AKVELON
SOFTWARE ENGINEERING REDEFINED.AI

AI-ASSISTED ENGINEERING

# AI-Assisted Engineering Workflow Playbook

*Reducing hidden Claude Code costs, rework, and workflow variance at team scale.*

A practical operational guide for managing AI-assisted engineering workflows with Claude Code — built from direct engineering experience with AI-assisted development workflows.

**~$13**
avg cost / developer / active day

**40–70%**
fewer tokens in focused sessions

**50×**
output vs a cached read

**4 areas**
cost · sessions · patterns · governance

# Contents

PART 0

# 0 The Operational Reality of AI-Assisted Development

AI coding assistants can improve engineering productivity. But unstructured AI-assisted workflows tend to create operational problems over time.

This playbook focuses on the **operational side** of AI-assisted engineering. It shows how engineering teams can structure AI-assisted workflows around measurable cost visibility, session discipline, reusable workflow patterns, cache management, hooks, and team-level governance. The workflow patterns it covers are drawn from real production usage.

## What Goes Wrong Without Workflow Structure

Most inefficiencies come from how AI is used day to day: sessions that run too long, context that grows unchecked, correction loops that repeat, and tool configurations that nobody has reviewed.

| Operational problem | What it looks like in practice | Where it shows up |
|---|---|---|
| **Hidden cost accumulation** | Teams reach end-of-month bills nobody predicted. Usage looks reasonable per developer but aggregates unexpectedly. | Finance, engineering leadership |
| **Correction loops & rework** | An AI-generated change requires multiple follow-up turns to fix. Session context becomes polluted. A restart was cheaper than continuing. | Developer time, output quality |
| **Context bloat** | Configuration files grow without audits. Sessions start carrying 50,000+ tokens of rarely-used instructions into every interaction. | Per-session cost, response quality |
| **Inconsistent outputs** | Different developers get different quality results from the same tools because personal settings, model choices, and session habits vary widely. | Code review, integration friction |
| **Low operational visibility** | Engineering leaders cannot see which teams are spending what, which workflows are efficient, or whether AI usage correlates with delivery outcomes. | Planning, budgeting, ROI assessment |
| **Workflow fragmentation** | Developers reinvent context for every session. No shared skills, no reusable patterns, no institutional workflow knowledge. | Onboarding time, team consistency |
| **Reasoning overhead on routine tasks** | Teams run the highest-capability models at maximum reasoning depth for boilerplate work. | Direct API cost, throughput |

## The Workflow Discipline Gap

Most engineering teams already have established practices for managing code quality, testing, deployment, and collaboration. AI-assisted development adds another layer of operational decisions that often go unmanaged.

Questions such as *when to start a new session, how much context to provide, which model to use, what should become a reusable skill,* or *how to control tool usage* are typically left to individual developers. That works for experimentation. It becomes harder to sustain as AI usage expands across projects and teams.

## What This Playbook Covers

This playbook is built on Akvelon engineers' direct experience with Claude Code across projects of varying scales and maturities. It covers four operational areas:

| Area | Core question | Primary audience |
|---|---|---|
| **Cost structure & visibility** | What does AI-assisted development actually cost, and how do you measure it? | Engineering leaders, AI platform teams |
| **Session & workflow structure** | How should AI work sessions be structured to reduce rework and produce consistent outputs? | Developers, technical leads |
| **Reusable workflow patterns** | How do you build team-level workflow knowledge that compounds rather than resets each session? | Technical leads, platform teams |
| **Team-scale governance** | How do you maintain cost predictability, output consistency, and operational visibility as AI usage scales? | CTOs, engineering leaders, platform teams |

**A NOTE ON SCOPE & IMPLEMENTATION SPECIFICITY**

The implementation examples in this playbook are Claude Code-specific. Claude Code is an AI coding assistant built by Anthropic that operates as a CLI tool and IDE integration. The operational challenges it addresses — cost visibility, session discipline, context management, workflow consistency — appear broadly across AI-assisted engineering contexts, but the specific configurations, commands, and mechanics described here apply to Claude Code as of May 2026.

AI coding tools evolve quickly. Models, pricing, default settings, and product capabilities can change within months. Treat the specific numbers and configurations here as reference points, not permanent benchmarks.

**How to Read This Document**

If your primary goal is to understand AI usage, cost, and governance at the team level, start with the operational framing, cost structure, and team-scale workflow sections.

If your focus is implementation and day-to-day usage, go directly to the sections on session management, context structure, reusable workflow patterns, and reference configurations.

Throughout the playbook, summary callouts highlight key operational takeaways. The final checklists provide practical guidance for both individual developers and engineering leaders.

PART 1

# 1 Understanding Your Cost Structure

> Most teams that encounter unexpected AI infrastructure costs share a common history: they did not establish a cost baseline before usage scaled.

## 1.1 Establish Your Baseline

The numbers in this section come from Anthropic's enterprise deployment data and represent typical Claude Code usage patterns. Use them for calibration. If your team's numbers are significantly higher, this playbook identifies where to look first.

*Actual costs depend on task mix, repository complexity, model selection habits, and session discipline.*

| Metric | Typical value | Planning implication |
|---|---|---|
| Average cost per developer per active day | **~$13** | Baseline for team budgeting |
| 90th percentile ceiling per active day | **$30** | Flag for investigation if sustained |
| Typical monthly cost per developer | **$150–$250** | Reasonable planning range |
| Background idle usage per session | **Under $0.04** | Close stale sessions; marginal but compounds |
| API users under $12 / active day | **90%** | Sustained outliers warrant workflow review |

> **NOTE**
>
> The Max plan (20× usage multiplier) breaks even against raw API billing at roughly **70 million tokens per month** of typical Sonnet-heavy usage.

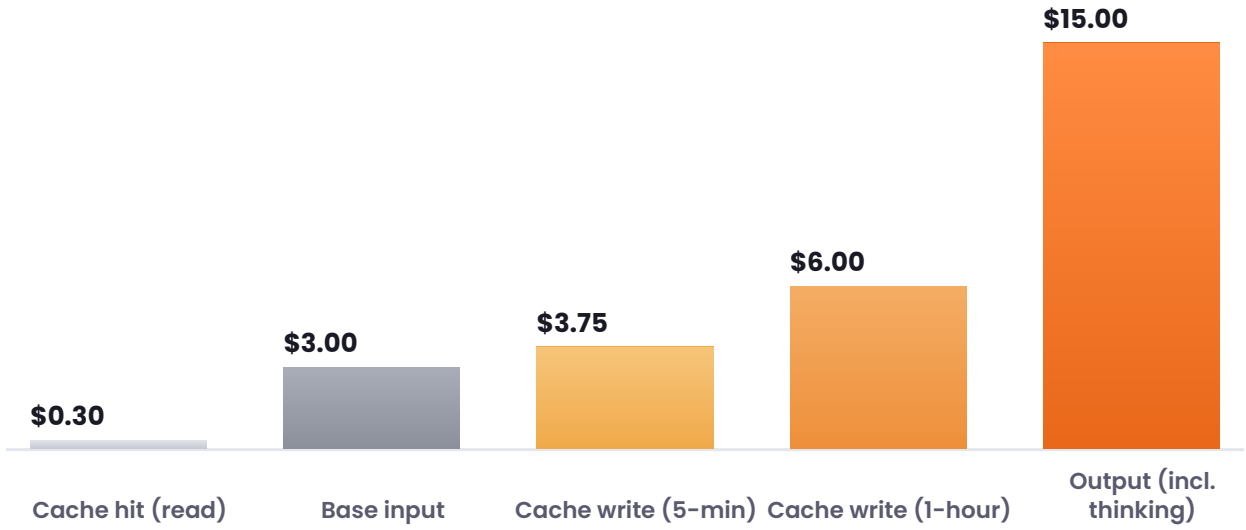
**☆ WHAT ACTUALLY DRIVES COST**

The primary cost drivers in Claude Code sessions are **output token length**, **reasoning depth**, and **cache hit rate** — not how much context you provide as input. A cached input token costs approximately **10%** of a normal input token. An output or thinking token costs approximately **5×** more than a base input token.

