

You probably don't need the bigger plan.

Ten things to check before you upgrade, in the order I'd check them. Most people running out of tokens are not doing too much work. They are carrying too much furniture into every single message.

BY DAVE BROWN

I've built two seven-figure software companies, and I shipped one of my mobile apps on a single \$20 subscription. So when someone tells me they're hitting limits by lunchtime and the fix is a plan four times the price, my first question isn't *how much do you code*. It's *what are you dragging along behind you*.

Almost always the answer is: a lot. And almost always they have never looked.

THE MECHANIC EVERYTHING ELSE FOLLOWS FROM

These tools are stateless. Every time you hit enter, the entire conversation is sent again from scratch — your prompts, every response, every file it read, every tool result, plus your instruction files, your skill definitions, and the description of every MCP tool you have installed. It is not reading a log or remembering where it left off. It is re-processing the whole pile, on every turn.

So a 2,000-token file you loaded forty messages ago isn't a one-time cost of 2,000 tokens. It is 2,000 tokens times every message you've sent since.

Caching softens this and it does not remove it. On Anthropic's published API pricing a cached token still bills at roughly 10% of the normal input rate, and writing to the cache costs about 25% more than a plain token the first time. Subscription plans don't publish their internal weighting, so I won't pretend to quote you a number there. But the shape holds: cheap is not free, and 10% of an enormous number is still a large number.

10%

What a cached input token still costs on the API. Discounted, not free.

5×

Input-price gap between Haiku 4.5 and Opus 5. Same dollar, five times the volume.

Every turn

How often your skills, instructions and tool list get resent. Not once. Every turn.

~30 min

The one-time audit in Part two. It's the highest-return half hour here.

Everything below applies whether you're on Claude Code or Codex, since the underlying economics are the same. Where a command or file path is specific to Claude Code I've said so, and I've checked each one against a current install rather than repeating what circulates on social. Two items that show up on nearly every list like this one are wrong; I've flagged both where they come up.

— WHAT'S IN HERE

- 01 **The audit.** Three things to measure and cut before you change a single habit. This is where the money is.
 - 02 **The habits.** Five working patterns that stop refilling the window you just emptied.
 - 03 **The two commands.** Both underused, one of them almost unknown.
 - 04 **What doesn't work.** Including the trick half the internet recommends, and why I don't.
 - 05 **If you've done all ten.** When a second subscription genuinely beats a bigger one.
-

Three things you are paying for on every message, whether you use them or not.

This is a one-time cleanup with a permanent payoff, which makes it the only part of this document I'd call urgent. Do it once and every conversation you have afterwards starts lighter.

01

Audit your skills, and tighten every trigger.

This is the single biggest hidden cost I see, and it's gotten worse as skill marketplaces filled up. You installed thirty of them over a few months. You actively use four. The other twenty-six are still being described to the model on every turn so it can decide whether they're relevant.

A well-built skill costs you around a hundred tokens to have sitting there — a name and a tight description, with the real content loaded only when it fires. A badly-built one can pull in thousands before you've seen a single word of response. The worst offenders are consistently the *"my voice"* and *"content coach"* genre, because their trigger conditions are written so broadly that the model concludes it needs the full skill loaded to shape almost any output. You ask a one-line question about a config file and you've paid for a 4,000-token essay on your brand tone.

So go through them. For each one, read the trigger description and ask whether it fires exactly when you'd expect and never otherwise. Tighten anything vague. Delete anything you haven't used in a month — you can always reinstall it. Progressive loading is a genuinely good design, but if a skill progressively loads on every turn, you are paying full price for a feature you think is saving you money.

WHERE TO LOOK	THE TEST	EFFORT
~/ <code>.claude/skills/</code> and <code>.claude/skills/</code> in each project	Would this fire on "fix the typo on line 12"? If yes, the trigger is too broad.	Fifteen minutes, once. The largest single win on this list.

Put CLAUDE.md on a diet, then split it up.

Your instruction file — CLAUDE.md, or AGENTS.md depending on the tool — is loaded in full, every message, forever. It is the most expensive real estate you own, and it is usually the least curated.

