ENTERPRISE DNA

A FIELD GUIDE FROM ENTERPRISE DNA

Working With Claude

The developer's field guide. How to read, write, and ship code with AI agents. What Claude Code is, the prompts that work, and how to build with it well.

FRONT MATTER

A note from Sam

You probably arrived here from a search. A code visualizer, an explainer, a pseudocode tool, something that helped you make sense of code or turn an idea into it. That is the right instinct, and this guide is the larger version of it.

The way we work with code just changed. You no longer only type the next line. You hand an agent a whole job. It reads across the codebase, plans, edits the files, runs the tests, fixes its own failures, and hands back a change you review. The developer moves up a level, from operating the editor to directing the work and approving what matters.

I have spent ten years at Enterprise DNA helping people do more with their data and their code. This is the most useful version of that mission we have ever shipped. Not a demo. Real agents doing real engineering work, every day, with a person holding the gate.

We do not write this from the outside. Our own company runs on a command center of Claude agents, and we build with Claude Code the same way these pages describe. Every pattern here is one we use ourselves.

Read it as a field guide. What Claude Code is, the prompts that actually work, what to delegate and what to keep, and how to adopt it without getting it wrong. If it lands, the last page tells you where to start.

Sam McKay

Founder, Enterprise DNA

CONTENTS

What is inside

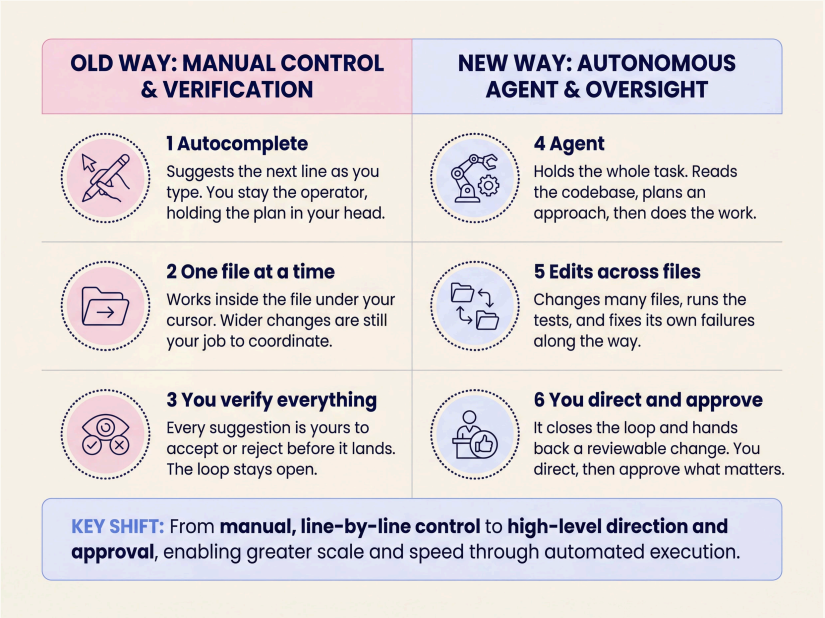
PART 1 · THE SHIFT

From autocomplete to agents

The unit of work for a developer just changed. Not the speed of typing. The size of the job you can hand off.

The old tool finished your line. Autocomplete watched your cursor and suggested what came next. You stayed the operator. You held the plan, you moved file to file, you verified every suggestion before it landed.

The new tool holds a whole job. It reads across the codebase, plans an approach, edits many files, runs the tests, fixes its own failures, and hands back a change you can review. The difference is not that it is faster. It is that it carries responsibility for the outcome.



Same chair, different altitude. You stop operating the editor and start directing the work.

You move up a level. Many readers arrived here while trying to understand unfamiliar code, or to turn an idea into something that runs. That instinct, putting a capable assistant on a specific piece of work, is exactly right. This guide scales it up. From approving lines to directing jobs and approving what matters.

The signal, honestly

> *A guide that only shows the upside is not worth your time.*

The traction is real. Anthropic reports its own engineers merged about 67% more pull requests per day after adopting Claude Code, even as the team grew. Over a year, it put measured productivity rising from roughly 20% to 50%, as the share of AI-assisted work climbed from about 28% to 59%. More than 80% of the code merged into Anthropic's own production codebase in May 2026 was written by Claude.

The honest counterweight

Adoption is not production. A widely-cited Gartner forecast expects more than 40% of agentic-AI projects to be cancelled by the end of 2027. Most fail on weak governance, unclear value, or loose cost controls, not on weak technology.

And faster can be a feeling. A METR randomized trial in 2025 found experienced open-source developers were about 19% slower with AI tools on their own mature codebases, while still believing they were faster. METR is clear this does not generalize to all developers. It was a narrow setting. Experienced devs, large mature repos, early-2025 tooling.

The thesis

This shift is real and it is early. Access to the technology is becoming universal, so access is not the edge. The edge goes to whoever adopts it well. That is what the rest of this guide is about.