A session that generates long outputs with deep reasoning on a cold cache is far more expensive than a focused session on the same task. Session habits and cache discipline, not model selection alone, account for the largest share of cost variance.

**Sonnet 4.6 token costs: output is 50× more than a cache hit**

$ per million tokens




*Output tokens cost 50× more than a cached read. That single ratio drives every optimization in this guide.*

## 1.2 Instrument Your Environment

Without measurement, optimization is guesswork. Teams tend to focus on model selection while overlooking the higher costs associated with session structure or context overhead. The commands and tools below are the minimum a team should have in place before changing anything else.

**✓ FOR TEAM-LEVEL VISIBILITY, DO THIS**

Run `/cost` and `/context` at the start of every session. Check the Anthropic Console weekly. You cannot optimize what you do not measure.

| Command / tool | What it shows | Where to find it |
|---|---|---|
| `/cost` | Current session spend (local estimate from token counts) | Type in Claude Code |
| `/context` | Breakdown of context window usage: system prompt, tools, memory, skills, history | Type in Claude Code |
| `/usage` | Session stats: total cost, API duration, wall time, lines changed | Type in Claude Code |
| `ccusage` | Daily and weekly spend breakdowns with trend graphs | `npm install -g ccusage` |
| Anthropic Console | Authoritative billing, per-user cost breakdown, workspace spend limits | platform.claude.com/usage |

> **NOTE**
>
> For Teams and Enterprise on Bedrock or Vertex, Claude Code does not send metrics to Anthropic. Use LiteLLM to track spend by API key in those setups.

## How to Set Up ccusage for Ongoing Tracking

**ccusage** is a third-party CLI that reads your Claude Code session logs and produces daily and weekly spend reports with trend graphs. Run it once a week.

terminal

```
# 1. Install once
npm install -g ccusage

# 2. Check today's spend
ccusage today

# 3. Weekly trend with cost by model
ccusage week --by-model

# 4. Monthly report exported to file
ccusage month --output report.txt
```

**FOR TEAMS ON BEDROCK OR VERTEX**

Claude Code does not send metrics to Anthropic in Bedrock and Vertex deployments. Use LiteLLM to track spend by API key in those setups. Set up this instrumentation before expanding team usage — retroactive cost attribution is difficult without it.

**LEADERSHIP SUMMARY — PART 1**

Typical Claude Code usage runs **$150–$250 per developer per month**. The gap between teams with and without workflow discipline can be substantial, but the exact range depends heavily on task mix and usage patterns. Establish measurement tooling (ccusage, Anthropic Console) before attempting any optimization. Per-user cost visibility is available in the Console and is the starting point for identifying outliers.

PART 2

# 2 Structuring AI Work Sessions

> The single largest source of preventable waste in AI-assisted engineering workflows is unstructured sessions: interactions that start without a clear scope, carry stale context forward, apply maximum reasoning depth to routine tasks, use the wrong model for the job, and require multiple rounds of correction to produce a usable output.

Each of the four concerns in this part — model selection, reasoning depth, session habits, and context management — represents a separate optimization lever. They compound. A team that has addressed all four is operating a materially different workflow than one that has addressed none, even with identical tooling.

**Session Anatomy: Structured vs. Unstructured**

**✗ Unstructured session**

- Session carries previous task context forward
- Developer asks an underspecified question
- Claude asks clarifying questions (turn 2)
- First response misses intent
- Developer corrects (turn 3)
- Context grows with each turn
- Mid-session model switch invalidates cache
- Wrong implementation path pursued
- Esc pressed after 20 minutes
- Restart with polluted history fragment

**Total: 12 turns, high token count, one usable output**

**✓ Structured session**

- `/clear` at session start
- `/context` verified
- Model matched to task
- Plan mode activated for multi-file work
- Specific prompt submitted
- First response usable
- One correction turn
- `/compact` with focus hint at task boundary
- Next task started in fresh session

**Total: 4 turns, lower token count, two usable outputs**

## 2.1 Match Model Capability to Task Complexity

One of the most common sources of unnecessary AI costs in engineering teams is using models more capable than the task requires. The strongest model can cost several times more than a mid-tier one. For routine tasks like bug fixes, test scaffolding, or renaming variables, that extra cost adds up quickly without improving the results.

The framework below focuses on using each model where it delivers the most value — reserving deep reasoning for complex tasks and using cheaper models for routine execution.

> ✓ **DO THIS**
>
> Start with the cheapest model that can do the job. Step up only if quality is insufficient.

| Task type | Examples | Recommended model | Cost (input/output per MTok) |
|---|---|---|---|
| Trivial / mechanical | Renames, getters, simple regex, boilerplate | **Haiku 4.5** | ~$0.80 / $4 |
| Everyday coding | Bug fixes, feature work, test writing, PR reviews | **Sonnet 4.6** | $3 / $15 |
| Complex reasoning | Architectural decisions, multi-file refactors, perf work | **Opus 4.7 at xhigh** | $5 / $25 |
| Large-context reading | Big codebase search, long document review | **sonnet[1m]** | $3 / $15 (1M window) |
| Plan + execute sessions | Multi-file work with planning phase | **opusplan alias** | ~40% less than full Opus |

To switch: type `/model sonnet` or `/model haiku`. Switch at session boundaries only.

> ⚠ **WATCH OUT — MODEL SWITCHING HAS A HIDDEN COST**
>
> Caches are per-model. Switching models mid-session forces a cold-start cache write for all prior context, which often costs more than staying with the original model. If you need to switch, do it at a session boundary, not in the middle of a task.

## How opusplan Works

**opusplan** is a built-in model alias that uses Opus in plan mode for architectural decisions, then automatically switches to Sonnet when execution begins. For sessions that would otherwise run entirely on Opus, opusplan reduces cost by roughly **40%** with no change in planning quality.

**Usage:** `/model opusplan` — or set as the project default in `.claude/settings.json`: `{"model": "opusplan"}`

**Best for:** multi-file refactors, migrations, new feature scaffolding — any session where planning matters but execution is mostly mechanical.

## Typical opusplan Session

1. Start Claude Code in the project root: `claude`
2. Switch to opusplan: `/model opusplan`
3. Enter plan mode: **Shift+Tab twice** (first press → auto-accept; second press → plan mode, shown in status bar)
4. Describe the work: *"Refactor the auth flow to use OAuth2 with PKCE."* Claude reasons with Opus and returns a written plan.
5. Review and approve the plan. Edit if needed. Exit plan mode (Shift+Tab) and tell Claude to proceed.
6. Execution runs on Sonnet automatically — visible in the status line. No manual switching required.

## Three Ways to Set opusplan

.claude/settings.json

```
# Option 1: switch for the current session only
/model opusplan

# Option 2: set as the project default (shared with team)
# Edit .claude/settings.json
{ "model": "opusplan" }

# Option 3: set as your personal default (gitignored)
```

```
# Edit .claude/settings.local.json
{ "model": "opusplan" }
```

> **NOTE**
>
> opusplan uses the standard **200K** context window during the Opus planning phase. The automatic 1M upgrade does not apply.

