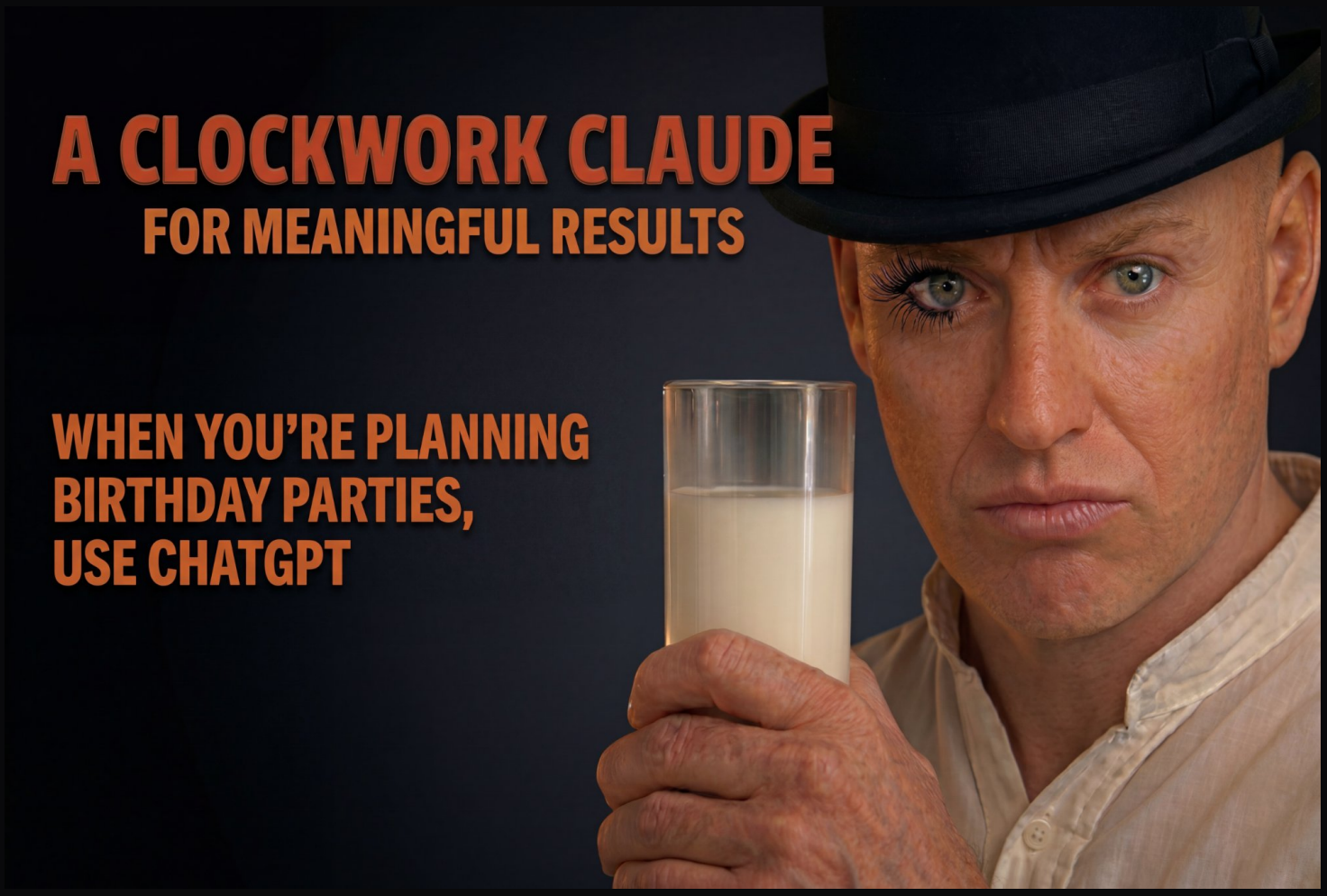

By Pete Connor

Dedicated to the TC_Leaders.

Ya'll keep on moving the needle.

Contents

Author's Note

This guide is not a vendor whitepaper. It's not a "10 Tips to Unlock AI Magic" listicle written by someone whose most complex prompt was asking ChatGPT to plan a birthday party.

It's the field manual I wished existed when I started using Claude in production work — not demo work, not "let me show leadership something impressive in a meeting" work, but work where deadlines are real and "it mostly works" is not an acceptable outcome. I've spent twenty-plus years in enterprise operations — contact centers, workforce management, healthcare RCM, five digital transformations (three of which I survived, two of which survived me) — and the single most consistent pattern across that career is the canyon between what technology promises and what it delivers when your phone lines open at 7 AM.

Claude narrows that canyon. Not eliminates. Narrows. This guide is for people who want to understand the tool well enough to work with its capabilities instead of against its limitations — people who do work, not people who attend meetings about work. If your primary job function involves producing slide decks about other people's slide decks, you'll find this guide aggressively unhelpful.

Every technical detail and configuration example comes directly from Anthropic's course curriculum. I haven't invented anything. I stripped away the instructional design voice — the careful neutrality, the "learning objectives" headers, the tone that treats you like you've never operated a computer — and replaced it with someone who's deployed this technology where failure has consequences. All the substance remains. The packaging has been upgraded from corporate beige to something you'll actually remember.

Where Claude is good, I'll say so — with the surprised respect of someone burned by enough vendor promises to know the difference. Where it has limitations, I'll say that too. Calibrated expectations and practical skill. Not cheerleading. Not cynicism for its own sake.

Let's get to work.

CHAPTER 1

Meet the Machine

Or, How a Statistical Parrot Learned to Fake Competence Better Than Most of Your Colleagues

Meeting Claude for the first time feels like meeting a new hire who showed up on day one having already read every document in your shared drive, memorized the org chart, and formed opinions about your formatting choices. The immediate instinct is suspicion. Nobody is that prepared. The second instinct — after a few hours of actual work — is reluctant respect. The kind you reserve for a colleague who delivers competent output without requiring three alignment meetings and a project charter first.

> *The immediate instinct is suspicion. Nobody is that prepared.*

Constitutional AI is what happened behind the curtain. Instead of humans manually flagging every bad output — a process that scales about as well as manual QA in a 500-seat contact center — Anthropic trained the model against a set of principles and let it evaluate its own work. Whether you find this reassuring or terrifying says more about your relationship with institutional governance than about the technology.

What it produced: predictability. In operations, unpredictability is the enemy. A tool that delivers brilliant output 60% of the time and incoherent garbage 40% of the time is worse than one that delivers competent output 95% of the time. Claude sits closer to the latter. That's not a sales pitch. That's the reason you're reading this guide instead of the one for the model that hallucinates your company's financials with confidence.

What Claude Can Do (and What It Cannot)

The capability map, delivered with the precision this deserves — because "Claude can help with lots of things!" is the kind of answer that makes experienced operators reach for the nearest exit.

Writing and content creation. Not autocomplete with delusions of grandeur. Given proper context — your voice, your tone, your structure — Claude produces draft-quality work that genuinely compresses the distance between "blank page" and "something

worth editing." You iterate on structure and clarity until your voice comes through, not Claude's default helpful-professional register (which, to be clear, is not a compliment).

Research and analysis. A context window that ingests 200K+ tokens — roughly 500 pages of text in a single conversation. Five hundred pages. Not a typo. Your quarterly report package, competitive analysis deck, customer feedback corpus, and strategy document, held in working memory simultaneously. The context window is the capacity. Your prompting is the steering wheel. One without the other produces expensive noise.

> *The context window is the capacity. Your prompting is the steering wheel.*

Coding assistance. Opus 4.5 is — Anthropic's claim, not mine, though the benchmarks support it — the best coding model in the world. Write, debug, and explain code across multiple languages. If you've ever lost three hours to a CSS bug that turned out to be a missing semicolon, you will develop an inappropriately emotional relationship with this capability.

Problem-solving and reasoning. Opus 4.5 and Sonnet 4.5 offer two modes: near-instant responses and extended thinking for deeper reasoning. The difference on complex analytical tasks is a napkin calculation versus a properly worked-out analysis. Use extended thinking when the stakes justify it. Skip it for straightforward questions.

Learning. The capability most people ignore. The one with the highest long-term ROI. Learning mode guides your reasoning process rather than handing you the answer — developing understanding rather than collecting facts. Using Claude to produce is linear. Using Claude to learn compounds.

Where Claude Lives

Same intelligence. Multiple interfaces. Each built for different work.

Claude.ai — the general-purpose interface. Conversations, writing, research, analysis, file creation. Projects, memory, and preferences sync across desktop, mobile, and web. Available on all plan types. This is where most people live.

Claude Code — the agentic coding tool. Terminal, IDE, browser, or Slack. Describe what you need in plain English; Claude writes the code, runs tests, creates commits. The competent junior engineer who works at 3 AM without complaint. Chapter 2 goes deep.

Claude in Slack — tag @Claude in a thread about a bug, and it spins up a session using the surrounding context. Claude meets you where the work is happening instead of demanding you context-switch to a separate tool. The workflow integration is genuinely clever — a phrase I deploy sparingly.

Claude for Excel — a sidebar that reads, analyzes, modifies, and creates workbooks. Traces formula dependencies across sheets. Debugs circular references. Explains calculation flows in plain language. If you've ever inherited a nightmare spreadsheet and spent two days reverse-engineering someone else's nested VLOOKUP logic, this is the colleague you wish you'd had.

Claude for Chrome — browser extension for working with web content directly. Select a passage, get a summary. Navigate a complex dashboard, get an explanation. Quietly one of the most useful integration points for anyone doing significant web-based research.

Your First Conversation

Open Claude.ai. Clean interface, text box, blinking cursor. No tutorial wizard. No seventeen-step onboarding flow. Refreshing or terrifying, depending on your relationship with unstructured tools.

Talk to it like a competent colleague. Not a search engine. It doesn't need keywords. It needs context.

Three elements separate a prompt that works from a prompt that produces something you immediately delete:

Set the stage. Who are you? What's the situation? "Write an email about the project delay" is a prompt. "Write an email to our enterprise client explaining that the software integration will be delayed by two weeks — they've been patient so far but this is the second delay, keep it professional but apologetic" is a prompt that produces something you'd actually send. The difference: "I need something" versus "here's exactly what I need and why."

Define the task. The verb matters. "Help me with this data" is a therapy session. "Analyze this dataset and identify the three strongest correlations between customer tenure and support ticket volume" is a work order. Precision in, precision out.

> *Precision in, precision out.*

Specify the format. Style, tone, structure, citations. Format specifications save you the editing pass you'd otherwise spend reformatting Claude's default output into something a human would recognize as useful.

Adding Context

Context is the single biggest determinant of output quality. Not the model. Not the prompt template you downloaded from LinkedIn. The context.

Uploads handle PDF, DOCX, CSV, TXT, HTML, PNG, JPEG — parsed automatically. No preprocessing, no format conversion, no middleware that was supposed to be "seamless" but requires three configuration files and a prayer. Upload your actual data — not hypothetical versions, not sanitized examples — and the quality of analysis improves dramatically. This is where Claude transitions from "smart chat partner" to something approaching an actual research assistant.

Preferences in Settings configure tone, format, technical depth, and communication style once. Applied across every conversation thereafter. Thirty seconds to set up. Thousands of words of repeated instructions eliminated over time.

Iterating on Claude's Responses

Claude reads your mind exactly as well as your actual colleagues do — which is to say, imperfectly. First responses won't always be what you wanted. This is collaboration, not a vending machine transaction.

Ask follow-ups. "Expand on the second point." "Make it more concise." "The data is right but the narrative is wrong." These refinements are where output goes from "acceptable first draft" to "this is actually what I needed."

Give direct feedback. "The tone is too formal — make it conversational." "Cut the first two paragraphs and make the conclusion more action-oriented." Claude is trained to take direction. Give it direction. The complaints about Claude not producing what people want are almost always complaints about people not telling Claude what they want.

Know when to restart. If a conversation has accumulated twenty messages of compounding misunderstanding, a fresh chat with a clearer opening prompt is faster than rehabilitation. This is true of Claude and also of most conference calls.

Getting Better Results

When the output isn't right — and it will occasionally not be right — the diagnosis is usually one of five things:

Too generic? Your prompt lacked context about your specific situation. The fix is details: audience, role, constraints. The prompt that could apply to any company on earth will produce output that sounds like it was written for any company on earth.

Wrong length? You didn't specify. "Two-paragraph summary." "Under 100 words." "Comprehensive analysis — length isn't a concern." Claude follows length instructions with surprising fidelity. Use them.

Wrong format? Show, don't tell. Attach a sample of the format you want. "Follow this structure" produces better results than a paragraph describing the structure you want. Claude calibrates from examples faster than from descriptions.

Confidently wrong? Claude occasionally generates plausible but incorrect information — the limitation that matters most for professional work. For high-stakes deliverables: verify key facts independently, enable web search to ground responses, and ask Claude to indicate its confidence level. Trust but verify, with the emphasis surgically placed on *verify*.

> *Trust but verify, with the emphasis surgically placed on verify.*

Wrong tone? Claude defaults to helpful-professional, which is the AI equivalent of business casual — inoffensive and forgettable. Provide an example of writing in the style you want. One sample paragraph teaches tone more effectively than a paragraph of adjectives describing tone.

Evaluating Claude for Your Workflows

Here's the question that separates operators from enthusiasts: does Claude perform well enough on the work you actually do, with the data you actually have, against the standards you actually apply? Not "is it impressive in a demo" — that's easy and meaningless.

The eval process is four steps. Gather 5-10 examples of work you've actually done — emails, reports, analyses. These are your ground truth. Create test prompts that would generate similar outputs. Run them and compare Claude's responses to your examples. Refine your approach based on what you learn.

This is the difference between confidently delegating recurring tasks and nervously checking every output because you never established a baseline. Invest the hour. The return is measured in weeks.

Creating with Artifacts

Artifacts sound unremarkable in description and become indispensable in practice. Standalone, interactive outputs rendered in a dedicated window alongside your conversation. A working website. An interactive chart. A downloadable document. A React component with actual logic — not a mockup, but working code that executes and responds to input.

Claude creates artifacts automatically when content is significant, self-contained, and something you'd want to reuse. Six types: **Documents** (structured written content), **Code** (highlighted, syntactically valid), **HTML** (fully rendered web pages), **SVGs** (vector graphics — diagrams, icons), **Mermaid diagrams** (flowcharts and architecture visualizations), and **React components** (interactive UI with real functionality).