The specific way these explode is worth naming, because almost nobody notices it happening. Somebody sets up a rule along the lines of *“any time I correct you, write it down in CLAUDE.md so you don’t do it again.”* That sounds like excellent hygiene. Six weeks later the file is nine hundred lines of accumulated micro-corrections, most of them contradictory, most of them about a file you deleted, and all of them being resent on every single turn.

The fix is two moves. First, cut it hard: a short, sharp root file with the things that are true everywhere — how to run the tests, the conventions that actually matter, what not to touch. Second, push the specifics down. Both Claude Code and Codex will pick up instruction files in subdirectories and load them only when work moves into that directory, which is how you get detail without paying for it constantly. Frontend conventions live in the frontend folder. They do not need to be in context while you’re editing a database migration.

THE NUMBER YOU’VE PROBABLY SEEN, WITH THE CAVEAT ATTACHED

A developer write-up that circulated widely reported cutting always-loaded context from 42,200 tokens to 1,900 — a 94% reduction — and extrapolated that to roughly 1.2 million tokens a day at thirty conversations. I’ve seen no independent verification of either figure and the daily number assumes a very particular usage pattern, so treat it as an illustration of the shape rather than a benchmark. The shape is right regardless: a bloated instruction file is a fixed tax multiplied by every message you will ever send.

TARGET

Root file under ~100 lines. If it’s longer, something belongs in a subdirectory.

WATCH FOR

Auto-append rules. They’re the mechanism that inflates the file invisibly.

EFFORT

Ten minutes, plus a quarterly re-read.

Run /context before you change anything.

This is the one I want people to run first, and it’s the reason the two items above are worth doing rather than guessing at. /context shows you what is actually sitting in your context window right now, broken out by what put it there: system prompt, instruction files, skills, MCP tool definitions, the conversation itself.

Run it in a brand-new session, before you've typed anything real. That number — whatever it is, before you've done a minute of work — is your floor. It is what every message you send today will carry. Most people are genuinely startled by it, and the startle is useful, because it tells you which of these ten items is *your* problem rather than somebody else's.

It costs nothing to run and it turns this entire guide from advice into arithmetic. Do it before the cleanup and after, so you know what the cleanup bought you.

RELATED

/usage shows where you stand
against your plan's limits

WHEN

Fresh session, before work.
Then again after you've cut.

EFFORT

Five seconds. There is no
excuse for skipping this one.

A fresh session that opens at 40,000 tokens isn't a fresh session. It's a bill you agreed to before you started working.

PART TWO · THE HABITS

Five ways to stop refilling the window you just emptied.

The audit is a one-time fix. These are working patterns, which makes them harder — they only pay if they become automatic. The good news is that every one of them also makes the output better, so you're not trading quality for cost.

04

Match the model to the task. Stop defaulting to the big one.

The published API rates make the case better than I can. Opus 5 runs \$5 per million input tokens and \$25 per million output. Sonnet 5 is \$2 and \$10. Haiku 4.5 is \$1 and \$5. That is a five-to-one spread on input between the top and the bottom, for work where the bottom is frequently indistinguishable.

Writing documentation for code that already exists. Renaming things across a repo. Formatting, boilerplate, test scaffolds, commit messages, turning notes into prose. None of that needs the most expensive reasoning model on the market, and if you're on a subscription the same logic applies to your limits even though you never see a dollar figure.

The pattern that works best is a split: plan with the strong model, execute with the cheap one. Use plan mode to think the problem through properly, get a specific written plan out of it, then hand that plan to a smaller model to implement. The expensive tokens go where judgment is required. The cheap tokens do the typing. A well-specified plan is exactly the thing that makes a smaller model reliable, which is why this works rather than just being cheaper.

STRONG MODEL	CHEAP MODEL	SWITCH WITH
Architecture, debugging something genuinely stuck, planning, security-sensitive work	Docs, renames, tests, boilerplate, anything with a written spec in front of it	<code>/model</code> , mid-session, as often as the work changes shape

05

MCP servers aren't Pokémon. You don't have to catch 'em all.

