TokenBuild

TokenBuild Help

The complete manual. It covers what TokenBuild does on your machine and why, its three products (Token Saver, Team Build, and ML Validation and Economics) with every feature, command and setting, the dashboard and Team workspace, every error you might see and how to fix it, and the habits that save the most.

Version 0.20.4 · Updated November 23, 2026 · 124 printed pages. Shorter pages: Documentation (overview), Commands (one-line reference), FAQ.

CONTENTS

GETTING STARTED

1. Introduction and key ideas
2. Getting started
3. What happens in a session

TOKEN SAVER · SAVE

4. Claude reads less
5. Claude writes less and wastes fewer turns
6. Measuring savings

TEAM BUILD · TEAM

7. Task state and planning
8. Team Relay and the team workspace

ML VALIDATION AND ECONOMICS · LEARN

9. Tests and the Verify Loop
10. Before you ship
11. How TokenBuild learns
12. Proof, strategies and repo economics

REFERENCE

13. The web app and dashboard
14. CLI command reference
15. MCP tools, skills and the scout
16. Configuration reference
17. Privacy, security and your data
18. Errors and fixes
19. Tips and tricks
20. Improving your results
21. Glossary

Appendix A: Worked examples

Appendix B: More questions answered

CHAPTER 1

Introduction and key ideas

This chapter explains what TokenBuild is, the problem it solves, and the handful of ideas the rest of the manual builds on. If you only read one chapter before you start, read this one.

What TokenBuild is

TokenBuild is a plugin for Claude Code, Anthropic's coding agent. It sits beside Claude Code on your own computer and takes over the parts of a coding session that don't need a large language model: finding the right files, reading long logs, remembering what has already been read, checking whether a change broke the tests, and carrying the important facts of a long conversation into a fresh one. Ordinary code does that work locally, so it costs no tokens.

TokenBuild doesn't replace Claude Code, change the model you use, or route your requests through anyone else's servers. You keep running official Claude Code with your own Claude plan or Anthropic API key. TokenBuild only uses the extension points that Claude Code publishes for this purpose: hooks, an MCP server, skills, a subagent and project instructions.

You use it in three places:

- **The CLI and plugin** (`tokenbuild`, published on npm). Every feature runs here, on your machine.
- **The web app** at app.tokenbuild.ai. You create your account, manage billing and CLI keys, and see your savings dashboard there. Team plans also get a Team workspace.
- **This website**: pricing, documentation, this manual, legal terms and Toki, the assistant in the corner of each page.

The three products

Every TokenBuild feature belongs to one of three products. Each one has its own command group in the CLI, and this manual has a part for each.

Product	What it does	Commands	Chapters
Token Saver	Keeps what wouldn't help out of Claude's context, cuts what Claude writes, avoids wasted turns, and counts every token it saves. Waste Guard, Output Shrinker, Context Ledger, Repo Index, Task Brief, Next-Read, Exploration Memo, Docs Oracle, Fingerprints, Local Boost, model routing, Anchor Edits, Codemod Mode, Edit Preflight, Loop Breaker, Fresh Start, Context GC, Cache Keeper, setup tuning, the savings ledger, Real Meter and dashboard sync.	`tokenbuild save`	4, 5, 6
Team Build	Keeps the goal, plan and decisions outside the conversation, plans projects, and coordinates every Claude on a team. Task State, Project State, Project Preflight, Autopilot budgets, Team Relay, the team plan with stages and dependencies, handoffs, Team Memory and coordination insights.	`tokenbuild team`	7, 8
ML Validation and Economics	Models trained on your own repo's history check the work and decide where tokens pay off. Verify Loop with predictive test selection and flaky-test detection, change risk and incidents, drift, the learning layer and budget forecasts, the control-group proof, Replay, the Recipe Compiler and the Repo Shape Optimizer.	`tokenbuild learn`	9 to 12

The groups are a way to find things. The older single-word commands (`tokenbuild stats`, `tokenbuild relay`, `tokenbuild verify` and the rest) still work, and most features run by themselves through Claude Code's hooks with nothing to type.

Why tokens matter

Claude reads and writes text in units called **tokens**. A token is roughly four characters of English or code, so a 400-line source file is often 4,000 to 6,000 tokens. Everything Claude knows

during a conversation sits in its **context window**: your messages, its replies, every file it read, every command output it saw, and the definitions of every tool it can call.

Three facts about how Claude Code works explain where most of the cost comes from:

1. **Every reply re-sends the whole conversation.** The model doesn't keep memory between turns. Each time Claude answers, the entire context goes back in. A 2,000-line log Claude read early in a session gets paid for again on every turn after that, until you clear or compact the conversation.
2. **Caching helps, but it expires.** Claude Code caches the conversation so re-sending it costs a fraction of the normal input price. The cache lasts only a few idle minutes. If you come back from lunch to a huge conversation, the next message writes the whole thing to the cache again at full price.
3. **Output tokens cost the most.** When Claude edits a file with the standard edit tool, it writes out the old text and then the new text. Repeating the old code is paid at the output price, which is several times the input price.

On a Claude Pro or Max subscription you don't pay per token, but the same tokens count toward your usage limits. Wasted tokens mean hitting the limit sooner. On an API key, they're money.

The kinds of waste TokenBuild removes

Waste	Example	What TokenBuild does
Reading things that never help	Claude opens `package-lock.json` or a minified bundle while searching	Waste Guard stops the read and suggests the source file instead
Noisy command output	A failing test run prints 3,000 lines, of which 40 matter	Output Shrinker shows the errors and the summary; the full log stays on disk

Reading the same thing twice	Claude re-reads a file it already has, which is unchanged	Context Ledger answers "already in your context" or sends only changed lines
Searching instead of knowing	Five Grep and Glob calls to find where a function lives	Repo Index, Task Brief and `tb_find` give `path:line` directly
Rewriting old code in edits	An edit that changes one line of a 60-line block	Anchor Edits (`tb_edit`) send only the new lines
Extra fix cycles	An edit with a syntax error, found only after running the tests	Edit Preflight catches it before the edit lands
Running the whole test suite	Every change runs 900 tests when 12 are relevant	Verify Loop runs only affected tests and reports one line
Loops	The same test fails the same way, again and again	Loop Breaker steps in after the third identical failure
Carrying a huge conversation	A 150K-token session re-sent on every turn	Fresh Start writes a handoff note so `/clear` continues from about 2K tokens
Setup overhead	An MCP server you never use adds 3,000 tokens of tool definitions every turn	Tool Diet finds it; `--apply` turns it off for the project

Design principles

A few rules shape every feature. Knowing them makes TokenBuild's behavior easier to predict.

- **Your code never leaves your machine.** Features that need intelligence beyond plain code either ask Claude (which you already use) or, with Local Boost, a model running on your own computer. Sync sends counts, never content. See Chapter 17.
- **Never get in the way.** Hooks finish in milliseconds and never wait on the network. If anything goes wrong, or you have no active plan, TokenBuild steps aside and Claude Code

behaves exactly as it would without it.

- **Blocks are advice, not walls.** When TokenBuild stops a read or an edit, it explains why. If Claude really needs it, asking again gets it through. A re-requested read is logged as an "undo" and its saving is subtracted, so the numbers stay honest.
- **Measure, don't guess.** A saving appears on your dashboard only if something counted it. Features whose benefit is fewer turns are proven with control groups, and their credit is capped. See Chapter 6 and Chapter 12.
- **Learn from this repo.** What helps in one codebase can cost tokens in another. TokenBuild learns per repo, from tasks it measured, and judges every choice on tokens to a *verified* result, so giving up early never counts as saving. See Chapter 11.

How the pieces fit together

When you install the plugin, Claude Code loads five kinds of extension from it:

Piece	What it is	What TokenBuild uses it for
Hooks	Small programs Claude Code runs at fixed moments: session start, each prompt, before and after each tool, when a reply ends, before compaction, at session end	Waste Guard, Context Ledger, Edit Preflight, Task Brief, Verify Loop, Loop Breaker, Fresh Start, formatting, sync at session end
MCP server	A local process that offers extra tools Claude can call	The `tb_*` tools: brief, map, find, pack, verify, task, edit, apply, docs, memo and more
Skills	Instructions Claude Code loads only when relevant, or that you run with a slash command	`/tokenbuild:setup`, `/tokenbuild:stats`, `/tokenbuild:handoff`, the rules skill, and recipes compiled from your work
Subagent	A helper Claude can send off to do a narrow job in its own context	The **scout**, a fast low-cost explorer for finding things and reading big files

CLI on the PATH	The `tokenbuild` command	Everything you run by hand: audit, stats, meter, proof, risk, relay and the rest

Claude Code starts one short-lived `tokenbuild hook <event>` process per event. Heavier work, such as running tests, rebuilding the index or compiling recipes, happens in detached background processes so the hook can return immediately.