The practical advice: Be specific. "Build a monthly budget tracker where I can input expenses by category, see a pie chart breakdown, and get a warning when I'm over budget" dramatically outperforms "build a budget tracker." Describe the end user. Iterate one feature at a time. If Claude responds in chat when you expected an artifact, say so.

Sharing: Copy, download, share within your org (Team/Enterprise), or publish publicly. Others can remix published artifacts in their own conversations. Only the published version goes public — your chat stays private.

File Execution

Upload slides, spreadsheets, contracts — .xlsx, .pptx, .docx, .pdf — and Claude manipulates them in a sandboxed environment. Create slides, perform analyses, add suggested edits. Download modified files or open directly in Google Drive.

Toggle "Allow limited network access" when prompted. Claude needs the sandbox to manipulate files without touching your local filesystem.

Security note: Skills powering file execution can include executable code. Anthropic's built-in Skills are tested and maintained. Custom Skills from external sources deserve the same scrutiny you'd give any npm package — which is to say, more than 90% of people will give them.

Skills — Expertise Packages for Claude

Skills are procedural knowledge that teaches Claude how to complete specific tasks repeatably. Folders of instructions, scripts, and resources loaded dynamically. If you've used Claude to create an Excel spreadsheet or a PowerPoint, you've already used Skills — those capabilities are powered by Skills running behind the scenes, which is why the document generation output is substantially better than raw prompting.

Anthropic Skills — built-in, maintained by Anthropic. Excel, Word, PowerPoint, PDF creation. Available to all paid users, activate automatically.

Custom Skills — you build them once, Claude applies them automatically when the task pattern matches. The creation process is conversational: tell Claude what you want, answer its questions about your workflow, upload reference materials, and Claude generates a downloadable ZIP. Upload it in Settings, and from that point forward, it activates whenever relevant.

Skills vs. Projects: Projects store knowledge — reference materials Claude draws on across conversations. Skills encode process — the specific steps and methodology for executing a task. A "customer call prep" skill might pull from customer profiles stored in a project's knowledge base. The project provides the *what*. The skill provides the *how*.

Organizing Your Work with Projects

Projects are self-contained workspaces — dedicated environments for specific work streams. Own memory, chat histories, knowledge bases, customized instructions.

The difference between one giant desk covered in every document from every initiative and a filing cabinet where each drawer contains exactly the materials for one project.

Project knowledge — upload documents once, Claude references them across every conversation in that project. No re-uploading. No re-explaining. **Project instructions** — specify tone, expertise level, response style, and process requirements applied to every conversation.

Scaling is automatic. When your knowledge base approaches context window limits, Claude enables RAG mode — expands capacity by up to 10x, intelligently retrieving only what's relevant instead of loading everything at once.

For Team and Enterprise: Three permission levels. Can view (read-only). Can edit (full collaboration). Owner (full control including access management).

Operational wisdom, earned: Start focused — one use case per project, not one project for everything. Keep your knowledge base current — outdated documents produce outdated responses. Name files descriptively — "Q4-2024-Brand-Guidelines.pdf" beats "document1.pdf" by a margin that shouldn't need explaining but apparently does. Reference documents by name in your questions — "Based on our Q3 report, what were the top customer concerns?" focuses Claude's search. "What are our customer concerns?" makes Claude guess which drawer to open.

Expanding Claude's Reach with Connectors and MCP

Connectors transform Claude from an assistant that knows what you tell it into an assistant that knows what your tools know. Your files, project boards, email threads, documentation — searchable, retrievable, and synthesizable from within a conversation.

The protocol underneath is MCP — Model Context Protocol. The course materials call it "USB-C for AI." Having spent years watching enterprise integration projects that required custom middleware, dedicated integration engineers, and six-figure budgets to connect two systems that should have talked to each other out of the box — the USB-C analogy is aspirational, but the protocol is genuinely well-designed. Open standard. Common interface. No bespoke integration per application.

Web connectors link Claude to cloud services — Google Drive, Notion, Slack, Asana, Linear, Stripe, Gmail. Work through claude.ai. Setup: find the connector, authenticate,

grant permissions, test. **Desktop extensions** run locally through the Claude Desktop app — local files, browser control, native application integration.

Once connected, Claude considers your tools when responding. Ask "What are my highest priority tasks due this week?" and it searches Asana. Ask "Find the email thread where we discussed the vendor contract" and it searches Gmail. Natural language in, synthesized answer out — pulling from across connected tools.

Security: Permissions scoped to what each connector needs. Claude accesses only what you can access. Connections revocable at any time. Connecting your work email doesn't give Claude access to your CEO's inbox (a clarification that apparently needs making, which tells you everything about enterprise security training).

Enterprise Search adds a dedicated sidebar option for finding and synthesizing knowledge across your organization's tools. Purpose-built for questions that span multiple sources. "What happened yesterday while I was out?" "What is our company's remote work policy?" "How does our authentication system work?" Claude searches SharePoint, Slack, Gmail, Google Drive — unified response with source citations. Requires Team or Enterprise plans.

Research Mode for Deep Dives

Research makes you reconsider what "doing research" means. Claude doesn't run a single web search and summarize the first result. It operates agentically — conducting multiple searches that build on each other, pursuing leads, filling gaps, synthesizing everything into a comprehensive cited report. Five to fifteen minutes for most queries. Up to forty-five for complex investigations. Hours of manual research compressed into minutes.

When to use what:

- **Research** — comprehensive reports, multi-source synthesis, comparative analysis, cited investigations
- **Web search** — quick specific facts, one or two sources needed, speed over depth
- **Extended thinking** — deep reasoning on complex problems without external information, math, code debugging

The prompting stakes are higher with Research because a poorly scoped prompt wastes 5-45 minutes instead of 5 seconds. Be specific. Provide structure. Include

constraints. "Analyze the electric vehicle battery market — identify key players, technology trends, and supply chain challenges affecting investment decisions" beats "Tell me about the EV market" by the same margin that a real RFP beats a napkin sketch.

With integrations connected, Research gets genuinely powerful. Claude pulls from your emails, calendar, and documents alongside web research. "Summarize what's been discussed about Project X across my emails and Slack, then research industry best practices for similar initiatives." Internal context plus external research in a single query. This is the use case that justifies the Enterprise plan cost.

Claude in Action — Use Cases by Role

Specific functions. Specific applications. Not the generic "AI can help everyone!" enthusiasm that pervades every vendor pitch deck on earth.

Cross-functional: Generate project status reports with consistent formatting. Analyze patterns in user feedback. Package brand guidelines in a Skill so Claude applies your standards automatically.

Sales: Build competitive intelligence battle cards. Research prospects and organize talking points. Turn pipeline data into actionable analysis instead of a spreadsheet everyone ignores.

Marketing: Extract insights from campaign metrics. Repurpose content across platforms and audiences without starting from scratch each time.

Finance: Build and refine financial models. Structure investment memos. Decode inherited spreadsheets — which, if you've ever inherited a workbook from someone who left two years ago with zero documentation, is worth the entire Claude subscription by itself.

HR: Create role-specific onboarding guides — the kind of task everyone agrees is important and nobody ever has time to do properly.

Legal: Organize case timelines. Identify patterns in discovery documents.

Research: Plan literature reviews. Verify statistics against raw data.

The pattern across all of them: Claude is most valuable when you give it specific context about your role, your data, and your goals. Least valuable when you treat it as a

question-answering machine. The investment in learning how to work with it pays for itself not in the first interaction, but in the hundredth — when you've built the muscle memory, the custom Skills, and the project knowledge bases that turn a tool into infrastructure.

You've met the machine. You know where it lives, what it can do, and how to work with it. The chapters that follow go deeper — into code, into frameworks, into the API, into the protocol architecture, into the agent patterns that turn Claude from a responsive assistant into an autonomous collaborator.

Let's keep going.

CHAPTER 2

Claude Code

Or, The Competent Junior Engineer Who Works at 3 AM Without Complaint

Sub-Chapter 2A: Foundations

Every coding assistant performs the same trick: making a language model that can only process text look like it can read files, run commands, and manage deployments. The trick isn't intelligence. It's plumbing. Understanding the plumbing matters because it determines what goes wrong when things go wrong — and things will go wrong.

How it actually works: You give a task. Claude formulates a plan and requests actions — read this file, run this command, edit this line. The tool system executes the actual operations. Results flow back. Claude generates its answer. At no point does the model directly touch your filesystem. It asks. The tool system does.

Claude Code ships with generic, abstract tools — bash, read, write, edit, glob, grep — rather than specialized tools like "refactor code." The abstraction is the design: Claude figures out how to combine basic tools for complex tasks, handling scenarios the developers never explicitly planned for. This also explains the security model — Claude navigates your code through direct file operations, reading what it needs when it needs it. Less data leaves your machine. But context management — what Claude knows about your project at any moment — becomes your most important job.

Setup

Terminal-based. macOS, Windows WSL, Linux.

```
# macOS (Homebrew)
brew install --cask claude-code

# macOS, Linux, WSL
curl -fsSL https://claude.ai/install.sh | bash

# Windows CMD
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del
install.cmd
```

Run `claude`. First run prompts authentication. Requires Node.js — verify with `npm help`. If that produces an error instead of output, install Node.js before proceeding. This is the prerequisite that documentation assumes you've handled and reality confirms you haven't.

Context Management

The single highest-leverage skill in Claude Code. Too much irrelevant context measurably decreases performance — the model gets distracted, misses signal in noise, produces worse output than if you'd given it less.

`/init` — run first in any new project. Claude scans your codebase and generates a CLAUDE.md summary: project purpose, architecture, key commands, coding patterns. Included as context in every subsequent request — a persistent system prompt for your project.

The CLAUDE.md hierarchy: Project-level (generated by /init, committed to source control, shared with team). Local CLAUDE.local.md (personal instructions, not committed). User-level ~/.claude/CLAUDE.md (universal preferences across all projects). Specific overrides general — same inheritance pattern as CSS specificity or git config.

`#` **memory command** — add instructions to CLAUDE.md mid-conversation. `# Use comments sparingly. Only comment complex code`. Claude merges it into the appropriate file. Persistent, conversational, no workflow interruption.

`@` **file mentions** — surgical context injection. `How does the auth system work? @auth` includes the file's contents directly. Reference files in CLAUDE.md too: `The schema is in @prisma/schema.prisma. Reference it for data structure questions.` Every piece of context costs tokens, adds latency, and competes for attention. The @ symbol gives you precision control.

The Three-Phase Workflow

Not "describe what you want and hope." A deliberate separation of understanding, planning, and execution.

Phase 1: Feed Context. Before building anything, point Claude at relevant files. "Read @components/UserTable.tsx and @hooks/useUsers.ts. Describe the patterns used." Claude can't infer your preferences from air — it needs examples.

Phase 2: Plan. "Based on what you've read, plan how we'd add pagination. Don't write code yet." Forces reasoning before committing. Plans are cheap to revise. Implemented code is expensive.

Phase 3: Implement. "Implement the pagination plan from above." Claude now has context, a plan, and a clear instruction.

The test-driven variant: feed context → Claude writes test cases → Claude implements tests → Claude writes code that passes the tests. Explicit success criteria instead of ambiguous vibes about what "working" means.

Making Changes — Planning Mode, Thinking Mode, Screenshots

Screenshots (Ctrl+V) — paste a broken layout directly into the terminal. Claude sees what you see. "Fix the alignment issue in this screenshot" is dramatically more effective than describing it.

Planning Mode (Shift+Tab twice) — handles breadth. Claude reads more files, creates detailed plans before executing. Use when a task touches routing, data model, and UI simultaneously.

Thinking Mode ("ultrathink") — handles depth. Extended reasoning budget for complex logic, subtle bugs, tricky algorithms. Use when Claude understands the problem space but keeps producing incorrect logic.

If Claude breaks things elsewhere: more breadth (planning mode). If Claude gets the logic wrong: more depth (thinking mode). If both, use both.

Context Control

Long sessions accumulate noise. Five techniques:

Escape — stops Claude mid-response. Interrupt wrong directions immediately.

Escape + `#` — stop, then add a memory. "Don't modify schema directly — use migrations." Prevents recurrence.

Double Escape — conversation rewind. Jump back to an earlier point, skip dead-ends.

`/compact` — summarizes conversation history while preserving learned knowledge. Context stays, noise goes.

`/clear` — nuclear option. Fresh start for unrelated tasks.

Custom Commands

Create `.md` files in `.claude/commands/`. Filename becomes the command. Use `$ARGUMENTS` for runtime parameters.

```
Write comprehensive tests for: $ARGUMENTS

Testing conventions:
- Use Vitest with React Testing Library
- Place test files in __tests__ directory
- Name as [filename].test.ts(x)
- Use @/ prefix for imports
```

Invoke: `/project:write_tests the use-auth.ts file in the hooks directory`. Conventions baked into the command, not remembered from a previous conversation. **The rule:** any prompt you've typed more than twice should be a custom command.

Hooks — Automated Guardrails

Commands that run before or after Claude executes tools. Automated feedback loops without requiring you to watch every operation.

Pre-tool hooks — block operations. A hook watching `read` and `grep` prevents Claude from accessing `.env` files. Exit code 0: proceed. Exit code 2: blocked, with stderr feedback to Claude.

Post-tool hooks — provide feedback. A TypeScript checker running after file edits catches type errors immediately, feeding them back for automatic correction. Claude edits a function signature → hook catches broken call sites → Claude fixes them.

Two that earn their cost immediately: a type-checker after file edits (catches the most common failure mode — editing a function without updating callers) and a duplicate-code prevention hook for query-heavy codebases.

SDK and GitHub Integration

The SDK — programmatic interface via CLI, TypeScript, and Python. Same tools, runs headless. Designed for pipelines, CI/CD, automated processes. Default read-only; write access requires explicit `options.allowTools` configuration. How Claude Code stops being something a developer uses and starts being something a system invokes.

GitHub integration — `/install GitHub app` adds two Actions: @Claude mention support in issues/PRs and automatic PR review on new pull requests. The PR review isn't a lint pass — it's architectural reasoning. The Terraform example: Claude traced data flow from a Lambda through an S3 bucket shared with an external partner and flagged PII exposure risk. That's not pattern matching. That's understanding how infrastructure components connect.

Sub-Chapter 2B: Workflows and Control

Effective Workflows in Practice

Performance Optimization. The Chalk demonstration: 429 million weekly npm downloads. Claude ran benchmarks to establish baseline, identified hotspots through profiling, created a prioritized fix list, implemented changes one at a time, re-benchmarked after each. Result: 3.9x throughput improvement. The pattern generalizes: give Claude a measurable objective, tools for measurement, and access to the code. Let it iterate. Measure → analyze → change → measure again.