PART 2 · WHAT CLAUDE CODE IS

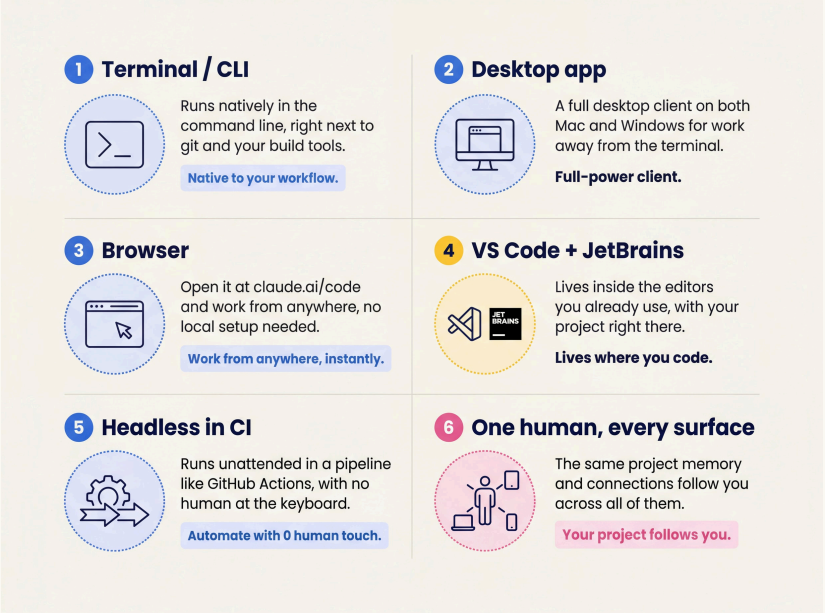
Meet Claude Code

> *An autocomplete tool guesses the next line. An agent holds the job.*

Claude Code is a terminal-native agent that runs where the work already is. You do not move to it. It comes to your project.

It runs everywhere you do. The command line, the desktop app on Mac and Windows, the browser at claude.ai/code, inside VS Code and JetBrains, and headless in a pipeline like GitHub Actions. The same project memory and connections follow you across all of them, so you start one place and continue in another without losing the thread.

It sees the real project. When you run it, it can read your files, run terminal commands like git, build tools, and package managers, check your git state, and read your CLAUDE.md and memory. It works against the actual codebase, not a copy you paste in.



One human, the same agent, every surface the work happens on.

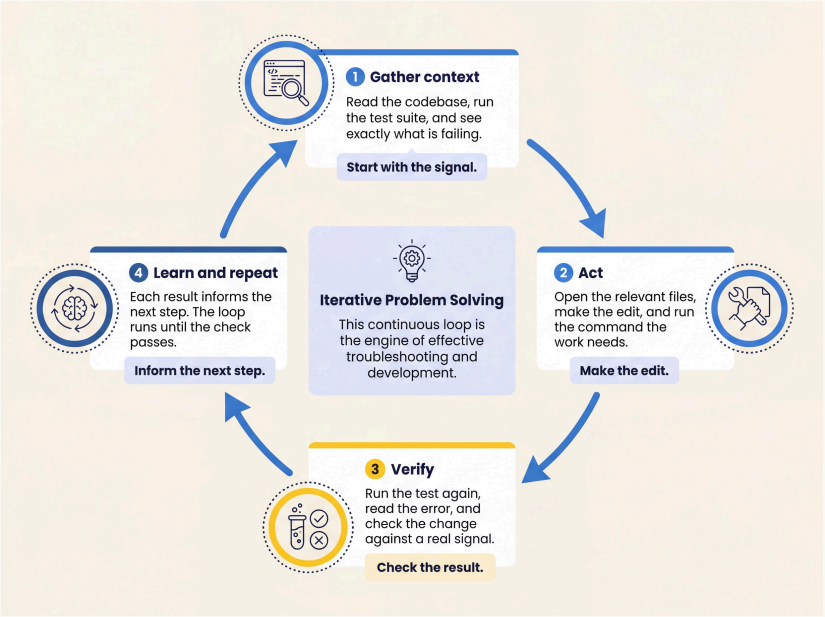
That whole-project view is the difference. Because it sees everything at once, it works across many files in a single task, traces dependencies through the code, and verifies its own fixes from end to end.

How the agent loop works

> *It keeps going until the work holds up, not until it produces a guess.*

Underneath, it runs a simple loop. Gather context, act, verify, repeat. It keeps cycling until the job holds up. Say "fix the failing tests" and it runs the suite to see what fails, reads the errors, finds and opens the relevant files, edits them, runs the tests again, and loops until they pass.

Each result informs the next step. A test failure points it at a file. The file points it at a fix. The next run tells it whether the fix worked. It is not guessing once and stopping. It is reasoning over real output, step after step, until the check comes back clean.



