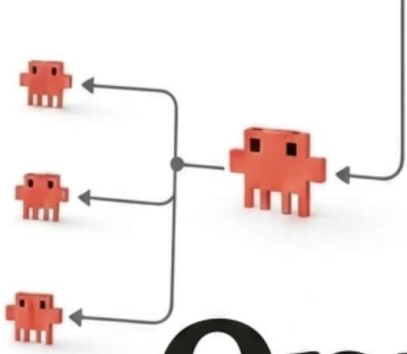
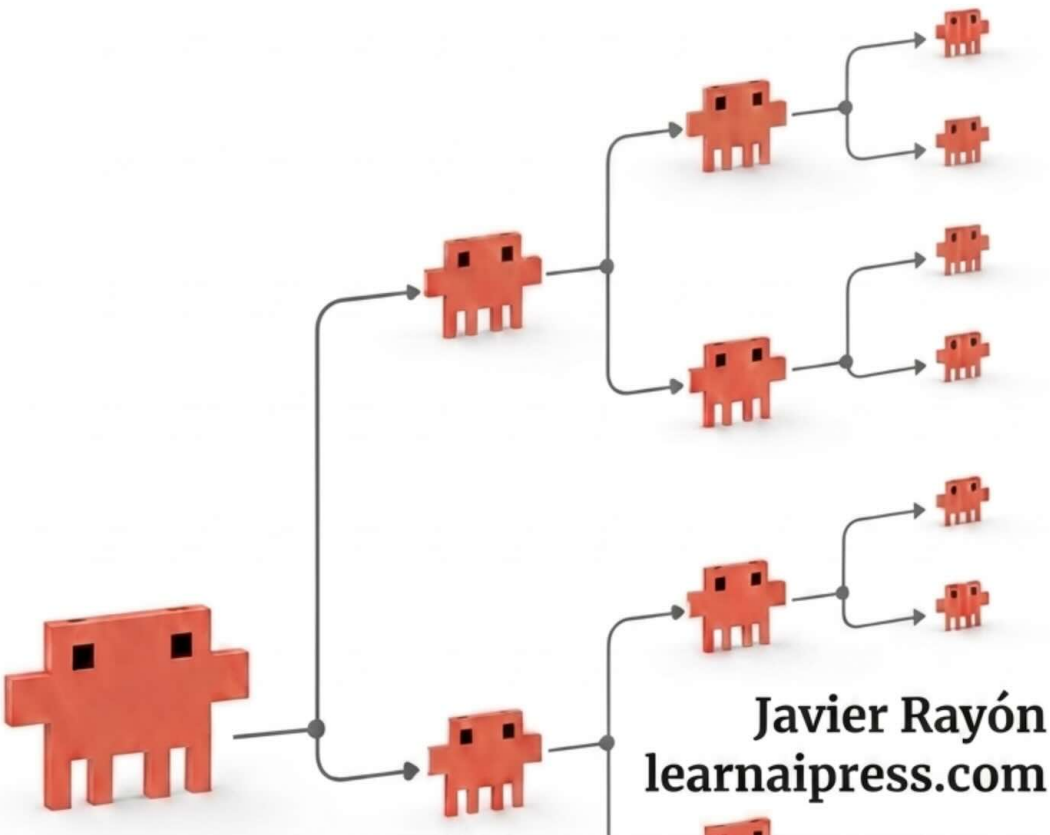


Opus 5 Migration Guide



Javier Rayón
learnaipress.com

Contents

1 The Four Models in the Claude 5 Family 2

2 Which Model to Use for Each Task 4

3 Six Prompt Patterns That No Longer Work 6

4 Effort vs. Length 10

5 Reasoning On by Default 12

6 New Commands and Background Subagents 14

7 The 20-Minute Migration Checklist 17

8 What Each Book in the Series Covers 20

Migrating to Claude Opus 5 What changed, what to delete from your prompts, and how to update in 20 minutes

© 2026 Javier Rayón. All rights reserved.

Published by learnaipress.com

Free-distribution guide. You may share the complete, unmodified file. No part of this publication may be reproduced in part, altered, or within another work without the author's prior written permission.

Product and service names mentioned here are trademarks of their respective owners. This guide is not affiliated with or sponsored by Anthropic.

