even though it would be technically capable of fixing many of the issues it finds.

View solution

EXERCISE 2

A review agent reports the finding "the error handling in this file could probably be better." Explain what's missing from this finding, and rewrite it in this chapter's own structured format with a concrete failure scenario.

View solution

EXERCISE 3

A review agent returns zero findings on a piece of code, and the developer concludes the code is definitely bug-free. Using this chapter's own warning box, explain what's wrong with that conclusion.

View solution

CHAPTER 6 QUICK REFERENCE

- A code-review agent **finds** problems — it never fixes them itself, kept honest by deliberately having no Edit/Write access
- A good finding names the **file/line, a summary, and a concrete failure scenario** — not a vague concern
- Rank findings by **severity**, and distinguish confirmed issues from merely plausible ones
- An empty findings list is a valid, good outcome — don't pad findings out to seem thorough
- Give it the actual diff, not the whole file, to keep the review focused
- Zero findings means nothing was caught — not that the code is proven bug-free

Chapter 6 closed on an honest limitation: code review catches what's visible from reading code, but some real bugs only surface once the code actually runs against real data or a genuine edge case. A testing agent exists to catch exactly that gap — writing and running tests, rather than reviewing or implementing.

Designing a Testing Agent's Definition

"*Use this agent to write tests for new or existing code, or to run the existing test suite and investigate a failure. Not for implementing the feature itself — use a coding agent for that.*" Tools need to cover both writing test files and actually running them: `Read`, `Edit`, `Write`, and `Bash` (to execute the test suite and read its real output, rather than guessing at whether tests would pass). Model choice should favor a genuinely capable model — writing tests that would actually catch a real regression requires real understanding of the code's intended behavior, not just producing tests that trivially pass.

Writing the System Prompt for a Testing Agent

A testing agent's system prompt should instruct it to:

- Write tests that would genuinely catch a real regression — not tests that simply re-assert whatever the code currently happens to do
- Cover genuine edge cases (empty input, boundary values, error conditions), not just the obvious happy path
- Run the **full** test suite after any change, not only the newly written tests, since a fix in one place can silently break something unrelated elsewhere
- Report a failure with the actual error output, not a paraphrase of what it thinks the error means

Test-Driven vs. Test-After

A testing agent can work in either order: **test-after**, writing tests against code that already exists, or **test-driven**, writing tests first against an intended behavior that hasn't been implemented yet. The brief needs to state which mode is wanted, since test-driven work requires describing the intended behavior explicitly — there's no existing implementation for the agent to read and infer correct behavior from.

Briefing a Testing Agent Well

Following Chapter 3's own principles, a good brief for a testing agent states exactly what to test, what correct behavior looks like — especially for edge cases that aren't obvious from a quick read — and whether the task is writing new tests, fixing broken ones, or diagnosing a specific failure.

A Worked Example

"Write tests for the discount calculation function in `src/orders/discount.ts`*. Cover: the normal multi-item case, an empty cart, a single-item cart, and specifically the off-by-one edge case flagged in the recent code review, where the last item in the cart was being excluded from the discount. Use the same test framework and file conventions already used in this module's existing test files."* This tells the agent exactly what to cover, names the specific edge case a prior review already flagged, and points at the existing conventions to follow rather than inventing a new style.

ASPECT	CODE-REVIEW AGENT (CH.6)	TESTING AGENT
What it catches	Issues visible from reading the code	Issues that only surface when the code actually runs
Output	A ranked list of findings	Test files, plus a pass/fail result with real output
Modifies code?	No	Yes — writes test files (not the implementation itself)

ALWAYS RUN THE FULL SUITE, NOT JUST THE NEW TESTS

A fix that makes new tests pass can still silently break something unrelated elsewhere in the codebase. Instructing a testing agent to run the entire existing test suite after any change — not only whatever it just wrote — catches that kind of regression before it goes unnoticed.

A PASSING TEST SUITE PROVES THE TESTS PASS — NOT THAT THE CODE IS CORRECT

"All green" means the code behaves as the existing tests check for — it says nothing about behavior nobody thought to write a test for. A test suite is only as good as the cases it actually covers; a genuinely subtle bug in an untested edge case can sit behind a fully passing suite indefinitely, undetected not because testing failed, but because that specific case was never asked about.

