# Token Book

Claude Code Token Optimization Guide

48 Diagnostic Symptoms · Copy-Paste Templates · 800+ Hours of Data

Revised edition — 2026-08-21

# Introduction: Why This Book Exists

Claude Code is expensive. On the Max plan ($200/month), sessions can burn through your daily allowance in 15-30 minutes. The Pro plan ($20/month) is worse — users report only getting 12 usable days out of 30.

GitHub Issue #16157 — Instantly hitting usage limits with Max subscription — has drawn 700+ reactions and is still open (checked 2026-08-21). It is not an isolated complaint. #46917 documents a version upgrade that added roughly 20,000 billed tokens per request for an identical payload, and #42796 drew 3,200+ reactions before it was closed. The message across all three is the same: **you are paying for tokens you cannot see**.

Free advice exists. Anthropic's own docs recommend hooks and CLAUDE.md optimization. Blog posts offer "18 hacks" and "6 tips." But none of them include:

- **Measured before/after data** from real operation
- **Copy-paste settings templates** that work immediately
- **Hook scripts** that automate token control without manual vigilance
- A **systematic approach** that covers CLAUDE.md, context management, hooks, subagents, and workflow design together

This book fills that gap.

## What's Inside

10 chapters. 44,000 words. Based on 800+ hours of autonomous Claude Code operation.

1. **Where Tokens Go** — The 4 layers of consumption. How to read `/cost`
2. **CLAUDE.md Optimization** — 5 patterns that cut consumption measurably
3. **Context Management** — When to `/compact`, `/clear`, and `--resume`
4. **Hook-Based Token Guards** — 9 hooks that auto-detect and block waste
5. **Subagent Strategy** — Budget limits, spawn controls, isolation patterns
6. **Workflow Design** — Task splitting for maximum cache efficiency
7. **Before & After Data** — Real measurements from 800h of operation
8. **Troubleshooting** — Cache breakage, token spikes, billing anomalies
9. **Template Collection** — settings.json, CLAUDE.md, hooks (copy-paste ready)

10. **(Bonus) April 2026 Update** — the cache TTL silent change, the v2.1.100 cache_creation inflation, the $1,446 unauthorized transfer, and nine more April incidents
11. **(Bonus) May 2026 Update** — the month single failures became clusters: silent skips, silent model swaps, sub-agent boundary gaps, and a $6,000 overnight quota burn

Each chapter stands alone. Start wherever your problem is.

## Who This Is For

- **Max plan users** ($200/month) who run out of session budget in under an hour
- **Pro plan users** ($20/month) who can only use Claude Code 12 days out of 30 — Claude chat and Code share the same quota pool, which changes your consumption strategy entirely
- **API users** who want to control costs before they spiral — Chapter 10 documents a $1,446 unauthorized transfer and a $367 subprocess incident, both traced to their cause
- **Anyone** who wants copy-paste templates instead of vague advice

No programming background required. Every hook and template is ready to use as-is. The author is a non-engineer who built and operates a business entirely through Claude Code.

## What "Half" Means

The title says "cut your token consumption in half." Here's what that's based on:

- A 100-line CLAUDE.md compressed to 35 lines improved cache read ratio from 89% to 95%
- Session token consumption dropped measurably (Chapter 7 has the full data)
- The combination of CLAUDE.md optimization + context management + hooks + workflow changes produced an overall reduction that averaged roughly 50%

Your results will vary. If your CLAUDE.md is already lean and you use hooks, the gains will be smaller. If you're running a 500-line CLAUDE.md with no hooks and single long sessions, you might save more than 50%.

## A Note on Timing

This book reflects Claude Code as of April 2026 (v2.1.x). Anthropic ships updates frequently — some of the specific issues described here (like the cache TTL regression or v2.1.100 inflation) may be fixed by the time you read this.

However, the principles are durable: - Shorter, structured CLAUDE.md files will always be more token-efficient - Hook-based automation will always beat manual vigilance - Context management will always matter - Workflow design will always affect consumption

The templates may need minor updates. The approach won't.

---

Chapter 1: Where Your Tokens Are Going →

# Chapter 1: Where Your Tokens Are Going

You open Claude Code in the morning. By lunch, your session budget is half gone. You haven't done anything unusual — a few file edits, some debugging, maybe a refactor. But the tokens are disappearing.

The latest example: a Max 20x plan user — the highest tier available — saw **50% of their weekly budget consumed in just 1.5 days** (#50325). At that rate, a "weekly" budget becomes a 3-day budget. Even paying top dollar doesn't make the problem go away.

This isn't a bug. It's how Claude Code works. Understanding the mechanism is the first step to fixing it.

## The 4 Layers of Token Consumption

Every Claude Code session burns tokens across four distinct layers. Most users only notice the first one.

### Layer 1: System Prompt (Every Session Start)

When Claude Code starts, it loads an internal system prompt that you never see. This is a fixed cost per session.