The model data, pricing, and behaviors described here reflect the platform documentation as consulted on August 1, 2026. Prices are expressed in dollars per million input and output tokens. This information is provided without warranty of any kind, and the author is not liable for damages arising from its use.

First edition: September 2026.

SECTION 1

The Four Models in the Claude 5 Family

Claude 5 is a model family that shipped in three stages between June and July 2026. Fable 5 arrived on June 9, Sonnet 5 on June 30, and Opus 5 on July 24. Anyone searching for "the Claude 5 launch" found three separate announcements weeks apart, each with its own pricing table and its own changelog. That's where most of the confusion still circulating comes from.

The last of the three, Opus 5, delivers frontier intelligence at half the price of Fable 5, and at the same price as the model it replaces, Opus 4.8. It also became the default Opus model in Claude Code, which means some teams have been working with it for weeks without ever consciously choosing it.

1.1 The family, at a glance

Model	Available since	Price $/Mtok	Context	Profile
Fable 5	Jun 9, 2026	$10 / $50	1M	The top rung of capability. Always reasons, with no option to turn it off
Opus 5	**Jul 24, 2026**	**$5 / $25**	**1M**	Frontier intelligence. Claude Code's default Opus model
Sonnet 5	Jun 30, 2026	$3 / $15	1M	The workhorse of everyday development

Model	Available since	Price $/Mtok	Context	Profile
Haiku 4.5	Oct 2025	$1 / $5	200K	Volume, mechanical tasks, reading subagents

The three Claude 5 family models share a million-token context window and a 128,000-token maximum output. Haiku 4.5 stays at 200,000 tokens of context and 64,000 of output, which is plenty for the work it typically does.

Opus 4.8 remains available, and it still costs exactly what Opus 5 costs. The difference between the two plays out in behavior, not on the invoice.

1.2 Two pricing traps

The first has circulated since launch day. Several secondary sources place Opus 5 at $10 / $50. That figure belongs to **fast mode**, the latency-optimized variant billed at its own rate. The model's actual price is $5 / $25.

The second affects Sonnet 5, which launched at a promotional price of $2 / $10 with an expiration date set for August 31, 2026. Anyone who budgeted several months ahead on that figure faces a 50% higher bill without having changed a single line of code. The number to plan around is $3 / $15.

1.3 The prompt-cache threshold drops to 512 tokens

Opus 5 lowers the minimum prompt-cache size to 512 tokens, down from the 1,024 tokens that Opus 4.8 and Sonnet 5 require. System prompts that used to fall below the threshold, and got billed in full on every call, now qualify for caching on their own. Revisit the small agents that missed out on that saving by a few hundred tokens.

SECTION 2

Which Model to Use for Each Task

Start from the task on the table rather than from a comparison of capabilities in the abstract, and walk away with two decisions made: which model, and how much effort.

Work	Model	Effort
Large refactors, architecture design, debugging that won't budge	Opus 5	`high` (the default)
Scoped feature, code review, tests, repetitive migrations	Sonnet 5	`medium`
Repo search, summaries, mass renames	Haiku 4.5	— (doesn't support effort)
A problem where you don't yet know what needs figuring out	Fable 5	always reasons
Interactive work where latency is what matters	Opus 5, fast mode	`high`

Getting the Fable 5 row wrong costs the most money. Its territory is genuine exploratory work, the kind where the problem doesn't yet have the shape of a task. That is what it was designed for, and the framing its own documentation uses. A large refactor, however hard, already has the shape of a task, and Opus 5 solves it at half the price.

Opus 5's fast mode keeps the same model and the same reasoning quality behind it, optimized for latency and billed at $10 / $50. It pays off in long interactive sessions, where wait time is paid for in human attention. For background work, it doesn't earn its keep.

2.1 When to scale down

Opus 5 performs well at low effort. That shift turns `low` and `medium` into the primary cost control, ahead of any adjustment to context or prompt. The effort ladder runs from `low` to `max` and defaults to `high`, so the default sits above what many tasks need.

Scale down when the task is fully specified from the first turn, when the pattern has already been solved several times in the same repository, or when a test suite verifies the result and returns the signal within seconds. In all three cases, the extra reasoning gets spent confirming something that was already decided.

Scale up when the problem has no known solution in the project, when constraints cross each other — performance, backward compatibility, and security all at once — or when a mistake costs more than the time it would take to think it through more carefully.