What TokenBuild keeps on disk

Each repo gets a `.tokenbuild` folder at its root, the folder that contains `.git`. Everything in it is local data you can read, back up or delete. The most useful files:

File or folder	Holds
`config.json`	Your settings for this repo (Chapter 16). Optional; it may not exist.
`index.json`	The Repo Index: files, symbols and imports
`logs/`	Full logs of commands the Output Shrinker condensed
`ledger.jsonl`	The savings ledger: one line per counted event
`memory.json`	Repo Memory: which files tasks changed, and which change together
`fingerprints.json`	Error fingerprints and the files that fixed them
`task.json`, `handoff.md`	The current task state and the latest handoff note
`tasks.jsonl`, `train.jsonl`	Measured tasks and the learning layer's training data
`fresh.jsonl`	Adaptive Fresh Start decisions (context numbers, choice, probability) and when you typed `/clear`; never leaves your machine

`edits.jsonl`	One line per edit (file, first line, line counts) for the field measures; never leaves your machine
`tests.jsonl`	Per-test history for predictive test selection
`risk.jsonl`	Logged change-risk checks and their outcomes
`meter.json`, `reads.json`	Real Meter totals and what each file cost to read
`memo.json`	Saved answers (Exploration Memo)
`relay.json`	Team Relay's latest view of teammates (Team plans)
`state/`	Per-session working state: guard, ledger, policies, warnings

Your computer-wide data lives in `~/.tokenbuild` (on Windows, `C:\Users\<you>\.tokenbuild`): the saved CLI key file, the signed plan answer (`license.json`) and pooled learning priors. Set the `TOKENBUILD_HOME` environment variable to move it.

> **TIP**
> Add `.tokenbuild/` to your `.gitignore`. It's per-machine data. The one exception is recipes: they're written to `.claude/skills/`, which you *should* commit so teammates get them.

Plans at a glance

Plan	Price	For
Waste Audit	Free, no account	Pricing the waste in your existing Claude Code logs
Solo	$9/month or $90/year	One developer. Every tool except the team features.

Team	$19/seat/month or $190/seat/year, 2–100 seats	Everything in Solo, plus Team Relay, shared memory and fingerprints, and the Team workspace
Enterprise	From $799/month	Larger organizations: volume pricing, invoicing, onboarding

Every paid plan starts with a 30-day trial for $5, credited to your first payment. Full details are on Pricing and in the Terms.

Back to top · Next: Getting started

CHAPTER 2

Getting started

From nothing to a working setup takes about five minutes. This chapter goes through each step, the choices along the way, and how to update, move or remove TokenBuild later.

What you need

- **Claude Code**, with a Claude Pro, Max, Team or Enterprise plan, or an Anthropic API key. TokenBuild never asks for your Claude login and never uses your Claude account.
- **Node.js 20 or newer.** TokenBuild's tools run on it. The one-line installer installs it for you if it's missing.
- **Windows, macOS or Linux.** On Windows, both PowerShell and Git Bash work.
- **A git repository** is recommended. TokenBuild treats the folder that contains `.git` (or an existing `.tokenbuild` folder) as the project root. Change risk, Team Relay and some learning features use git history.

The free audit needs only Node.js and some past Claude Code sessions on the machine.

Step 1: See your waste (free)

Open a terminal and run:

```
npx tokenbuild audit
```

The audit reads the session logs Claude Code already keeps on your computer and prices the avoidable waste in the last 30 days: lockfile and build-output reads, oversized whole-file reads, repeated reads, noisy command output and more. It lists your costliest files and shows spend by model. Nothing is uploaded and no account is needed.

Useful options:

- `--days 7` looks at a shorter window.
- `--project myapp` limits it to sessions in one project folder.
- `--json` prints machine-readable output, with the top 20 files.

> **TIP**
> Run the audit again a few weeks after installing. The difference is a simple before-and-after you can show your team.

Step 2: Create an account and start a trial

1. Go to app.tokenbuild.ai/signup. Enter your email and a password, and agree to the Terms and Privacy Policy.
2. On **Billing**, choose **Solo** or **Team**, and **Monthly** or **Yearly** (yearly gives two months free). For Team, pick the number of seats (2–100).
3. Tick the auto-renewal agreement and choose **Continue to secure payment**. Stripe handles the payment; TokenBuild never sees your card.
4. You return to Billing with "Thanks! Your trial has started." It can take a few seconds for the plan to show.

The trial costs $5 for 30 days of every tool. On day 31 the plan you chose starts, and the $5 is credited to your first payment. You're emailed 7 days before the trial converts. Cancel any time before then and you won't be charged again.

Step 3: Install

Open the CLI key page and choose **Create key**, then confirm. The key is shown **once**, together with a one-line installer that already contains it. Copy the line for your system.

Windows: paste the line into PowerShell. It looks like this:

```
$env:TB_KEY='tb_…'; irm https://app.tokenbuild.ai/install.ps1 | iex
```

Mac or Linux: paste it into Terminal:

```
curl -fsSL https://app.tokenbuild.ai/install.sh | sh -s -- tb_…
```

The installer works in three steps and prints **OK** after each:

1. **Node.js.** It uses Node 20+ if you have it. Otherwise it installs the LTS release from an official source (on Windows, with winget).
2. **Claude Code's command-line tool.** It's installed from Anthropic's official installer if missing.
3. **TokenBuild.** It installs the plugin into Claude Code (or the Claude desktop app), saves your key in the plugin's secure settings, and confirms your plan with the server.

When it prints **All set**, restart the Claude app, or open a terminal in a project and type `claude`.

If you already have Node.js, this does the same thing:

```
npx tokenbuild install <key>
```

> **WATCH OUT**
> Save the key somewhere safe, such as a password manager, before you close the page. If you lose it, create a new one. The old key stops working immediately, so every computer that used it needs the new key.

If a step fails, the installer says what to do and saves a log: `%TEMP%\tokenbuild-install.log` on Windows, `$TMPDIR/tokenbuild-install.log` (or `/tmp`) on Mac and Linux. The log contains no keys or code, so it's safe to send to support. See Installation errors.

Step 4: Restart and check

Claude Code loads plugins when it starts, so restart it after installing. Then check that everything is on:

- In Claude Code, type `/tokenbuild:setup`, or
- in a terminal at your repo's root, run `npx tokenbuild doctor`.

Both show your plan and status, which tools are active in this repo, and the single most useful thing to do next, such as building the index or connecting your key.

From here, work as you always do. There are no new commands to learn. Your dashboard fills in as you work: it syncs every 30 minutes while a session is open, and again when the session ends.

Installing per project instead of the plugin

The plugin is the easiest choice and covers every project on your machine. Some teams prefer settings that live in the repo and are committed to git. For that, use project mode:

```
npx tokenbuild login <key>     # once per computer
npx tokenbuild init             # once per repo
```

`init` writes TokenBuild's hooks, MCP server, model routing and rules into the project's own `.claude` settings. It's safe to run again: it only ever touches its own entries. Options:

- `--route balanced` (default) runs Claude Code's helper subagents on Sonnet and the scout on Haiku. `--route lean` runs all helpers on Haiku, the cheapest option but weaker on hard subtasks. `--route off` leaves models alone.
- `--no-format` turns off formatting edited files with your formatter at the end of each reply.

If both the plugin and project hooks are present, the plugin steps aside in that project, so nothing runs or counts twice.

Updating

The quickest way:

```
npx tokenbuild release update
```

It updates the plugin to the latest version and keeps your CLI key. Or, in a terminal:

```
claude plugin marketplace update tokenbuild
claude plugin update tokenbuild@tokenbuild
```

Restart Claude Code after either one. If you use project mode, `npx tokenbuild init` again picks up the newest hook layout.

More computers, keys and CI

- **Another computer.** Install the same way with the same key. One key works on all your machines. If you've lost it, create a new key and reinstall, or run `npx tokenbuild login <new key>`, on each machine.
- **Changing the key in the plugin.** In Claude Code, run `/plugin configure tokenbuild@tokenbuild` and paste the new key, or run `npx tokenbuild login <key>`.
- **Where keys are looked for**, in this order: the `TOKENBUILD_KEY` environment variable, the plugin's settings, then the key file that `tokenbuild login` saves in `~/.tokenbuild`. If several keys are present, TokenBuild uses the one your current signed plan belongs to, so an old key left in one place can't switch the tools off.
- **CI and containers.** Set `TOKENBUILD_KEY` as a secret environment variable. Nothing is written to disk except the signed plan answer.
- **Signing out a computer.** `npx tokenbuild logout` deletes the saved key and plan answer on that machine.