Hands-On Exercises

EXERCISE 1

Explain the difference between test-driven and test-after work, and why a brief for test-driven work needs to include something a test-after brief doesn't.

View solution

EXERCISE 2

A testing agent writes and runs new tests for a bug fix, and reports all new tests pass. It did not re-run the rest of the existing test suite. Explain why this is a genuine gap, using this chapter's own tip box.

View solution

EXERCISE 3

A developer sees a fully passing test suite and concludes the code has no remaining bugs. Using this chapter's own warning box, explain what's wrong with that conclusion.

View solution

CHAPTER 7 QUICK REFERENCE

- A testing agent catches issues that only surface when code actually **runs** — the gap code review can't cover
- Needs `Read`, `Edit`, `Write`, and `Bash` — to write tests and actually execute them
- **Test-driven** briefs must describe intended behavior explicitly; **test-after** briefs can point at existing code
- Good tests cover genuine edge cases, not just the happy path
- Always run the **full** test suite after a change, not just the newly written tests
- A passing suite proves the code matches what the tests check for — not that it's free of untested bugs

Every agent covered so far succeeds by producing something new or catching something wrong. A refactoring agent has a stranger, almost inverted goal: success means the code's *observable behavior* stays exactly, provably the same, while its internal structure changes. This chapter covers designing an agent for that specific, unusual discipline.

What Makes Refactoring a Different Kind of Task

Chapter 5's coding agent succeeds by producing new, correct behavior. A refactoring agent succeeds by producing *no* behavior change at all — only a structural one (renaming, extracting a shared function, simplifying convoluted logic). This constraint changes what "done correctly" even means: it's not enough for the new code to look cleaner or work correctly in isolation — it has to be provably indistinguishable, from the outside, from the code it replaced.

Designing a Refactoring Agent's Definition

"Use this agent to restructure or clean up existing code — renaming, extracting shared logic, simplifying convoluted conditionals — without changing what the code actually does. Not for adding features or fixing bugs; use a coding agent for that." Tool access looks similar to the coding agent's own — `Read`, `Edit`, `Grep`, `Glob`, and `Bash` — but `Bash` here serves a specifically different purpose: confirming behavior didn't change, not just confirming new behavior works.

Writing the System Prompt for a Refactoring Agent

A refactoring agent's system prompt should instruct it to:

- Run the existing test suite **before** making any change, establishing a baseline of current behavior
- Make the structural change without altering any public-facing behavior
- Run the exact same test suite **after** the change and confirm identical results
- If the code being touched has no adequate existing test coverage, flag this explicitly rather than proceeding as though the absence of failing tests proves nothing broke

Briefing a Refactoring Agent Well

A good brief states what to restructure and why (reducing duplication, improving readability, isolating a pattern), and — critically — names explicit boundaries on what must not change, especially any exported or public-facing function signature that other code elsewhere depends on. It should also state what test coverage already exists for the code being touched, since that coverage is literally what the "before and after" comparison depends on.

A Worked Example

"Extract the repeated validation logic duplicated across the three functions in `src/forms/signup.ts`*,* `login.ts`*, and* `resetPassword.ts` *into a single shared helper function. The exported public function signatures in all three files must not change — only their internal implementation. Run the existing form-validation test suite before and after the change to confirm identical behavior."* This gives the agent a specific structural goal, an explicit boundary (unchanged public signatures), and a concrete way to verify nothing else changed (the existing test suite, run both before and after).

ASPECT	CODING AGENT (CH.5)	REFACTORING AGENT
Goal	Produce new, correct behavior	Change structure while behavior stays identical
Success criterion	The new feature works as specified	Before/after test results are identical
Bash's role	Verify new behavior works	Confirm behavior didn't change at all

CHECK TEST COVERAGE BEFORE REFACTORING, NOT AFTER

The "before and after" comparison a refactor depends on is only as trustworthy as the tests it's measured against. If the code being touched has thin or no existing coverage, confirm adequate coverage exists first — potentially delegating to a testing agent (Chapter 7) beforehand — rather than refactoring first and hoping nothing important was missed.