Switching to a smaller model is a different move from lowering effort, and it belongs to high-volume mechanical work: locating where a function lives, extracting a module's signatures, summarizing a long log. Haiku 4.5 costs a fifth of what Opus 5 costs and handles that work comfortably. Subagents accept their own model, separate from the main session's, which lets you reserve Opus 5 for the thread that decides and leave the searching to Haiku.

2.2 A number to calibrate against

Claude Code's cost documentation publishes a per-developer reference: around $13 per active day, $150–250 a month, with 90% of users under $30 a day. Use it as a thermometer, not a target.

Anyone comfortably above that band is rarely there because they chose the wrong model. The two usual causes are uncapped delegation and context loaded with dead weight, and both have a fix in sections 3 and 6 of this guide.

SECTION 3

Six Prompt Patterns That No Longer Work

Opus 5 verifies its own work without being asked. It reviews what it just wrote and corrects itself before responding. That behavior ships by default, and it **compounds** with any verification instructions you already had written. The model verifies once on its own, and again because your prompt told it to. The result is token spend that doesn't improve the answer.

Anthropic's official prompting documentation for Opus 5 is unusually direct on this point, and it uses the imperative *remove them* when talking about those instructions. Take that literally.

The inventory covers six patterns that held up until July 2026 and now work against you.

What you had written	What happens now	What to do
"Include a final verification step"	Over-verification: spends tokens without improving the result	Delete it
"Use a subagent to verify the work"	Doubled cost and redundant verification	Delete it. Verification lives in the main loop
"Review it twice before responding"	Compounds with native verification and duplicates it	Delete it
"Summarize progress every three calls"	The model already narrates its progress, and then some	Delete it
"Use subagents whenever you can"	Over-delegates; multiplies cost and time	Replace with an explicit cap
"Don't think, just answer directly"	Increases leakage of internal tags	Delete it

3.1 The three verification instructions

The first three rows are variants of the same order, and they get deleted the same way. It doesn't matter whether they're phrased as a plan step, an acceptance criterion, or a closing line tacked onto the prompt.

```
# Before
Implement the endpoint according to the spec. When you're done,
include a final verification step where you review your own work
twice and confirm it meets every requirement.

# Now
Implement the endpoint according to the spec.
```

The saving that matters is on the output side rather than in the input tokens the cut removes, because every review order opens an additional round of reasoning and rewriting over work the model had already checked on its own.

The subagent variant is the most expensive of the three. Delegating verification opens a new session that inherits context, rereads the files, and produces its own report, all to confirm something the main thread had already confirmed. The cost doubles, and the judgment call splits between two threads that share no state.

3.2 The progress scaffolding

Opus 5 narrates what it's doing in noticeably more detail than Opus 4.8. In a long agentic session it explains which file it's opening, what it finds, and what it decides next, without being asked.

Instructions like "summarize progress every three tool calls" or "keep an updated task list on every turn" were written for models that stayed too quiet. Applied now, they stack narration on top of narration.

```
# Before
Every three tool calls, summarize what you've done and what's left.

# Now
(nothing)

# And if the session comes out too narrated
Report only when you finish a step of the plan or get stuck.
```

3.3 The push toward delegation, now reversed

With Opus 4.8, you had to push the model to delegate. It resisted opening subagents, and that's how the advice to add "use subagents whenever you can" to any moderately large request spread. That advice appears in earlier books in this same series, written for the model of that era.

Opus 5 delegates far more readily. The same line that used to unlock parallelization now opens subagents for tasks that were better handled in the main thread, and each one starts with its own context, its own file reads, and its own bill.

This row of the inventory gets replaced rather than deleted:

```
# Before
Use subagents whenever you can to parallelize the work.

# Now
Launch at most two subagents, and only for parallel searches over
different parts of the repository. Everything else runs in the
main thread.
```

Two is a reasonable starting point for normal development work; a large-repository audit justifies more. Leaving the decision open-ended no longer works.

3.4 The order not to think

"Don't think, just answer directly" and its variants were written to save latency on trivial answers. With reasoning on by default, that order pushes the model into a mode its outputs are no longer calibrated for, and one observed effect is that the internal reasoning tags leak into the visible response.

Asking for concision explicitly is the better tool for shortening a response. Section 4 covers it in detail.