### sonnet[1m]: 1M Context Window at Sonnet Pricing

**sonnet[1m]** gives you a 1 million token context window at Sonnet pricing ($3/$15 per MTok) instead of Opus pricing ($5/$25). Use it when the task is primarily about reading large context — a large codebase, long documents — rather than complex multi-step reasoning.

| Option | Input / MTok | Output / MTok | Context | Best for |
|---|---|---|---|---|
| opus | $5.00 | $25.00 | 1M | Hard multi-step reasoning, long agentic loops |
| sonnet[1m] | $3.00 | $15.00 | 1M | Reading large context, code search, document review |
| Difference | 40% less | 40% less | Identical | Quality of reasoning |

```
.claude/settings.json

# Switch for the current session
/model sonnet[1m]

# Or set as project default (.claude/settings.json)
{ "model": "sonnet[1m]" }
```

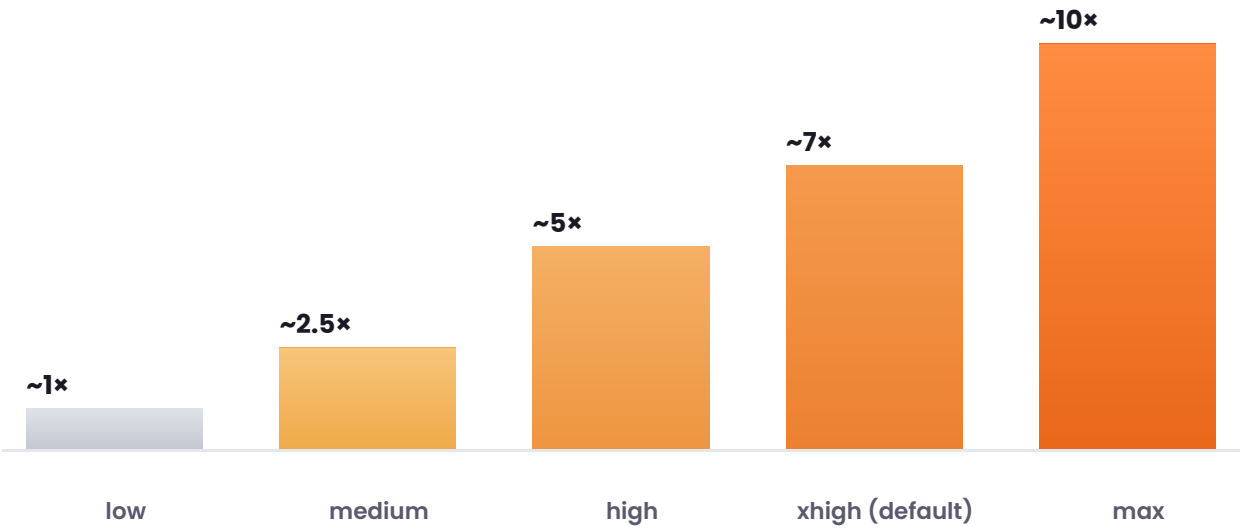
## 2.2 Control Reasoning Depth and Thinking Overhead

Reasoning models are valuable for complex problems, but often unnecessary for routine tasks. Increasing reasoning depth from low to maximum can raise thinking-token usage by around **10×**. When teams keep the highest setting enabled by default, they pay that extra cost on every interaction, including simple ones.

The solution is simple: use a lower reasoning level by default and increase it only when the task genuinely requires deeper analysis.

**Opus 4.7 effort levels: xhigh on trivial work wastes tokens**

Relative thinking tokens (low = 1×)




*Going from low to max multiplies the number of thinking tokens by 10×. xhigh on trivial tasks is where teams overspend most.*

✓ **DO THIS**

Use medium or low effort for boilerplate, formatting, and routine bug fixes. Reserve xhigh and max for multi-step reasoning and agentic sessions.

| Effort | Use it for |
|---|---|
| **low** | Classification, extraction, formatting, short summaries, one-line edits |
| **medium** | Routine bug fixes, simple feature work, well-specified tests |
| **high** | Multi-file changes, debugging with several suspects, code review |
| **xhigh (default)** | Agentic coding, architectural decisions, multi-step reasoning |
| **max** | Hardest algorithmic problems; test before committing; can show diminishing returns |

## Three Ways to Lower Thinking Costs

- Switch effort mid-session: `/effort low` or `/effort medium`

- Cap the thinking budget: set `MAX_THINKING_TOKENS=8000` in your environment
- Disable thinking as your default: set `alwaysThinkingEnabled: false` in `settings.local.json`

### ultrathink: High Effort for a Single Turn

If your session default is medium effort, you save tokens on every routine turn. When you hit a hard problem, add `ultrathink` anywhere in your prompt to trigger high effort for that one turn only — without changing the session default.

1. Confirm your session default: run `/effort`. If it is above medium, lower it: `/effort medium`.
2. Work normally at the lower level.
3. Add `ultrathink` to any prompt that needs deep reasoning. Claude detects it and runs that turn at high effort.
4. Check `/usage` afterward to verify the thinking tokens spent.

### Example Prompts Using ultrathink

```prompt
# ultrathink can appear anywhere in the prompt
ultrathink: why is this race condition only reproducible under load?

# Or inline
review the auth module and ultrathink the edge cases
```

> **OPUS 4.7 SPECIFICS**
>
> - Adaptive thinking is always active — you cannot set a fixed budget. Steer with effort levels and prompt phrasing.
> - xhigh is the default for every plan tier — appropriate for hard work, wasteful for boilerplate.
> - The tokenizer uses up to **35% more tokens** for the same text compared to previous models.
> - Later turns cost more. Opus 4.7 thinks more at higher effort on longer sessions. Compact or restart at task boundaries.

## 2.3 Session Discipline: Where Most Waste Originates

Most AI workflow waste doesn't come from a single session — it comes from carrying unnecessary context between sessions. Developers often start new tasks in old conversations, keep irrelevant context around, or let chat history grow unchecked. Over time, that extra context increases both cost and overhead.

The practices below are ranked by impact. The first three — startup commands, focusing on one task per session, and using `/compact` and `/clear` correctly — deliver the biggest reductions in unnecessary overhead.

### 8 Habits to Build Into Every Session (the first three are the highest-impact)

#### 1. Start Every Session with Three Commands

✓ **DO THIS**

Run `/cost`, `/context`, and `/clear` at the start of each session. `/cost` shows last session's spend. `/context` shows exactly what is eating your window. `/clear` if you are switching domains.

#### 2. Run One Task Per Session

Focused sessions use **40–70% fewer tokens** than sprawling ones. When the task is done, close the session and start fresh for the next thing.

#### 3. Use /compact and /clear Correctly

`/compact` summarizes history and replaces it with a compressed version. Checkpoint after a discrete sub-task, not at 90% capacity.

⚠ **WATCH OUT**

`/compact` sends the full conversation under a different system prompt, which breaks the cache. You pay full uncached input rates for the entire history. Restart when work is done; compact only to keep going on the same thread.

Give `/compact` a focus hint so it keeps what matters:

```
prompt

/compact Focus on code samples and API usage
```

Or set permanent compaction rules in CLAUDE.md:

```
CLAUDE.md

When compacting, always preserve: test file paths, API endpoint names, and error
messages. Drop chat banter.
```

### 4. Rename Before You Clear

Run `/rename my-task` before `/clear` so you can `/resume my-task` later if needed.

### 5. Use /btw for Quick Side Questions

Every question in the main session enters conversation history and is re-read on every subsequent message. `/btw` opens a side question that uses your full session context but the response never enters history.

1. Type `/btw` followed by your question. Works mid-session.
2. Read the answer in the overlay — it appears in a dismissible side panel.
3. Dismiss and continue. The main session is unaffected. The same question asked normally would add **500–2,000 tokens** to every future turn.

```
prompt

# Quick lookups during a long session
/btw what was the name of that config file again?
/btw what does the --noEmit flag actually do?
/btw how is settings.local.json different from settings.json?
```

**Limits of /btw:** Full context access (Claude sees everything in the current conversation). No tool access (cannot read new files or run commands — use a subagent for that). Single response only (no follow-up turns within /btw).