"TESTS STILL PASS" ONLY COVERS WHAT THE TESTS ACTUALLY CHECK

Exactly as Chapter 7 warned for testing generally, a refactor confirmed only by "the existing tests still pass" has only been verified against whatever behavior those tests happen to check — not against every behavior the code actually has. A refactor that subtly changes an untested edge case can pass every existing test while still having genuinely altered real behavior somewhere nobody thought to check.

Hands-On Exercises

EXERCISE 1

Explain how "success" is defined differently for a coding agent (Chapter 5) versus a refactoring agent, and why that difference changes what each agent's own Bash access is actually used for.

View solution

EXERCISE 2

A refactoring agent is asked to simplify a function that has no existing tests at all. Explain what it should do before making any structural change, using this chapter's own guidance.

View solution

EXERCISE 3

A refactor is completed, and the existing test suite passes identically before and after. The developer concludes the refactor definitely introduced no behavior change anywhere. Using this chapter's own warning box, explain what's wrong with that conclusion.

View solution

CHAPTER 8 QUICK REFERENCE

- A refactoring agent's goal is **no observable behavior change** — only structure changes
- Run the test suite **before and after** to establish and confirm a behavior baseline
- Flag missing test coverage before refactoring rather than proceeding without a way to verify nothing changed
- A good brief states what to restructure, why, and which public-facing signatures must not change
- Bash's role here is confirming behavior **didn't** change — a different purpose than a coding agent's own use of it
- Identical before/after test results only cover what those tests actually check, not every possible behavior

A documentation agent's job is to describe what code actually does — in a README, in docstrings, in a changelog entry — accurately, for a human reader. Its job is never to change what the code does. This chapter covers designing one, keeping it honest against the single biggest risk documentation faces, and knowing where its responsibility actually ends.

What a Documentation Agent Actually Does

This covers a range of related tasks: writing or updating a README, generating docstrings or comments, keeping a changelog current (recall Chapter 2's own `changelog-writer` example — a small, specialized version of exactly this agent type), or writing a longer-form guide. What unifies all of these is describing existing or newly-added behavior for a reader, not altering that behavior.

Designing a Documentation Agent's Definition

"Use this agent to write or update documentation — README sections, docstrings, changelog entries, guides — describing what code does. Not for changing code behavior itself; use a coding agent for that." Tool access needs care here: broad `Edit` access across any source file risks the agent "helpfully" adjusting actual logic while documenting it — a genuine temptation for a capable model trying to be useful. A tighter design scopes `Edit` to documentation files and comment/docstring regions specifically, keeping the same separation of concerns Chapter 6's review agent relies on: this agent describes, it doesn't modify behavior.

Writing the System Prompt for a Documentation Agent

A documentation agent's system prompt should instruct it to describe what the code **actually does** — verified by reading the real, current implementation — not what it was originally intended to do, or what older documentation claims it does. It's worth drawing one honest distinction here: a README or API doc legitimately explains *what* something does, for a reader who's never seen it before — that's the whole point of documentation. Inline code comments are a different context entirely, and should stay minimal, focused only on non-obvious *why* (a hidden constraint, a subtle workaround) rather than restating what well-named code already makes clear. A good documentation agent should know which context it's writing in and match its style accordingly.

Keeping Documentation Honest and Current

The single biggest risk with any documentation is **drift** — docs describing behavior that used to be true but no longer is, since nothing forces documentation to update automatically when code changes. A documentation agent's system prompt should explicitly instruct it to treat the actual current code as the source of truth, verifying behavior by reading the real implementation rather than trusting existing documentation, which may itself already be stale.

Briefing a Documentation Agent Well

A good brief states what needs documenting (a specific module, feature, or changelog entry), the intended audience (an end user reading a README versus a fellow developer reading an API reference), and whether the task is updating existing documentation or writing something fresh.

A Worked Example

"Update the README's API section to document the new `/users/:id/preferences` *endpoint added in* `src/routes/users.ts`*. Follow the existing README's format for documenting other endpoints exactly — same structure, same level of detail."* This names the specific addition to document, points at the actual source of truth (the real route file), and asks for consistency with the existing documentation's own established format.