3.5 What stays untouched

This rule targets the model's redundant self-verification. It leaves quality engineering exactly as necessary as it's always been.

Planning **how** a change will be verified is human work, and good practice. Asking for a verification plan as a deliverable — which tests to write, which edge cases to cover, what to check by hand before deploying — produces a document another person can review. That stays.

Automated tests, linters, continuous integration, human code review, and any check that runs outside the model also stay. All of them are independent systems that produce external signal, so none of them fall inside the inventory's "final verification step."

One question sorts them: does the instruction ask the model to review itself, or does it ask for an artifact someone else will use? The first gets deleted; the second gets kept.

3.6 Where all of this lives

The inventory is rarely in one place. The six patterns accumulate over months across five different locations.

- The project's `CLAUDE.md` and the user's global one.
- Custom skills, especially in their "acceptance criteria" sections.
- Custom commands in the configuration folder.
- Prompt templates saved outside the repository.
- The system prompts of any agent you've put into production against the API.

The last two get forgotten most often and cost the most, because they run with nobody watching.

3.7 The 80% cut in Claude Code's own system prompt

Anthropic makes the case itself. Claude Code's system prompt went from roughly 800 tokens to 164 — an 80% cut — without losing performance.

Much of what accumulates in a prompt compensates for the shortcomings of an earlier version, rather than steering the model's behavior. Once the shortcoming disappears, the instruction stays there collecting a toll.

Deleting is, in this context, an engineering decision with a measurable result: fewer input tokens, fewer rounds of reasoning, and less competition among instructions for the model's attention.

SECTION 4

Effort vs. Length

Effort controls how much the model **thinks** before responding. Response length gets requested in the prompt. They are two separate controls, and confusing them is the most common mistake since Opus 5 launched.

Both controls seem to say "less," which explains the confusion. Someone gets a two-screen answer when they expected a paragraph, sees a parameter called effort, and lowers it from `high` to `low`. The answer still runs two screens long, only worse thought through.

	Effort	Length
What it controls	How much the model reasons before responding	How much text it delivers
Where it's set	In the model configuration, from `low` to `max`	In the prompt, with an explicit instruction
What it spends	Reasoning tokens	Visible output tokens
If you use it for the other one	Worsens the result without shortening the response	—

There is no verbosity parameter. Length is governed with natural language, inside the prompt, the same way you'd specify any other requirement for a deliverable.

4.1 Where the extra length comes from

Opus 5 writes more than Opus 4.8 on three fronts at once. Its responses to the user are longer, the documents it writes to disk are longer, and it narrates its own work in more detail. Each of the three gets fixed with a different instruction, and none of them with effort.

A fourth tendency adds to that: widening the assignment. Given a scoped task, Opus 5 tends to also fix what's around it — it patches the test that was already failing, updates

the dependency it noticed was outdated, renames something along the way that struck it as confusing. All of that fattens the diff, and with it, the response.

4.2 The four instructions, ready to copy

Put them in the project's `CLAUDE.md` if you want the behavior permanently, or in the prompt if you want it just this once.

Response concision.

```
Answer in five sentences or fewer. No preamble, no closing recap,
and don't repeat what's already in the previous message.
```

Length of a deliverable written to disk.

```
The document must not exceed 400 words. One section per item in
the outline, no introduction and no conclusion.
```

Scope discipline.

```
Stick to what was asked. If you notice adjacent work worth doing,
list it at the end in three lines or fewer, without doing it.
```

Narration of corrections.

```
Narrate only the corrections that change a prior decision. Apply
detail-level corrections without commenting on them.
```

An instruction like "be concise" leaves the bar wherever the model estimates it, and Opus 5's estimate runs generous. The specific number — five sentences, 400 words, three lines — settles the argument.

4.3 Combinations that work

The common case in day-to-day development is a well-specified task where you want a correct result and a short answer. The combination that works is `medium` effort with a concision instruction.

A hard problem that needs documenting for the team calls for `high` effort and a generous length instruction, with the document's outline given up front.

High effort with no length instruction is the combination that happens by accident, and it produces the three-thousand-word reports nobody asked for.

SECTION 5

Reasoning On by Default

On Opus 4.8, omitting the reasoning block from a request meant the model didn't reason. On Opus 5, that same request, byte-for-byte identical, reasons. The default behavior switched sides quietly, because nothing in the response flags that reasoning is now happening where it wasn't before.