Give it a check it can run

This is the single most important habit. Tests, a build, a linter, a screenshot to compare against. With a check, the agent closes the loop itself and knows when it is actually done. Without one, "looks done" is the only signal, and you become the verification loop, reading every change by hand. Ask it to show evidence rather than assert success.

Plan mode and the permission dial

> *The approval gate is the point. It is where your supervision lives.*

You control how much the agent does on its own. Cycle the setting with Shift+Tab and choose it per task.

Default
Asks before each file edit and most shell commands. The safe baseline you start from.

Auto-accept
Hands trusted, repetitive work to the agent. It edits and runs safe commands on its own, and you review the diff after.

Plan
It explores and proposes a plan without touching your source. Nothing moves until you read and approve it.

Allowlist
Mark trusted commands in settings, like npm test or git status, so you are not asked every single time.

There is also a research-preview Auto mode. A background classifier screens actions for you. Whichever setting you pick, the approval gate is doing the real job.

Why it asks before some things

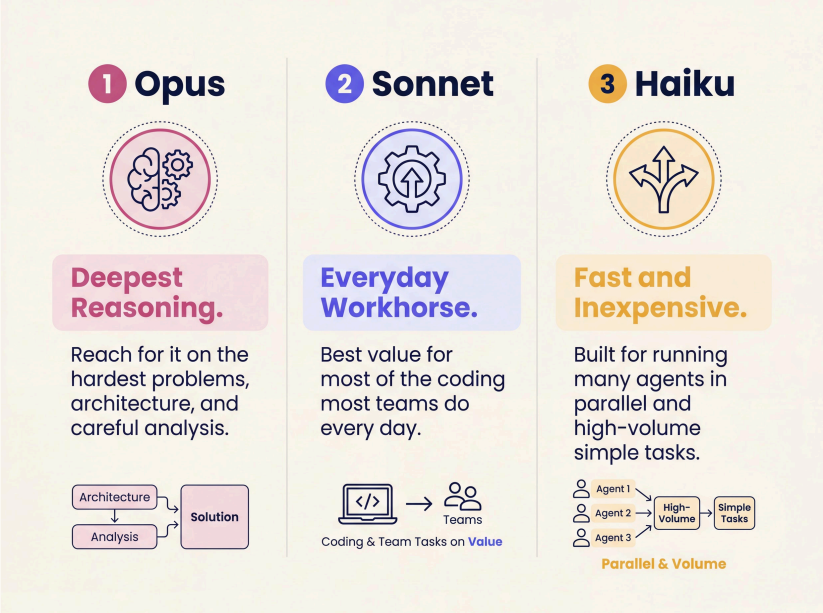
File edits are checkpointed and reversible. If a change is wrong, you roll it back. **Actions on remote systems cannot be undone.** A database write, an API call, a deploy. There is no checkpoint for those, which is exactly why the agent stops and asks before it touches them.

Choosing your model

> *Match the model to the difficulty of the work, not the prestige of the name.*

Three everyday models, three jobs. Opus is the deepest reasoning, for the hardest problems, architecture, and careful analysis. Sonnet is the everyday workhorse, the best value for most of the coding most teams do. Haiku is fast and inexpensive, ideal for running many agents in parallel and for high-volume simple tasks.

The context windows are generous. The agent can hold a large slice of your codebase at once and reason over the whole picture, not just the file in front of it. That is part of why it can trace a problem across a project.



Most teams mix. Sonnet for daily work, Haiku for scale, Opus when it is genuinely hard. You can switch mid-task too, so a cheap model does the legwork and a strong one makes the call. The skill is knowing which job you are on.

PART 3 · PROMPTING THAT WORKS FOR CODE

The core principles

> *Think of the agent as a brilliant but new employee who lacks context on your norms and workflows.*

It is fast and capable. It has not read your codebase, your conventions, or your reasons. Everything good follows from one golden rule: show your prompt to a colleague with minimal context. If they would be confused, the agent will be too.

Five principles carry most of the weight. Be explicit and ask for more than the bare minimum. Give the why, the context and the constraints, not just the rule. Tell it to do the thing, not suggest it. Give it a way to verify its own work, a test or a build. Point at specific files and example patterns to follow.

Weak
fix the login bug

Strong
users report login fails after session timeout. check the auth flow in src/auth/, especially token refresh. write a failing test that reproduces the issue, then fix it

The strong prompt names the symptom, the place to look, and the proof of done.

Reading code you did not write

> *The agent reads a codebase like a senior engineer. You do not need special prompting. You just have to ask.*

Start by understanding, not changing. Drop into an unfamiliar repo and ask plain questions. "Explain what this function does in plain English." "Trace the request flow from the API route to the database write." The agent opens the files, follows the calls, and answers from what is actually there.

Find the blast radius before you touch anything. The most useful question before a change is who depends on this. Ask it to search first, then reason.

Blast radius
if I change getUser to return null instead of throwing, what calls break? search for every caller first, then list the ones that assume it throws

Document and visualize. Ask it to write a short readme for a module, or to draw a Mermaid diagram of the top-level modules and how they connect. A picture of the architecture lands faster than a wall of prose.