Working offline

Your plan is confirmed online when you log in, run `init` or `doctor`, and when the MCP server starts with a Claude Code session. The server signs its answer; the CLI checks the signature with a public key built into TokenBuild, so the answer can't be forged. The signed answer lasts 14 days, so TokenBuild keeps working on a plane or behind a firewall. When you're back online, the next session renews it. Only dashboard sync needs a connection.

Uninstalling

1. In a terminal, run `claude plugin uninstall tokenbuild@tokenbuild`.
2. In project mode, also remove TokenBuild's entries from `.claude/settings.json`. They're the hooks whose command contains `tokenbuild`.
3. Optionally, delete the `.tokenbuild` folder in each repo, and `~/.tokenbuild`, to remove all local data.

4. To stop billing, cancel on Billing. Uninstalling doesn't cancel your subscription.

Recipes TokenBuild wrote under `.claude/skills/tb-recipe-*` are plain skills. Keep them or delete them.

Back to top · Next: What happens in a session

CHAPTER 3

What happens in a session

TokenBuild works through Claude Code's events. This chapter follows a session from start to finish and explains what TokenBuild does at each moment, what you'll see, and what it costs. Later chapters cover each feature in depth.

When a session starts

Claude Code fires a **SessionStart** event when you open a session, resume one, or type `/clear`. TokenBuild then:

- **Refreshes the Repo Index** in the background if files have changed, so `tb_find`, `tb_map` and the Task Brief point at current line numbers.
- **Injects the project state**: recent decisions, recent work, where work happens and what's still open. This happens on a new session or after `/clear`, and stays within about 600 tokens.
- **Injects a handoff note** if one is waiting, from Fresh Start, Cache Keeper, Context GC or a teammate's handoff. The new session starts from the note instead of from nothing. When a note is injected, only the durable parts of the project state are added alongside it, so nothing is said twice.
- **Starts background jobs**: the weekly Setup Check, and on Local Boost the daily outlines of your largest files.
- **Resets per-session state**: the Context Ledger and guard start fresh, because the new conversation hasn't read anything yet.

The MCP server starts at the same time. It confirms your plan online if the saved answer is due for renewal, runs the background sync if it's been six hours, and on Team plans starts the Team Relay heartbeat.

The project state block looks something like this at the top of a session:

```
# Project state (kept by TokenBuild)
## Decisions
- Use Postgres advisory locks for the job queue, not Redis
## Recently done
- Fix flaky upload test → src/upload.ts
## Where work happens
src/api/ (14), src/jobs/ (6)
## Outstanding
- failing check `npm test`
```

When you send a prompt

Before Claude sees your message, the **UserPromptSubmit** hook can add a few lines of context. Each part appears only when it has something useful to say:

- **The Task Brief**: likely files with line numbers, files usually opened with them, the tests to check, and past failures fixed in those files. It's built from the index and Repo Memory by plain code and typically costs about 150 tokens to read.
- **A clarifying nudge** if your request names nothing TokenBuild can match. Claude is asked to check one thing with you first, because a short question is cheaper than a long search in the wrong place.
- **The budget forecast**: how many tokens similar tasks took here, with an 80% range. With enough history, Autopilot replaces it with a split across planning, coding, testing and correction.
- **Saved answers** from the Exploration Memo that relate to your request and whose files haven't changed.
- **A learned tip**, at most once per session, such as "use tb_find before Grep in this repo". Tips that are ignored five times are dropped.
- **Change risk**, when you ask Claude to push, merge, deploy or ship and the risk isn't low.
- **Team notes** on Team plans: a plan item assigned to you just became ready, or a teammate passed you a task.

The Cache Keeper also runs here. If you've been away long enough for the prompt cache to expire and the conversation is very large, it holds your message once and offers a smaller fresh

start. See Cache Keeper.

Before Claude uses a tool

The **PreToolUse** hook (`tokenbuild guard`) sees each tool call before it runs and can let it through, change it, or send back advice:

- **Reads.** Waste Guard stops reads of lockfiles, generated, vendored and binary files, and whole-file reads of files over about 15K tokens without a line range. The Context Ledger answers repeat reads of unchanged lines with "already in your context", or with only the lines that changed.
- **Edits and writes.** Edit Preflight applies the edit in memory and checks the syntax of JavaScript, TypeScript, JSON and Python. If the edit would break a file that was fine, Claude gets the exact error and line before anything is written.
- **Shell commands.** Noisy test, build and lint commands are wrapped with the Output Shrinker. This happens automatically only for commands that would run without a permission prompt anyway. Otherwise Claude is told the wrapped form to use.
- **Team Relay.** Before Claude edits a file a teammate is editing right now, it's told who and on which branch.

After Claude uses a tool

The **PostToolUse** hook records what happened:

- The Context Ledger notes which lines of which files are now in context. Claude's own edits are tracked too, so they're never mistaken for outside changes.
- After an edit, the **Verify Loop** works out from your imports which tests the change can affect and runs them in a detached background worker. The result reaches Claude as one line on a later turn.
- When a check fails, the error is **fingerprinted**. When it later passes, the files changed in between are recorded as the fix.
- The **Loop Breaker** counts identical failures and edits that undo each other.
- **Next-Read** offers the outlines of the files that usually come next after the one Claude just opened.

- Task State is updated with files changed and check results.

When Claude finishes a reply

The **Stop** hook runs once per reply:

- **Formatting.** Files Claude edited are formatted with your project's own formatter (Prettier, Black, gofmt and so on), if one is configured.
- **Failing checks.** If a Verify Loop check failed during the turn and hasn't been fixed, Claude is asked to keep going and fix it before handing back to you.
- **Decisions.** If Claude's last reply says "decided to …", the decision is added to the project state.
- **Warnings**, each at most once per task: the early warning (when a task looks like it's heading for trouble at 30% of its forecast), a phase running more than twice its Autopilot budget, or the conversation growing big enough that a fresh start is cheaper.
- **Fresh Start.** Past about 100K tokens of context, a handoff note is written, and you're told you can `/clear` and continue from it.
- **Measurement.** The Real Meter reads the token usage Claude Code recorded for the reply.

Before compaction, and at session end

PreCompact: before Claude Code compacts a conversation (automatically or with `/compact`), TokenBuild saves a handoff note, so the details compaction drops are still on disk. The Context Ledger resets, because the compacted conversation no longer holds those reads.

SessionEnd: TokenBuild closes the session's last task, so it is measured even when no prompt follows it (a one-prompt `claude -p` run, or `tokenbuild plan run`). It then starts a detached `sync --usage` that sends the session's counters to your dashboard, if anything changed. It doesn't delay closing Claude Code. Turn it off with `"syncOnEnd": false`.

What TokenBuild itself costs

Everything TokenBuild adds to the context is counted against its savings. The Task Brief is typically 100–200 tokens, the project state at most about 600, and tips and warnings a line each. MCP tool definitions add a fixed amount per turn, like any MCP server's.

The control groups (Chapter 6) measure features with their own cost included, so a feature that costs more than it saves shows up as a loss, and the learning layer turns it down in that repo.

Hooks are written to finish in under 50 milliseconds, not counting your formatter's own run time. They never wait on the network.

Back to top · Next: Token Saver: Claude reads less

CHAPTER 4

Token Saver: Claude reads less

The first of TokenBuild's three products, **Token Saver** (`tokenbuild save`), keeps text that wouldn't help Claude out of its context. This chapter covers what Claude reads: what is blocked, condensed, answered from memory or handed to a cheaper model. Each section explains how the feature works, what you and Claude see, how it's counted, and how to adjust it.

Waste Guard

What it does. Stops reads that are almost never useful, before the file enters Claude's context.

What it blocks:

- Dependency and build folders: `node_modules` , `.git` , `dist` , `build` , `out` , `coverage` , `.next` , `.nuxt` , `.turbo` , `.cache` , `vendor` , `target` , `__pycache__` , `.venv` and `venv` .
- Lockfiles: `package-lock.json` , `yarn.lock` , `pnpm-lock.yaml` , `bun.lockb` , `Cargo.lock` , `poetry.lock` , `Gemfile.lock` , `composer.lock` and `go.sum` .
- Minified files and source maps (`*.min.js` , `*.min.css` , `*.map`), images, archives, fonts, video and compiled binaries.
- **Oversized whole-file reads**: any file over about 60 KB (about 15K tokens) read without a line range.