This is the collection problem. Supabase ships an MCP server — installed. GitHub — installed. Chrome DevTools, Figma, Sentry, Linear, Notion — all installed, over a few months, each one for a specific afternoon's task that ended six weeks ago.

Every one of those servers contributes a full description of every tool it exposes — names, parameters, usage notes — to the model on every turn, because the model can't know whether it needs the tool without knowing the tool exists. A single mid-sized server can run to several thousand tokens of definitions. A dozen of them is a second instruction file you didn't write and never read.

Run `/mcp`, look at what's connected, and be ruthless. Better still, scope them per project rather than globally: the Figma server belongs in the project with the design work in it, not in every terminal you open. Reconnecting one takes under a minute on the day you need it.

CHECK WITH	RULE OF THUMB	EFFORT
<code>/mcp</code> , and the MCP section of <code>/context</code>	Global config gets what you use weekly. Everything else is project-scoped.	Five minutes, and it stays fixed until you start collecting again.

Point at the file. Don't make it go looking.

"Find where we handle password resets and fix the redirect" and "fix the redirect in `src/auth/reset.ts`" produce roughly the same edit. They do not produce remotely the same bill.

The first version runs a search, which returns matches from a dozen files. It reads several of them to work out which is the right one. Nine of those reads were wrong guesses — and all nine are now in the context window, permanently, getting resent on every subsequent message for the rest of the session. You paid to read them once and you'll keep paying to carry them.

So spend the ten seconds. Tell it the path. Tell it the function name. Paste the error with the file and line number already in it. If you genuinely don't know where something lives, that's a fine reason to let it search — but then start a fresh session once you've found the answer, so the search debris doesn't follow you into the actual work.

CHEAPEST INPUT

A file path. Second cheapest: an error message with a stack trace.

MOST EXPENSIVE

A vague description of behaviour in a large unfamiliar repo

SIDE BENEFIT

Precision also cuts the odds it edits a file you didn't mean

Fence off the directories it should never read.

Some directories have no business being searched, ever: `node_modules`, build output, `dist`, vendored dependencies, fixture data, anything generated. A single stray read inside one of them can dump a minified bundle into your context and end the session's usefulness on the spot.

CORRECTING SOMETHING YOU'LL SEE REPEATED EVERYWHERE

Nearly every list like this tells you to create a `.claudeignore` file. I checked the current Claude Code build and there is no such thing — the string doesn't appear anywhere in the binary. If you've created one, it is doing nothing, which is worse than doing nothing because you think you're covered.

The mechanism that does work is a deny rule in settings. In `.claude/settings.json`, under `permissions.deny`, add entries like `"Read(./node_modules/**)"` and `"Read(./dist/**)"`. That is enforced by the harness rather than suggested to the model, which is the distinction that matters. Your `.gitignore` also does useful work here already, since ignored files are excluded from most search paths by default.

FILE	SYNTAX	VERIFY
<code>.claude/settings.json</code> — project, or user-level for global rules	<code>"permissions": { "deny": ["Read(./node_modules/**)"] }</code>	Ask it to read a denied path. It should refuse, not comply.

08

Plan in one session. Build in another. Compact on purpose.

The most expensive session you will ever run is the one you've been in all day. You planned in it, explored three dead ends in it, read a dozen files you ended up not touching, and now every message carries all of that. The work you're doing at 4pm is paying for the wrong turn you took at 10am.

Two habits fix this. The first is to treat the end of a phase as the end of a session. When planning is done, take the plan — written down, in the repo, as a file — and open something new to build it in. The plan is the only thing worth carrying forward, and a file carries it better than eighty messages of deliberation do.

The second is `/compact`, used deliberately rather than as an emergency measure. Auto-compaction will eventually fire on its own when the window fills, but by then you've already paid full freight for everything it's about to discard. Compacting at a natural boundary — feature done, bug fixed, tests green — keeps the summary while dropping the deliberation, and you choose what survives instead of finding out afterwards.

COMPACT WHEN	START FRESH WHEN	CARRY FORWARD WITH
A unit of work just finished, not when the window is already full	The topic changes, a plan is written, or you just backed out of a dead end	A file in the repo, not the conversation history

One is underused. The other one almost nobody knows exists.