ASPECT	CODING AGENT (CH.5)	DOCUMENTATION AGENT
What it changes	The code's actual behavior	The description of that behavior
Source of truth	The task's own spec/acceptance criteria	The actual current code, not existing (possibly stale) docs
Edit scope	Source/logic files	Documentation files and comment/docstring regions

POINT IT AT THE CODE, NOT THE EXISTING DOCS, AS THE SOURCE OF TRUTH

Existing documentation might already be out of date — treating it as authoritative risks faithfully reproducing an inaccuracy rather than fixing it. Instructing the agent to verify behavior against the real, current implementation catches drift instead of perpetuating it.

ACCURATE DOCUMENTATION ISN'T THE SAME AS VALIDATED CORRECTNESS

A documentation agent describing exactly what a function does — including any bug in its current behavior — has done its job accurately, even if that behavior is wrong. Clear, well-written documentation of buggy code is still documentation of buggy code; documenting something is not the same as reviewing or validating it. That's a code-review agent's job (Chapter 6), not this one's.

Hands-On Exercises

EXERCISE 1

Explain why a documentation agent's Edit access is often scoped specifically to documentation files rather than granted broadly across the whole codebase.

View solution

EXERCISE 2

Explain the distinction this chapter draws between what a README should describe and what an inline code comment should describe, and why a documentation agent needs to know the difference.

View solution

EXERCISE 3

A documentation agent accurately documents a function that actually contains a bug, describing its current (buggy) behavior correctly. A developer says this means the documentation agent failed at its job. Using this chapter's own warning box, explain why this criticism is misplaced.

View solution

CHAPTER 9 QUICK REFERENCE

- A documentation agent describes what code **actually does** — it never changes the code's own behavior
- Edit access is best scoped to documentation files/comment regions, not the whole codebase
- **READMEs/API docs** legitimately explain "what," for a new reader — inline code comments should stay minimal, focused on non-obvious "why"
- The real, current code is the source of truth — not existing documentation, which may already be stale
- A good brief states what to document, for which audience, and whether it's a fresh write or an update
- Accurate documentation of buggy code is still accurate — validating correctness is a code-review agent's job, not this one's

Chapter 1 already used a search agent as its very first concrete example of context isolation. This chapter goes deeper into designing that agent type properly — the fastest, most narrowly-scoped agent in this course, built purely to locate code and answer structural questions, and calibrated by how broad a search actually needs to be.

What a Research/Explore Agent Is For

This agent answers "where is X handled" or "how is Y structured" questions by searching a codebase and reporting back file paths, patterns, and structure — nothing more. It's distinct from Chapter 4's teaching agent: a teaching agent explains a concept once it's found, building understanding; a research agent's job stops at finding and reporting *where* and *how* something is structured, without necessarily explaining the deeper conceptual "why" behind it.

Designing a Research Agent's Definition

"Use this agent to locate where something is implemented, or to answer a factual question about the codebase's structure — especially when you're not confident you'll find the right match yourself in just a few tries. Not for making changes, and not for deep conceptual explanation." Tools are the clearest, most direct example of Chapter 1's isolation principle in this entire course: `Read`, `Grep`, and `Glob` only — no `Edit` or `Write` whatsoever, since this agent's entire job is finding and reporting, never changing.

Model choice here is a genuine trade-off, unlike most other agents in this course. A quick, narrow lookup can often use a faster, lighter model perfectly well. A genuinely wide, ambiguous search across an unfamiliar codebase — where the right search terms aren't obvious upfront — benefits from a more capable model able to reason about where to look next as each search comes back.

Search Breadth: Quick vs. Medium vs. Thorough

Specifying how broad a search should be helps a research agent calibrate its own effort correctly:

- **Quick** — a single targeted lookup, when you're fairly confident where the answer lives
- **Medium** — moderate exploration across a few likely locations
- **Thorough** — searching across multiple locations and naming conventions, for a genuinely uncertain or unfamiliar area of the codebase

Without this signal, a research agent might either under-search a genuinely hard question (stopping at the first plausible-looking match) or over-search a trivial one (spending effort exploring alternatives that were never really in doubt).

Writing the System Prompt for a Research Agent

A research agent's system prompt should instruct it to report exact file paths and line numbers — not vague locations like "somewhere in the auth folder" — to summarize findings concisely rather than dumping every raw search result verbatim, and to state honestly when it did *not* find a confident answer, rather than reporting an uncertain guess with unwarranted confidence.

Briefing a Research Agent Well

A good brief states the actual question clearly and, per the section above, the expected search breadth — "quick" for something you're fairly sure about, "thorough" for something genuinely unclear.