### 6. Be Specific in Your Prompts

Vague prompts generate clarification rounds that compound across session history.

```
prompt

# Vague: Claude asks follow-up questions
"make this better"

# Specific: Claude acts immediately
"Add input validation to login() in auth.ts:
  reject empty strings, trim whitespace,
  return a 400 with a descriptive error message."
```

### 7. Use Plan Mode Before Multi-File Work

Press `Shift` + `Tab` twice to enter plan mode. Claude reads the codebase and proposes an approach before touching any files. Five minutes of planning prevents 30 minutes of wrong implementation.

### 8. Cut Off Wrong Paths Early

Press `Esc` to stop Claude immediately. Press `Esc` twice to open the rewind menu: restore the conversation and code to a previous checkpoint, optionally keeping the conversation history while reverting only the file changes. Or type `/rewind`.

> ⚠ **WATCH OUT**
>
> Fast mode costs $30/$150 per MTok (6× standard Opus pricing). It persists across sessions by default. Enable `fastModePerSessionOptIn: true` in settings.local.json to force a reset each session.

## 2.4 Managing Context and Configuration

Context management is one of the areas where team practices matter most. Without clear rules, configuration files grow, memory accumulates, and more context gets carried into every session than is actually needed. Over time, that creates unnecessary cost and overhead across the team.

The framework below separates context into four categories: what should always be loaded (team conventions), what should be loaded only when relevant (language- or project-specific rules), what should be loaded only when explicitly needed (workflows and procedures), and what should never be included at all (generated files, lock files, credentials, and other low-value context).

| File | When it loads | Cost profile | Rule of thumb |
|---|---|---|---|
| `CLAUDE.md` | Every session, every turn | High: baseline tax on every message | Under 200 lines, facts only |
| Auto memory | Every session, every turn | Medium, grows quietly over time | Audit with `/memory` regularly |
| Skill descriptions | Session start only | Low: names only | Write matchable descriptions |
| SKILL.md body | On invoke only | Medium per use | Under 5,000 tokens, imperative |
| `.claude/rules/` files | When matching files are open | Proportional to task | Language-specific rules |
| `settings.local.json` | Read once at session start* | Personal cost overrides only | Model, effort, thinking |

** model and outputStyle apply on next restart. All other keys including effortLevel reload live.*

### THE TRIGGER RULE FOR DECIDING WHAT GOES WHERE

| | | |
|---|---|---|
| **Claude gets a convention wrong twice** | → | add it to CLAUDE.md |
| **You keep typing the same prompt to start a task** | → | save it as a skill |
| **A rule applies to one file type only** | → | put it in .claude/rules/ |
| **You want personal overrides teammates skip** | → | use settings.local.json |

## CLAUDE.md: What to Keep, What to Move

**Keep in CLAUDE.md**

- Package manager: 'use pnpm not npm'
- Test command: 'always run pytest before committing'

**Move to a Skill or Rules File**

- PR review checklists
- Migration procedures
- Deployment steps

- File naming conventions and linting rules
- Anything that applies to 80%+ of sessions

- API endpoint reference material

## Auto Memory: The Hidden Growing Cost

Auto memory is notes Claude writes based on your corrections. It loads every session alongside CLAUDE.md and grows quietly over time.

✓ **DO THIS**

Run `/memory` once a month. Read what is there. Remove anything outdated or redundant with CLAUDE.md.

## .claude/rules/: Scoped Rules for File Types

Rules files load conditionally based on which files are open. A `.claude/rules/typescript.md` only enters context when Claude touches TypeScript files — it costs nothing during a Python debugging session.

✓ **BEST PRACTICE**

Reserve CLAUDE.md for conventions that apply to every session. Language-specific conventions belong in .claude/rules/.

## Setting Up .claude/rules/

1. `mkdir -p .claude/rules`
2. Add one file per language: `typescript.md`, `python.md`, `sql.md`
3. Write imperative instructions, no preamble: *"Use strict null checks. Prefer interfaces over types. Never use `any`."*
4. Commit to git. Rules are project-level; every team member gets them.

> **GOVERNANCE NOTE: PERMISSIONS.DENY**
>
> There is no `.claudeignore` in Claude Code. The correct mechanism is `permissions.deny` in `.claude/settings.json`. Every file Claude reads adds tokens to context for the rest of the session — and without deny rules, Claude can read credentials, environment files, and sensitive configuration. Committing a baseline permissions.deny block to project repositories ensures this protection is in place for every developer from day one.

`.claude/settings.json`

```
// .claude/settings.json
{
  "permissions": {
    "deny": [
      "Read(node_modules/**)", "Read(dist/**)",
      "Read(.next/**)",        "Read(coverage/**)",
      "Read(.env*)",           "Read(*.lock)"
    ]
  }
}
```

> **NOTE**
>
> permissions.deny restricts the Read tool. For full protection including Grep and Glob, store secrets outside the project directory entirely.

## settings.local.json: Personal Cost Controls

`settings.local.json` sits in `.claude/` alongside `settings.json` and is gitignored automatically. It is where personal cost overrides go — the ones teammates should not share.

1. Navigate to your project root: `cd /path/to/your/project`
2. Ensure Claude Code has run at least once (it creates `.claude/` automatically)
3. Create the file: `touch .claude/settings.local.json`
4. Add your personal overrides:

`.claude/settings.local.json`

```
{
  "model": "claude-sonnet-4-6",
  "effortLevel": "medium",
  "alwaysThinkingEnabled": false,
```

```
  "fastModePerSessionOptIn": true
}
```

**What Each Setting Does**

- **model:** Override the team default without affecting teammates.
- **effortLevel: medium** saves tokens on routine tasks. Pair with `ultrathink` in prompts for turns that need deeper reasoning.
- **alwaysThinkingEnabled: false** disables extended thinking by default. Enable selectively with `ultrathink` or `/effort`.
- **fastModePerSessionOptIn: true** forces fast mode to reset each session. Prevents silently running at 6× Opus cost.

> **LEADERSHIP SUMMARY — PART 2**
>
> Model selection, reasoning depth, session discipline, and context management are the primary sources of both cost variance and output quality variance across teams. None requires infrastructure changes — all are configurable through settings files that can be distributed and enforced at the team level. Part 4 describes how to do that.

PART 3

# 3 Building Reusable Workflow Patterns

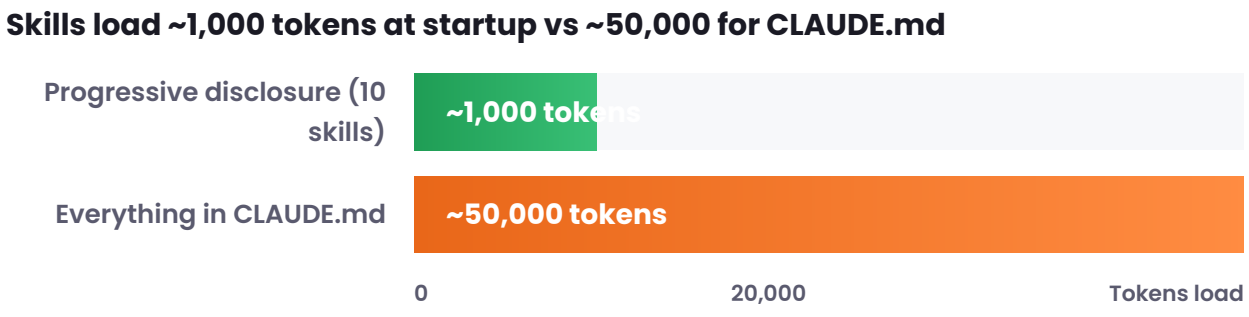
> Individual workflow discipline produces per-developer efficiency gains. Reusable workflow patterns produce team-level ones, and the gains compound. A skill built once is available to every developer on the project. A hook that catches formatting errors automatically eliminates that correction loop for the whole team. The investment is per-team, not per-developer.