Both of these attack the same thing from different directions: the cost of saying the same thing over and over, and the cost of interrupting work that was going fine.

09

Turn repeated instructions into slash commands.

If you find yourself typing the same four-step instruction every few days — run the tests, check the types, review the diff against main, write the commit — you are paying for that paragraph every time, plus the tokens spent working out what you meant by it.

A custom slash command is a markdown file in `.claude/commands/`. The filename becomes the command, and the contents are the instructions. Typing `/review` then pulls in a precise, pre-written sequence instead of your best improvised description of it.

The token saving is real but honestly secondary. The bigger win is that natural language is probabilistic — the same request phrased slightly differently gets interpreted slightly differently, and the variance costs you retries. A command is the same words every time, which means the same behaviour every time, which means fewer runs that you throw away and do again. Retries are where the tokens actually go.

WHERE

`.claude/commands/name.md`
— project, or
~/.claude/commands/
for global

GOOD CANDIDATES

Review sequences, release
checklists, your standard
debugging opener

REAL PAYOFF

Determinism. Fewer retries
beats fewer tokens per attempt.

10

/btw — stop interrupting it.

This is the one I almost never see mentioned, and it addresses a genuinely expensive habit.

It's working on something. Halfway through, you think of a question — *wait, does this thing even use Redis?* — and you interrupt to ask. That question and its answer are now permanently in the main conversation, resent on every future turn, and worse, you've derailed a run that was going fine and will now need re-orienting.

/btw opens a side channel instead. It answers against the current session's knowledge and your codebase, but the exchange is never injected into the main conversation history, and the task in progress continues undisturbed. You get the answer; the main thread pays nothing and loses nothing.

ONE LIMITATION WORTH KNOWING

/btw can't run tools. It answers from what's already in the session and the model's own knowledge, so it won't go read a file or run a command to check. If any tool calls appear in its output, they weren't executed and may not reflect your actual files. For "what did we decide about X" and "does this approach have an obvious problem" it's ideal. For "go check whether that file exists," ask in the main conversation.

USE FOR

Side questions, sanity checks, "what does this even do" — while work continues

NOT FOR

Anything requiring a file read, a command, or a real edit

THE ALTERNATIVE

Holding the thought. Free, though historically unreliable.

PART FOUR · WHAT DOESN'T WORK

The trick that gets recommended constantly, and why I'm not convinced.

Every thread on this topic eventually surfaces a prompting style meant to compress the model's output. It has several names. The reasoning behind all of them has the same hole in it.

The idea is that you instruct the model to drop articles, skip pleasantries, and answer in clipped fragments — "caveman speak," or one of its cousins. Fewer words out, fewer tokens billed. It feels like a free win, and people report dramatic-sounding savings.

The hole is that it optimises the wrong number. In a coding session the overwhelming majority of your token spend is *input* — the full conversation, your instruction files, your skill and tool definitions, and every file that's been read, all resent on every turn. Output is the small side of the ledger. Compressing it is like negotiating your coffee budget while the lease renews itself every hour.

And it isn't free, because there are two costs that don't show up in a token counter. The instruction itself lives in your context permanently, on every turn, working against you. And terse output means less explanation of what the model did and why, which means more misunderstandings, which means more rounds of correction — and a retry costs you the entire conversation again, wiping out a great many saved articles and prepositions.

I'll happily be talked out of this by measurement. Run `/context` and look at where your tokens actually sit before you spend effort here. My honest read is that anyone with the headroom to fuss over output style has not yet done items 1 through 3, where the real money is.

THE LEVER	HONEST SIZE	WHY IT LANDS WHERE IT DOES
01 · Skill audit	Large, permanent	Fixed cost on every turn, and the one most people have never looked at
02 · CLAUDE.md diet	Large, permanent	Same mechanic, and these files inflate silently over months
03 · <code>/context</code>	Zero on its own	Saves nothing directly. Tells you which of the other nine is your problem.
04 · Model matching	Large, ongoing	Up to a 5× input spread. Requires you to actually switch, which is the hard part.
05 · MCP pruning	Medium to large	Scales with how much of a collector you've been
06 · Point at the file	Medium, compounding	Avoided reads never enter the window, so they're never resent
07 · Deny rules	Small, occasionally huge	Insurance. Most days nothing; the day it catches a bundle it pays for itself
08 · Session hygiene	Medium, ongoing	Hardest to make habitual, which is why it's usually the one still undone
09 · Slash commands	Small directly	Pays through determinism. Retries cost far more than verbosity does.