Weak
why does ExecutionFactory have such a weird api?

Strong
look through ExecutionFactory's git history and summarize how its api came to be

One guardrail. Tell it never to claim anything about the code before it has opened the files. Grounded answers, not guesses.

Generating code that holds up

> *Constrain it to your shape. Then make it prove the work.*

Pseudocode first. Hand it the logic and tell it where to match. "Here is the pseudocode. Implement it in TypeScript matching the style and error handling in src/middleware/auth.ts. Do not add anything the pseudocode does not call for."

Test first for anything tricky. Split it into two phases so the agent cannot quietly bend the tests to fit the code.

Phase 1
write a failing test for [feature]. do NOT write the implementation yet

Phase 2
write the implementation. do not modify the tests. keep going until all tests pass

Guard against fake passes. Add one line so it solves the real problem: "write a general-purpose solution that works for all valid inputs, not just the test cases. Do not hard-code values."

Weak
add a calendar widget

Strong
look at how existing widgets are implemented on the home page. HotDogWidget.php is a good example. follow the pattern to add a calendar widget that lets the user pick a month and paginate by year. build from scratch without libraries other than the ones already used in the codebase

Scope tightly. "add tests for foo.py" gets you noise. "Write a test for foo.py covering the edge case where the user is logged out. Avoid mocks." gets you the one that matters. Only the change you asked for, nothing extra.

Refactoring and review loops

> *A green suite is your contract. Refactor against it.*

Refactor without changing behavior. The tests are the safety net, so protect them. Tell it to leave them untouched and prove nothing broke.

Refactor
refactor src/parser.ts for readability without changing behavior. run the suite before and after, the same tests must stay green. do not modify the tests

Review a diff like a senior engineer. Open a fresh context, lead with correctness, and name the file. A clean context reviews harder than the one that just wrote the code.

Reviewer
review the rate limiter in src/middleware/rateLimiter.ts. look for edge cases, race conditions, and consistency with our existing middleware patterns

Ask for the counterargument. Make it attack its own work: "argue against your own implementation. Where is it most likely to be wrong? List the three riskiest parts."

One caveat. A reviewer told to find gaps will usually report some even when the work is sound. So tell it to flag only gaps that affect correctness or the stated requirements. That keeps the signal real.

Plan, then execute

> *Separate research from action, or it will confidently solve the wrong problem.*

The spine of safe agentic coding is sequence. Let it explore and plan before it writes a line, so you catch a wrong approach on paper instead of in a diff.

Four phases. Explore in plan mode, read-only, no edits. Plan next: "What files need to change? What is the session flow? Create a plan." Then code, but only after you have read the plan. Then commit with a descriptive message and open a PR. Skip the plan only when you could describe the diff in one sentence.

The gate. Let it edit files and run tests freely. Pause before anything hard to reverse: deleting files or branches, dropping tables, git push --force, git reset --hard, pushing code, commenting on PRs, sending messages, modifying shared infrastructure. Agents draft and stage. Humans authorize the merge, the deploy, the migration.

The prompt cheat sheet

> *Pin this above the keyboard.*

Most good prompts come back to the same handful of moves. Name the symptom and where to look. Give it a way to check itself. Point at real files to match. Plan before anything that spans files. Tell it to act, not advise. And read the diff yourself before it lands. The card below is the whole of Part 3 on one page.

Keep it close. The habits compound faster than any single trick.

PART 4 · BUILDING WITH AGENTS

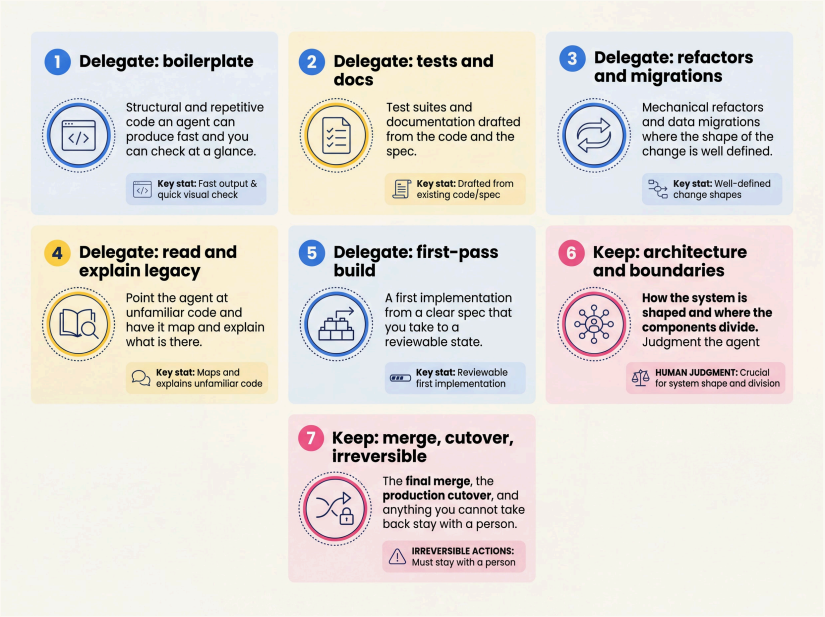
What to delegate, what to keep

> *The skill is not prompting. It is knowing which work is the agent's and which is yours.*

An agent that can write code is not the same as an agent you should hand everything to. The first decision on every task is where the line sits.