A Worked Example

"Find where rate limiting is implemented for the public API. I'm not sure whether it's applied per-endpoint or globally at the router level — a medium-depth search should be enough." This states the actual question, flags the genuine uncertainty driving the question (per-endpoint vs. global), and gives an explicit breadth expectation so the agent neither stops too early nor searches far wider than the question actually warrants.

ASPECT	TEACHING AGENT (CH.4)	RESEARCH/EXPLORE AGENT
Job stops at	Understanding — explains the concept once found	Locating — reports where/how, not necessarily deep "why"
Typical tools	Read, Grep	Read, Grep, Glob — no Edit/Write, ever
Model choice	Favor a capable reasoning model	Depends on search breadth — light for quick lookups, capable for wide/ambiguous ones

STATE THE SEARCH BREADTH EXPLICITLY

Saying "quick," "medium," or "thorough" up front is a small addition to a brief that meaningfully changes how the agent allocates its own effort — preventing both an overly shallow answer to a genuinely hard question and an unnecessarily wide search for something that was never really in doubt.

A CONFIDENT ANSWER IS ONLY AS GOOD AS WHAT THE SEARCH ACTUALLY COVERED

A research agent reporting "found it in file Y" is only as reliable as the search terms and locations it actually tried. If a codebase uses non-obvious naming, or the real answer lives somewhere the search never thought to look, the agent can miss the genuine answer entirely — and still sound just as confident about whatever it did find instead. Treating an unusually confident answer to a genuinely hard question with the same "trust but verify" instinct from Chapter 3 is worth applying here too.

Hands-On Exercises

EXERCISE 1

Explain why a research agent's ideal model choice is described in this chapter as a genuine trade-off, unlike most other agent types in this course.

View solution

EXERCISE 2

A brief simply says "find the caching logic," with no stated search breadth. Explain what could go wrong in either direction (too shallow or too wide a search), and rewrite the brief to include an appropriate breadth signal.

View solution

EXERCISE 3

A research agent confidently reports that a particular feature "is not implemented anywhere in the codebase." Using this chapter's own warning box, explain why this claim deserves some scrutiny before being accepted at face value.

View solution

CHAPTER 10 QUICK REFERENCE

- A research/explore agent **locates** code and structure — it doesn't explain deep concepts or make changes
- Tools: `Read`, `Grep`, `Glob` only — the clearest isolation example in this course
- Model choice is a genuine trade-off: lighter for quick lookups, more capable for wide/ambiguous searches
- State search breadth explicitly — **quick**, **medium**, or **thorough** — so effort matches the actual question
- It should report exact file paths/lines, summarize concisely, and admit when it didn't find a confident answer
- A confident "found it" (or "not found anywhere") is only as good as what the search actually covered — treat it with a "trust but verify" instinct

Every prior chapter covered one agent type in isolation. This capstone applies all seven — teaching, coding, code-review, testing, refactoring, documentation, and research — to one real, already-familiar project: **website-content**, the very course-generation system this course itself was built inside. Rather than a hypothetical scenario, this walks through which of these agents would genuinely help with the real work this project actually requires, and why.

The Scenario

A new course chapter needs to be generated: written in full compliance with the project's own rules (the required banner comment, the code-block/copy-button convention, the correct wrapper CSS class and folder placement), checked for accuracy against existing cross-references, and — once the course is actually finished — the project's own tracking files need updating to reflect what was really built. This is the exact workflow this project follows for every chapter, done here deliberately with a real agent assigned to each step.

Step 1 — Confirming the Convention Before Writing Anything (Chapter 10)

Before generating anything new, a **research agent**, given a thorough search breadth, checks the project's own mapping and index files for an existing entry under a different name or prefix — exactly the kind of overlap check that catches a proposed course accidentally duplicating one that already exists, before a single line of content gets written.

Step 2 — Explaining an Unfamiliar Convention (Chapter 4)

If someone new to this project needs to understand why every chapter requires a specific banner comment, or what the copy-button convention actually requires and why, a **teaching agent** — briefed with what that person already knows about HTML/CSS generally — walks them through the project's own specific conventions, rather than requiring them to read every rules file cover to cover unassisted.