On top of that, it loads: - **CLAUDE.md** — both global (`~/.claude/CLAUDE.md`) and project-local (`./CLAUDE.md`) - **Tool definitions** — descriptions of every available tool. Deferred Tool Loading (v2.1.89+) delays some, but core tools are always included - **MCP configs** — if you use MCP servers, their tool definitions add to the prompt

The implication: **a longer CLAUDE.md means higher base cost per turn**. A 500-line CLAUDE.md costs thousands of extra tokens on every single message.

### Layer 2: Context Window (Conversation Accumulation)

As you work, every message, tool call, and response accumulates in the context window. Claude Code supports a 1M token context, but as it fills:

- Each turn sends more tokens to the API
- Prompt cache efficiency becomes critical — when caching works, repeated content is cheap. When it breaks, you pay full price for everything

- **Prompt cache breakage is the #1 cause of sudden token spikes** (covered in Chapter 8)

**Layer 3: Tool Calls (Read/Write/Execute)**

Every tool call has a cost — both the input (parameters) and the output (results):

- **Read**: File contents enter the context. A 3,000-line file costs proportionally
- **Glob/Grep**: Search results enter the context
- **Bash**: Command output enters the context. `ls -la` vs `cat large-file.txt` is a 100x difference
- **Edit/Write**: Changes are recorded in the context

There's also an internal 200K token budget for tool results. Exceed it and results get silently truncated — leading to incomplete information, retries, and wasted tokens.

**Layer 4: Subagents (Parallel Processing Cost)**

The `Agent` tool spawns subagents, each with its own context window. Five parallel agents = five separate API calls. Subagents are powerful but expensive when uncontrolled (Chapter 5).

## Check Your Current Consumption

Before optimizing, measure your baseline:

- `/cost` — For API billing users. Shows token counts and dollar costs per session
- `/stats` — For Max/Pro plan users. Shows usage patterns (no dollar amounts since it's subscription-based)

You can also add token info to your status line via `/config`. Get in the habit of checking `/cost` before and after optimization — it's the only way to know if changes are working.

## What Burns the Most Tokens (Ranked)

From 800+ hours of autonomous operation, the biggest token consumers are:

| Rank | Source | Impact |
|---|---|---|
| 1 | **Large file reads** | Reading multi-thousand-line files in one shot |
| 2 | **Context bloat** | Long sessions without `/compact` |

| Rank | Source | Impact |
|---|---|---|
| 3 | **Cache breakage** | When prompt cache fails, costs spike 5-10x |
| 4 | **Unnecessary tool calls** | File listings, git logs, exploratory reads |
| 5 | **CLAUDE.md bloat** | 500+ lines = thousands of tokens per turn |
| 6 | **State file updates** | Updating mission.md or task files every tool call |

## The Optimization Map

Each problem has a chapter:

| Problem | Solution | Chapter |
|---|---|---|
| CLAUDE.md bloat | Compress to <100 lines, 5 writing patterns | Ch. 2 |
| Context accumulation | `/clear`, `/compact`, `--resume` strategy | Ch. 3 |
| Unnecessary tool calls | Hook-based detection and blocking | Ch. 4 |
| Subagent overuse | Budget limits and spawn controls | Ch. 5 |
| Inefficient workflows | Task splitting and sequencing | Ch. 6 |
| Cache breakage | Root cause identification and prevention | Ch. 8 |

## One Counterintuitive Finding

Most people assume "use fewer tokens" means "tell Claude to be brief." This backfires.

When you add "save tokens" or "be concise" to your CLAUDE.md, Claude Code starts **avoiding work** — it skips detailed analysis, avoids spawning helpful subagents, and produces shallow results. You save tokens but get worse output.

The right approach is not "use less" but **"waste less."** Same quality output, fewer wasted tokens. That's what the rest of this book is about.

---

Next: Chapter 2 — CLAUDE.md Optimization (the single highest-impact change you can make)

# Chapter 2: CLAUDE.md Optimization

CLAUDE.md is your instruction file for Claude Code. Everything in it gets loaded into the context on **every single turn**. One extra line means higher base cost for every message in every session.

But "make it shorter" is wrong advice. A CLAUDE.md that's too brief leads to ambiguity, which leads to retries, which costs far more than a few extra lines.

The goal isn't "short." It's **efficient** — same intent, fewer tokens, higher precision.

## The 100-Line Rule

From 800+ hours of operation:

| Length | Assessment |
|---|---|
| Under 50 lines | Basic instructions only. Often insufficient for autonomous work |
| 50-100 lines | Core rules and priorities fit. Best balance |
| 100-200 lines | Detailed coverage but fixed cost starts adding up |
| 200+ lines | Per-turn overhead becomes noticeable. Split into `.claude/rules/` |

When your CLAUDE.md exceeds 200 lines, move detailed procedures to `.claude/rules/` — Claude Code loads these files too (v2.1.x+) but they're easier to manage. Keep priorities and judgment criteria in CLAUDE.md; move procedures and conditionals to rules files.