This is where teams start building shared workflow knowledge into the tooling itself — not as documentation that developers have to find and follow, but as context that loads when it is relevant and costs nothing when it is not.

## 3.1 Progressive Context: Skills Over Static Configuration

The default approach to AI context configuration — put everything the model might need into a single file that loads at every session start — works at a small scale and breaks at a larger scale. A **200-line CLAUDE.md** is manageable. A **500-line** one that has grown through accumulation over months, on a 15-person team, represents thousands of tokens of overhead per session per developer, much of it for workflows that are rarely used.

Skills solve this through **progressive disclosure**: a short description token (around 100 tokens) loads at startup and tells Claude what the skill does. The full skill body, which might be 2,000–5,000 tokens, loads only when the task matches the description. Reference files linked from the skill body load only when the skill explicitly asks for them.




*10 skills with progressive disclosure cost ~1,000 tokens at startup vs ~50,000 if the same content lived in CLAUDE.md.*

✓ **DO THIS**

Move any section of CLAUDE.md that is not needed every session into a skill under **.claude/skills/**. Each skill is a directory with one SKILL.md file.

## How to Create a Skill

1. Identify a workflow you keep repeating — PR review, test writing, deployment, migration. Anything you have explained more than twice.
2. Create the skill directory: `mkdir -p .claude/skills/write-tests`
3. Write `SKILL.md` with YAML frontmatter. Two keys: `name` and `description`. The description is what Claude uses to decide when to load the skill, so be specific.
4. Write imperative instructions in the body. No narrative. Tell Claude what to do, in steps.
5. (Optional) Add bundled scripts in `scripts/`. When Claude runs a script, only the output enters context, not the script code.
6. Commit to git. Skills are project-level by default. Personal skills go in `~/.claude/skills/` instead.

## A Well-Structured SKILL.md

SKILL.md

```
---
name: write-tests
description: Use when writing or expanding tests. Covers Vitest and Playwright.
---
# Test writing conventions
- Use Vitest for unit and integration tests
- Use Playwright for end-to-end tests
- Do not mock the database
- Run `pnpm test` and confirm all pass before finishing
```

☆ **KEY INSIGHT**

When Claude runs a bundled script, the script's code never enters context. Only the output consumes tokens. Validation, parsing, transforms, and lookups all belong in scripts inside the skill directory.

Once a skill loads, its content stays in context across turns. Auto-compaction caps each re-attached skill at **5,000 tokens** with a shared **25,000-token budget**. Keep individual skills under that threshold and commit them to the project repository. A skill available to one developer costs the same to build as one available to the whole team.

### Where Skills Live

```tree
your-repo/
  .claude/
    skills/
      write-tests/
        SKILL.md
        scripts/
          run_vitest.sh   # runs as code; only output enters context

# Personal skills across all your projects:
~/.claude/skills/<skill-name>/
```

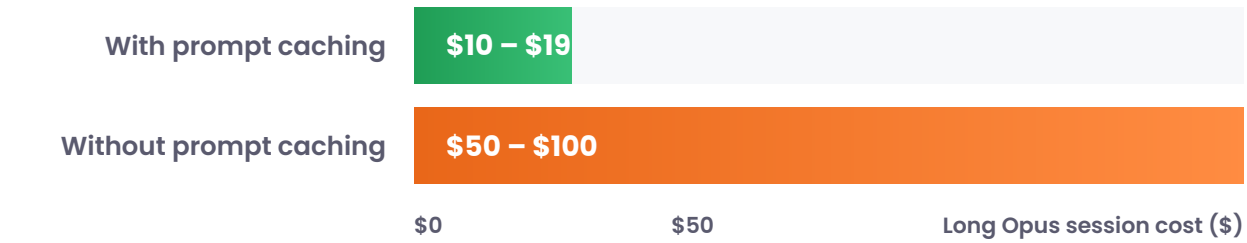
## 3.2 Cache Discipline: The Economics of Consistency

Prompt caching is the mechanism that makes extended AI coding sessions economically viable. A cached input token costs approximately **10%** of a normal input token. In a long session with a large system prompt, every turn either hits the cache, paying 10%, or misses it and pays full price for the entire prior context.

> ☆ **KEY INSIGHT**
>
> A cache hit costs ~10% of a normal input token. Break the cache and you pay full price for every token in your history, every turn.

Cache discipline is worth treating as a team practice, not just a personal one. The most common cache-breaking patterns (model switching mid-session, toggling MCP servers during a session, timestamps in the system prompt) can be prevented by configuration and habit.

Minimum prefix length for caching: **1,024 tokens** for Sonnet; **4,096** for Opus and Haiku 4.5. Below those thresholds, caching does nothing.




*With caching, a long Opus session costs $10–19 instead of $50–100.*

## Five Rules to Protect Your Cache

⚠ **RULE 1: NEVER SWITCH MODELS MID-SESSION**

Caches are per-model. Switching Opus to Haiku cold-starts a new cache. The cold-start write often costs more than staying with the original model.

⚠ **RULE 2: NEVER TOGGLE MCP SERVERS MID-SESSION**

Tool definitions are part of the cached prefix. Toggle one MCP server, and everything after it is invalidated. Pick your MCP setup at the start of the session and leave it alone.

⚠ **RULE 3: KEEP DYNAMIC CONTENT OUT OF THE SYSTEM PROMPT**

Timestamps, current dates, git status output — anything that changes between requests must go in messages, not in the system prompt. This is why Claude Code uses system-reminder tags inside messages.

⚠ **RULE 4: NORMALIZE WHITESPACE IN ANY WRAPPERS OR HOOKS**

A trailing newline that sometimes gets stripped invalidates the byte-for-byte match. If anything in your stack modifies Claude Code inputs programmatically, normalize aggressively.

⚠ **RULE 5: USE /CLEAR BETWEEN UNRELATED TASKS**

A fresh session with cache hits from a stable system prompt is cheaper than a long polluted session that recomputes everything from an ever-changing history.

✓ **BEST PRACTICE**

Claude Code handles caching mechanics automatically. Your job is to not break it. Do not toggle MCP servers. Do not switch models. Do not let the system prompt change between turns.

| Token type | Multiplier vs base input | $/MTok (Sonnet 4.6) |
|---|---|---|
| Base input | 1.0× | $3.00 |
| Cache write (5-min) | 1.25× | $3.75 |
| Cache write (1-hour) | 2.0× | $6.00 |
| Cache hit (read) | 0.1× | $0.30 |
| Output (incl. thinking) | 5.0× | $15.00 |

## 3.3 Tool Overhead and MCP Governance

MCP servers extend Claude Code with external tool capabilities — GitHub, databases, cloud services, and other integrations. Each connected server loads its tool definitions into context on every message turn. This is one of the more underappreciated ongoing cost drivers in configured Claude Code setups.

For individual developers, an unaudited MCP configuration is a moderate concern. For teams with shared or standardized MCP configurations, it becomes a budget item. A configuration with four servers averaging **20 tools each** adds **10,000–18,000 tokens** of overhead on every turn. That number grows each time someone adds a server without checking whether it is actually used.

| MCP setup | Approximate tokens per turn |
|---|---|
| 1 server, 5 tools | 500 – 2,500 tokens |
| 1 server, 20 tools | Up to 8,000 tokens |

| MCP setup | Approximate tokens per turn |
| --- | --- |
| 4 servers, 20 tools each | 10,000 – 18,000 tokens |

## Tool Search: The Default Mitigation

Tool Search is enabled by default in recent versions. MCP tool definitions are deferred rather than loaded upfront. Claude uses a search tool to discover relevant tools only when a task needs them.