Data Analysis. Claude performed churn analysis on CSV data using Jupyter notebooks — executing code cells iteratively, reading output, deciding what to investigate next. Claude Code isn't restricted to code-editing tasks. Any workflow involving scripts, execution, output reading, and iteration fits the model.

Visual Feedback Loop. Connect the Playwright MCP server. Claude opens a browser, navigates to your app, generates a component, takes a screenshot, analyzes the visual result, updates the code based on what it sees, tests again. Generate → observe → evaluate → improve. The evaluator-optimizer pattern from Chapter 4C, running inside a

single Claude Code session.

Git Integration

Claude handles staging, committing, and writing descriptive messages without leaving the terminal. Documentation mentions this in a bullet point. Working developers use it fifty times a week.

Commit messages reflect actual reasoning — Claude was there, did the work. Not a post-hoc summary of changes you half-remember. Add conventional commit formats or ticket number requirements to CLAUDE.md and Claude follows them automatically.

MCP Servers in Development Workflows

Playwright — the most immediately impactful integration for web development. Claude reads component code, opens a browser, identifies a styling issue, edits, refreshes, verifies. Complete development cycle without leaving the terminal.

Sentry — Claude discovers production bugs, reads stack traces, locates relevant code, proposes fixes. Collapses the bug-to-fix feedback loop from hours to minutes.

Composition — multiple servers matching your development process. Sentry for error detection, Jira for ticket context, Slack for notifications, Playwright for browser testing, custom servers for internal APIs. Five systems, one conversation, one coherent workflow.

GitHub Actions — Claude Code in CI/CD

Automatic PR Review. Every new PR triggers architectural reasoning about what the change does, what it might break, what it doesn't account for. The cost is API calls per PR. The return: a review process that catches things human reviewers miss — not because humans are incompetent, but because the twenty-third PR review on a Friday afternoon gets less attention than the first one Monday morning.

The Composability Story

Custom commands encode conventions. CLAUDE.md ensures architectural context. Hooks enforce quality gates. MCP servers extend reach. GitHub Actions run the apparatus on every PR without human initiation. The three-phase workflow structures each interaction.

The sum: a development environment with intelligence baked in at every layer. Not a coding assistant. Infrastructure.

Sub-Chapter 2C: Extensions and Automation

Or, When One Claude Isn't Enough and You Start Building an Army

Parallelizing with Git Worktrees

The largest single productivity multiplier available. Multiple Claude instances working simultaneously — a team of virtual engineers on your project. The concurrent-write problem has a standard solution: Git worktrees. Complete copies of your project in separate directories, each on its own branch, each running its own Claude instance.

Create worktrees → launch Claude Code in each → assign different tasks → let them work in parallel → commit → merge back to main. Standard branching strategy applied not to humans working asynchronously over days, but to Claude instances working simultaneously over minutes.

Custom commands automate both ends: `/project:create_worktree feature_b` handles creation, symlinking, and editor launch. A merge command handles the other end — Claude resolves conflicts intelligently because it understands the context of changes from both branches.

Scaling limit: not compute — decomposition. You need tasks that separate cleanly. Three independent API endpoints parallelize well. A feature touching routing, data model, and UI simultaneously doesn't. The skill is in the task decomposition, not the tooling.

Automated Debugging — Claude as Night Shift Operations

Removes the human entirely for production error detection and remediation.

The pipeline: A GitHub Action runs daily. Claude queries CloudWatch for errors from the last 24 hours. Filters and deduplicates. Analyzes each error and attempts fixes. Commits code and opens a pull request — clear error description, root cause analysis, specific fix. By the time the team opens their laptops, the fix is waiting for review.

Real-world example: Production deployment uses an invalid model identifier that only breaks in production because dev uses a different API endpoint. Claude reads the CloudWatch error, recognizes the pattern, locates the configuration, applies the correct identifier, opens a PR. No human discovered it. No human diagnosed it. No human wrote the fix. A human reviews and approves. That's the appropriate level of involvement for this class of problem.

The pattern extends beyond debugging: scheduled dependency updates, automated documentation generation, codebase migration tasks — any repeatable workflow involving reading context, reasoning about problems, and producing code changes.

Computer Use — When Code Isn't Enough

Extends Claude from text into the visual world. Same tool use system, applied to desktop automation. Schema specifies display dimensions. Claude requests actions — clicks, keystrokes, screenshots. Your system executes in a Docker container. Screenshots go back to Claude for observation.

```
{
  "type": "computer_20250124",
  "name": "computer",
  "display_width_px": 1024,
  "display_height_px": 768,
  "display_number": 1
}
```

The QA testing pattern: Describe test cases in natural language. Claude navigates, clicks, types, takes screenshots, generates pass/fail reports. In the course example, two tests passed and one caught a real bug — autocomplete dropdown appearing in the wrong location on Backspace. No Selenium required.

The economics: Computer Use tests are slower and more expensive per execution (each step is an API call with image processing). But writing them costs minutes instead of hours, and modifying them is a sentence edit instead of a code refactor.

Security: Sandboxed Docker container. Completely isolated. Nothing Claude does inside affects anything outside unless you explicitly mount volumes or expose ports. Architectural isolation, not optional configuration.

The Extension Architecture

Parallelization, automated debugging, and Computer Use share a structural principle: they extend Claude Code by changing its operating model, not by adding features.

Parallelization changes concurrency (multiple instances via worktrees). Automated debugging changes initiation (GitHub Action triggers instead of human prompts). Computer Use changes interaction (screenshots and clicks instead of file reads and writes).

None require Claude to be smarter. They require the environment around Claude to be structured differently. **That's the insight this chapter builds toward:** Claude Code's power comes not from the model's intelligence alone, but from how the tooling, configuration, and infrastructure around it shapes what that intelligence can reach.

> *Claude Code's power comes not from the model's intelligence alone, but from how the tooling around it shapes what that intelligence can reach.*

Chapter 5: MCP — Connecting the Dots

Or, The Protocol That Means You Never Have to Write Another GitHub Integration from Scratch

The Problem MCP Solves

You're building a chat interface for GitHub data. User asks "What open PRs are there across all my repos?" Without MCP, you write every tool yourself — schemas, functions, error handling, authentication, pagination. GitHub alone requires dozens of tools. Add Jira, Slack, Sentry, your internal APIs — you've invented a full-time job that produces nothing except plumbing.

MCP shifts the burden. Someone else — often the service provider — has already built and maintains the tool definitions inside a dedicated MCP server. You connect. You use. You move on to the part of your application that actually differentiates it.

The common misconception: MCP is "just tool use." It's not. Tool use is how Claude calls tools. MCP is about who creates and maintains those tools. With direct integration,

you author everything yourself. With MCP, it's packaged in a server you connect to. The protocol handles communication. You handle application logic.

Architecture — Client, Server, Transport

MCP Server — wraps external service functionality through a standardized interface. Exposes three capability types: tools (actions), resources (data), and prompts (instruction templates).

MCP Client — communication bridge between your application and MCP servers. Simple methods: `list_tools()`, `call_tool()`. Handles all protocol details.

Transport — communication channel between client and server. Stdio (local development), HTTP, WebSockets. Transport choice affects capabilities — Chapter 6 covers this in depth.

The complete flow for a user query involves ten steps through the chain. That's a lot of steps. Each component has a clear responsibility. The MCP client abstracts the complexity.

Building an MCP Server

The Python SDK reduces server creation to decorated functions:

```
from mcp.server.fastmcp import FastMCP
mcp = FastMCP("DocumentMCP", log_level="ERROR")

@mcp.tool(
    name="read_doc_contents",
    description="Read the contents of a document and return it as a string."
)
def read_document(
    doc_id: str = Field(description="Id of the document to read")
):
    if doc_id not in docs:
        raise ValueError(f"Doc with id {doc_id} not found")
    return docs[doc_id]
```

No manual schema writing. The decorator handles registration. Field descriptions help Claude understand arguments. The SDK transforms verbose schema exercises into standard Python function definitions. Test with the built-in inspector: `mcp dev mcp_server.py` — a browser-based dashboard for testing tools individually without full application setup.

Building an MCP Client

Two core methods:

```
async def list_tools(self) -> list[types.Tool]:
    result = await self.session().list_tools()
    return result.tools

async def call_tool(self, tool_name: str, tool_input: dict):
    return await self.session().call_tool(tool_name, tool_input)
```

In most projects, you implement either a client or a server, not both. Server to expose your service. Client to connect to existing servers.

The Three Primitives

Tools — actions Claude can perform. The tool use pattern from Chapter 4B, packaged in a server someone else maintains.

Resources — data without tool calls. Analogous to GET handlers. Expose document lists, file contents, reference data. Inject directly into prompts — faster than a tool call round-trip. Two types: direct resources (static URIs) and templated resources (parameterized URIs).

Prompts — pre-built instruction templates. Thoroughly tested workflows that users invoke instead of writing their own. Appear as slash commands in the client. You spend time crafting and evaluating prompts once; users benefit from your domain expertise without becoming prompt engineering specialists.

The mental model: Tools let Claude do things. Resources let Claude see things. Prompts tell Claude how to approach things. A well-designed server uses all three — tools for operations, resources for data, prompts for workflows that tie them together. The composition handles what none could do alone.

Chapter 6: MCP Advanced — Sampling, Transports, and the Edges of the Protocol

Or, What Happens When Your MCP Server Needs to Think, Talk Back, and Scale

Sampling — When the Server Needs a Brain

Standard MCP: server receives instructions and executes them. It doesn't think. Sampling changes this — allows an MCP server to request language model generation from the connected client.

Why it matters: Your server fetches Wikipedia data and needs to summarize it. Without sampling, the server needs its own API key, authentication, and token budget. For a publicly accessible server, that's random users generating unlimited text at your expense. Sampling eliminates this — each client pays for their own AI usage. The server provides functionality without incurring LLM costs.

Implementation: Server uses `ctx.session.create_message()` during tool calls. Client implements a sampling callback that calls Claude and returns the result. One detail the docs don't emphasize: MCP message formats differ from Anthropic SDK formats. You need conversion logic.

Log and Progress Notifications

When tools take time, users see nothing until completion. The operational equivalent of a loading spinner providing zero information.

Fix: Tool functions receive a `context` argument with `context.info()` for log messages and `context.report_progress()` for progress updates. Client implements callback functions for each. For anything taking more than a few seconds, the difference between "nothing happening" and "research complete, writing report..." is the difference between a user who waits and a user who kills the process.

Roots — Filesystem Access with Boundaries

The problem: User asks Claude to "convert biking.mp4 to MOV." Server can't search the entire filesystem. Roots let users grant access to specific directories. Claude calls ListRoots → ReadDirectory → ConvertVideo with full path. Permission boundary and autonomous discovery simultaneously.

Critical detail: The SDK does not automatically enforce root restrictions. You must implement access checks manually. `is_path_allowed()` called before every file operation is not optional — it's the entire security model.

Transports — How Messages Move

Stdio — default for local development. Full bidirectional communication. Client and server must run on the same machine.

StreamableHTTP — enables remote hosting. HTTP is fundamentally unidirectional, so server-to-client communication uses server-sent events (SSE). Session IDs, long-lived connections, two types of SSE channels. Clever infrastructure that works but introduces configuration complexity.

The Configuration Trap

Two flags control StreamableHTTP behavior. Both destroy capabilities when enabled.

Stateless HTTP (true): Needed for horizontal scaling behind load balancers. Cost: no sampling, no progress logging, no resource subscriptions, no server-to-client communication of any kind.

JSON Response (true): Disables streaming on POST requests. No intermediate progress, no log statements during execution.

The deployment trap: Everything works in local development with stdio. Deploy with StreamableHTTP and flags set to true. Features depending on bidirectional messaging silently stop working. No errors. No warnings. Messages have no path to the client.

The rule: Use the same transport in development as production. Discover constraints early.

Transport Selection

Stdio: Same machine, full bidirectional, development/testing.

StreamableHTTP (default flags): Remote hosting, need sampling/progress, traffic doesn't require horizontal scaling.

StreamableHTTP (stateless=true): Horizontal scaling, can live without server-initiated messages.

No configuration gives you horizontal scalability, full bidirectional communication, and remote hosting simultaneously. Pick two.

Chapter 7: Claude on Bedrock — Enterprise Reality

Or, Where Your Cloud Bill Goes to Die

Same Claude models. Different everything else. `boto3` instead of the Anthropic SDK. Inference profiles instead of model IDs. AWS documentation, not Anthropic documentation. If you're in an enterprise running on AWS, this isn't optional.

The critical distinction: Mixing up which documentation you're reading produces errors that are technically correct and practically useless — the kind where the message tells you what went wrong without telling you why.

Making Requests

Three components: `boto3` client with correct region, model ID, structured message. **Model IDs and the Region Problem:** not all models exist in every region. Solution: inference profiles — found in the console under "cross region inference," not the model catalog. Use profile IDs for automatic routing. Response text lives at `response["output"]["message"]["content"][0]["text"]` — a path that tells you everything about how AWS thinks about nested data structures.

Multi-Turn, System Prompts, Temperature

Multi-turn: Stateless. Same pattern as Chapter 4 — maintain full history, send with every request. Messages must alternate user/assistant.

System prompts: Via `system` keyword parameter. Same principles as Chapter 3.

Temperature: 0.0 deterministic, 1.0 maximum variability. The right value is determined by testing against your use case, not by reading someone else's blog post.

Streaming and Output Control

Streaming: `converse_stream()` instead of `converse()`. For user-facing applications, not optional — it's the difference between "responsive" and "broken."

Output control: `max_tokens`, `stop_sequences`, system prompts for format. **Structured data extraction** uses tools without performing actions — the forced-tool-use pattern from Chapter 4B through the Bedrock surface.

Evaluation

Generate test dataset → run through prompt → grade (code-based for objective correctness, model-based for subjective quality) → iterate. The principle worth emphasis: every configuration decision — temperature, extended thinking, prompt structure — should be eval-driven, not assumption-driven. The number of production deployments running default parameters because nobody bothered to test alternatives is, conservatively, most of them.

Extended Thinking, Vision, Prompt Caching

Extended thinking: Reasoning before final response. Higher cost, higher latency, higher accuracy on complex tasks. Enable after prompt optimization, not before.

Vision: Up to 20 images per request. Accuracy depends on prompt engineering, not image quality. Structured analysis prompts with verification steps turn vision from novelty to production capability.