What Claude sees. For a generated file: `TokenBuild: package-lock.json is generated/vendored (~48K tokens). Read the source file or package.json instead. Retry only if you truly need it.` For a big file, Claude is told its size and asked to get line numbers with `tb_find` and read only the ranges it needs. With Local Boost outlines on, it gets an outline of the file instead, made on your machine.

Overriding it. Each block happens once per file and range. If Claude asks for the same read again, it goes through. That's deliberate: sometimes the lockfile really is the problem.

How it's counted. A blocked read counts the estimated tokens of the file it kept out. A retried read is logged as an undo and subtracted.

Output Shrinker

What it does. Test runners, builds, package installs and linters can print thousands of lines, and every line Claude sees stays in its context for the rest of the session. The Output Shrinker runs the same command with the same exit code, but Claude sees only what matters: errors with their code frames, warnings, and the final summary.

How it's used. When Claude is about to run a noisy command, the guard wraps it as `tokenbuild run <command>`. It does this automatically only when the command would run without a permission prompt anyway: it's covered by an allow rule, you're in bypass mode, or it's a known test, build or lint runner in auto mode. It never wraps installs or commands with environment-variable prefixes on its own. Otherwise, Claude is told the wrapped form and can choose to use it.

You can use it yourself too:

```
npx tokenbuild run npm test
npx tokenbuild run -- pytest -x tests/
```

The full log is saved in `.tokenbuild/logs`, and the condensed output ends with its path, so Claude can read a specific part when it needs to.

Typical effect. On a failing test run, output usually shrinks by more than 90%. Successful runs often shrink to a single summary line.

How it's counted. Full log size minus what Claude was shown.

> **TIP**
> Add allow rules for your test and build commands (the Permission Learner can do it for you, see below). Then the Shrinker wraps them automatically, with no prompt and no instruction for Claude to follow.

Context Ledger

What it does. Keeps a private record, on your machine, of which lines of which files Claude has read in the current conversation. When Claude asks for lines it already has:

- If they're **unchanged**, Claude is told they're already in its context, with the line range. Nothing is re-sent.
- If the file **changed** since (you edited it, a formatter ran, a tool regenerated it), Claude gets only the changed lines, with line numbers.
- Claude's own edits are tracked, so they never count as outside changes.

When it resets. At a new session, `/clear` or compaction, because the conversation no longer holds those reads. `tokenbuild guard --reset` resets it by hand.

Override. If Claude really needs the full text again, for example after it summarized a file and lost detail, asking a second time gets it. That saving is taken back out of the ledger.

The learning layer never turns the Context Ledger off. Control groups still measure it in about 10% of sessions.

Repo Index, map, find and context packs

What it does. Builds a local map of your code: files, the symbols each defines (functions, classes, types, constants) with line numbers, and which files import which. Claude uses it to go straight to the right lines instead of exploring with many Grep, Glob and LS calls.

Languages: TypeScript, JavaScript, Python, Go, Ruby, Java, Kotlin, C#, Rust, PHP and Swift. Other files are still covered by Waste Guard.

Four ways to use it:

Use	CLI	Tool for Claude	Returns
Build or refresh	`tokenbuild index`	(automatic at session start)	File and symbol counts

Overview	`tokenbuild map --budget 2500`	`tb_map`	The most-imported files and their main symbols, under the token budget
Locate	`tokenbuild find <name>`	`tb_find`	`path:line` for a symbol or file
Context pack	`tokenbuild pack "<task>" --budget 4000`	`tb_pack`	The files and symbols a task most likely needs, with line numbers

Packs are ranked with Repo Memory, so files that past tasks like this one actually changed come first.

> **TIP**
> In a new or unfamiliar repo, ask Claude to "use tb_map first". A 2,500-token map replaces most of the exploration Claude would otherwise do over its first several turns.

Task Brief

What it does. When you send a prompt, TokenBuild builds a short brief before Claude starts:

```
TokenBuild brief:
Likely files: src/billing/invoice.ts (createInvoice:42, applyCredit:118); src/billing/stripe.ts (web
Related: src/billing/invoice.test.ts
Check: npm test -- src/billing (node --test)
Past fix here: "Cannot read properties of undefined (reading 'amount')" → src/billing/stripe.ts
```

Where it comes from. Words in your prompt matched against the Repo Index, files that similar past tasks changed (Repo Memory), files usually opened together, the test files linked to those files, and error fingerprints recorded there. The file list is re-ranked by a model trained on which files tasks in this repo really opened and edited (Chapter 11).

Size. About 150 tokens. The learning layer picks how many likely files to list (3, 5 or 8) for each kind of task. Pin it with `"briefSize"`.

Vague requests. If your prompt names nothing TokenBuild can match, such as "it's broken again", the brief nudges Claude to ask you one short question first.

Turning it off. `"brief": false`. The learning layer may also hold it off for kinds of tasks where it doesn't pay in your repo.

> **TIP**
> Name a file, function, route or error message in your prompt when you can. "Fix the 500 on `POST /invoices` when credit is applied" gives the brief much more to match than "fix the invoice bug".

Next-Read

When Claude opens a file, Next-Read checks which files usually come next in this repo, from Repo Memory and the Real Meter's read history. It offers their outlines (symbols and line numbers) in one short line, so Claude can jump to the right lines instead of opening each file whole. Turn it off with `"predict": false`. Like the brief, it's measured and learned per kind of task.

Exploration Memo

Answers Claude works out about your repo, such as "where are webhooks verified?" or "how does auth middleware get the user?", are saved with the files they're based on. `tb_memo` returns a saved answer only while those files are unchanged. As soon as one changes, the answer is treated as stale. Questions that took several searches to answer are saved automatically. On Team plans at relay level 2, saved answers are shared with teammates.

Docs Oracle

Claude's knowledge of a library may be a version or two behind the one you installed. The Docs Oracle answers from the version in your project:

- **JavaScript and TypeScript**: type declarations and doc comments from `node_modules`. Use `"pkg#name"`, for example `tokenbuild docs "zod#object"`.
- **Python**: docstrings from the installed package.
- **Go**: `go doc`.

Claude calls it as `tb_docs`. It's cheaper than reading library source and more accurate than memory.

Fingerprints

What it does. When a check fails, TokenBuild reduces the error to a fingerprint: numbers, quoted values and machine-specific paths are removed, then the rest is hashed. When that check later passes, the files changed in between are recorded as the fix. The next time the same error appears, in any session or on any branch, Claude is pointed at those files first.

Team plans share fingerprints as one-way hashes with file paths and counts, so one teammate's fix helps everyone. The error text never leaves your machine.

Shared fingerprints across users. When an error was fixed by changing a well-known config file (`package.json`, `tsconfig.json`, `pyproject.toml` and similar), the hash and that file name are shared. The server keeps an entry once at least two teams fixed it that way. If you hit the error with no fix of your own on record, Claude is told which config files fixed it for others, and by how many teams. Turn it off with `"shareFingerprints": false`.

Local Boost

What it does. An optional helper that uses a free, open model running on your own computer through Ollama. Claude can ask it a question about a large file or log (`tb_digest`) and get a short answer with line numbers, so the large text never enters Claude's context. It costs zero Claude tokens.

Setup:

1. Install Ollama and pull a coding model, for example `ollama pull qwen2.5-coder:7b`. It runs on most laptops with 16 GB of memory.
2. Run `tokenbuild boost on`, or `/tokenbuild:boost`. Pick a different model with `--model name`.
3. Try it: `tokenbuild ask "where is the retry logic?" src/client.ts`.

Boost Outlines. With Boost on, a daily background job outlines your largest files. When Waste Guard stops a whole-file read of one, Claude gets the outline immediately, as long as the file

hasn't changed since, which saves a `tb_find` round. Turn it off with `"outlines": false`.

Privacy. Local Boost only ever connects to your own machine (localhost). This is enforced in code. Your code still never leaves your computer.

Limits. Small local models are good at reading and locating, not at writing code. TokenBuild uses them only to digest text for Claude.

Model routing and the scout

Your main session keeps the model you chose. Routing sets the model only for Claude Code's helper subagents that don't name one, plus TokenBuild's read-only **scout** subagent for finding things and reading big files.