Good agent fits. Boilerplate and structural code, tests, documentation, mechanical refactors, migrations, reading and explaining legacy code, first-pass implementations from a spec, and parallel exploration of a few approaches. This is the work that is well-defined, bounded, and easy to check.

What stays human. Architecture and component boundaries, interpreting ambiguous requirements, the production cutover, the final merge, anything irreversible, and the judgment calls about the business. These need context the agent does not hold and consequences it cannot own.



The line for code is one sentence. The agent produces, the human authorizes. The agent gets you to a reviewable change. A person decides it goes live.

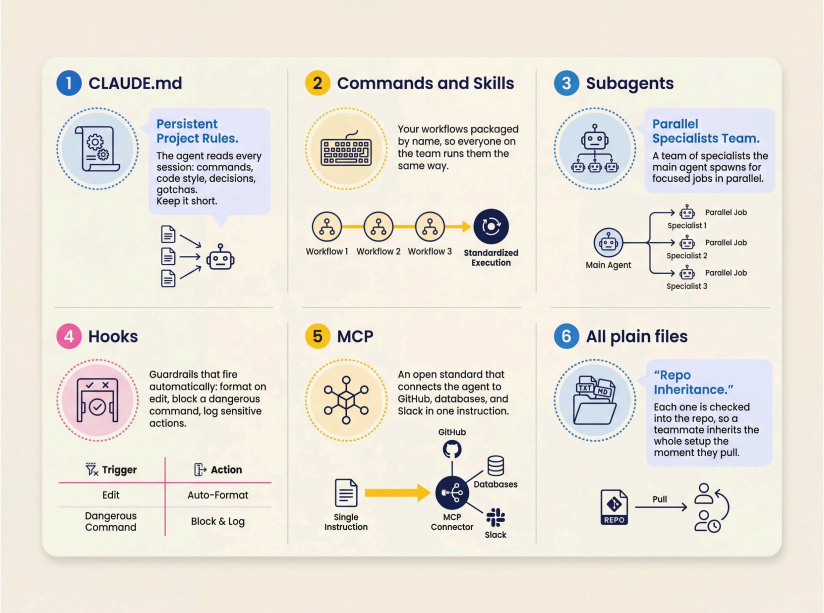
The building blocks

> *Five plain files turn a general model into a specialist for your team.*

They are all checked into the repo, so a teammate inherits them the moment they pull. No setup call, no shared doc that drifts.

CLAUDE.md. Persistent project rules the agent reads at the start of every session: build and test commands, code style that differs from defaults, architecture decisions, repo etiquette, known gotchas. Keep it short. A bloated CLAUDE.md gets ignored.

Commands and Skills. Your workflows packaged by name, so anyone on the team runs them the same way.



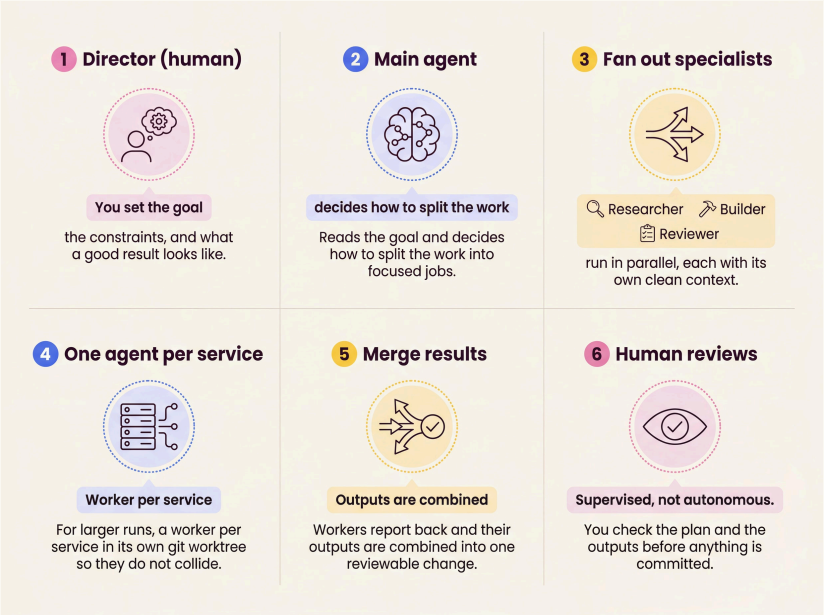
Subagents. A team of specialists the main agent spawns for focused jobs in parallel. Covered next.
Hooks. Guardrails that fire automatically: format on edit, block a dangerous command before it runs, log every sensitive action to your audit trail. A policy in a hook runs every time, unlike a policy people are asked to remember. **MCP.** An open standard that connects the agent to your real systems, GitHub, databases, Slack, so one instruction can span several of them.