### CONFIGURING ENABLE_TOOL_SEARCH

1. Choose your loading strategy: `auto` loads schemas when they fit within 10% of context; `auto:5` at 5%; `true` always defers; `false` always loads.
2. Set the variable in `.claude/settings.json` under the env block, or export in your shell profile.
3. Verify with `/context` after restarting — the tool-definition section should show "deferred."

.claude/settings.json

```
{ "env": { "ENABLE_TOOL_SEARCH": "auto" } }
```

## Four More MCP Cost Levers

- **Prefer CLI tools over MCP.** Tools like `gh`, `aws`, `gcloud`, and `sentry-cli` add zero per-tool overhead — Claude runs them directly without loading a schema.
- **Allowlist only the tools you need.** If your GitHub MCP server has 40 tools but you use 3, exclude the other 37 using `allowed_tools`.
- **Watch MCP output size.** Claude Code warns when output exceeds 10,000 tokens. Don't raise `MAX_MCP_OUTPUT_TOKENS` without a specific reason.
- **Disable servers you are not actively using.** Run `/mcp` to see all configured servers. A connected-but-unused server still contributes tool names on every turn.

## Allowlist Example: Restrict a 40-Tool GitHub Server to Three

.claude/settings.json

```
"tool_configuration": {
  "enabled": true,
```

```
  "allowed_tools": ["create_pr", "list_issues", "get_file"]
}
```

> ⚠ **WATCH OUT — AGENT TEAMS MULTIPLY MCP OVERHEAD**
>
> In agent teams, each teammate loads all connected MCP servers independently. A 3-agent team with a heavy MCP configuration pays the overhead three times per turn. Allowlist aggressively in agentic setups.

> ⓘ **GOVERNANCE NOTE**
>
> MCP configuration should be a team decision, not an individual one. Left unmanaged, servers accumulate and tool definitions bloat every turn's context — often without anyone noticing until the bill arrives. Reviewing MCP server configurations as part of project setup and periodic workflow audits is low-effort with consistent returns.

## 3.4 Automation with Hooks

Hooks are shell commands that run automatically at lifecycle events in a Claude Code session: before a tool executes, after a file write, and when a session ends. Anything a script can do reliably does not need to be done with tokens.

The main value of hooks is catching errors before they become rework. A formatting error caught immediately after a file write costs a hook invocation. The same error caught three turns later, after Claude has built further on top of it, costs multiple correction turns and potentially a rewind.

> ✓ **DO THIS**
>
> Add a PostToolUse hook to run your formatter and type-checker after every file write. Errors caught in seconds do not become 10-turn rework loops.

### How to Add a PostToolUse Hook

1. Open or create `.claude/settings.json` in your project root.
2. Add a hooks block. The `matcher` field is a regex over tool names.
3. Save the file. Claude Code reloads hook config live; no restart needed.
4. Verify with `Ctrl` + `O` — toggle verbose mode to see hook stdout/stderr in the terminal.
5. Trigger a test edit. The hook should run prettier and tsc after the edit completes.

.claude/settings.json

```
// .claude/settings.json
{
  "hooks": {
    "PostToolUse": [{
      "matcher": "Edit|Write|MultiEdit",
      "hooks": [
        { "type": "command", "command": "prettier --write $CLAUDE_FILE_PATH" },
        { "type": "command", "command": "tsc --noEmit" }
      ]
    }]
  }
}
```

## Hook Preprocessing: Filter Verbose Output

A PreToolUse hook can intercept a command before Claude sees the output. For test runs, filter to failures only — 10,000 lines of test output becomes 100 lines that fit in context.

hook.sh

```
# PreToolUse hook: filter test output to failures only
if [[ "$cmd" =~ ^(npm test|pytest|go test) ]]; then
  filtered="$cmd 2>&1 | grep -A 5 -E '(FAIL|ERROR)' | head -100"
fi
```

## Other High-Value Hooks

- PreToolUse hook blocking writes to protected paths
- Stop hook running a final lint pass before handing back to you
- PostToolUse on Bash to log commands for debugging long agentic runs

**LEADERSHIP SUMMARY — PART 3**

Skills reduce per-session startup overhead and make workflow knowledge shareable. Cache discipline ensures that investment pays off on every turn. MCP governance prevents new servers from silently undoing it. Hooks remove correction loops before they start. Together, they are what separate a team of developers each using AI independently from a team running AI-assisted workflows with any consistency.

PART 4

# 4 Operating at Team Scale

> Parts 2 and 3 describe what individual and project-level workflow discipline looks like. Part 4 covers what happens when you need to operate this across a team.

Three concerns dominate at the team scale: **cost predictability** — can you forecast AI infrastructure spend before the bill arrives? **Output consistency** — are developers getting comparable results, or does quality vary by who is asking? And **operational visibility** — can engineering leadership see what is happening, identify outliers, and act on it?

None of this requires new infrastructure. It requires consistent configuration and the willingness to treat AI workflow decisions as team standards rather than personal preferences.

## 4.1 Agent Workflows and Subagents

Subagents are self-contained Claude Code processes with their own context windows. The main session receives only the subagent's output, keeping its own context lean regardless of how much work the subagent performs. This makes subagents the right pattern for heavy exploration, parallel investigation, and codebase audits where you want the findings without the context pollution.

The tradeoff: subagents save your main context window, not your total token spend. Each subagent has its own bootstrap overhead and runs its own full session. For small lookups or quick questions, spawning a subagent costs more than asking directly. Use the pattern when the task is genuinely heavy, when the output is a file or summary rather than an inline response, and when context isolation matters.

### When to Use a Subagent

- Heavy exploration that would pollute your main thread (reading 30 files to map an unfamiliar feature)
- Parallel investigations where multiple subagents do read-only research simultaneously
- Codebase audits where the output is a summary, not a code change
- Specialized roles with restricted tools (a "reviewer" agent that only reads, never writes)

> ⚠ **WATCH OUT**
>
> Subagent bootstrap overhead can run 10–20K tokens before any user task begins. Don't spawn subagents for trivial questions.

✓ **DO THIS**

Have the subagent write findings to a file, not return them inline. The main session reads only the parts it needs. This is the pattern that actually reduces total spend.

## How to Create and Invoke a Subagent

1. Create the agents directory: `mkdir -p .claude/agents`
2. Create one Markdown file per agent role: `explorer.md`, `reviewer.md`, `security-audit.md`
3. Add YAML frontmatter with required fields: `name`, `description`, `tools`, `model`. Tools restricts what the agent can do.
4. Write the agent's instructions: imperative steps. Tell it where to write findings and what to summarize back.
5. Invoke from your main session: *"Use the explorer agent to map the auth module."* Claude delegates automatically and returns only the summary.

**.claude/agents/explorer.md**

```
# Example: .claude/agents/explorer.md
---
name: explorer
description: Reads many files to map a feature area. Returns short summary,
  writes detailed findings to a file.
tools: [Read, Grep, Glob, Write]
model: claude-haiku-4-5
---
1. Read all files relevant to the requested area.
2. Write detailed findings to docs/exploration-<topic>.md.
3. Return a 5-sentence summary to the parent session
   plus the path to the detailed file.

# Invoke from main session:
"Use the explorer agent to map the auth module"
```

> ⚠ **AGENT TEAMS: USE WITH INTENT**
>
> Agent teams use approximately **7×** more tokens than a standard single-agent session when teammates run in plan mode. They are disabled by default. The pattern fits complex, genuinely parallel workstreams, not routine coding tasks. Enable them intentionally: `CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS=1`
>
> Every active teammate consumes tokens even when idle. Clean up agent teams when work is done.

### If You Use Agent Teams

1. Use Sonnet for teammates. Teammate tasks rarely require Opus-level reasoning.
2. Keep teams small. Token usage scales roughly proportionally to team size.
3. Keep spawn prompts focused. Teammates automatically load CLAUDE.md, MCP servers, and skills — everything extra adds to starting context.
4. MCP overhead multiplies. Each teammate loads all connected MCP servers independently. Allowlist aggressively.