This section matters most to anyone calling the API from their own code. Anyone working exclusively through Claude Code inherits a configuration that's already sorted out and can skim this one.

5.1 `max_tokens` now covers reasoning too

`max_tokens` is a hard cap on reasoning **plus** the response text. The two share the same budget.

```
{
  "model": "claude-opus-5",
  "max_tokens": 4096,
  "messages": [ ... ]
}
```

Until July 2026, a request like that reserved 4,096 tokens for the response. Today it reserves 4,096 tokens for reasoning and response combined, and reasoning goes first. A workload that used to run with a tight but working margin starts coming back cut off halfway through, on code that never changed.

Search the logs for responses whose stop reason is the token limit. If they start appearing from the date you switched models, that's the cause. Opus 5's maximum output reaches 128,000 tokens, so there's plenty of room to raise the cap.

5.2 Disabling reasoning conflicts with high effort

Reasoning can be disabled, with one caveat that tends to surface in production. A request that combines disabled reasoning with effort set to `xhigh` or `max` returns a 400.

```
{
  "model": "claude-opus-5",
  "thinking": { "type": "disabled" },
  "max_tokens": 4096,
  "messages": [ ... ]
}
```

The combination is validated on every request, not when the session opens or when the service deploys. A service that sets effort by task type can run fine for days and start failing the first day a task arrives with effort set high.

5.3 The two artifacts of reasoning turned off

With reasoning disabled, two behaviors show up, and neither one presents itself as an error.

The first is a tool call written **as text**. The model drafts the invocation inside its response instead of emitting the structured block the client executes. The turn ends cleanly, the response looks reasonable, and the tool never runs. It's the harder of the two failures to diagnose, because nothing raises an exception or an error code. The only symptom is the file that never got written or the query that never ran.

The second is the internal reasoning tags leaking into the visible output. This one detects itself, because the end user sees it.

5.4 The mitigation

The same fix applies to both artifacts. Leave reasoning on and control cost by lowering effort instead.

It works because Opus 5 performs well at low effort. The savings lever that used to be "turn reasoning off" becomes "reason less," and this second version carries neither artifact. In practice, the low-cost configuration for mechanical work is reasoning on with `low` effort.

Fable 5 is the exception. Reasoning is always on there, with no option to disable it. Any code that assumes it can turn reasoning off before calling Fable 5 needs a separate branch.

SECTION 6

New Commands and Background Subagents

The Claude Code changes that ship alongside Opus 5 are extensive. The ones below are ordered by how much they change day-to-day work.

6.1 Subagents move to the background

Subagents now launch **in the background by default**. The main session stays available while they work, instead of sitting idle waiting for them to finish. That changes the rhythm of a session, because you dispatch a long-running job, keep working on something else, and pick up the result later.

Nesting now goes up to depth 3. Previously, a subagent couldn't open another one; now a subagent can open a subagent that opens a subagent. Combined with Opus 5's tendency to delegate, this is the change that turns into a bill fastest if nobody sets a limit.

There are also caps on subagents per session and on how many can run at once. Hitting them is rare in normal work, and when it happens, it's usually a sign of undisciplined delegation. The explicit cap from section 3 keeps you well away from them.

6.2 The session commands

Three new commands change how long sessions get handled.

`/fork` copies the current conversation into a new session that runs in the background. It's useful for exploring an alternative — a different design approach, a different library — without losing the thread of the main work or polluting its context.

`/subtask` launches a subagent from inside the session. It's explicit delegation, decided by the person doing the work, and it fits the discipline section 3 asks for: instead

of instructing the model to delegate whenever it feels like it, you delegate by hand when it's needed.

`/resume` opens a picker of past sessions and lets you pick one back up, also in the background. Alongside `/rewind`, which rolls back what's been done, and `/rename`, which gives a session a name you can find later, these three form the minimum toolkit for working across several live sessions without losing track.

One sequence that works well with Opus 5: open the session with the whole task specified up front, launch the slow search with `/subtask`, keep working in the main thread while it runs, and when an alternative worth exploring shows up, pull it out with `/fork` into a separate session instead of arguing about it inside the main context. At the end of the day, `/rename` leaves the live sessions with recognizable names, so `/resume` earns its keep the next morning.