One developer, a fleet of agents

> *Workers fan out, report back, and you review before anything is committed.*

Subagents are specialized helpers the main agent spawns, each with its own clean context, often running in parallel. Fan out a researcher, a builder, and a reviewer at once, then merge the results.

Real patterns. A domain split, one agent on frontend, one on backend, one on schema. A writer-reviewer split, where a fresh context reviews the code so it is not biased toward what it just wrote. One agent per service in a nightly run, each in its own git worktree so they do not collide on shared files.



The honest cost note. Anthropic reports a multi-agent setup beat a single agent by about 90% on one internal research evaluation. But multi-agent runs use far more tokens, on the order of fifteen times a normal chat, so it is worth it for high-value work, not everything. The human stays the director. You review the plan and the outputs before anything lands. It is supervised parallelism, not autonomy.

Claude Code in the pipeline

> *Wake up to results, not a queue.*

The same agent runs headless, with no human at the keyboard. That is where unattended work lives.

Headless mode. Pass a prompt with claude -p and get the result on stdout, with json output that includes the run cost so you can track CI spend.

GitHub Actions. Anthropic publishes an official action with two modes. Interactive, where the agent responds to @claude mentions in issues, PRs, and review threads. Automation, where a prompt in the workflow runs on events like a PR opening or CI failing, for automated review.

The Agent SDK. The same agent loop, tools, and context management, callable from Python or TypeScript, so you can embed the agent in a product or an unattended pipeline. The honest edge: persistent always-on background agents are more limited, so the reliable path is headless mode driven by your own scheduler, a cron or a scheduled GitHub Action.

Subagents: specialists with isolated contexts, spawned in parallel.	**Headless mode:** one prompt in, result and cost out, no keyboard.
Official GitHub Action: @claude in your PRs, or automated review on events.	**Agent SDK:** the same loop embedded in your own code.

The gate holds even in CI. The agent stages a change. A person approves the merge.

PART 5 · THE PROOF

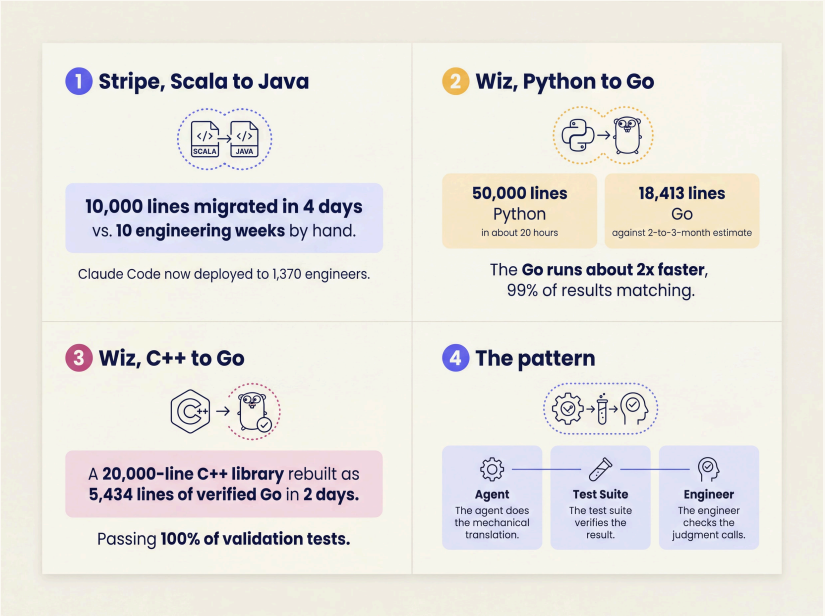
Migrations at a new speed

> *The work nobody had time for is now the clearest win.*

Big rewrites used to sit at the bottom of the backlog forever. They are now some of the strongest results teams report, because the translation is mechanical and a test suite tells you when it is right.

Stripe migrated 10,000 lines of code from Scala to Java in 4 days. The project had been estimated at 10 engineering weeks by hand. Stripe has since deployed Claude Code to 1,370 engineers.

Wiz moved 50,000 lines of Python into an 18,413-line Go codebase in about 20 hours, against a 2-to-3-month manual estimate. Wiz reports the Go version runs about 2x faster, with 99% of results matching. A second migration took a 20,000-line C++ library to 5,434 lines of verified Go in 2 days, passing 100% of validation tests.



Numbers as reported by each company on Anthropic's customer pages.

The pattern under all of it is the same. The agent does the mechanical translation. The engineer checks the judgment calls. The test suite is the contract that says the new code still does what the old code did.