Prompt caching: Stores processing work for 5 minutes. Cache point marks where caching occurs — everything before is cached. Minimum 1024 tokens. Content must be exactly identical. Best targets: system prompts, tool schemas, stable prefix content.

Tool Use, RAG, and MCP on Bedrock

Same conceptual foundations as Chapters 4-6, different API surface. Tool use, RAG pipeline (chunking, embeddings, hybrid search, reranking), and MCP all work through Bedrock identically in concept. Implementation uses `boto3` instead of the Anthropic SDK.

Claude on Bedrock is Claude with an AWS accent — same intelligence, different dialect. Learn the dialect. Ship the deployment. Budget for the moment procurement asks why the Bedrock line item tripled. The answer is always "because it's working," which is the one answer procurement never wants to hear.

Chapter 8: Agent Skills — Teaching Claude New Tricks

Or, The Part Where You Build the Instruction Manual Your Organization Will Never Read

Every time you explain your coding standards to Claude, you're repeating yourself. This is the AI equivalent of training a new employee who develops amnesia every evening — and if that sounds like a contact center with 40% annual turnover, you now understand why skills exist.

What Skills Are

A folder containing a `SKILL.md` file with YAML frontmatter (name, description) and instructions. Claude Code scans at startup, loads only names and descriptions, matches requests using semantic matching. On match, full content enters context.

```
---
name: pr-description
description: Writes pull request descriptions. Use when creating a PR,
  writing a PR, or when the user asks to summarize changes for a pull request.
---

When writing a PR description:
1. Run `git diff main...HEAD` to see all changes on this branch
2. Write a description following this format:
```

The description field is the most important element. It's what Claude uses for matching. Skill not triggering? The description doesn't overlap semantically with how you phrase your requests. Almost always the problem.

Where Skills Live

Personal (`~/.claude/skills`) — follow you across projects. **Project** (`.claude/skills` in repo) — shared via version control. **Priority:** Enterprise (highest) → Personal → Project → Plugins (lowest). Use descriptive names — "frontend-review" not "review" — to avoid collisions you'll spend forty-five minutes debugging before discovering the answer was nomenclature.

Skills vs. Everything Else

CLAUDE.md — loads into every conversation, always. Project-wide standards. **Skills** — load on demand when task matches. **Slash commands** — explicit invocation. **Hooks** — fire on events. **Subagents** — isolated execution contexts. **MCP servers** — external tools.

A production setup uses all five: CLAUDE.md for always-on standards, skills for task-specific expertise, hooks for automated operations, subagents for delegated work, MCP servers for external integrations.

Configuration

`allowed-tools` — restricts which tools Claude can use when skill is active. A codebase-onboarding skill set to `Read, Grep, Glob, Bash` prevents file modifications during exploration.

Progressive disclosure — keep SKILL.md under 500 lines. Put detailed reference material in separate files. Include instructions about when to load each. The context window contains a table of contents, not the entire document.

Scripts — execute without loading contents into context. Only output consumes tokens. Tell Claude to *run* the script, not *read* it.

Sharing

Repository — commit to `.claude/skills`, propagates via Git. **Plugins** — package and distribute through marketplaces. **Enterprise managed settings** — administrators deploy organization-wide with highest priority. `strictKnownMarketplaces` controls plugin sources.

Troubleshooting

Doesn't trigger: Description doesn't match your phrasing. Add trigger phrases reflecting how you actually word requests.

Doesn't load: Structural problem. Must be inside a named directory, filename exactly `SKILL.md`. Run `claude --debug`.

Wrong skill fires: Descriptions too similar. Make them more distinct.

Runtime failure: Missing dependencies, permission issues (`chmod +x`), path separators (forward slashes everywhere).

Start with the validator. Catches structural problems before you debug anything else.

The rule of thumb: if you find yourself explaining the same thing to Claude twice, that's a skill waiting to be written. Second rule: if your skill description doesn't contain the words people actually use when requesting the task, it will sit in your skills directory like a fire extinguisher behind a locked cabinet — technically available, practically useless.

> *Like a fire extinguisher behind a locked cabinet — technically available, practically useless.*

Chapter 9: Agents & Workflows — The Architecture of Autonomy

Or, When You Let the Machine Drive and Hope It Read the Manual

Throughout this guide, you've been building both without naming them. Claude with tools figuring out tasks: agent. Complex task broken into predetermined steps: workflow. This chapter names the patterns and gives you the decision framework.

Workflows — Predetermined Steps for Known Problems

Evaluator-Optimizer. Producer creates output. Grader evaluates against criteria. Rejected output cycles back with feedback. Repeat until acceptance. If this sounds like every QA review you've suffered through — except the feedback loop runs in seconds and nobody sends a passive-aggressive email between iterations. The producer and grader can be separate Claude calls or the grader can be code-based.

Parallelization. Split a complex task into simultaneous sub-tasks, aggregate results. Instead of one prompt evaluating all six material types (mediocre analysis across the board), six parallel requests with focused criteria each. Advantages compound: focused attention, independent optimization, easy scaling, reduced cognitive load per request.

Chaining. Sequential subtasks where each output feeds the next input. Solves the long-prompt problem — when Claude gets a content generation prompt with 15 constraints, it reliably violates some. Not because it can't follow instructions, but because

juggling creation and compliance simultaneously is hard. A fact anyone who's tried writing a creative brief while adhering to brand guidelines, legal requirements, and their manager's unspoken preferences will recognize immediately. Solution: first call generates, second call revises.

Routing. Categorize incoming requests, direct to specialized pipelines. Different content types, different optimized prompts. Each pipeline fully optimized for its category.

Agents — Tools and Goals for Unknown Problems

Give Claude tools and an objective. Claude figures out the combination. Three datetime tools seem trivial individually — Claude chains them to handle "When does my 90-day warranty expire?" by recognizing it needs to ask when you purchased the item.

Tools should be abstract. Claude Code's toolset: bash, read, write, edit, glob, grep. No "refactor code" tool. Hyper-specialized tools constrain. Abstract tools expand.

Environment inspection is the missing requirement. After every action, Claude needs observation — screenshot after clicking, file contents after editing, API response after sending. Without it, Claude can't know if actions succeeded. "How will Claude know if this worked?" If you can't answer that for every tool, you haven't built an agent — you've built a wish and a prayer with a JSON schema.

> *You haven't built an agent — you've built a wish and a prayer with a JSON schema.*

The Decision Framework

Workflows: higher accuracy, easier testing, predictable execution, reliable. Use when you can picture the steps ahead of time.

Agents: flexibility, creative tool combination, handles novel situations. Lower completion rates. Harder to test. Less predictable. Use when requirements genuinely can't be predetermined.

The recommendation: Always implement workflows where possible. Resort to agents only when truly required. Users don't care that you built a sophisticated agent. They care that the product works consistently. The distinction between "this is cool" and "this was necessary" is the entire difference between engineering and demo theater.

> *The distinction between 'this is cool' and 'this was necessary' is the entire difference between engineering and demo theater.*

Back Matter: Now What?

Or, The Part Where You Close the Guide and Open the Terminal

You've covered everything Anthropic's course curriculum teaches — compressed from 193,500 words of instructional content into something that fits in your head and stays there.

A mental model. Claude is a stateless text-prediction system with a massive context window, tool use capabilities, and an increasingly sophisticated ecosystem. It doesn't remember. It doesn't understand. It predicts the next token based on everything you've given it. Every technique in this guide — prompt engineering, system prompts, CLAUDE.md files, skills, MCP servers, the three-phase workflow — is a different way of giving Claude better material to predict from. The model is the constant. Your inputs are the variable.

> *The model is the constant. Your inputs are the variable.*

A toolkit. Nine chapters of deployable techniques. Prompt structures that work. API patterns that scale. Claude Code configurations encoding team conventions. MCP servers extending reach. Workflow architectures solving known problems reliably. Agent patterns handling unknown problems flexibly. Skills capturing expertise once and applying it automatically. None theoretical. All operational the moment you use them.

A calibration. You know where Claude is strong and where it isn't. You know extended thinking costs latency and tokens. You know prompt caching saves money on repeated content. You know the difference between a good interaction and a mediocre one is almost always input quality, not model quality. Calibrated expectations are the most valuable thing any technical guide can produce.

The Gap This Guide Doesn't Close

This guide teaches how Claude works and how to build with it. Not what to build. That's harder and matters more.

The people who get transformative value aren't the ones who master prompt engineering. They're the ones who look at their actual work — the repetitive analysis, manual data processing, document creation, operational reporting — and recognize which parts are patterns Claude can execute. The recognition is the skill. The execution is what this guide teaches.

If you came here using Claude as a slightly fancier search engine, you now have the knowledge to use it as infrastructure. The distance between those two things is measured in problems solved.

Three Things to Do Now

First: Pick one real workflow from your actual job and build it. The PR description skill. The automated debugging pipeline. A custom command encoding your commit format. Something small, something real, something you'll use tomorrow. The compound interest starts with the first deposit.

Second: Build a CLAUDE.md for your primary project. Coding conventions, architectural decisions, testing requirements — everything you'd tell a new team member in week one. The single highest-leverage action in Claude Code.

Third: Run an eval. Pick a task Claude handles regularly. Five test cases with known-good outputs. Run. Score. Change prompt. Run again. This is how you stop guessing and start measuring.

The Author's Note, Revisited

Claude narrows the gap between what technology promises and what it delivers — in proportion to how well you understand it. A power drill in the hands of someone who doesn't know which end to hold is an expensive noisemaker. The same drill in the hands of someone who understands torque, bit selection, and material properties builds things.

> *A power drill in the hands of someone who doesn't know which end to hold is an expensive noisemaker.*

You now understand the tool. Go build things.

And when the next "digital transformation" initiative lands on your desk with an 80-slide strategy deck and a timeline no one believes — you'll have something the deck doesn't: a working system, built by someone who actually read the manual.

Pete Connor builds things that work in environments where failure has consequences. He can be found wherever the gap between the strategy deck and the production floor is widest.

CHAPTER 3

AI Fluency — The 4D Framework

Or, A Surprisingly Non-Terrible Framework for Not Being Terrible at AI

Here's something that sounds like a contradiction until you've lived it: the people worst at using AI are overwhelmingly the people who understand AI best as a technology and least as a collaboration. They can explain transformer architecture. They can debate RLHF versus DPO. They can quote benchmark scores from memory. And they produce mediocre outputs because they treat Claude like a database with delusions of grandeur instead of a collaborator that requires the same communication skills you'd use with a competent but unfamiliar colleague.

AI Fluency is the ability to collaborate with AI effectively, efficiently, ethically, and safely. The framework — developed through research collaboration between Professor Rick Dakan at Ringling College and Professor Joseph Feller at University College Cork — has the unusual distinction of being an academic framework that translates directly to practical competence without requiring a three-day workshop to decode.

Four competencies. Not sequential steps — overlapping capabilities you deploy simultaneously and improve through practice. They apply whether you're automating tasks, augmenting your thinking, or configuring Claude to work autonomously. The competencies don't change as AI evolves. The specifics of how you apply them do. Which is exactly what makes this worth learning instead of memorizing prompting tricks that'll be obsolete in six months.

What Generative AI Actually Does

You don't need to become an AI researcher. You need to understand the mechanism well enough to work with it intelligently — the same way understanding that a car engine needs oxygen changes how you troubleshoot a stall.

Generative AI creates new content by predicting the most likely next token based on patterns learned during training. The output has never existed before in exactly that form. That's both the capability (novel content tailored to your request) and the limitation (no guarantee the novel content is factually accurate). Three developments made this possible: transformer architecture (2017, processing entire text sequences in

parallel), vast training data (a meaningful fraction of everything humanity has published), and computational power that makes your company's IT budget look like a rounding error.

The capabilities are real: versatility across language tasks, conversational awareness, seamless task-switching without reconfiguration, and external tool integration through MCP.

The limitations are equally real. Knowledge cutoffs mean Claude doesn't know about events after its training data ends without web search. Hallucinations — the polite term for generating incorrect information with the confidence of someone who just checked their notes — remain fundamental, especially for specific facts and numerical data. Context windows, even at 200K+ tokens, have limits. And complex multi-step reasoning still requires human oversight, because Claude can follow a chain of logic off a cliff with the same serene confidence it brings to everything.

The most effective applications combine both. Humans provide judgment, creativity, ethical oversight, domain expertise. AI provides speed, consistency, pattern recognition, tireless availability. Neither is sufficient alone.

The 4D Framework

Delegation — The Strategic Layer. Deciding what to do yourself, what to collaborate on, and what to hand off entirely. The "should I even involve Claude in this?" question that most people skip, jumping straight to prompting without asking whether the task is appropriate for AI in the first place.

Description — The Communication Layer. Communicating effectively with AI systems. Goes beyond "writing good prompts" into creating an environment where collaboration actually works. The skill most people think they're good at. Most people are mediocre at it, because clear communication is hard with humans and doesn't get easier when your audience processes tokens instead of emotions.

Discernment — The Quality Control Layer. Evaluating whether what Claude produced is actually good — not just whether it sounds good. Where domain expertise becomes irreplaceable, because you can't evaluate output quality in a domain you don't understand.

Diligence — The Responsibility Layer. Using AI responsibly and ethically. The layer most people consider optional until it becomes a crisis — making thoughtful choices about

systems, maintaining transparency about AI's role, and taking accountability for outputs you share.

These form a cycle. Delegate → Describe → Evaluate → Ensure responsibility → Refine everything based on what you learned. Every iteration makes you more fluent.

Delegation in Practice

Three components that build on each other:

Problem Awareness — understanding your goals before you open a chat window. What are you trying to accomplish? What does good look like? The number of people who start a Claude conversation without a clear picture of what they need is — to borrow from my contact center years — roughly the same percentage of callers who can't describe the problem they called about. You wouldn't walk up to a colleague and say "help me with the thing." Same rules apply.

Platform Awareness — knowing what Claude can and cannot do. Excels at writing, analysis, coding, research synthesis, pattern recognition. Struggles with precise numerical calculations (use code execution), real-time information (use web search), and anything requiring physical-world interaction. Knowing boundaries before you delegate prevents the frustration of asking Claude to do something it fundamentally can't — the AI equivalent of asking your accountant to fix the plumbing.

Task Distribution — the strategic split between you and Claude. Not everything should be delegated. Not everything should be kept. Sometimes Claude does 90% and you refine the last 10%. Sometimes you do 90% of the thinking and Claude handles production. Sometimes it's genuine 50/50 iteration. The ratio changes with every task. The framework for determining the ratio doesn't.