6.3 The dead-weight audit

`/doctor` — with `/checkup` as an alias — diagnoses the installation and the **dead weight in your context**: what's loaded and going unused.

Three pieces round out the audit:

- `/context` shows what's taking up space in the window. MCP server definitions are deferred by default, so a connected server no longer costs context until it is used. Anyone who avoided connecting servers for fear of filling the window can revisit that decision.
- `/usage` attributes consumption to each skill, subagent, plugin, and MCP server, and explicitly flags anything over 10% of the total. It's the fastest route to finding what's behind a bill that doesn't add up.
- Prompt caching lasts one hour on a subscription plan, and drops to five minutes on usage credits or the API. The "five-minute rule" that's been circulating for a while describes only the second case; for anyone on a subscription, the window for resuming a session while still benefiting from the cache is twelve times longer.

6.4 Review, teams, and workflow size

`/code–review ultra` launches a multi-agent review in the cloud. The user triggers it explicitly, so it doesn't run on its own on every commit. Its natural territory is a large change before it goes to human review.

Agent teams remain off by default, and that default holds up unless the work justifies otherwise, because they consume on the order of **seven times more tokens** than a standard session, with teammates running in plan mode. It's a tool for problems that benefit from several simultaneous perspectives, at a cost you have to want to pay.

In `/config`, the `workflowSizeGuideline` setting fixes the reference size for dynamic workflows. Check it if the workflows you generate consistently come out longer or shorter than they should.

6.5 Changes on the API side

Three platform changes shift integration decisions.

Mid-conversation tool changes let you add or remove tools between turns while preserving the prompt cache. Until now, touching the tool set invalidated the cache and forced you to pay for the entire prompt again. With that resolved, an agent can load tools in phases — read tools first, write tools later — without a penalty.

Server-side fallback routes by refusal category without maintaining a model list in your code. Refusals in the *cyber* category, for instance, route to Opus 4.8. The fallback credit offsets the cache cost of the retry on the other model.

A refusal arrives as a **valid HTTP response**, with the stop reason marked as refusal and an associated category, not as an error. That is the usual cause of one specific class of production failure. Code that reads the response content without first checking the stop reason breaks the day the first one shows up. Anthropic doesn't publish how often this happens, so the only sensible policy is to treat it as something that will happen.

On billing for refusals: a request refused before generating any output isn't billed; a refusal mid-stream does bill whatever had already been emitted up to that point.

SECTION 7

The 20-Minute Migration Checklist

Six steps, each with its own check. The clock is budgeted for a normal-sized setup: one project `CLAUDE.md`, one global one, a handful of skills, and the odd custom command. Anyone running an arsenal of thirty skills will need more time on step 2, and the same amount on the other five.

7.1 1. Confirm which model you're using (2 min)

Open Claude Code in the project you work in daily and check the active model with `/model`. If you call the API from your own code, look up the model identifier in the service configuration.

Check: you know for certain whether you're on `claude-opus-5`, on `claude-sonnet-5`, or on an earlier generation. Without this, the next five steps run blind.

7.2 2. Comb through `CLAUDE.md`, skills, and commands (7 min)

This is the long step, and the one that returns almost all of the value. From the project root:

```
grep -rniE "verif|review.{0,3}twice|double.?check|summarize .*progress|use
↪  subagents|don't think" \
  CLAUDE.md ~/.claude/CLAUDE.md .claude/ 2>/dev/null
```

Check each match against the table in section 3 and apply that section's rule: if the line asks the model to review itself, delete it; if it asks for an artifact a person is going to use, keep it.

Add by hand the prompt templates that live outside the repository and the system prompts of any agent you have running in production. The grep doesn't reach those, and they're the two places where the toll gets paid with nobody watching.

Check: the same grep, run again once you're done, returns only lines you've consciously decided to keep.

7.3 3. Run `/doctor` and clear the dead weight (3 min)

Run `/doctor`, then `/context`. Look at what's loaded and what of it goes unused: skills you installed to try out, MCP servers from a previous project, leftover context files.

Check: `/usage` doesn't flag anything above 10% that you can't justify in a sentence.

7.4 4. Set a delegation cap (2 min)

Add a line to the project's `CLAUDE.md`:

```
Launch at most two subagents at a time, and only for parallel
searches over different parts of the repository. Everything else
runs in the main thread.
```

Adjust the number if your work calls for it.