Across the dev team

> *It is not only migrations. The named results run right through day-to-day work.*

Wiz reports that more than 90% of its engineers now use Claude Code daily, and merged pull requests rose about 1.5x among its top 100 contributors.

Ramp: up to 80% reduction in incident-investigation time.

Ramp: more than 1,000,000 lines of suggested code implemented in 30 days, 50% weekly active.

Smartsheet: 3x more code and 31% more pull requests merged than peers on the same teams.

Smartsheet: top performers at 5x to 7x prior output, org-wide cycle time down 28%.

TELUS reports engineering teams shipping code about 30% faster.

Anthropic's own engineers merged about 67% more pull requests per day after adopting Claude Code, even as the team grew. They report productivity rising from about 20% to 50% over a year. More than 80% of the code merged into Anthropic's own production codebase in May 2026 was written by Claude.

These are strong, real numbers from teams that adopted the tool well. Read them as the ceiling that careful adoption can reach, not the floor everyone lands on by default. The next page is the counterweight.

Productivity, honestly

> *Measure it honestly or you oversell it and lose trust, or undersell it and lose the budget.*

The wins on the last page are real. But four things are just as real for the developer holding the keyboard, and ignoring them is how pilots quietly fail.

Over-trust is the common failure

In the 2025 Stack Overflow survey, the single biggest frustration, cited by 66% of developers, is code that is almost right but not quite. Only about a third of developers trust the accuracy of the output, even as 84% use or plan to use these tools. The fix is structural. Give the agent a check it must pass, then review the diff yourself.

Speed can be a feeling

A METR randomized trial in 2025 found experienced developers were about 19% slower with AI tools on their own mature codebases, while believing they were faster. METR is clear this does not generalize to all developers. It was a narrow setting of experienced devs on large mature repos with early-2025 tooling.

Two more to plan for

The review burden. When agents write more code, review time rises, because the bottleneck moves from writing to reading. **Adoption is not production.** Gartner expects more than 40% of agentic-AI projects to be cancelled by the end of 2027, mostly on cost and weak controls, not weak technology. The bottleneck is the quality of adoption, not access to the technology.

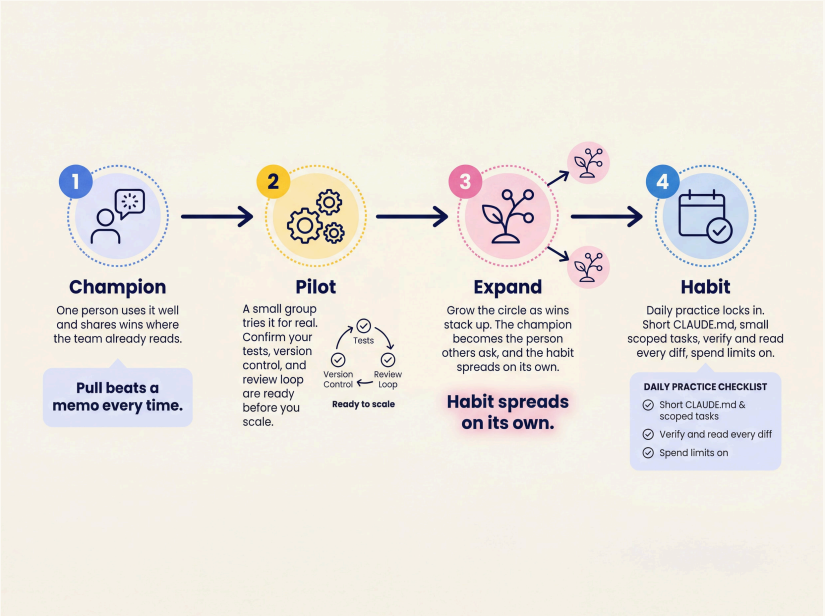
PART 6 · DOING IT WELL

Getting started well

Adoption does not happen by mandate. It happens by example.

Start with a champion, not a memo. One person uses the tool well, shares wins where the team already reads, and becomes the person others ask. Then the circle grows on its own. A broadcast announcement creates pressure. A visible win creates pull.

Crawl, walk, run. Pilot before you scale. A small pilot shows whether your tests, version control, and review loop are ready before you multiply the change volume. If those foundations are shaky, more output just exposes the cracks faster.



The discipline is what pays. The DORA 2025 read is that AI amplifies what is already there. It relates positively to throughput, but it can pull on delivery stability. The team that already has good tests and a tight review loop gets a multiplier. Start the habits today: keep CLAUDE.md current and short, work in small scoped tasks and plan before code, give the agent a way to verify, read every diff, and put spend limits on from day one.

The pitfalls, and how to avoid them

> *Every failure mode has a fix. None of them is "use it less."*

Over-trust

Confident output gets accepted without a check. In the 2025 Stack Overflow survey, the top frustration for 66% of developers is code that is almost right but not quite. The fix is a check the agent must pass, reading the diff yourself, and asking for the counterargument before you ship.