Description in Practice

Three types most people collapse into one:

Product Description — what you want created. Output, format, length, detail level, audience, tone. "Write me a report" is a product description. "Write a 2,000-word analysis of Q3 customer churn patterns for our VP of Customer Success, structured with an executive summary, three key findings with supporting data, and recommended actions

— professional but direct tone" is a product description that produces useful output. The gap between those two prompts is the gap between vague intent and actionable specification.

Process Description — how you want Claude to approach the work. "First review all uploaded feedback data, then identify the top five recurring themes, then for each theme provide specific quotes illustrating the pattern, then synthesize into a narrative about what's driving customer behavior." You're not just telling Claude what to produce — you're telling it how to think about the problem. Especially valuable for complex analytical tasks where the reasoning path matters as much as the conclusion.

Performance Description — how you want Claude to behave during the collaboration. Concise or detailed? Challenging or supportive? "Be a thoughtful critic — push back if my assumptions seem weak, ask probing questions rather than just executing." That instruction produces a fundamentally different collaboration than the default helpful-and-agreeable mode.

Six techniques that sharpen Description:

Give context. "Preparing for a board presentation on Tuesday" primes differently than "brainstorming for a blog post." Context shapes everything downstream.

Show examples. One good example communicates things fifty words of description can't — tone, structure, formality, the specific way you organize information.

Specify constraints. "Under 300 words." "Table format — pros, cons, cost." "Technical level appropriate for someone with a CS degree." Constraints prevent Claude from defaulting to its own preferences.

Break complex tasks into steps. "First summarize the data. Then identify the three strongest trends. Then explain what's driving each. Then suggest three actions." Step-by-step instructions produce step-by-step thinking.

Ask Claude to think first. "Before writing the email, think through the recipient's main concerns and how we should address each one." Especially valuable for complex analytical or persuasive tasks.

Define the role. "You're a skeptical financial analyst reviewing this proposal" produces different output than "You're a supportive mentor helping me improve." Role definition changes the lens Claude uses for everything.

The secret weapon most people never use: Ask Claude to improve your prompt. "Here's what I want to accomplish. Can you suggest a better way to frame this request?" Claude can often identify exactly what information it needs to do its best work — which is the information you should have included.

Discernment in Practice

The flip side of Description. Three types, mirroring the three types above:

Product Discernment — evaluating output quality. Is the information accurate? Is the analysis appropriate? Does the tone match? The challenge: "good-sounding" and "good" are different things, especially in domains where Claude produces fluent, confident text about topics it has subtly wrong.

Process Discernment — assessing how Claude arrived at its output. Did it follow logical reasoning? Are there gaps? Did it skip steps that matter? This is where asking Claude to show its reasoning (or using extended thinking) becomes valuable — not because Claude's reasoning is always sound, but because visible reasoning is easier to evaluate than invisible reasoning.

Performance Discernment — evaluating how Claude behaved during the interaction. Was it responsive to direction? Was terminology appropriate? Did the communication style help or hinder? A collaboration where Claude consistently misreads your intent produces worse outcomes even when individual responses seem acceptable.

The exercise that builds Discernment fastest: Return to a topic where you have genuine expertise. Ask Claude for three different analyses. Apply your expert knowledge to evaluate each across all three discernment types. Identify the strongest and weakest. Tell Claude specifically why. Work together on an improved version. This reveals the relationship between domain knowledge and evaluation quality — without expertise, you can't reliably distinguish good output from confident-sounding mediocre output. Which is precisely why Discernment and Delegation work together: delegate tasks in domains where you have enough expertise to evaluate the results.

The Description-Discernment Loop

This is where the framework stops being four separate competencies and becomes an integrated practice.

Describe what you need → Claude produces output → Evaluate the output → Refine your description → Claude produces improved output → Evaluate again. The loop continues until the output meets your standards. Each pass produces better results because your descriptions get more precise and your discernment gets sharper.

The patterns you'll notice: specific product descriptions and clear process guidance consistently produce the best outcomes. Description requires more initial effort than Discernment — it's harder to communicate what you want than to evaluate what you get. And your execution will diverge from your plan, because the loop reveals things you didn't anticipate. That's not failure. That's how iterative collaboration works.

Diligence in Practice

Three components the other competencies don't cover:

Creation Diligence — thoughtful choices about which AI systems you use and what data you share. Using Claude for customer data analysis is a Creation Diligence decision. Including PII in your prompts is a Creation Diligence decision. Using an unvetted AI system is a Creation Diligence decision. Every choice has implications. Think about them before they become problems.

Transparency Diligence — honesty about AI's role in your work. Different contexts have different expectations — academic settings require explicit disclosure, professional settings vary, personal projects may have none. The principle is consistent: people who rely on your work deserve to know how it was produced.

Deployment Diligence — taking ownership of AI-assisted outputs. If you share a Claude-assisted report with inaccurate data, "but the AI wrote that part" is not a defense. You deployed it. You own it. This is the component most people skip and the one that matters most when something goes wrong.

Three Modes of Engagement

The 4D framework applies across three increasingly autonomous modes:

Automation — Claude completes specific tasks based on your instructions. "Summarize this document." "Format this as a table." You define, Claude executes, you evaluate.

Augmentation — you and Claude collaborate as thinking partners. The mode most of this guide assumes, because it's where Claude delivers the most value for knowledge workers. You're thinking together, iterating together, building something neither would produce alone. The Description-Discernment loop runs at its most intense here.

Agency — Claude works independently on your behalf. Claude Code operating autonomously on your codebase. Research mode investigating through dozens of searches. Skills executing complex workflows without step-by-step supervision. Requires the highest levels of all four competencies, because autonomous action without careful delegation, precise description, rigorous discernment, and stringent diligence is how organizations discover their AI liability insurance doesn't cover what they assumed it covered.

The progression isn't a hierarchy — it's a spectrum. Effective users move fluidly between modes depending on the task. Format this CSV (automation). Help me analyze this data (augmentation). Research this topic thoroughly (agency). The framework applies to all three.

The Trajectory

AI Fluency isn't mastered in a course or a chapter. It develops through the repeated cycle of delegating, describing, evaluating, ensuring responsibility, and doing all of it slightly better next time.

The difference between Claude producing mediocre output and Claude producing exceptional output is almost never the model. It's the person using it. The 4D framework gives you a structure for being that person. The investment pays dividends immediately and compounds over time — better delegation, sharper descriptions, more precise evaluation, more refined ethical instincts. Every iteration.

Fluency isn't a destination. It's a trajectory.

CHAPTER 4

Building with the API

Or, Where We Leave the GUI Behind and Start Talking to the Machine Directly

Everything up to this point has been Claude through its consumer interfaces. For most people, that's enough. But if you're building applications, integrating Claude into production systems, or doing anything requiring programmatic control, you need the API — the raw pipe between your code and Claude's intelligence.

Prerequisites are real: proficiency in Python, basic JSON handling, an Anthropic API key. If that's you, this chapter takes you from "I can write a prompt in the chat interface" to "I can build a production application that uses Claude as its reasoning engine." If not, Chapter 3's framework applies regardless — the concepts transfer even if the implementation details don't.

Three sub-chapters. 4A: fundamentals — requests, conversations, output control, prompt evaluation, prompt engineering. 4B: tool use and RAG. 4C: advanced features. Let's start with the plumbing.

Sub-Chapter 4A: API Fundamentals & Prompt Engineering

The Request Flow

Every interaction follows five phases: client to your server, your server to Anthropic API, model processing, Anthropic to your server, your server to client.

The architecture point that matters most: never make API requests from client-side code. The API key is a secret. Exposing it in client code is a security vulnerability anyone can exploit. Your app sends requests to your server; your server communicates with Anthropic using the securely stored key. Standard practice for any third-party API, but worth stating because the first instinct of every developer who just got their key is to paste it into frontend JavaScript — an instinct that will cost you money and sleep.

SDKs for Python, TypeScript, JavaScript, Go, and Ruby. Four required fields per request: API key, model name (e.g., `"claude-sonnet-4-0"`), messages list, and max_tokens.

How Claude Processes Your Request

Four stages: **Tokenization** breaks input into tokens (roughly one per word). **Embedding** converts tokens into numerical vectors capturing semantic meaning. **Contextualization** refines embeddings using the transformer's attention mechanism — "quantum computing" gets different treatment than "quantum of solace." **Generation** calculates probabilities for each possible next token, selects one (influenced by temperature), adds it to the sequence, and repeats.

When generation stops: Claude checks three conditions after each token — max_tokens limit hit, end-of-sequence token generated, or predefined stop sequence encountered. The `stop_reason` field tells you which triggered, and that matters for your application logic.

Getting Connected

Navigate to console.anthropic.com. Create a key. Copy it when displayed — shown once only. Store in a `.env` file, never in code. Add `.env` to `.gitignore`. This isn't optional security theater — it's the minimum viable practice for not having your key scraped from a public repo.

```
%pip install anthropic python-dotenv
```

```
from dotenv import load_dotenv
load_dotenv()
from anthropic import Anthropic

client = Anthropic()
model = "claude-sonnet-4-0"
```

The core function is `client.messages.create()` — three required parameters: model, max_tokens (ceiling, not target), and messages:

```
message = client.messages.create(
    model=model,
    max_tokens=1000,
    messages=[
        {
            "role": "user",
            "content": "What is quantum computing? Answer in one sentence"
        }
    ]
)
```

Extract response text with `message.content[0].text`. That's your first API call. Everything that follows is about making it more sophisticated.

Multi-Turn Conversations

Claude stores nothing between requests. Each call is independent — no memory of previous exchanges. Follow up "What is quantum computing?" with "Write another sentence" and Claude has zero context. It will write a sentence about something completely unrelated.

To maintain conversation context: manually maintain a message list in your code, send the complete history with every request. Three helper functions make this practical:

```
def add_user_message(messages, text):
    messages.append({"role": "user", "content": text})

def add_assistant_message(messages, text):
    messages.append({"role": "assistant", "content": text})

def chat(messages):
    message = client.messages.create(
        model=model,
        max_tokens=1000,
        messages=messages,
    )
    return message.content[0].text
```

These helper functions are your workhorses throughout any Claude API integration.

System Prompts

System prompts shape Claude's behavior across an entire conversation. Without one, Claude gives generic answers. With one, you get specialized, role-appropriate responses.

The canonical example — a math tutor: without the system prompt, Claude solves the problem immediately. With "You are a patient math tutor. Do not directly answer. Guide them step by step," Claude asks guiding questions instead.

```
def chat(messages, system=None):
    params = {
        "model": model,
        "max_tokens": 1000,
        "messages": messages,
    }
    if system:
        params["system"] = system
    message = client.messages.create(**params)
    return message.content[0].text
```

Implementation detail: Claude's API doesn't accept `system=None` — conditionally include the parameter only when provided.

Temperature

Decimal between 0 and 1. Controls token selection randomness. **Low (0.0-0.3):** factual responses, coding, data extraction — consistency matters. **Medium (0.4-0.7):** summarization, educational content, constrained creativity. **High (0.8-1.0):** brainstorming, creative writing, idea generation — variety matters. Default is 1.0. Match to your use case.

Response Streaming

Standard calls: wait for complete response before seeing anything. Long responses mean users staring at blank screens assuming something broke.

Streaming sends response chunks as generated — the "typing" effect from the chat interface. Use `client.messages.stream()` (not `create()`):

```
with client.messages.stream(
    model=model,
    max_tokens=1000,
    messages=messages
) as stream:
    for text in stream.text_stream:
        print(text, end="", flush=True)
```

For web apps, send chunks via Server-Sent Events or WebSockets instead of printing.

Controlling Output

Prefilled assistant messages let you provide the beginning of Claude's response. Add an assistant message at the end of your message list — Claude continues from that point without repeating it. "Coffee is better because" steers the entire response toward coffee. Steer anywhere.

Stop sequences force generation to end when Claude produces a specific string. `stop_sequences=["5"]` on "Count from 1 to 10" returns "1, 2, 3, 4,".

The combination is where the real power lives. Generating clean JSON for programmatic consumption:

```
messages = []
add_user_message(messages, "Generate a short EventBridge rule as JSON")
add_assistant_message(messages, "```json")
text = chat(messages, stop_sequences=["```"])
clean_json = json.loads(text.strip())
```

Prefill makes Claude think it started a code block. Stop sequence catches the closing. Result: clean JSON, no markdown, no explanations. This pattern works for any structured output — Python, CSV, anything where you want content without commentary.

Prompt Evaluation

Writing a good prompt is the beginning. Measuring whether it works is the discipline.

The eval workflow in five steps: **Draft** a prompt (doesn't need to be good — needs to exist as a baseline). **Build a dataset** — sample inputs representing production use cases. **Feed through Claude** — merge each test case with your prompt, collect responses. **Grade** — model-based grading (Claude evaluates its own output) and/or code-based grading (syntax checks, format validation). **Change the prompt and repeat** — compare scores, keep what improves, revert what doesn't.

```
def generate_dataset():
    prompt = """
    Generate an evaluation dataset for prompts that generate Python,
    JSON, or Regex for AWS-related tasks. Generate an array of JSON objects,
    each representing a task.
    Please generate 3 objects.
    """
    messages = []
    add_user_message(messages, prompt)
    add_assistant_message(messages, "```json")
    text = chat(messages, stop_sequences=["```"])
    return json.loads(text)
```

The course's running example: naive prompt scored 2.32, clear rewrite scored 3.92, added specificity scored 7.86. Each technique measurable. Each improvement validated. That's what separates prompts that survive production from prompts that worked in the demo.

Prompt Engineering Techniques

Four techniques that stack — each measurably improves output, validated through the eval pipeline above.

Be clear and direct. First line matters most. Start with an action verb. "Generate a one-day meal plan for an athlete that meets their dietary restrictions" beats "What should this person eat?" by a margin of 2.32 → 3.92 on the eval scale.

Be specific. Two types: output quality guidelines ("Include calorie totals, macros, meal timing, portion sizes in grams") and process steps ("First examine market metrics, then assess industry changes, then review individual performance"). Adding guidelines jumped the eval from 3.92 → 7.86.