## 4.2 Team Controls and Governance

Individual workflow discipline scales only as far as individual habits. Consistent AI-assisted development practices across a team require governance at the configuration level, not just through documentation and training.

Claude Code provides several levers for team-level control, organized below by the concern they address.

### Cost Predictability Controls

| Control | What it does | Where to configure |
|---|---|---|
| `availableModels` | Restricts which models developers can select. Prevents individual drift toward high-cost models for routine work. | `managed-settings.json` on each developer machine |
| Workspace spend limits | Monthly cap that blocks usage when exceeded. Blunt but effective as a ceiling control. | Anthropic Console → Settings |

| Control | What it does | Where to configure |
|---|---|---|
| `fastModePerSessionOptIn: true` | Requires opt-in to fast mode each session. Prevents accidental 6× billing from a mode that persists by default. | `managed-settings.json` |
| `CLAUDE_CODE_DISABLE_FAST_MODE=1` | Removes fast mode entirely for all users. Use when the cost multiplier is not acceptable for any use case. | `managed-settings.json` env block |
| Per-user cost breakdown | Shows individual spend in the Console. Starting point for identifying outliers and coaching conversations. | platform.claude.com/usage |

## Output Consistency Controls

| Control | What it does | Where to configure |
|---|---|---|
| Team default model in `settings.json` | Sets a shared model baseline for the project. Developers can override personally in `settings.local.json`. | `.claude/settings.json` committed to repo |
| Shared skills library | Ensures all developers have access to the same workflow context for common tasks. Reduces variance in output quality for defined procedures. | `.claude/skills/` committed to repo |
| `.claude/rules/` files | Language-specific conventions committed to the project. Every developer picks them up automatically. | `.claude/rules/` committed to repo |
| Default effort level | Sets a consistent reasoning depth baseline. Prevents xhigh on routine tasks across the team. | `managed-settings.json` or `settings.json` |

## Operational Visibility Practices

| Practice | Cadence | What to look for |
|---|---|---|
| Per-user cost review in Console | Weekly | Sustained outliers above $30/active day; sudden spikes |

| Practice | Cadence | What to look for |
|---|---|---|
| ccusage weekly trend report | Weekly | Week-over-week cost trends; model mix changes |
| Shared skills audit | Monthly | Skills that haven't been invoked; descriptions that don't match usage |
| CLAUDE.md and auto memory review | Monthly | Accumulated content that no longer reflects current practice |
| MCP server configuration review | Quarterly or on new project setup | Unused servers; tools that could be replaced with CLI equivalents |

## Lock Model Selection with availableModels

Admins on Team and Enterprise plans can restrict which models users can select. The `model` field sets the starting default; `availableModels` is the allowlist.

managed-settings.json

```
// managed-settings.json
{
  "model": "sonnet",
  "availableModels": ["sonnet", "haiku"]
}
```

When `availableModels` is set, users cannot switch to any model not on the list via `/model`, the `--model` flag, or the `ANTHROPIC_MODEL` environment variable.

## How to Deploy managed-settings.json

1. Choose the install path for your OS.
2. Create the file at that path.
3. Distribute via MDM (Jamf, Intune, Kandji) for production rollout, or commit a deployment script to your infra repo.
4. Verify on a user machine: have a teammate run `/model` after picking up the deployment. Restricted models should not appear in the picker.

install-paths

```
# macOS
/Library/Application Support/ClaudeCode/managed-settings.json
```

```
# Linux / WSL
/etc/claude-code/managed-settings.json

# Windows
C:\Program Files\ClaudeCode\managed-settings.json
```

> **NOTE**
>
> The Default option in the `/model` picker is not restricted by `availableModels`. To control what Default resolves to, also set `ANTHROPIC_DEFAULT_SONNET_MODEL` in the env block.

### Full Model Reference

| Model | Input / Output per MTok | Context | Best for |
|---|---|---|---|
| **Opus 4.7** | $5 / $25 | 1M tokens | Long agentic tasks, outputs over 64K, vision |
| **Opus 4.6** | $5 / $25 | 1M tokens | Same capability as 4.7, no API breaking changes |
| **Sonnet 4.6** | $3 / $15 | 1M tokens | Most coding work; best price/quality ratio |
| **Haiku 4.5** | ~$0.80 / $4 | 200K tokens | High-volume, low-complexity, background tasks |

> **NOTE**
>
> Sonnet 4.6 achieves **95–99%** of Opus-level effectiveness on coding tasks at **40% lower cost**. Use Opus 4.7 when you need outputs longer than 64K tokens or the January 2026 knowledge cutoff. Sonnet 4.6's cutoff is August 2025.

## 4.3 Reference Configuration

Team configuration standards distribute through two mechanisms: the project repository (`.claude/settings.json`, skills, rules files) handles workflow context — shared skills, project conventions, hook configurations. `Managed-settings.json` deployed via MDM handles cost controls and model governance.

## Recommended Deployment Sequence for New Teams

1. **Instrument first.** Set up ccusage and Anthropic Console visibility before configuring anything else. You need a baseline to evaluate changes against.
2. **Set defaults in settings.json.** Commit a project-level default model and effort level. This establishes the team baseline without restricting individual flexibility.
3. **Add permissions.deny.** Commit baseline deny rules for credentials, generated output, and lock files. This has immediate cost and security value.
4. **Build the first two or three skills.** Start with the workflows developers repeat most — PR review, test writing, migration procedures. Commit them to the repository.
5. **Configure hooks.** Add PostToolUse hooks for formatting and type-checking. These reduce correction loops immediately.
6. **Add governance controls via MDM.** Once the team has established workflow habits, add `availableModels` and spend limits as a ceiling, not a starting point.

This layout applies every technique in the guide. Copy and adapt to your project.

project-layout

```
your-repo/
  CLAUDE.md                # under 200 lines, invariants only
  .claude/
    settings.json          # hooks + permissions.deny + availableModels
    settings.local.json    # personal overrides: model, effort, thinking
    rules/
      typescript.md        # loads only when editing .ts files
      python.md            # loads only when editing .py files
    skills/
      auth-flows/
        SKILL.md
        scripts/
          validate_jwt.py  # only output enters context
      db-migrations/
        SKILL.md
        REFERENCE.md       # loaded only when SKILL.md says so
        scripts/
          generate_migration.py
    agents/
      explorer.md          # subagent for heavy research (Haiku)
      reviewer.md          # subagent for code review (Sonnet)
  src/
    ...
```

## Why Each Piece Is There

- **CLAUDE.md under 200 lines** — it loads every session. Only what applies to 80%+ of tasks.

- **permissions.deny in settings.json** — blocks credentials, generated output, and lock files.
- **.claude/rules/** — language-specific conventions at zero cost in unrelated sessions.
- **settings.local.json** — lets each developer run leaner without affecting teammates.
- **Hooks in settings.json** — run formatters and type-checkers without LLM tokens.
- **Subagents use Haiku or Sonnet**, not Opus, with file-based handoff.
- **Bundled scripts inside skills** — handle deterministic work; only output enters context.

**LEADERSHIP SUMMARY — PART 4**

Team-scale governance requires consistent configuration distributed through standard developer tooling — not new infrastructure. The primary mechanisms: managed-settings.json for cost controls and model governance; the project repository for shared workflow context (skills, rules, hooks); the Anthropic Console for operational visibility. Start with instrumentation, then add controls progressively. Trying to govern everything at once tends to produce friction without the workflow foundation to support it.

R

REFERENCE

# Checklists

Two checklists follow. The developer checklist covers daily and per-session habits. The team lead checklist covers setup, governance, and review practices. Both can be used independently — print or copy these pages and tick items off as you go.