THE LEVER	HONEST SIZE	WHY IT LANDS WHERE IT DOES
10 · /btw	Small to medium	Keeps side questions out of the permanent history, and out of the model's way
Output compression	Marginal, possibly negative	Targets the small side of the ledger and buys you extra rounds of clarification

PART FIVE · IF YOU'VE DONE ALL TEN

Then the honest answer might be two subscriptions, not a bigger one.

This is the mildly controversial part, and it only applies once the ten above are genuinely done rather than skimmed.

If you've run the audit, cut the files, pruned the servers, and you are still hitting limits doing real work, then you have a capacity problem rather than a hygiene problem, and it's worth solving. But before you quadruple your spend on a single provider, do the arithmetic on a second \$20 subscription instead — either a second one with your provider of choice, or one with a different provider entirely.

Two independent limits at \$40 is often more usable capacity than one larger limit at a higher price, and it buys you something a bigger plan can't: a second opinion. Different models fail differently. When one is stuck in a loop on a bug, handing the same problem to a different family frequently unsticks it in one turn, which is a token saving in its own right.

The thing that makes or breaks this is switching friction. If moving between models means reconfiguring your environment, you will not do it, and you'll have bought a subscription you use twice. Pick a harness that makes switching trivial — conductor.build is what I use, and it also lets you run several agents in parallel, which changes the calculus again. Whatever you pick, test the switching flow before you commit to the second subscription.

THE ORDER I'D ACTUALLY DO THIS IN

Run `/context` in a fresh session and write the number down. Audit skills, then `CLAUDE.md`, then MCP servers. Run `/context` again and compare. Work for a week pointing directly at files and switching models by task. If you are still constrained after that, and only then, buy the second subscription — at which point you'll know precisely what you're buying and why, instead of guessing.

Running out of tokens feels like a capacity problem, which is why the pitch for a bigger plan lands so easily. Most of the time it's a carrying problem. You are moving the same furniture into every room and being surprised that it's heavy.

The whole audit takes about half an hour and costs nothing. A plan upgrade takes thirty seconds and costs you every month from here on. It seems worth doing them in that order.

ONE MORE THING

If you run the audit and the number still doesn't make sense.

Sometimes the floor is high for a reason that isn't on this list, and it's hard to spot from inside your own setup.

Send me your `/context` breakdown from a fresh session — just the numbers, no code, nothing proprietary — along with roughly what you're building and which plan you're on. I'll tell you which of the ten above is eating your budget and whether upgrading would actually fix it. Often it wouldn't, which is a cheaper answer than finding out in a month.

GET IN TOUCH

Email hi@davecto.com with the subject line **"Token Diet"** and paste the breakdown.

More guides like this one, for people trying to use AI without embarrassing themselves.

Weekly, plain-language breakdowns on Instagram.

[@davecto](#)

A note on sourcing: per-token prices and the caching multipliers are Anthropic's published API rates as of writing, and they describe API billing rather than subscription-plan limits, which aren't published in comparable terms — the ratios are the useful part, not the dollar figures. Commands, file paths and settings syntax were checked against a current Claude Code install (2.1.x) rather than taken from secondary write-ups; that is also how I established that `.claudeignore` is not a real feature, and it's worth re-checking after major releases since this surface moves. The 42,200-to-1,900-token figure is one developer's self-reported result, repeated here with that caveat and not independently verified. The relative sizing in the Part four table is my judgement from my own usage, not measurement across a population — your `/context` output is better evidence about your setup than my table is. I have no commercial relationship with Anthropic or OpenAI. I do use Conductor and recommend it on that basis; it's named because switching friction is the thing that decides whether a second subscription pays, not because anyone paid for the mention.