Structure with XML tags. When prompts include data to analyze, code to debug, or documentation to reference, XML tags create unambiguous boundaries. `<sales_records>` wrapping data prevents Claude from confusing instructions with content. Descriptive names: `<athlete_information>` beats `<data>`.

```
<athlete_information>
- Height: 6'2"
- Weight: 180 lbs
- Goal: Build muscle
- Dietary restrictions: Vegetarian
</athlete_information>

Generate a meal plan based on the athlete information above.
```

Provide examples. One-shot or multi-shot. Show Claude what good output looks like — examples communicate things descriptions can't. Structure with XML tags (`<sample_input>`, `<ideal_output>`). Mine your eval results for highest-scoring outputs and use those as examples. Include examples addressing your most common failure cases.

The secret weapon most people never use: Ask Claude to improve your prompt. "Here's what I want to accomplish. Can you suggest a better way to frame this?" Claude can identify exactly what information it needs to do its best work.

These four layers stack. A production prompt uses all of them — clear opening, specific guidelines, XML-structured data, examples for edge cases. The eval pipeline measures the cumulative impact.

Sub-Chapter 4B: Tool Use & Retrieval Augmented Generation

Or, Teaching the Machine to Pick Up the Phone and Read the File

Claude knows a lot. Claude does not know what time it is, what's in your database, or what happened after its training cutoff. The intelligence is there. The plumbing isn't. Tools are the plumbing.

How Tool Use Works

Four phases: you send Claude a question plus descriptions of available tools. Claude analyzes the question and responds with a structured tool request. Your server executes the function. You send results back to Claude, which generates the final response.

Claude never directly calls external APIs. It requests information; your code decides whether and how to fulfill that request. You control what Claude can access — an important security boundary.

Building Tool Functions

A tool function is ordinary Python. What makes it a "tool" is the JSON schema describing it to Claude and the conversation loop connecting requests to function calls.

Three guidelines: descriptive names, input validation, meaningful error messages ("Location cannot be empty" is actionable; a stack trace is not).

```
def get_current_datetime(date_format="%Y-%m-%d %H:%M:%S"):
    if not date_format:
        raise ValueError("date_format cannot be empty")
    return datetime.now().strftime(date_format)
```

Tool Schemas

The JSON schema tells Claude what arguments a function expects and when to use it. Three parts: name, description (3-4 sentences — what it does, when to use it, what it returns), and input_schema:

```
get_current_datetime_schema = {
    "name": "get_current_datetime",
    "description": "Returns the current date and time formatted according to t
    he specified format. Use when a user asks about the current time or date."
    ,
    "input_schema": {
        "type": "object",
        "properties": {
            "date_format": {
                "type": "string",
                "description": "Python strftime format codes for the returned
                datetime.",
                "default": "%Y-%m-%d %H:%M:%S"
            }
        },
        "required": []
    }
}
```

The description is crucial — it's what Claude reads to decide whether to call your tool. Vague description, unreliable behavior. Precise description, predictable behavior. You can use Claude itself to generate schemas — paste your function and the docs, ask Claude to write the schema. Using Claude to write schemas for Claude is not recursive madness. It's the most reliable path to properly formatted specs.

The Tool Use Conversation

When Claude decides to use a tool, responses become multi-block — Text Block (human-readable explanation) plus ToolUse Block (which tool, what parameters, unique ID). Enable with `tools` parameter in your API call.

Send results back with a tool result block. **Critical:** the `tool_use_id` must match the request ID. Claude can issue multiple tool calls per response — return a result block for each, with matching IDs.

```
messages.append({
    "role": "user",
    "content": [{
        "type": "tool_result",
        "tool_use_id": response.content[1].id,
        "content": "15:04:22",
        "is_error": False
    }]
})
```

The Conversation Loop

Complex queries require multiple tool calls in sequence. Automate the back-and-forth with a loop that checks `stop_reason`:

```
def run_conversation(messages):
    while True:
        response = chat(messages, tools=all_tool_schemas)
        add_assistant_message(messages, response)

        if response.stop_reason != "tool_use":
            break

        tool_results = run_tools(response)
        add_user_message(messages, tool_results)

    return messages
```

```
def run_tools(message):
    tool_requests = [
        block for block in message.content if block.type == "tool_use"
    ]
    tool_result_blocks = []
    for tool_request in tool_requests:
        try:
            output = run_tool(tool_request.name, tool_request.input)
            tool_result_blocks.append({
                "type": "tool_result",
                "tool_use_id": tool_request.id,
                "content": json.dumps(output),
                "is_error": False
            })
        except Exception as e:
            tool_result_blocks.append({
                "type": "tool_result",
                "tool_use_id": tool_request.id,
                "content": f"Error: {e}",
                "is_error": True
            })
    return tool_result_blocks
```

Error handling matters — when a tool fails, you still provide a result block with `is_error: True`. Claude can often recover, retry with corrected parameters, or inform the user. Swallowing errors silently breaks the conversation flow.

Adding new tools means adding a schema and a routing case. The loop doesn't change. This separation of concerns keeps the system maintainable as your tool library grows.

RAG — Making Claude an Expert on Your Data

The problem: 800-page document, Claude needs specific answers. Stuffing the whole document into the prompt has limits — context window constraints, cost, degraded performance on very long prompts.

The solution: Break the document into chunks, find the chunks most relevant to the user's question, include only those. That's RAG.

Chunking strategies — three options:

Size-based — equal-length strings. Simple, universal, but cuts mid-sentence and loses context. Adding overlap between chunks helps. Often the production default because it works with any content type.

Structure-based — split on headers, paragraphs, sections. Cleanest, most meaningful chunks. Only works when document formatting is guaranteed — well-structured Markdown splits cleanly, OCR output from scanned PDFs doesn't.

Sentence-based — splits into sentences, groups with optional overlap. Middle ground between the two. Good for most text documents.

Embeddings convert text to math. An embedding model converts text into a vector — a list of numbers capturing semantic meaning. Similar texts produce similar vectors. Find relevant chunks by comparing their vectors to the user question's vector using **cosine similarity** (range: -1 to 1, higher = more similar).

The complete RAG pipeline — six steps:

- **Chunk** source text using your chosen strategy
- **Embed** each chunk into a numerical vector
- **Store** in a vector database (Steps 1-3 are preprocessing — done once, before any query)
- **Process** the user query through the same embedding model
- **Retrieve** chunks with the most similar embeddings
- **Assemble** the prompt — user question plus relevant chunks in XML tags — and send to Claude

```
Answer the user's question about the financial document.
<user_question>{question}</user_question>
<report>{relevant_chunk}</report>
```

Beyond basic RAG: Semantic search alone misses things. Incident number "INC-2023-Q4-011" doesn't have strong semantic meaning — it's an identifier, not a concept. **BM25 lexical search** solves this with exact term matching. The production solution: **hybrid search** combining both indexes, with **reciprocal rank fusion** to normalize and merge rankings.

Reranking adds one more layer — send merged results to Claude with the original question, ask Claude to reorder by relevance. Adds latency and cost, but for applications where retrieval accuracy is critical, the quality improvement justifies it.

The engineering wisdom that applies to every system: RAG trades simplicity for capability. The prompt with the entire document stuffed in is simple and often good enough. The multi-index hybrid pipeline with reranking handles edge cases that simple

stuffing can't. Match architecture to actual requirements, not to the most complex system the technology allows.

Sub-Chapter 4C: Advanced Features

Or, Everything Else the API Can Do (And When You Should Actually Use It)

4A gives you conversations. 4B gives you external data. This sub-chapter covers everything on top — the features that turn a functional integration into a production system. Each exists because someone hit a wall the basic API couldn't solve. Understanding which wall each feature addresses matters more than memorizing implementation details.

Extended Thinking — Claude's Scratch Paper. Gives Claude time to reason before responding. Response becomes two blocks: thinking (reasoning process) and text (final answer). Trade-offs: higher cost, increased latency, more complex response handling. **Decision framework:** run without thinking first. If accuracy isn't meeting requirements after prompt optimization, enable it. Not a default — a tool for when standard prompting isn't getting you there.

```
params["thinking"] = {
    "type": "enabled",
    "budget": thinking_budget
}
```

Not compatible with message prefilling or temperature settings. Responses include a cryptographic signature preventing reasoning manipulation.

Vision — Processing Images. Include image blocks alongside text in user messages — base64 or URL. Constraints: 100 images per request, 5MB per image, 8000px max dimensions, cost = (width × height) / 750 tokens. The same prompt engineering techniques from text apply — structured analytical frameworks dramatically outperform "analyze this image." The fire risk assessment example: instead of "provide a score," break analysis into five structured steps with specific criteria per level. The difference between a toy and a tool.

PDF Support and Citations. Change type to "document," media_type to "application/pdf," encode as base64. Claude extracts text, images, charts, tables, document structure. One block, one media type change. **Citations** add verifiability — when enabled, responses include exact source text, document index, title, and page range for each claim. Enable with `"citations": {"enabled": True}`. Citations transform Claude from "it said so" to "here's exactly where it found it."

Prompt Caching — Paying Less for Repeated Work. Every request preprocesses your input from scratch — tokenization, embedding, contextual analysis. For conversations with the same 6,000-token system prompt and 1,700-token tool schema every turn, you're paying for identical work repeatedly. Caching saves the preprocessing. Follow-up requests with identical content read from cache instead of reprocessing. Cache lives one hour.

Implementation: add `"cache_control": {"type": "ephemeral"}` to blocks you want cached. Content must be at least 1024 tokens. Cache only hits if content is identical to the breakpoint — even adding "please" invalidates it. System prompts and tool schemas are the highest-value targets.

Code Execution and Files API. Files API: upload files ahead of time, reference by ID (no base64 encoding every request). Code execution: Claude writes and runs Python in an isolated Docker container with no network access. The combination: upload data via Files API → Claude writes analysis code → executes in sandbox → generates outputs you download. Multiple executions per conversation, iteratively refining analysis.

MCP — Standardized Tool Ecosystems. Shifts tool definitions and execution from your server to specialized MCP servers. Instead of writing every GitHub integration tool yourself, connect to a GitHub MCP server where someone already did that work. The `@mcp.tool` decorator auto-generates JSON schemas from Python type hints:

```
from mcp.server.fastmcp import FastMCP
mcp = FastMCP("DocumentMCP")

@mcp.tool(
    name="read_doc_contents",
    description="Read the contents of a document."
)
def read_document(
    doc_id: str = Field(description="Id of the document to read")
):
    if doc_id not in docs:
        raise ValueError(f"Doc with id {doc_id} not found")
    return docs[doc_id]
```

Ecosystem includes servers for Sentry, Playwright, Figma, Atlassian, Slack, and more. MCP's value: the difference between "I need to write 47 GitHub integration tools" and "I need to connect to the GitHub MCP server." Tool use conversation flow stays the same — MCP changes who writes and maintains the tools.

Claude Code — AI-Powered Development. Command-line tool giving Claude generic, abstract tools — bash, read, write, edit, glob, grep — and letting it figure out how to combine them. No specialized "refactor code" tool. The abstraction is the design. Two production patterns: **parallel development with Git worktrees** (separate workspaces per feature, avoiding file conflicts), and **automated debugging** (Claude monitors production errors via Sentry/CloudWatch, reads relevant code, applies fixes, pushes changes).

Computer Use — Desktop Automation as Tool Use. Same conversation loop, different tool. Include a computer tool schema specifying display dimensions. Claude responds with action requests (clicks, keystrokes, screenshots). Your system executes in a Docker container. Screenshots go back to Claude. Practical use: QA testing — describe test cases in natural language, Claude clicks through every interaction, generates pass/fail reports.

Workflows vs. Agents — The Architectural Decision

When your application needs multi-step tasks: **workflows** define steps explicitly. **Agents** give Claude tools and a goal, letting it figure out the steps.

Workflow patterns:

Chaining — sequential steps. Generate content in step one, fix constraint violations in step two. Use when Claude ignores constraints in long prompts.

Parallelization — simultaneous sub-tasks aggregated. Instead of one request evaluating all six material types, six parallel requests with focused criteria each, then a final aggregation step.

Routing — categorize incoming requests, route to specialized pipelines. Different content types, different prompts.

Evaluator-Optimizer — producer creates output, grader evaluates, rejected output cycles back with feedback. Repeat until acceptance.

Agent patterns: Give Claude tools and a goal. The power is combining simple tools in unexpected ways. Three datetime tools seem trivial individually — Claude chains them to handle "When does my 90-day warranty expire?" by recognizing it needs to ask when you purchased the item first.

Critical agent design principle: environment inspection. Claude operates blindly without observation. Every action should be followed by verification — screenshots after UI interactions, file reads before modifications, response checks after API calls.

The decision framework: Workflows provide higher accuracy, easier testing, predictable execution. Agents provide flexibility, creative tool combination, ability to handle novel situations. **Start with workflows. Add agent flexibility only where requirements genuinely can't be predetermined.** Users don't care that you built a sophisticated agent. They care that the product works consistently. This isn't philosophy — it's engineering backed by completion rate data that anyone who's deployed both approaches has seen.

Chapter 2: Claude Code — Where Theory Meets Terminal

Or, The Part Where You Stop Clicking Buttons and Start Getting Dangerous

Sub-Chapter 2A: Foundations

Every coding assistant performs the same trick: making a language model that can only process text look like it can read files, run commands, and manage deployments. The trick isn't intelligence. It's plumbing. Understanding the plumbing matters because it determines what goes wrong when things go wrong — and things will go wrong.

How it actually works: You give a task. Claude formulates a plan and requests actions — read this file, run this command, edit this line. The tool system executes the actual operations. Results flow back. Claude generates its answer. At no point does the model directly touch your filesystem. It asks. The tool system does.

Claude Code ships with generic, abstract tools — bash, read, write, edit, glob, grep — rather than specialized tools like "refactor code." The abstraction is the design: Claude figures out how to combine basic tools for complex tasks, handling scenarios the developers never explicitly planned for. This also explains the security model — Claude navigates your code through direct file operations, reading what it needs when it needs it. Less data leaves your machine. But context management — what Claude knows about your project at any moment — becomes your most important job.

Setup

Terminal-based. macOS, Windows WSL, Linux.

```
# macOS (Homebrew)
brew install --cask claude-code

# macOS, Linux, WSL
curl -fsSL https://claude.ai/install.sh | bash

# Windows CMD
curl -fsSL https://claude.ai/install.cmd -o install.cmd && install.cmd && del
install.cmd
```

Run `claude`. First run prompts authentication. Requires Node.js — verify with `npm help`. If that produces an error instead of output, install Node.js before proceeding. This is the prerequisite that documentation assumes you've handled and reality confirms you haven't.

Context Management

The single highest-leverage skill in Claude Code. Too much irrelevant context measurably decreases performance — the model gets distracted, misses signal in noise, produces worse output than if you'd given it less.