My CLAUDE.md settled at 141 lines after starting at 500+. Chapter 9 includes a minimal 30-line template. Adjust between 30-100 depending on your project's complexity.

## Pattern 1: Allowlist Over Denylist

### Wasteful

```
## Forbidden
- Don't use rm -rf
- Don't use git reset --hard
```

```
- Don't use git push --force
- Don't delete .env files
- Don't delete node_modules
- Don't use rm with paths containing /
- Don't use sudo
```

7 lines for 7 rules. But "anything not forbidden is allowed" — so `find . -delete` is fair game. Denylists are always incomplete.

### Efficient

```
## Allowed Operations
- File reads: always OK
- git commit: OK (no direct push to main)
- npm install: OK (--save-dev only)
- File deletion: only files you created

Any destructive operation not listed above: ask first.
```

6 lines covering the same scope, plus a safe default for unlisted operations. Allowlists are a security fundamental.

**Token impact**: Similar line count, but retries drop significantly. Denylist thinking causes "is this forbidden? → try it → blocked by hook → try alternative → blocked again" loops. Allowlist thinking produces "not on the list → skip it" in one step.

## Pattern 2: One Example (Few-Shot)

### Wasteful

```
Write clear commit messages.
```

Claude Code generates: "Update files", "Fix bug", "Add feature." No standard to measure "clear" against.

### Efficient

```
Commit messages: include the "why."
✓ fix: prevent duplicate API calls on rapid button clicks
✗ fix bug
```

3 lines added. Quality transforms. LLMs excel at extracting patterns from examples (few-shot prompting). One good example + one bad example creates an implicit standard.

**Token impact**: Adding 2 lines costs ~20 tokens per turn. Eliminating "bad message → rewrite request → regeneration" retries saves hundreds.

## Pattern 3: One Line of Reasoning

### Wasteful

```
Put test files in `__tests__/`.
```

Rules without reasons get ignored at edge cases. Claude Code may decide "next to src/ is cleaner" and break the rule.

### Efficient

```
Put test files in `__tests__/`.
  Why: files in src/ get included in build output, inflating bundle size.
```

One line of reasoning. Claude Code now understands the principle and applies it to similar cases (config file placement, fixture files, etc.) without being told.

**Note**: Keep reasoning to 1-2 lines. Three paragraphs of explanation is counterproductive — in our data, verbose reasoning increased consumption by roughly 10%. One line stays within noise.

## Pattern 4: Tables for Decision Criteria

### Wasteful

```
 When choosing technology, pick the most popular option.
For unclear requirements, implement the simplest interpretation.
When performance and readability conflict, prefer readability.
But if profiling confirms a bottleneck, prefer performance.
For security decisions, always choose the safer option.
```

5 lines of prose. Claude Code reads all of it every time, searching for the relevant rule.

### Efficient

```
| Situation | Decision |
|-----------|----------|
| Tech choice | Most popular option |
| Unclear requirements | Simplest interpretation |
| Performance vs readability | Readability (profiled bottleneck: exception) |
| Security | Always the safer option |
```

Same information, clear structure. LLMs parse table-formatted conditionals with higher accuracy than prose. Fewer misreads means fewer retries.

**Token impact**: Tables tend to express the same information in fewer tokens than prose. Plus, decision accuracy improves, reducing retry loops.

## Pattern 5: Move Enforceable Rules to Hooks

### Wasteful

```
## Safety Rules
- Never use rm -rf. Delete files individually instead.
- Don't use git reset --hard. Use git reflog to recover.
- Don't git add .env files. Make sure .gitignore covers them.
- Don't push directly to main. Create a feature branch.
- Don't use sudo.
- Don't use chmod 777.
```

These are "requests." Claude Code might forget them. And even if it remembers, it's reading 6 lines on every turn that could be enforced mechanically.

### Efficient

```
## Safety
Hooks auto-block destructive commands, branch violations, and secret exposure.
Don't retry operations that hooks block.
```

2 lines. The hooks enforce the rules technically — Chapter 4 shows how. Writing enforceable rules in both CLAUDE.md and hooks is double management: wasted tokens and wasted effort.

**Token impact**: 6 lines → 2 lines saves ~100 tokens per turn in fixed cost. More importantly, hook enforcement means the retry cycle is "attempt → blocked → stop" (1 cycle) instead of "attempt → succeeds → accident → rollback" (10x tokens).

## Quick Reference

| Pattern | Core Idea | How It Saves |
|---|---|---|
| Allowlist | Define what's permitted | Eliminates retry loops |
| One example | Show one good + one bad | Prevents quality rework |
| One-line reason | 15-30 words explaining "why" | Enables edge-case autonomy |
| Table format | Condition → decision table | Faster parsing, fewer misreads |
| Hook delegation | Enforce mechanically, not textually | Removes duplicate rules |

These five patterns won't cut your CLAUDE.md to zero. But they'll compress 500 lines into 100 without losing information — and the token savings compound on every turn of every session.

---

Next: Chapter 3 — Context Management (`/clear`, `/compact`, `--resume`)

---