Step 3 — Writing the Actual Chapter (Chapter 5)

The chapter itself is implemented by a **coding agent**, briefed with the exact course name, chapter number, accent color, wrapper class, and content outline — matching this project's own established per-course template precisely, the same way every chapter in this very course was actually produced.

Step 4 — Checking Rules Compliance (Chapter 6)

Rather than assuming the newly-written chapter is compliant, a **code-review agent** — restricted to reading only, no editing — checks it against the project's actual rules: is the banner comment present and correctly formatted, does every code block use the site's established copy-button markup, is the wrapper class consistent with the rest of the course. It reports findings; it never quietly fixes them itself.

Step 5 — Verifying Nothing Broke During a Reorganization (Chapter 7)

When files get moved between folders — exactly as happened during this project's own recent folder-structure migration — a **testing-agent**-style verification pass confirms the numbers genuinely reconcile: every file accounted for, none silently lost, matching the kind of before/after reconciliation check that migration actually relied on.

Step 6 — Restructuring Old Content Without Changing Its Meaning (Chapter 8)

Upgrading an old single-line course-header comment to the project's current, fuller banner format, or wrapping an existing code block in the copy-button markup, is a refactoring task in exactly this course's own sense: the lesson's actual content and meaning must stay identical — only its formatting structure changes, verified by confirming the visible lesson content reads the same before and after the change. This is a genuine **refactoring agent** job, not a coding-agent one.

Step 7 — Updating the Project's Own Tracking Files (Chapter 9)

Once a course is genuinely finished, the project's tracking files need updating to reflect the real, current state — describing what was actually built, verified against the real generated files, not against whatever a prior outline said before the course started. This is a **documentation agent**'s job specifically: describing current reality accurately, not modifying the course content itself.

SCENARIO STEP	AGENT USED	CHAPTER
Step 1 — Confirming the convention first	Research/Explore Agent	Chapter 10
Step 2 — Explaining an unfamiliar convention	Teaching Agent	Chapter 4
Step 3 — Writing the chapter	Coding Agent	Chapter 5
Step 4 — Checking rules compliance	Code-Review Agent	Chapter 6
Step 5 — Verifying a reorganization	Testing Agent	Chapter 7
Step 6 — Restructuring old formatting	Refactoring Agent	Chapter 8
Step 7 — Updating tracking files	Documentation Agent	Chapter 9

WHERE THIS SCALES UP NEXT

These seven agents, used one at a time as needed, are exactly what Claude Code Agents: Advanced Orchestration's own capstone formalizes into a real, chained pipeline — running several of these steps automatically, one after another, rather than launching each agent individually by hand every time a new chapter is generated.

NOT EVERY STEP NEEDS A DEDICATED AGENT EVERY TIME

For a single course chapter on its own, doing several of these steps by hand — as this project genuinely does for most of its day-to-day work — is often perfectly reasonable; assembling a full agent team has a real setup cost of its own. The actual payoff of formalizing this into a repeatable team comes from doing this work at real, repeated volume, not from using agents as a goal in themselves for their own sake.

Hands-On Exercises

EXERCISE 1

Explain why Step 1 (confirming the convention) uses a research agent rather than simply having the coding agent check for an existing course itself before writing.

View solution

EXERCISE 2

Explain why Step 6 (upgrading an old banner comment format) is described as a refactoring task rather than a coding task, using this course's own Chapter 8 distinction.

View solution

EXERCISE 3

A developer decides to build and deploy all seven of these agents before generating even a single course chapter for this project. Using this chapter's own warning box, explain what consideration they should weigh first.

View solution

COURSE COMPLETE

Claude Code Agents: Fundamentals — 11 of 11 chapters complete. Claude Code Agents: Advanced Orchestration remains outstanding as this two-course track's own next planned course.

CHAPTER 11 QUICK REFERENCE

- Every agent type from this course maps to a real, specific step in this project's own actual workflow
- **Research** agent confirms conventions first, **teaching** agent explains them, **coding** agent implements, **review** agent checks compliance, **testing** agent verifies reorganizations, **refactoring** agent restructures old formatting, **documentation** agent keeps tracking files current
- None of these steps requires a new capability beyond what Chapters 1–10 already covered — this capstone only applies them together
- Formalizing a full agent team has its own real setup cost — it pays off at repeated volume, not for a single one-off task