`/init` — run first in any new project. Claude scans your codebase and generates a CLAUDE.md summary: project purpose, architecture, key commands, coding patterns. Included as context in every subsequent request — a persistent system prompt for your project.

The CLAUDE.md hierarchy: Project-level (generated by /init, committed to source control, shared with team). Local CLAUDE.local.md (personal instructions, not committed). User-level ~/.claude/CLAUDE.md (universal preferences across all projects). Specific overrides general — same inheritance pattern as CSS specificity or git config.

`#` **memory command** — add instructions to CLAUDE.md mid-conversation. `# Use comments sparingly. Only comment complex code.` Claude merges it into the appropriate file. Persistent, conversational, no workflow interruption.

`@` **file mentions** — surgical context injection. `How does the auth system work? @auth` includes the file's contents directly. Reference files in CLAUDE.md too: `The schema is in @prisma/schema.prisma. Reference it for data structure questions.` Every piece of context costs tokens, adds latency, and competes for attention. The @ symbol gives you precision control.

The Three-Phase Workflow

Not "describe what you want and hope." A deliberate separation of understanding, planning, and execution.

Phase 1: Feed Context. Before building anything, point Claude at relevant files. "Read @components/UserTable.tsx and @hooks/useUsers.ts. Describe the patterns used." Claude can't infer your preferences from air — it needs examples.

Phase 2: Plan. "Based on what you've read, plan how we'd add pagination. Don't write code yet." Forces reasoning before committing. Plans are cheap to revise. Implemented code is expensive.

Phase 3: Implement. "Implement the pagination plan from above." Claude now has context, a plan, and a clear instruction.

The test-driven variant: feed context → Claude writes test cases → Claude implements tests → Claude writes code that passes the tests. Explicit success criteria instead of ambiguous vibes about what "working" means.

Making Changes — Planning Mode, Thinking Mode, Screenshots

Screenshots (Ctrl+V) — paste a broken layout directly into the terminal. Claude sees what you see. "Fix the alignment issue in this screenshot" is dramatically more effective than describing it.

Planning Mode (Shift+Tab twice) — handles breadth. Claude reads more files, creates detailed plans before executing. Use when a task touches routing, data model, and UI simultaneously.

Thinking Mode ("ultrathink") — handles depth. Extended reasoning budget for complex logic, subtle bugs, tricky algorithms. Use when Claude understands the problem space but keeps producing incorrect logic.

If Claude breaks things elsewhere: more breadth (planning mode). If Claude gets the logic wrong: more depth (thinking mode). If both, use both.

Context Control

Long sessions accumulate noise. Five techniques:

Escape — stops Claude mid-response. Interrupt wrong directions immediately.

Escape + `#` — stop, then add a memory. "Don't modify schema directly — use migrations." Prevents recurrence.

Double Escape — conversation rewind. Jump back to an earlier point, skip dead-ends.

`/compact` — summarizes conversation history while preserving learned knowledge. Context stays, noise goes.

`/clear` — nuclear option. Fresh start for unrelated tasks.

Custom Commands

Create `.md` files in `.claude/commands/`. Filename becomes the command. Use `$ARGUMENTS` for runtime parameters.

```
Write comprehensive tests for: $ARGUMENTS

Testing conventions:
- Use Vitest with React Testing Library
- Place test files in __tests__ directory
- Name as [filename].test.ts(x)
- Use @/ prefix for imports
```

Invoke: `/project:write_tests the use-auth.ts file in the hooks directory`. Conventions baked into the command, not remembered from a previous conversation. **The rule:** any prompt you've typed more than twice should be a custom command.

Hooks — Automated Guardrails

Commands that run before or after Claude executes tools. Automated feedback loops without requiring you to watch every operation.

Pre-tool hooks — block operations. A hook watching `read` and `grep` prevents Claude from accessing `.env` files. Exit code 0: proceed. Exit code 2: blocked, with stderr feedback to Claude.

Post-tool hooks — provide feedback. A TypeScript checker running after file edits catches type errors immediately, feeding them back for automatic correction. Claude edits a function signature → hook catches broken call sites → Claude fixes them.

Two that earn their cost immediately: a type-checker after file edits (catches the most common failure mode — editing a function without updating callers) and a duplicate-code prevention hook for query-heavy codebases.

SDK and GitHub Integration

The SDK — programmatic interface via CLI, TypeScript, and Python. Same tools, runs headless. Designed for pipelines, CI/CD, automated processes. Default read-only; write access requires explicit `options.allowTools` configuration. How Claude Code stops being something a developer uses and starts being something a system invokes.

GitHub integration — `/install GitHub app` adds two Actions: @Claude mention support in issues/PRs and automatic PR review on new pull requests. The PR review isn't a lint pass — it's architectural reasoning. The Terraform example: Claude traced data flow from a Lambda through an S3 bucket shared with an external partner and flagged PII exposure risk. That's not pattern matching. That's understanding how infrastructure components

connect.

Sub-Chapter 2B: Workflows and Control

Effective Workflows in Practice

Performance Optimization. The Chalk demonstration: 429 million weekly npm downloads. Claude ran benchmarks to establish baseline, identified hotspots through profiling, created a prioritized fix list, implemented changes one at a time, re-benchmarked after each. Result: 3.9x throughput improvement. The pattern generalizes: give Claude a measurable objective, tools for measurement, and access to the code. Let it iterate. Measure → analyze → change → measure again.

Data Analysis. Claude performed churn analysis on CSV data using Jupyter notebooks — executing code cells iteratively, reading output, deciding what to investigate next. Claude Code isn't restricted to code-editing tasks. Any workflow involving scripts, execution, output reading, and iteration fits the model.

Visual Feedback Loop. Connect the Playwright MCP server. Claude opens a browser, navigates to your app, generates a component, takes a screenshot, analyzes the visual result, updates the code based on what it sees, tests again. Generate → observe → evaluate → improve. The evaluator-optimizer pattern from Chapter 4C, running inside a single Claude Code session.

Git Integration

Claude handles staging, committing, and writing descriptive messages without leaving the terminal. Documentation mentions this in a bullet point. Working developers use it fifty times a week.

Commit messages reflect actual reasoning — Claude was there, did the work. Not a post-hoc summary of changes you half-remember. Add conventional commit formats or ticket number requirements to CLAUDE.md and Claude follows them automatically.

MCP Servers in Development Workflows

Playwright — the most immediately impactful integration for web development. Claude reads component code, opens a browser, identifies a styling issue, edits, refreshes,

verifies. Complete development cycle without leaving the terminal.

Sentry — Claude discovers production bugs, reads stack traces, locates relevant code, proposes fixes. Collapses the bug-to-fix feedback loop from hours to minutes.

Composition — multiple servers matching your development process. Sentry for error detection, Jira for ticket context, Slack for notifications, Playwright for browser testing, custom servers for internal APIs. Five systems, one conversation, one coherent workflow.

GitHub Actions — Claude Code in CI/CD

Automatic PR Review. Every new PR triggers architectural reasoning about what the change does, what it might break, what it doesn't account for. The cost is API calls per PR. The return: a review process that catches things human reviewers miss — not because humans are incompetent, but because the twenty-third PR review on a Friday afternoon gets less attention than the first one Monday morning.

The Composability Story

Custom commands encode conventions. CLAUDE.md ensures architectural context. Hooks enforce quality gates. MCP servers extend reach. GitHub Actions run the apparatus on every PR without human initiation. The three-phase workflow structures each interaction.

The sum: a development environment with intelligence baked in at every layer. Not a coding assistant. Infrastructure.

Sub-Chapter 2C: Extensions and Automation

Or, When One Claude Isn't Enough and You Start Building an Army

Parallelizing with Git Worktrees

The largest single productivity multiplier available. Multiple Claude instances working simultaneously — a team of virtual engineers on your project. The concurrent-write problem has a standard solution: Git worktrees. Complete copies of your project in separate directories, each on its own branch, each running its own Claude instance.

Create worktrees → launch Claude Code in each → assign different tasks → let them work in parallel → commit → merge back to main. Standard branching strategy applied not to humans working asynchronously over days, but to Claude instances working simultaneously over minutes.

Custom commands automate both ends: `/project:create_worktree feature_b` handles creation, symlinking, and editor launch. A merge command handles the other end — Claude resolves conflicts intelligently because it understands the context of changes from both branches.

Scaling limit: not compute — decomposition. You need tasks that separate cleanly. Three independent API endpoints parallelize well. A feature touching routing, data model, and UI simultaneously doesn't. The skill is in the task decomposition, not the tooling.

Automated Debugging — Claude as Night Shift Operations

Removes the human entirely for production error detection and remediation.

The pipeline: A GitHub Action runs daily. Claude queries CloudWatch for errors from the last 24 hours. Filters and deduplicates. Analyzes each error and attempts fixes. Commits code and opens a pull request — clear error description, root cause analysis, specific fix. By the time the team opens their laptops, the fix is waiting for review.

Real-world example: Production deployment uses an invalid model identifier that only breaks in production because dev uses a different API endpoint. Claude reads the CloudWatch error, recognizes the pattern, locates the configuration, applies the correct identifier, opens a PR. No human discovered it. No human diagnosed it. No human wrote the fix. A human reviews and approves. That's the appropriate level of involvement for this class of problem.

The pattern extends beyond debugging: scheduled dependency updates, automated documentation generation, codebase migration tasks — any repeatable workflow involving reading context, reasoning about problems, and producing code changes.

Computer Use — When Code Isn't Enough

Extends Claude from text into the visual world. Same tool use system, applied to desktop automation. Schema specifies display dimensions. Claude requests actions — clicks, keystrokes, screenshots. Your system executes in a Docker container. Screenshots go back to Claude for observation.

```
{
  "type": "computer_20250124",
  "name": "computer",
  "display_width_px": 1024,
  "display_height_px": 768,
  "display_number": 1
}
```

The QA testing pattern: Describe test cases in natural language. Claude navigates, clicks, types, takes screenshots, generates pass/fail reports. In the course example, two tests passed and one caught a real bug — autocomplete dropdown appearing in the wrong location on Backspace. No Selenium required.

The economics: Computer Use tests are slower and more expensive per execution (each step is an API call with image processing). But writing them costs minutes instead of hours, and modifying them is a sentence edit instead of a code refactor.

Security: Sandboxed Docker container. Completely isolated. Nothing Claude does inside affects anything outside unless you explicitly mount volumes or expose ports. Architectural isolation, not optional configuration.

The Extension Architecture

Parallelization, automated debugging, and Computer Use share a structural principle: they extend Claude Code by changing its operating model, not by adding features.

Parallelization changes concurrency (multiple instances via worktrees). Automated debugging changes initiation (GitHub Action triggers instead of human prompts). Computer Use changes interaction (screenshots and clicks instead of file reads and writes).

None require Claude to be smarter. They require the environment around Claude to be structured differently. **That's the insight this chapter builds toward:** Claude Code's power comes not from the model's intelligence alone, but from how the tooling, configuration, and infrastructure around it shapes what that intelligence can reach.

CHAPTER 5

MCP — Connecting the Dots

Or, The Protocol That Makes Claude Reach Further Than Its Own Context Window

The Problem MCP Solves

You're building a chat interface for GitHub data. User asks "What open PRs are there across all my repos?" Without MCP, you write every tool yourself — schemas, functions, error handling, authentication, pagination. GitHub alone requires dozens of tools. Add Jira, Slack, Sentry, your internal APIs — you've invented a full-time job that produces nothing except plumbing.

MCP shifts the burden. Someone else — often the service provider — has already built and maintains the tool definitions inside a dedicated MCP server. You connect. You use. You move on to the part of your application that actually differentiates it.

The common misconception: MCP is "just tool use." It's not. Tool use is how Claude calls tools. MCP is about who creates and maintains those tools. With direct integration, you author everything yourself. With MCP, it's packaged in a server you connect to. The protocol handles communication. You handle application logic.

Architecture — Client, Server, Transport

MCP Server — wraps external service functionality through a standardized interface. Exposes three capability types: tools (actions), resources (data), and prompts (instruction templates).

MCP Client — communication bridge between your application and MCP servers. Simple methods: `list_tools()`, `call_tool()`. Handles all protocol details.

Transport — communication channel between client and server. Stdio (local development), HTTP, WebSockets. Transport choice affects capabilities — Chapter 6 covers this in depth.

The complete flow for a user query involves ten steps through the chain. That's a lot of steps. Each component has a clear responsibility. The MCP client abstracts the complexity.

Building an MCP Server

The Python SDK reduces server creation to decorated functions:

```
from mcp.server.fastmcp import FastMCP
mcp = FastMCP("DocumentMCP", log_level="ERROR")

@mcp.tool(
    name="read_doc_contents",
    description="Read the contents of a document and return it as a string."
)
def read_document(
    doc_id: str = Field(description="Id of the document to read")
):
    if doc_id not in docs:
        raise ValueError(f"Doc with id {doc_id} not found")
    return docs[doc_id]
```

No manual schema writing. The decorator handles registration. Field descriptions help Claude understand arguments. The SDK transforms verbose schema exercises into standard Python function definitions. Test with the built-in inspector: `mcp dev mcp_server.py` — a browser-based dashboard for testing tools individually without full application setup.

Building an MCP Client

Two core methods:

```
async def list_tools(self) -> list[types.Tool]:
    result = await self.session().list_tools()
    return result.tools

async def call_tool(self, tool_name: str, tool_input: dict):
    return await self.session().call_tool(tool_name, tool_input)
```

In most projects, you implement either a client or a server, not both. Server to expose your service. Client to connect to existing servers.

The Three Primitives

Tools — actions Claude can perform. The tool use pattern from Chapter 4B, packaged in a server someone else maintains.

Resources — data without tool calls. Analogous to GET handlers. Expose document lists, file contents, reference data. Inject directly into prompts — faster than a tool call round-trip. Two types: direct resources (static URIs) and templated resources (parameterized URIs).

Prompts — pre-built instruction templates. Thoroughly tested workflows that users invoke instead of writing their own. Appear as slash commands in the client. You spend time crafting and evaluating prompts once; users benefit from your domain expertise without becoming prompt engineering specialists.

The mental model: Tools let Claude do things. Resources let Claude see things. Prompts tell Claude how to approach things. A well-designed server uses all three — tools for operations, resources for data, prompts for workflows that tie them together. The composition handles what none could do alone.