Check: the line is in the file, and none remain that encourage unlimited delegation — step 2's grep would already have surfaced them.

7.5 5. Sweep effort downward (4 min)

Take a representative task from your daily work, one of the kind you repeat every week, and run it with effort one notch below what you use now. If you've been running `high`, try `medium`.

Check: compare two things. The result, against whatever objective criteria you already have — tests passing, the diff matching expectations — and the consumption, with `/usage`. If the result holds up, effort stays lowered. If it doesn't, that specific task needs `high`.

7.6 6. Review `max_tokens` (2 min)

This step only applies if you call the API from your own code and some of your workloads used to run with reasoning omitted. Search recent logs for responses cut off by the token limit.

Check: no recent response ends by exhausting its token budget. If any do, raise `max_tokens` — there's room up to 128,000.

7.7 Checklist summary

#	Step	Time	Done when
1	Confirm the model	2 min	You know the exact identifier you're using
2	Comb through prompts and config	7 min	The grep returns only lines kept on purpose
3	`/doctor` and `/context`	3 min	Nothing above 10% without justification
4	Delegation cap	2 min	The line is in `CLAUDE.md`
5	Effort one notch down	4 min	Result and cost compared on a real task
6	`max_tokens`	2 min	No cut-off responses in the logs

Step 5 repays repeating. One task isn't enough to calibrate an entire team, and dropping a notch and measuring works just as well the fifth time as the first.

SECTION 8

What Each Book in the Series Covers

This guide covers the migration and nothing more: what changed with Opus 5, which instructions stopped working, what replaces them, and how to bring your setup up to date in a morning. It assumes the rest. If you made it this far and just wanted to get current, you're done.

What's left is covered in three books from the same series. Each answers a different question, and none of them is required for what came before.

Getting Started with Claude Code is for anyone new to the tool who wants to be productive quickly. It covers the first hour of work, how to phrase requests that return what you expect, how to get oriented in someone else's repository, tests, `CLAUDE.md`, and the git workflow. If anything in this guide sounded like a foreign language, that's the book to start with.

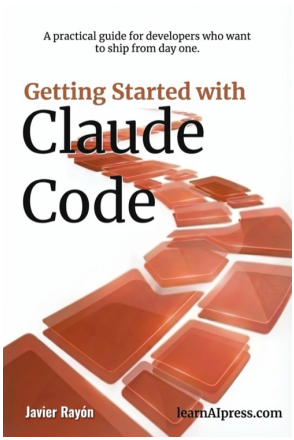
Claude Code Advanced is for anyone already working with the tool daily who wants to get more out of it. It goes into parallelization, subagent orchestration, automation, cost control, and security, with a chapter dedicated to Fable 5 and when it's worth paying for. It's the book that develops, with room and examples, the delegation caps that fit in a single line here.

Claude Code for CTOs is for anyone who has to decide for a team and defend the decision. It covers governance by model — data retention, safety classifiers, refusals, and fallback — total cost of ownership, and how to build the business case with the numbers the vendor publishes. It's written for whoever signs the budget.

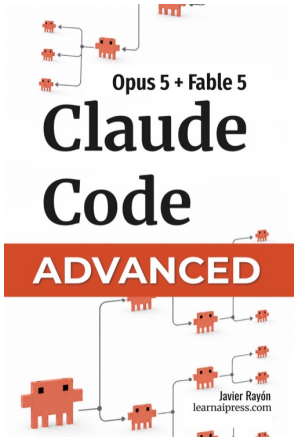
All three are paid titles, available at learnaipress.com. This guide was written apart from them and stands on its own.

The Claude Code Series

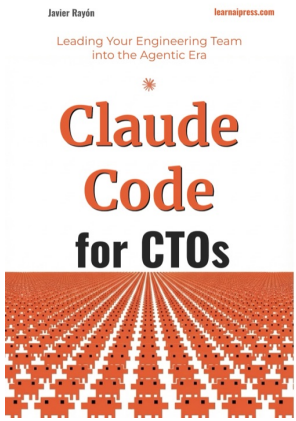
The titles in the collection, at learnaipress.com.



Getting Started with Claude Code



Claude Code Advanced



Claude Code for CTOs