Route	Helper subagents	Scout	When to use
`balanced` (default)	Sonnet	Haiku	Most teams
`lean`	Haiku	Haiku	Cheapest; weaker on hard subtasks
`off`	Unchanged	Unchanged	You manage models yourself

If you've already chosen a helper model yourself, TokenBuild keeps your choice. The audit shows spend by model, so you can see the effect.

Back to top · Next: Token Saver: Claude writes less and wastes fewer turns

CHAPTER 5

Token Saver: Claude writes less and wastes fewer turns

Every turn re-sends the conversation, and output tokens cost the most. These Token Saver features cut what Claude writes, catch broken edits and loops before they cost turns, and keep the fixed cost of every turn small.

Anchor Edits and Codemod Mode

Anchor Edits (`tb_edit`). With Claude Code's standard Edit tool, Claude writes the exact old text and then the new text, so every edit pays output tokens, the most expensive kind, to repeat code that already exists. With `tb_edit`, Claude names a line range it has already read and sends only the new lines. TokenBuild applies the change only if those lines are unchanged since Claude read them, and refuses an edit that would break the syntax.

Codemod Mode (`tb_apply`). One find-and-replace (text or pattern) across many files in one call, applied locally. Claude gets a compact summary of what changed where, not every file's diff. Use it for renames, import path moves and API migrations.

> **WATCH OUT**
> Codemods are powerful. Ask Claude to show the match count or a dry run first on anything broad, and commit before large changes so `git diff` and `git checkout` can review or undo them.

Edit Preflight

Before an edit lands, TokenBuild applies it in memory and checks the syntax of JavaScript, TypeScript, JSON and Python files. If the edit would break a file that was fine before, it goes straight back to Claude with the exact error and line:

```
TokenBuild preflight: this Edit would leave src/api.ts with a syntax error at line 88: ')' expected.
```

```
Fix the edit and send it again. If the broken state is intentional, resend the same edit.
```

That replaces an edit, run, read error and fix cycle, which can take several full turns. If the broken state is intentional, for example halfway through a two-step change, resending the same edit lets it through. Edit Preflight is never turned off by learning. Set `"preflight": false` to disable it.

Loop Breaker

Claude sometimes gets stuck: it runs the same failing test again and again with small edits that don't touch the cause, or undoes and redoes its own change. Each lap re-sends the whole conversation. The Loop Breaker watches for:

- the **same command failing with the same error three times**, and
- Claude **reverting its own edit** in a file repeatedly.

It then tells Claude what was already tried and asks it to state what each attempt was trying to fix, pick one root cause, and make one deliberate change, or ask you. It speaks once per problem.

> **TIP**
> When you see a Loop Breaker message, it's often a good time to step in with a hint. You know things about the system that the logs don't show.

Fresh Start

The problem. Every reply re-sends the whole conversation. A session at 150K tokens pays for 150K tokens of input on every turn, even with caching, and long contexts also make models less precise.

What it does. When the conversation passes about 100K tokens (the `handoffAt` setting), TokenBuild writes a handoff note from the session: the goal, your latest requests, the plan, files changed, which checks pass or fail, verified facts, open questions and where work left off. It tells you once. When you type `/clear`, the new session starts from the note, typically 1–2K tokens.

By hand. Run `tokenbuild handoff` or `/tokenbuild:handoff` any time, for example before switching tasks or at the end of the day. A note is also saved automatically before compaction.

How it's counted. The context the new session no longer carries.

> **TIP**
> A good moment to `/clear` is right after a verified milestone: tests pass and the change is committed. The note is then short and accurate, and the new session starts clean.

Adaptive Fresh Start

The problem. A fixed size limit is a blunt rule. Sometimes a 60K conversation has moved on to a new topic and is all dead weight. Sometimes a 120K one is in the middle of a delicate change, and clearing it would only make Claude read everything again.

What it does. Every 5 prompts (you choose: any whole number from 1 to 10000), TokenBuild decides whether this conversation is better continued or restarted from a handoff note, and tells you why in one line. It doesn't cost any of Claude's tokens. The line gives the context size, how much it grew since the last check, how much of your request is new to the conversation, whether the prompt cache has expired, and what the model expects a fresh start to save. When the answer is `/clear`, the handoff note is already saved. Conversations under about 20K tokens get no advice (`freshMin`).

How it learns. Each decision is logged locally with its context. Afterwards TokenBuild measures what happened: tokens per reply over the next 5 tasks compared with the 5 before, following a `/clear` into the new session. Tasks that failed or had to be redone count against the decision, so a fresh start that loses needed context doesn't pay. Ignoring a `/clear` recommendation counts slightly against it, so you're asked less often if you usually keep going. A Bayesian model for each choice learns from these outcomes and picks by Thompson sampling. Until it has 8 measured outcomes of each choice, a starting rule decides, and 10% of those decisions go the other way so the model sees both.

Judged on whole tasks (0.20.4). A cleared session can be cheaper per reply but need more replies to rebuild what it lost. So `tokenbuild fresh` also compares tokens per *finished task* after a followed clear with tokens per task after continuing, including the new session's rereads,

with a 95% interval. Until both sides have enough outcomes, or while the interval includes no difference, it says **insufficient evidence** rather than picking a side.

Settings. `tokenbuild fresh --every 10` sets the interval. `tokenbuild fresh off` and `fresh on` switch it, and `tokenbuild fresh` shows the decisions, how often you followed them, the measured savings and whether the model has taken over from the starting rule. While it's on, the fixed reminder past `handoffAt` follows its last decision: no reminder after "continue normally".

How it's counted. Measured from your own task records: per-reply tokens before and after each followed `/clear`. The savings themselves are credited the same way as Fresh Start, from the context the new session no longer carries.

Context Profiler and Context GC

Context Profiler (`tokenbuild profile`, `tb_profile`) shows what fills Claude's context right now, by category: files read, command output, search results, tool definitions, conversation text. It also shows how much is **dead weight**: content nothing has used since it arrived.

Context GC. When a large conversation is mostly dead weight, TokenBuild saves a clean note with only the live parts (the files still in use, verified facts, the plan) and suggests `/clear`. You continue at a fraction of the size, without losing the thread.

Cache Keeper

The problem. Claude Code caches the conversation so re-reading it costs a fraction of the input price. The cache expires after a few idle minutes. Coming back to a huge conversation after a break means your next message re-caches all of it at a higher price.

What it does. TokenBuild learns how long the cache lasts from your own sessions. When you send a message after the cache has likely expired, on a conversation big enough for it to matter, it saves a handoff note first and holds the message once. Claude explains: you were away N minutes, re-caching would cost about X tokens, and `/clear` followed by sending your message again starts small.

Your choice. Type `/clear` and resend for the small start, or just send the same message again to carry on where you are. Turn it off with `"cacheKeeper": false`.

Tool Diet, CLAUDE.md Compiler and Permission Learner

These three tune your Claude Code setup to how you actually work. Each shows what it would change and changes nothing until you add `--apply`.

- **Tool Diet** (`tokenbuild diet`). Every MCP server's tool list is sent on every turn. Diet measures how many tokens each server adds against how often you use it, and lists unused ones. `--apply` disables the project-level ones for this project. It needs at least 5 sessions of usage before it suggests removals.
- **CLAUDE.md Compiler** (`tokenbuild claudemd`). A root `CLAUDE.md` loads in every session. Sections that only matter in one folder ("in `ios/`, use…") can move into that folder's own `CLAUDE.md`, which Claude Code loads only when it works there.
- **Permission Learner** (`tokenbuild permissions`). Commands you approve every time become allow rules. That saves your time, and lets the Output Shrinker wrap them automatically.

The weekly **Setup Check** runs these in the background and suggests findings. With `"autoTune": true` it applies them for you.

Back to top · Next: Token Saver: measuring savings

CHAPTER 6

Token Saver: measuring savings

TokenBuild's rule is simple: a saving appears only if something counted it. This chapter covers the savings ledger, the Real Meter, the Context Profiler and the dashboard sync. The control-group proof of each feature's effect is part of ML Validation and Economics (Chapter 12).