Skill atrophy

The doing is what keeps judgment sharp. A 2025 Anthropic-run study on learning a library found the AI-assisted group scored about 17% lower on a comprehension quiz, roughly two letter grades, with no significant time saving. Treat it as suggestive. The lesson holds: protect the hands-on work, especially for junior developers.

Cost blowups

Usage-based cost runs ahead of seat budgets. Agentic sessions consume large context, so spend can outpace a seat-count plan if nobody is watching. Microsoft is moving its engineers to its own GitHub Copilot CLI, a standardization on a tool it owns with cost named as a factor. Uber reportedly burned through its planned 2026 AI coding budget in about four months. The fix is spend limits and per-model choices from day one.

Review burden

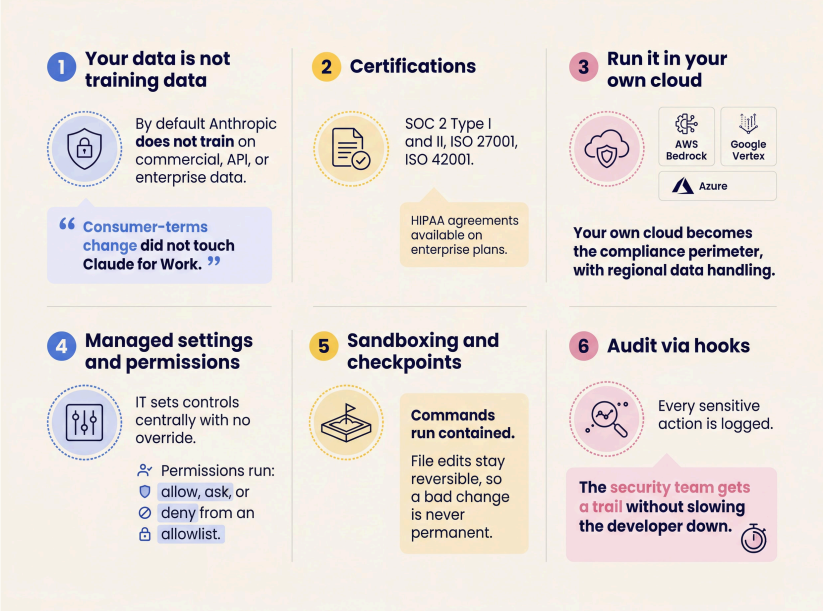
The bottleneck moves from writing to reading. Scope tasks small so diffs stay reviewable, and keep a real reviewer in the loop. Agents run, developers direct. The bottleneck is the quality of adoption, not access to the technology.

Governance, security, and your data

By default, Anthropic does not train on commercial, API, or enterprise data.

The headline that scared people did not apply to you. The 2025 consumer-terms change applied to consumer accounts, not to Claude for Work, the API, Bedrock, or Vertex. That single sentence unblocks most security reviews before they start.

The certifications a security team asks for. SOC 2 Type I and II, ISO 27001, and ISO 42001, with HIPAA agreements available on enterprise plans. **Run it inside your own perimeter.** You can run Claude inside AWS Bedrock, Google Vertex, or Azure, which makes your own cloud the compliance perimeter, with regional data handling options.



Claude Code's own controls

Managed settings: IT sets them centrally and no one can override.	**Permissions:** allow, ask, or deny, starting from an allowlist.
Sandboxing: commands run contained.	**Checkpoints:** file edits are reversible.

PART 7 · WHERE WE FIT

Where Enterprise DNA fits

> *You landed here through one of our tools. This is the larger version of that idea.*

The free tools that may have brought you here, the Code Visualizer, the Code Explainer, the Pseudo-Code Generator, each do one job well. They live in MENTOR, our set of more than 40 specialized tools for working with data and code. They are small examples of a larger idea. Put a capable assistant on a specific piece of work and let it carry the load. That same idea, applied to a whole business, is what we build.

Our own company runs on it. A fleet of Claude agents wired into the systems we already use, with one person holding the approval gate. Nothing reaches a customer or a system of record until that person says yes. There are three ways we help, and you can start with any one.

Learn it
Training and workshops to get your team fluent with Claude Code and agentic workflows.

Build it
We install your operating layer. The agents, the connections, the guardrails, the review surface.

Run it
We operate the layer alongside your team and keep improving it, with a report of what ran each week.

Start with a conversation

Thirty minutes, free. We look at where the repeated work lives and give you a clear picture of what an operating layer would change. You leave with the map either way.

Email: sam.mckay@enterprisedna.co.nz

Web: enterprisedna.co

Tools: mentor.enterprisedna.co

The decision: yours. No pressure.

ENTERPRISE DNA

A NOTE IN CLOSING

You no longer only type the next line. You hand an agent a whole job, and you supervise. The developers who learn to direct a fleet of agents, with a person holding the gate, will run circles around the ones who did not.

FROM INSIDE THIS GUIDE.

Put Claude to work on your code.