## Developer Checklist

Daily and per-session practices

### BEFORE EACH SESSION

- [ ] Run `/cost` to see last session's spend and gauge where you are relative to baseline ($13/day average).
- [ ] Run `/context` to see what is loaded in your context window. Identify anything that should not be there.
- [ ] Run `/clear` if you are starting a new task. Don't carry yesterday's context into today's work.
- [ ] Choose the right model for the task. Haiku for boilerplate; Sonnet 4.6 for everyday coding; Opus 4.7 only when reasoning quality is the bottleneck.
- [ ] Set your effort level. `effortLevel: medium` in `settings.local.json` for most sessions. Use `ultrathink` in prompts for turns that need deep reasoning.

### DURING EACH SESSION

- [ ] Run one task per session. When the task is done, close the session. Start fresh for the next thing.
- [ ] Use `/btw` for quick side questions that should not enter conversation history.
- [ ] Write specific prompts. Include scope, file location, and expected output. Vague prompts generate clarification turns that compound across history.
- [ ] Press `Shift` + `Tab` twice before multi-file work. Plan mode. Review the plan before proceeding.
- [ ] Press `Esc` immediately when Claude takes a wrong path. Use `/rewind` to restore to a checkpoint rather than correcting forward in a polluted context.
- [ ] Do not switch models mid-session. If you need to switch, do it at a session boundary.
- [ ] Do not toggle MCP servers mid-session. Both actions invalidate the prompt cache.

### AT SESSION BOUNDARIES

- [ ] Run `/rename` before `/clear` if there is any chance you will need to resume the thread.
- [ ] Use `/compact` with a focus hint when you need to continue a long session: `/compact Focus on API endpoints and error messages`.
- [ ]

Run `/usage` to see total session cost, API duration, and lines changed before closing.

## WEEKLY

- [ ] Run `ccusage week --by-model` to review weekly spend trend and model mix.
- [ ] Check per-day cost against the $13 average and $30 ceiling. Investigate sustained outliers

## MONTHLY

- [ ] Run `/memory` and audit auto memory. Remove anything outdated or redundant with CLAUDE.md.
- [ ] Review CLAUDE.md length. If it is over 200 lines, move low-frequency content to skills or `.claude/rules/`.
- [ ] Check which skills you have used. Uninvoked skills may have descriptions that do not match how you actually phrase tasks.

## STRUCTURAL SETUP (ONE-TIME OR ON NEW PROJECTS)

- [ ] Create `settings.local.json` with your personal overrides: model, effort level, alwaysThinkingEnabled, fastModePerSessionOptIn.
- [ ] Install ccusage: `npm install -g ccusage`
- [ ] Verify `permissions.deny` is configured in `.claude/settings.json`. Block: node_modules, dist, .next, coverage, .env files, *.lock files.
- [ ] Confirm `.claude/rules/` exists with language-specific convention files for the languages you work with most.

## Team Lead / Engineering Leader Checklist

Setup, governance & review cadence

### INITIAL SETUP (BEFORE TEAM USAGE SCALES)

- [ ] Set up Anthropic Console access for per-user visibility and distribute ccusage. Do this before anything else.
- [ ] Record average per-developer daily spend before making configuration changes. You need a before to evaluate the after.
- [ ] Commit baseline `.claude/settings.json` to the project repository — model, effort default, permissions.deny.
- [ ] Commit a baseline permissions.deny block. Both a cost control and a security measure.
- [ ] Deploy `managed-settings.json` via MDM — `fastModePerSessionOptIn: true` at minimum.

### SHARED WORKFLOW INFRASTRUCTURE

- [ ] Identify the three to five workflows developers repeat manually: PR review, test writing, migration procedures, deployment steps.
- [ ] Build and commit skills for those workflows to `.claude/skills/`. Include bundled scripts for deterministic work.
- [ ] Commit `.claude/rules/` files for each primary language in the project.
- [ ] Configure PostToolUse hooks for formatting and type-checking in `.claude/settings.json`. Commit to the repository.
- [ ] Define standard agent configurations if agentic workflows are in use (`explorer.md`, `reviewer.md` in `.claude/agents/`).

### COST AND MODEL GOVERNANCE

- [ ] Set `availableModels` in `managed-settings.json` to restrict model selection to Sonnet and Haiku for routine work. Add Opus only if the team regularly needs it.
- [ ] Set a workspace spend limit in the Anthropic Console.
- [ ] Set `CLAUDE_CODE_DISABLE_FAST_MODE=1` in `managed-settings.json` unless fast mode is explicitly required (it bills at 6× standard Opus pricing).
- [ ] Document the team's model selection policy. Which tasks warrant Opus? When should effort level be escalated? Write it down.

## ONGOING REVIEW CADENCE

- [ ] Weekly: Review per-user cost in Anthropic Console. Flag sustained outliers above $30/active day for a workflow conversation, not a reprimand.
- [ ] Weekly: Run `ccusage week --by-model`. A shift toward Opus without a corresponding complexity increase warrants a look.
- [ ] Monthly: Audit shared skills. Remove ones that haven't been invoked. Fix descriptions for skills being triggered inconsistently.
- [ ] Monthly: Review CLAUDE.md and auto memory across projects. Remove accumulated content that no longer reflects current practice.
- [ ] Quarterly: Review MCP server configurations. Identify unused servers. Replace with CLI equivalents where possible.
- [ ] Quarterly: Assess whether AI workflow investment is producing visible engineering outcomes. Focus on qualitative signals: developer perception, rework rates, time-to-first-usable-output trends.

## TEAM ONBOARDING

- [ ] Include the developer checklist in the team's engineering onboarding documentation.
- [ ] Walk new developers through `settings.local.json` setup on their first day. Personal defaults are easy to set and compound from day one.
- [ ] Review the shared skills library with new team members. Ensure they know what exists and how to invoke it.
- [ ] Set expectations explicitly: model selection, effort levels, and session habits are team practices, not personal preferences.

### ☆ A NOTE ON CURRENCY

The specific numbers and configuration examples in this playbook will change as Claude Code evolves. The underlying practices — lean context, protected caches, matched model capability, automated checks — are more stable. When a specific number looks wrong, check the official documentation. When a practice looks wrong, it probably still holds.

R

REFERENCE

# Reading List

### Official Documentation

- Claude Code: Manage costs effectively
- Claude Code: Subagents
- Claude Code: Hooks reference
- Claude Code: Slash Commands and skills migration
- Claude Code: Settings and permissions
- Anthropic: Best practices for Opus 4.7 with Claude Code
- Anthropic: Lessons from Claude Code: Prompt caching is everything
- Anthropic engineering: Equipping agents with Agent Skills
- Adaptive thinking and effort levels
- What's new in Opus 4.7

### Community Guides

- "18 Claude Code Token Management Hacks"
- Claude Code Token Optimization: Stop the $1,600 Bill
- Stop Wasting Tokens: Optimize Claude Code Context by 60%
- ClaudeFast: Context window guide and Opus 4.7 best practices
- How Prompt Caching Actually Works in Claude Code
- Sub-Agents in Claude Code (when they actually save tokens)
- Claude Code Skills: The 98% Token Savings Architecture
- Effort, Thinking, and How Opus 4.7 Changed the Rules

AKVELON
SOFTWARE ENGINEERING REDEFINED.AI

ABOUT AKVELON

## Built from production experience, not theory.

This playbook was built on direct engineering experience with Claude Code in production workflows across projects of varying scales and levels of complexity. Akvelon engineers use AI-assisted development tooling as a standard part of their delivery practice.

### WHY AKVELON

- **50+ AI-powered projects** delivering measurable ROI
- **Full AI lifecycle support** — from strategy to deployment
- **Industry-leading AI expertise** across fintech, healthcare, e-commerce
- **Security-first builds** with built-in compliance controls