The savings ledger (`tokenbuild stats`)

Every time a tool keeps tokens out of Claude's context, an event is written to `.tokenbuild/ledger.jsonl` with the tokens it kept out. `tokenbuild stats` (or `/tokenbuild:stats`) reports them for this repo in **three tiers that are never added together** (0.20.4):

```
Savings in this repo, in three tiers that are never added together:
  Measured   48.2M tokens used by Claude Code (~$61.40 at API prices) (Real Meter)
  Estimated  ~3.9M tokens kept out of context: Waste Guard ~640K, Context Ledger ~1.9M, Output Shrin
             avoided text, not proven bill savings; the parts can overlap
  Proven     ~2.1M tokens over 184 tasks (control groups, credited only when the 95% interval exclud
  Overhead   ~96K tokens of context TokenBuild added (Project State ~41K, hooks ~55K; tokenbuild ove
  Break-even last 30 days: proven $2.67 − overhead $0.12 = $2.55 vs the $9/month fee: below the plan
```

- **Measured** is what Claude Code actually used, read from its own session records.
- **Estimated** is text TokenBuild kept out of context. Each event is counted once, but the tools can overlap (a read Waste Guard blocked might never have been made), and avoided text is not the same as a lower bill. So it is labelled an estimate, and no headline adds it to anything else.
- **Proven** is the control-group credit (Chapter 12): only a feature whose 95% interval shows a real effect is credited, and at the low end of that interval.
- **Overhead** is the context TokenBuild itself added. **Break-even** prices the proven tier, minus overhead, at your own metered cost per token and compares it with your plan fee. It can

honestly say "below the plan fee" or "no proven saving yet".

The per-tool activity list follows below. `tokenbuild stats --json` prints the tiers for scripts.

How each is counted:

Tool	Counted as
Waste Guard	Estimated tokens of the file a blocked read would have sent
Output Shrinker	Full log minus what Claude was shown
Context Ledger	Lines not re-sent, or the unchanged part of a changed file
Local Boost	The file the local model read, minus its short answer
Anchor Edits	Old text Claude didn't have to write out
Fresh Start, Context GC	Context the new session no longer carries
Brief, Verify Loop, Next-Read, Edit Preflight	Proven tier: credited from the control-group proof (Chapter 12) only when its 95% interval excludes no effect, at the interval's low end, capped at a 50% cut
Auto-format, repo lookups	Uses only. They save turns, not a countable number of tokens.

When Claude re-requests something TokenBuild held back, the event is logged as an undo and subtracted.

The Real Meter (`tokenbuild meter`)

Claude Code records the exact token usage of every reply in its session transcript on your machine: new input, cache reads, cache writes and output. The Real Meter reads those records incrementally, without sending any transcript text anywhere, and shows real tokens and cost per task and in total for this repo.

- Costs are shown at API list prices. On Pro or Max you pay a subscription instead, but the same tokens count toward your usage limits.
- `--json` gives machine-readable output.
- Per task, it also records the kind of task, which `tb_*` tools and subagents were used, tokens by phase (plan, code, test, fix), and whether the last check passed.

Context Profiler (`tokenbuild profile`)

A snapshot of the current conversation: what's in the context window by category, and how much is dead weight. Use it when a session feels slow or expensive, to decide whether to `/clear`. See Context GC.

Sync and the dashboard

Counts reach your dashboard four ways:

- every 30 minutes while a session is open, if the ledger changed (`"syncEvery"`, in minutes; 0 turns it off),
- at session end (`"syncOnEnd"`),
- on a schedule, with no session open, once you turn on auto-sync: `npx tokenbuild autosync on --every 15` (Task Scheduler on Windows, cron on macOS and Linux; it covers every repo TokenBuild has run in, and logs one line per run to `~/.tokenbuild/autosync.log`; `autosync status` shows it, `autosync off` removes it), and
- on demand with `npx tokenbuild sync` (this repo) or `npx tokenbuild sync --all` (every repo).

A fuller background sync also runs every 6 hours when Claude Code starts. Sync sends counts only: per-tool daily totals, per-task token totals for the proof, noised learning counts and error hashes. On Team plans, it also sends file paths and the other Team Relay data your relay level allows. Never code, file contents, prompts or error text.

Reading the numbers honestly

- **"Tokens kept out" isn't the same as "money saved".** A token kept out early in a long session would have been re-sent on every later turn, mostly at cache prices. The Real Meter shows what you actually spent. The proof shows the difference per task.
- **Small repos save less.** If your tasks are already small, there's less to cut. The free audit tells you upfront how much waste there is to remove.
- **The first week is learning.** Policies, rankings and test selection improve as history builds. Compare weeks, not days.
- **If you're not saving,** the dashboard shows it plainly. Cancel in one click; see Chapter 20 first for the usual causes.

TokenBuild's own overhead (`tokenbuild overhead`)

Every hook and tool has a cost: time on each Claude Code event, context it adds, and messages that interrupt you. Since 0.20.4 TokenBuild measures its own cost and keeps to a budget:

```
TokenBuild's own overhead in this repo (4.1K hook runs over 63 sessions):
  post         2210 runs · p50 6 ms · p95 21 ms · ~31.0K tokens added
  prompt        980 runs · p50 9 ms · p95 34 ms · ~18.2K tokens added
  ...
  Context added: ~55.0K tokens in all, ~873 per session (budget 12.0K)
  Nudges: 1.4 per session (budget 5)
  All hooks within the 50 ms budget at p95.
```

- **Latency**: each hook's own time, against the 50 ms rule. A hook over budget at the 95th percentile is named, with what to do.
- **Context**: text hooks add to Claude's context, about 4 characters per token. It also counts toward the Overhead line in `tokenbuild stats`.
- **Nudges**: messages shown to you from the prompt, stop and edit hooks. After `"nudgeBudget"` of them in a session (default 5), further ones are held back. Information Claude needs, such as a failing test, is context, not a nudge, and is never held back. The SessionStart summary isn't counted.

Everything is local (`.tokenbuild/state/overhead.jsonl`). Settings: `"nudgeBudget"` (0 = no nudges, `false` = no limit) and `"contextBudget"` (tokens per session to report against,

default 12000).

Back to top · Next: Team Build: task state and planning

CHAPTER 7

Team Build: task state and planning

The second product, **Team Build** (`tokenbuild team`), keeps the plan and its state outside the conversation, so a new session, a subagent or a teammate picks up exactly where the work stopped. This chapter covers the parts that work for one developer too: Task State, Project State, Project Preflight and Autopilot budgets.

Task State

TokenBuild keeps a live record of the current task: the goal, the plan, files changed, check results, verified facts and open questions. See it with `tokenbuild task` ; clear it with `tokenbuild task --reset` .

- A **new session or a subagent** reads it with `tb_task` and resumes immediately, instead of re-reading files to work out where things stand.
- **Parallel agents** claim plan items before working on them, so two agents never do the same work.
- **Handoff notes** include it.
- Claude can mark the task done with `tb_task done` , which reports whether it's verified.

Project State

Task State covers the current task. Project State is the durable layer above it, the things worth knowing at the start of any session in this repo:

- **Decisions**: recorded with `tokenbuild state --decide "use X, not Y"` or `tb_task decide` , or picked up when Claude's last reply says "decided to …". Remove one with `tokenbuild state --forget <n>` .
- **Recently done**: from measured tasks.
- **Where work happens**: the folders with the most recent activity.

- **Outstanding**: failing checks and open questions.

It's injected at the start of new sessions and after `/clear`, within about 600 tokens. Turn it off with `"state": false`.

> **TIP**
> Record decisions that Claude would otherwise re-debate: "we don't use an ORM here", "dates are stored in UTC", "the mobile app is frozen until March". One line saves an argument in every future session.

Project Preflight

`tokenbuild preflight "<goal>"` (or `tb_preflight`) plans a project before you start. For each step, it lists:

- the **files** likely to change,
- the **tests** that cover them,
- a token **estimate** from similar past tasks,
- **dependencies** on other steps, and
- the **stage** each step falls into (steps that can run at the same time share a stage).

Give your own steps with `--steps "add the schema; write the endpoint; update the client"`, or let it propose them. On Team plans, Claude can share the result straight to the team plan.

Project Preflight is different from Edit Preflight, which syntax-checks individual edits.

Autopilot budgets

`tokenbuild autopilot "<task>"` (or `tb_autopilot`) gives a token budget for a task, split across planning, coding, testing and correction. It's learned from the most efficient half of similar tasks in this repo, and calibrated against how past budgets turned out. Each figure comes with an 80% range.

```
Autopilot: ~38K tokens (80%: 24K–61K)
  plan 6K · code 17K · test 9K · fix 6K
```

Pass `--budget 30000` to fit the split to a budget you set. Once there's phase history, the Autopilot line replaces the plain forecast in the brief. During the task, if a phase runs more than twice its budget, Claude and you are told once.