CHAPTER 6

MCP Advanced

Or, What Happens When You Stop Treating MCP as a Tutorial

Sampling — When the Server Needs a Brain

Standard MCP: server receives instructions and executes them. It doesn't think. Sampling changes this — allows an MCP server to request language model generation from the connected client.

Why it matters: Your server fetches Wikipedia data and needs to summarize it. Without sampling, the server needs its own API key, authentication, and token budget. For a publicly accessible server, that's random users generating unlimited text at your expense. Sampling eliminates this — each client pays for their own AI usage. The server provides functionality without incurring LLM costs.

Implementation: Server uses `ctx.session.create_message()` during tool calls. Client implements a sampling callback that calls Claude and returns the result. One detail the docs don't emphasize: MCP message formats differ from Anthropic SDK formats. You need conversion logic.

Log and Progress Notifications

When tools take time, users see nothing until completion. The operational equivalent of a loading spinner providing zero information.

Fix: Tool functions receive a `context` argument with `context.info()` for log messages and `context.report_progress()` for progress updates. Client implements callback functions for each. For anything taking more than a few seconds, the difference between "nothing happening" and "research complete, writing report..." is the difference between a user who waits and a user who kills the process.

Roots — Filesystem Access with Boundaries

The problem: User asks Claude to "convert biking.mp4 to MOV." Server can't search the entire filesystem. Roots let users grant access to specific directories. Claude calls ListRoots → ReadDirectory → ConvertVideo with full path. Permission boundary and autonomous discovery simultaneously.

Critical detail: The SDK does not automatically enforce root restrictions. You must implement access checks manually. `is_path_allowed()` called before every file operation is not optional — it's the entire security model.

Transports — How Messages Move

Stdio — default for local development. Full bidirectional communication. Client and server must run on the same machine.

StreamableHTTP — enables remote hosting. HTTP is fundamentally unidirectional, so server-to-client communication uses server-sent events (SSE). Session IDs, long-lived connections, two types of SSE channels. Clever infrastructure that works but introduces configuration complexity.

The Configuration Trap

Two flags control StreamableHTTP behavior. Both destroy capabilities when enabled.

Stateless HTTP (true): Needed for horizontal scaling behind load balancers. Cost: no sampling, no progress logging, no resource subscriptions, no server-to-client communication of any kind.

JSON Response (true): Disables streaming on POST requests. No intermediate progress, no log statements during execution.

The deployment trap: Everything works in local development with stdio. Deploy with StreamableHTTP and flags set to true. Features depending on bidirectional messaging silently stop working. No errors. No warnings. Messages have no path to the client.

The rule: Use the same transport in development as production. Discover constraints early.

Transport Selection

Stdio: Same machine, full bidirectional, development/testing.

StreamableHTTP (default flags): Remote hosting, need sampling/progress, traffic doesn't require horizontal scaling.

StreamableHTTP (stateless=true): Horizontal scaling, can live without server-initiated messages.

No configuration gives you horizontal scalability, full bidirectional communication, and remote hosting simultaneously. Pick two.

CHAPTER 7

Claude on Bedrock

Or, Running Claude in Environments Where You Control the Infrastructure

Same Claude models. Different everything else. `boto3` instead of the Anthropic SDK. Inference profiles instead of model IDs. AWS documentation, not Anthropic documentation. If you're in an enterprise running on AWS, this isn't optional.

The critical distinction: Mixing up which documentation you're reading produces errors that are technically correct and practically useless — the kind where the message tells you what went wrong without telling you why.

Making Requests

Three components: `boto3` client with correct region, model ID, structured message. **Model IDs and the Region Problem:** not all models exist in every region. Solution: inference profiles — found in the console under "cross region inference," not the model catalog. Use profile IDs for automatic routing. Response text lives at `response["output"]["message"]["content"][0]["text"]` — a path that tells you everything about how AWS thinks about nested data structures.

Multi-Turn, System Prompts, Temperature

Multi-turn: Stateless. Same pattern as Chapter 4 — maintain full history, send with every request. Messages must alternate user/assistant.

System prompts: Via `system` keyword parameter. Same principles as Chapter 3.

Temperature: 0.0 deterministic, 1.0 maximum variability. The right value is determined by testing against your use case, not by reading someone else's blog post.

Streaming and Output Control

Streaming: `converse_stream()` instead of `converse()`. For user-facing applications, not optional — it's the difference between "responsive" and "broken."

Output control: `max_tokens`, `stop_sequences`, system prompts for format. **Structured data extraction** uses tools without performing actions — the forced-tool-use pattern from Chapter 4B through the Bedrock surface.

Evaluation

Generate test dataset → run through prompt → grade (code-based for objective correctness, model-based for subjective quality) → iterate. The principle worth emphasis: every configuration decision — temperature, extended thinking, prompt structure — should be eval-driven, not assumption-driven. The number of production deployments running default parameters because nobody bothered to test alternatives is, conservatively, most of them.

Extended Thinking, Vision, Prompt Caching

Extended thinking: Reasoning before final response. Higher cost, higher latency, higher accuracy on complex tasks. Enable after prompt optimization, not before.

Vision: Up to 20 images per request. Accuracy depends on prompt engineering, not image quality. Structured analysis prompts with verification steps turn vision from novelty to production capability.

Prompt caching: Stores processing work for 5 minutes. Cache point marks where caching occurs — everything before is cached. Minimum 1024 tokens. Content must be exactly identical. Best targets: system prompts, tool schemas, stable prefix content.

Tool Use, RAG, and MCP on Bedrock

Same conceptual foundations as Chapters 4-6, different API surface. Tool use, RAG pipeline (chunking, embeddings, hybrid search, reranking), and MCP all work through Bedrock identically in concept. Implementation uses `boto3` instead of the Anthropic SDK.

Claude on Bedrock is Claude with an AWS accent — same intelligence, different dialect. Learn the dialect. Ship the deployment. Budget for the moment procurement asks why the Bedrock line item tripled. The answer is always "because it's working," which is the one answer procurement never wants to hear.

CHAPTER 8

Agent Skills

Or, Teaching Claude What You Know So It Can Do What You Do

Every time you explain your coding standards to Claude, you're repeating yourself. This is the AI equivalent of training a new employee who develops amnesia every evening — and if that sounds like a contact center with 40% annual turnover, you now understand why skills exist.

What Skills Are

A folder containing a `SKILL.md` file with YAML frontmatter (name, description) and instructions. Claude Code scans at startup, loads only names and descriptions, matches requests using semantic matching. On match, full content enters context.

```
---
name: pr-description
description: Writes pull request descriptions. Use when creating a PR,
  writing a PR, or when the user asks to summarize changes for a pull request.
---

When writing a PR description:
1. Run `git diff main...HEAD` to see all changes on this branch
2. Write a description following this format:
```

The description field is the most important element. It's what Claude uses for matching. Skill not triggering? The description doesn't overlap semantically with how you phrase your requests. Almost always the problem.

Where Skills Live

Personal (`~/.claude/skills`) — follow you across projects. **Project** (`.claude/skills` in repo) — shared via version control. **Priority:** Enterprise (highest) → Personal → Project → Plugins (lowest). Use descriptive names — "frontend-review" not "review" — to avoid collisions you'll spend forty-five minutes debugging before discovering the answer was nomenclature.

Skills vs. Everything Else

CLAUDE.md — loads into every conversation, always. Project-wide standards. **Skills** — load on demand when task matches. **Slash commands** — explicit invocation. **Hooks** — fire on events. **Subagents** — isolated execution contexts. **MCP servers** — external tools.

A production setup uses all five: CLAUDE.md for always-on standards, skills for task-specific expertise, hooks for automated operations, subagents for delegated work, MCP servers for external integrations.

Configuration

`allowed-tools` — restricts which tools Claude can use when skill is active. A codebase-onboarding skill set to `Read, Grep, Glob, Bash` prevents file modifications during exploration.

Progressive disclosure — keep SKILL.md under 500 lines. Put detailed reference material in separate files. Include instructions about when to load each. The context window contains a table of contents, not the entire document.

Scripts — execute without loading contents into context. Only output consumes tokens. Tell Claude to *run* the script, not *read* it.

Sharing

Repository — commit to `.claude/skills`, propagates via Git. **Plugins** — package and distribute through marketplaces. **Enterprise managed settings** — administrators deploy organization-wide with highest priority. `strictKnownMarketplaces` controls plugin sources.

Troubleshooting

Doesn't trigger: Description doesn't match your phrasing. Add trigger phrases reflecting how you actually word requests.

Doesn't load: Structural problem. Must be inside a named directory, filename exactly `SKILL.md`. Run `claude --debug`.

Wrong skill fires: Descriptions too similar. Make them more distinct.

Runtime failure: Missing dependencies, permission issues (`chmod +x`), path separators (forward slashes everywhere).

Start with the validator. Catches structural problems before you debug anything else.

The rule of thumb: if you find yourself explaining the same thing to Claude twice, that's a skill waiting to be written. Second rule: if your skill description doesn't contain the words people actually use when requesting the task, it will sit in your skills directory like a fire extinguisher behind a locked cabinet — technically available, practically useless.

CHAPTER 9

Agents & Workflows

Or, When You Let the Machine Drive and Hope It Read the Manual

Throughout this guide, you've been building both without naming them. Claude with tools figuring out tasks: agent. Complex task broken into predetermined steps: workflow. This chapter names the patterns and gives you the decision framework.

Workflows — Predetermined Steps for Known Problems

Evaluator-Optimizer. Producer creates output. Grader evaluates against criteria. Rejected output cycles back with feedback. Repeat until acceptance. If this sounds like every QA review you've suffered through — except the feedback loop runs in seconds and nobody sends a passive-aggressive email between iterations. The producer and grader can be separate Claude calls or the grader can be code-based.

Parallelization. Split a complex task into simultaneous sub-tasks, aggregate results. Instead of one prompt evaluating all six material types (mediocre analysis across the board), six parallel requests with focused criteria each. Advantages compound: focused attention, independent optimization, easy scaling, reduced cognitive load per request.

Chaining. Sequential subtasks where each output feeds the next input. Solves the long-prompt problem — when Claude gets a content generation prompt with 15 constraints, it reliably violates some. Not because it can't follow instructions, but because juggling creation and compliance simultaneously is hard. A fact anyone who's tried writing a creative brief while adhering to brand guidelines, legal requirements, and their manager's unspoken preferences will recognize immediately. Solution: first call generates, second call revises.

Routing. Categorize incoming requests, direct to specialized pipelines. Different content types, different optimized prompts. Each pipeline fully optimized for its category.

Agents — Tools and Goals for Unknown Problems

Give Claude tools and an objective. Claude figures out the combination. Three datetime tools seem trivial individually — Claude chains them to handle "When does my 90-day warranty expire?" by recognizing it needs to ask when you purchased the item.

Tools should be abstract. Claude Code's toolset: bash, read, write, edit, glob, grep. No "refactor code" tool. Hyper-specialized tools constrain. Abstract tools expand.

Environment inspection is the missing requirement. After every action, Claude needs observation — screenshot after clicking, file contents after editing, API response after sending. Without it, Claude can't know if actions succeeded. "How will Claude know if this worked?" If you can't answer that for every tool, you haven't built an agent — you've built a wish and a prayer with a JSON schema.

The Decision Framework

Workflows: higher accuracy, easier testing, predictable execution, reliable. Use when you can picture the steps ahead of time.

Agents: flexibility, creative tool combination, handles novel situations. Lower completion rates. Harder to test. Less predictable. Use when requirements genuinely can't be predetermined.

The recommendation: Always implement workflows where possible. Resort to agents only when truly required. Users don't care that you built a sophisticated agent. They care that the product works consistently. The distinction between "this is cool" and "this was necessary" is the entire difference between engineering and demo theater.

Now What?

Or, The Part Where You Close the Guide and Open the Terminal

You've covered everything Anthropic's course curriculum teaches — compressed from 193,500 words of instructional content into something that fits in your head and stays there.

A mental model. Claude is a stateless text-prediction system with a massive context window, tool use capabilities, and an increasingly sophisticated ecosystem. It doesn't remember. It doesn't understand. It predicts the next token based on everything you've given it. Every technique in this guide — prompt engineering, system prompts, CLAUDE.md files, skills, MCP servers, the three-phase workflow — is a different way of giving Claude better material to predict from. The model is the constant. Your inputs are the variable.

A toolkit. Nine chapters of deployable techniques. Prompt structures that work. API patterns that scale. Claude Code configurations encoding team conventions. MCP servers extending reach. Workflow architectures solving known problems reliably. Agent patterns handling unknown problems flexibly. Skills capturing expertise once and applying it automatically. None theoretical. All operational the moment you use them.

A calibration. You know where Claude is strong and where it isn't. You know extended thinking costs latency and tokens. You know prompt caching saves money on repeated content. You know the difference between a good interaction and a mediocre one is almost always input quality, not model quality. Calibrated expectations are the most valuable thing any technical guide can produce.

The Gap This Guide Doesn't Close

This guide teaches how Claude works and how to build with it. Not what to build. That's harder and matters more.

The people who get transformative value aren't the ones who master prompt engineering. They're the ones who look at their actual work — the repetitive analysis, manual data

processing, document creation, operational reporting — and recognize which parts are patterns Claude can execute. The recognition is the skill. The execution is what this guide teaches.

If you came here using Claude as a slightly fancier search engine, you now have the knowledge to use it as infrastructure. The distance between those two things is measured in problems solved.

Three Things to Do Now

First: Pick one real workflow from your actual job and build it. The PR description skill. The automated debugging pipeline. A custom command encoding your commit format. Something small, something real, something you'll use tomorrow. The compound interest starts with the first deposit.

Second: Build a CLAUDE.md for your primary project. Coding conventions, architectural decisions, testing requirements — everything you'd tell a new team member in week one. The single highest-leverage action in Claude Code.

Third: Run an eval. Pick a task Claude handles regularly. Five test cases with known-good outputs. Run. Score. Change prompt. Run again. This is how you stop guessing and start measuring.

The Author's Note, Revisited

Claude narrows the gap between what technology promises and what it delivers — in proportion to how well you understand it. A power drill in the hands of someone who doesn't know which end to hold is an expensive noisemaker. The same drill in the hands of someone who understands torque, bit selection, and material properties builds things.

You now understand the tool. Go build things.

And when the next "digital transformation" initiative lands on your desk with an 80-slide strategy deck and a timeline no one believes — you'll have something the deck doesn't: a working system, built by someone who actually read the manual.

Pete Connor builds things that work in environments where failure has consequences. He can be found wherever the gap between the strategy deck and the production floor is widest.