Back to top · Next: Team Build: Team Relay and the team workspace

CHAPTER 8

Team Build: Team Relay and the team workspace

Team plans connect your teammates' Claude Code sessions, so the team's Claudes share what they learn, don't collide, and don't repeat each other's work. This chapter covers Team Relay, the Team workspace, plans and stages, handoffs, and the coordination insights.

Team Relay

While Claude Code is open, the TokenBuild MCP server sends a small heartbeat about once a minute. Hooks never call the network: they read the latest team view from `.tokenbuild/relay.json`. Relay starts automatically when a teammate opens Claude Code in a repo with a git remote. Teammates are matched by repo.

What Relay does:

- **Conflict radar.** Before Claude edits a file a teammate is editing right now, it's told who and on which branch, so it keeps the change small, works in a different part of the file, or picks other work.
- **Shared plan.** `tokenbuild relay share` puts your plan on the team plan. Each teammate's Claude claims items before working on them, so no two do the same work.
- **Baton pass.** `tokenbuild relay send maya --note "auth done, tests left"` hands your task state and handoff note to Maya. Her next Claude prompt in that repo starts from it.
- **Ask the expert.** When Claude is stuck on an error a teammate fixed before, it's pointed at that teammate and the files they changed.
- **Team board.** Who's working where, who's stuck, the plan, and tasks waiting to be picked up.

From the CLI: `tokenbuild relay` shows teammates' live sessions in this repo; `relay claim <n>` and `relay done <n>` claim or finish plan step *n*; `relay name <you>` sets the name teammates see (your git user name by default).

Relay levels: what's shared

	Level 1 (default)	Level 2
Display name, repo folder name, branch	✓	✓
Paths of recently edited files	✓	✓
One-way hashes of errors	✓	✓
Task goals and plan item titles		✓
Saved answers about the repo (memos)		✓
Handoff notes of tasks you pass on		✓
Code, file contents, prompts	Never	Never

An admin sets the level for the whole team on the Team dashboard's Overview, or on the main dashboard's Team Relay card. The board, shared plans and handoffs need level 2. To turn Relay off for one repo, set `"relay": false` in that repo's `.tokenbuild/config.json`.

Setting up a team

1. **Create the team.** Choose Team at signup, or upgrade from Solo on Billing or the Team dashboard (prorated).
2. **Each teammate installs TokenBuild** as in Chapter 2, with their own account.
3. **Invite people.** In **People**, choose **Invite people** and send the link or code. Teammates can also ask to join using the owner's email address.
4. **Approve requests.** Every join request waits for an admin, who approves it as **coder** or **viewer**.
5. **Turn on level 2** on the Overview, if you want plans and handoffs.
6. **Add projects** (below).

Roles

Role	Can
Admin	Everything coders can, plus: approve join requests, change roles, remove people, invite, set the Team Relay level, manage billing, approve, merge and revoke Team Knowledge, connect GitHub, set retention, export or delete data, and read the audit log. The owner (the paying account) is always an admin.
Coder	Add, assign and finish plan items; set dependencies, specs and stages; deploy stages; hand off tasks; review and send handoff contracts; act on or dismiss Conflict Radar alerts; write Team Knowledge drafts and correct their own; add projects. Whoever adds a project is its admin.
Viewer	See the board, plans, pull requests, alerts, knowledge, people, handoffs and savings. Can't change anything.

Anyone can leave a team from **People** → **Leave team**. A person can be on one team at a time.

The Team workspace

The Team dashboard (/team) has five main views, plus People and Admin:

- **Overview**: where each data source stands (GitHub, Claude sessions, usage, plans: *confirmed*, *stale* or *unavailable*, never a guess), six widgets (active work, verified completions, review blockers, at-risk dependencies, potential collisions, spend anomalies) and **Prioritized actions**, the Team Intelligence Brief.
- **Work & Planning**: tasks by teammate, the Conflict Radar, pull requests linked to plan items and sessions, and suggested allocation. Projects keep each plan, its stages and progress.
- **Agents & Sessions**: live Claude sessions (repo, branch, goal, stuck or at risk), handoff contracts to review and send, and quick handoffs.
- **Economics & Quality**: cost per verified task, delivery quality, Conflict Radar accuracy, handoff quality and the coordination models.
- **Team Knowledge**: decisions, verified solutions and known failure modes your team agreed on.

- **People**: members, roles, pending requests and invites. **Admin** (admins only): GitHub, controls (kill switch, radar blocking, budget), workflow experiments, retention, export and deletion, and the audit log.

GitHub

An admin connects GitHub in **Admin** → **GitHub**. It's a repository or organization webhook: TokenBuild gets a payload URL and a secret (shown once), and you add them in GitHub under Settings → Webhooks, with content type `application/json` and the pull request, review, push, check run, check suite, status and delete events. This needs no token and gives no code access. Commit messages and PR bodies are dropped on arrival. PR titles are kept only at Relay level 2.

Every delivery is checked against the secret. Its delivery id is its key, so a redelivered webhook never makes a second record. The dashboard rebuilds its view from these events every time it loads, so replaying them always gives the same state. **Disconnect** stops collection at once, and **New secret** rotates it. The panel shows deliveries, the last one, the repos seen and any deliveries with a bad signature.

A pull request is linked to the Claude sessions on its branch, and to a plan item. The item is either tagged in the branch name or title as `tb-` plus the first 6 characters of its id, or it's the one item in progress in a session on that branch. Open PRs with changes requested, failing CI, or no review a day after they were ready are listed as review blockers.

Conflict Radar

The radar checks four rules. Each alert explains itself, names its owners and shows its evidence:

- **Overlapping changes**: two people's current work touches the same file. It compares live session edits, the files a session's brief predicted, open PR branches and in-progress plan items. High confidence when both are actual edits.
- **Same high-risk module**: different files in a folder where fixes for past failures landed, or files your team's history says change together.
- **Conflicting ownership**: an item assigned to one person is being worked on by another, or two in-progress items name the same files.

- **Out-of-order work**: an item was started, or has an open PR, before the items it comes after are done.
- **Shared API contract**: two people change API definitions (api/, schema, OpenAPI, protobuf, GraphQL, .d.ts files), even different ones.
- **Overlapping migrations**: two people add database migrations at once.
- **Duplicate exploration**: two sessions are headed for the same files, with nothing changed yet and goals in common.

The same people on the same evidence make one alert, whichever sources saw it. Coders can act on an alert (**Share context**, **Sequence**, **Narrow scope**, **Assign an owner**) or dismiss it with a reason. A dismissed alert comes back only if its evidence changes. **Economics & Quality** shows how the radar is doing: alerts raised, acted on and dismissed (by reason), precision, the lead time before the merge, and missed collisions. A miss is a merged PR that overlapped another person's open PR with no alert first. The rules are checked against a fixed set of labelled collision scenarios on every release. The learned collision estimate (below) stays alongside as a second opinion.

Team Knowledge

Shared Engineering Memory: records of four types (decision, verified solution, known failure mode, note). Each has a title, details, the project and paths it covers, an owner, a version history, a review date and an optional expiry. Each also has its provenance: where it came from (the dashboard, a Claude session or a handoff), who wrote it, the task, the PR and the tests that back it.

A Claude session's stated decisions ("decided to …", `tb_task decide`) arrive as **drafts**. Nothing an agent said reaches another session as fact until a team admin approves it. Approved, unexpired records reach Claude sessions **in their own repo only**, ranked by the files the session is working on and its goal, once per change. Owners and admins can correct a record (a new version), mark it reviewed or revoke it. Admins merge duplicates, and likely duplicates are flagged. Needs Relay level 2. Turn the session side off with `"teamKnowledge": false`.

Handoff contracts

A Verified Handoff Contract has ten fields: objective, owner, recipient, changed files, constraints, decisions, acceptance criteria, tests, unresolved risks and next actions. Each one holds a value or an explicit *unknown*, never a silent blank. In a session, `tokenbuild team relay send <name> --contract` drafts one from local state: the goal, files changed, recent decisions, real test runs with their times, failing checks and open questions, and the plan item's spec. It's uploaded as a draft. `tokenbuild handoff --contract` prints it without sending anything.

A person reviews the draft in **Agents & Sessions**. Missing fields are highlighted, and **Review and send** refuses until each is filled in or marked unknown. A contract is **verified** only when its tests passed after the last edit and within the last 48 hours. Older evidence shows as *stale*. When the receiver's Claude session picks it up, the contract is marked picked up, linked to where it came from. The receiver's Claude is asked to run the contract's tests first. Test evidence carries the repository revision it ran at. The first check after pickup is reported back as the **resumption check**, and the time to resume is measured. You can also write a contract on the dashboard. **Economics & Quality** shows the share with every field filled, verified and picked up.

Intelligence Brief and suggested allocation

The **Prioritized actions** on the Overview are a digest: open collisions, stuck reviews, items waiting on work that has stalled for over 3 days, duplicate claims, two people chasing the same error (the same error fingerprint), handoffs without fresh test evidence, and spend anomalies. A spend anomaly is a day at 3 times or more a person's own usual, which often means a loop. Each entry has an owner, the reason, a confidence, the expected benefit and a link to its evidence.

Suggested allocation (Work & Planning) is advisory:

- **AI**: the item is ready, scoped with acceptance criteria and named files, in an area with verified work.
- **Paired**: it touches a high-risk module, or there's no verified work in that area yet.
- **Human**: it needs scoping first.

Each suggestion comes with the suggested person. **Accept** assigns that person, and **Reject** just records it. Nothing is reassigned automatically.

Cost per verified task keeps metered (observed) tokens apart from estimates. Dollar figures are API-price equivalents, not billed spend, because subscription plans aren't billed per token. People are never ranked by token use.

Engineering Intelligence Graph and evidence grades

Everything the dashboard knows about a repository forms one graph. People, Claude sessions, plan items, files, pull requests, checks, knowledge records and handoffs are linked by typed edges: *owns*, *modifies*, *depends on*, *blocks*, *verifies*, *contradicts*, *supersedes*, *learned from* and *handed to*. The graph is rebuilt from the stored events each time, never kept as a second copy. **Work & Planning** → **Engineering graph** shows it per project: edges by type, the most-connected files, and each file's neighbours.

Every edge, alert, brief item and knowledge record carries an **evidence grade**. A grade is never raised without new evidence:

- **observed**: seen directly, such as an edit, a claim or a webhook.
- **inferred**: derived from other signals, such as a predicted file or a shared module.
- **proposed**: someone or some agent said so, not confirmed yet.
- **verified**: backed by tests or checks.
- **invalidated**: no longer true: stale, revoked or superseded.

Each insight also shows when its evidence was last seen, its confidence and its owner. Graph queries are limited to one repository of your own team.

Knowledge that stays current

A knowledge record covers paths. When a pull request that changed any of those paths merges after the record was last validated, the record becomes **stale**. It's no longer given to Claude, and it appears in the revalidation queue and the Intelligence Brief until its owner or an admin revalidates or revokes it. Records can be marked **superseded by** a newer one. Records that likely contradict each other are flagged. An admin can mark a pair as contradicting, and contested records aren't served until one side is resolved.

Claude sessions get a **context pack**: the fewest relevant records (up to 3), each linked to its evidence (PR, tests) rather than whole conversations, plus any open Conflict Radar alert on the

same files. Every change in what a session is served is recorded (who, which repo, which records), and each record shows how often it was served. A session's stated decisions are shared as drafts only from machines whose config has `"shareDecisions": true`.

Workflow experiments

Admins can test a workflow change on part of the team before adopting it (**Admin** → **Workflow experiments**). A proposal declares, in advance:

- the hypothesis and expected benefit
- the metric: verified share, tokens per verified item, or unverified share
- the cohort (projects), and the intervention: a setting such as the Adaptive Fresh Start interval, Team Knowledge context packs or the brief confidence gate
- the measurement window, the minimum number of items per group, and when to roll back

An admin approves and starts it. Half of the teammates, chosen by a stable random hash, get the setting on their next heartbeat. A setting someone chose themselves in `config.json` always wins, and `"experiments": false` opts a machine out.

A separate **Judge** scores the result from finished plan items. It never sees the expected benefit. It reports the effect with a 95% interval, sample sizes, the verified-rate guardrail and caveats (few people, an open window). Its verdict is *pass*, *fail*, *insufficient* or *rollback*. Only a *pass* can be promoted, and only by an admin. An experiment that meets its rollback criterion is rolled back automatically. The **kill switch** stops every experiment and setting at once.

Delivery forecast (research)

For a project's remaining plan, a simulation forecasts how long it takes with 1 to 6 people working in parallel: median, 80% and 90%. It uses the plan's dependencies and your team's own claim-to-done times, and it adds rework where work in flight shares files. It shows the critical path, which no amount of parallel work gets under, and where adding people stops helping. A backtest replays finished plans and reports how often the actual time fell inside the forecast's 10–90% range (target 80%). It's a research prototype: advisory only, and it states when it's working from a default assumption because there's too little history.

Controls: blocking, budget and reconciliation

Radar blocking (off by default): a plan claim is refused when another person is already changing the same files, in their in-progress item or an open PR. Only these high-confidence cases block; every other alert stays advisory.

Monthly token budget: the brief warns at 80% of metered tokens.

Reconcile (GitHub): replays the stored events and lists open PRs with no event for 14 days and deliveries with a bad signature, with instructions to redeliver from GitHub. Redelivered events are stored once.

Allocation now shows ranked execution plans for each ready item: AI, paired or human, each with a workflow, confidence, trade-off and prerequisite checks. Confidence comes from how this team's accepted suggestions of that kind turned out. Items reassigned three or more times in a week get no suggested person, and the panel warns when more parallel work than the team can integrate is in flight.

Economics & Quality leads with the primary outcome: verified items per million tokens and per week, with unverified and abandoned work counted. It also measures Conflict Radar recall against actual merge conflicts reported by GitHub, the override rate, handoff time to resume and resumption checks.

Admin: retention, export, deletion and the audit log

Admins set how long work events, radar alerts and dashboard events are kept (7 to 730 days, default 180). They can export the team's data as JSON (never the webhook secret) and delete work events, alerts, knowledge, contracts or dashboard events. The **audit log** records privileged actions: invites, approvals, role changes, linking sessions, projects, the Relay level, deploys, handoffs, GitHub connect and disconnect, knowledge approvals and merges, contract sends, alert decisions, retention, exports and deletions. Deleting data is itself logged, and the audit log can't be deleted from the dashboard. Every dashboard action checks the person's role.

Projects

New project takes a name, the code repo link (`https://…` or a `git@host:org/repo` address) and, optionally, a link to project files such as a design folder or spec (`https://…`). The project lines up with teammates' sessions in that repo automatically, because both sides derive the same project id from the repo.

Plans, dependencies and specs

Each project has a plan of up to 40 items.

- **Add items** from the dashboard, from `tokenbuild relay share`, or from `tb_preflight` shared to the team plan.
- **Assign** items to people. If someone else's Claude tries to claim an item assigned to you, the claim is refused.
- **Dependencies.** Set **after #** on an item (or `tb_task after` in a session). An item can't be claimed until the items it depends on are done. Cycles are refused. When an item assigned to you becomes ready, your next prompt says so once.
- **Specs.** For each item, write *done when*, *constraints* and *files to start with*. The spec travels with the item to whoever's Claude picks it up.
- **Verified.** When Claude finishes an item with `tb_task done`, the plan records whether it was verified. Marking an item done by hand on the dashboard leaves it unverified.
- **Estimates.** Claims carry the token forecast and completions carry the actual tokens and check spend. Items over twice their estimate are flagged.

Stages and deploying work

Items group into **stages** by their dependencies: stage 1 has no prerequisites, stage 2 depends only on stage 1, and so on.

- **Deploy stage** sends each ready, assigned item in the stage to its person's next Claude prompt as a staged task, with its spec. On pickup, the CLI adds likely files locally and starts a fresh task.
- **Auto-advance** sends the next stage once everything in the current one is done and none of it is unverified.
- `tokenbuild plan` shows the plan for this repo by stage. `tokenbuild plan run` claims your ready items in stage order and runs official Claude Code (`claude -p`) on each one with its prompt. It stops at the first item that isn't verified. Add `--yes` to skip the confirmation and `--max <n>` to limit the number of items. The permission mode is `acceptEdits` by default; set `"planRunMode"` to change it.

> **WATCH OUT**
> `plan run` lets Claude edit files without asking for each one. Run it on a clean working tree on its own branch, so you can review everything with `git diff` before merging.

Handoffs

Pass a task from a session with `tokenbuild relay send <name> [--note "…"]`, or from the dashboard's **Handoffs** view. The receiver:

- gets an email from tasks@tokenbuild.ai, and
- on their next Claude prompt in that repo, starts from your task state and handoff note instead of from scratch.

A handoff that hasn't been picked up can be cancelled. Handoffs need Relay level 2.

Coordination insights

- **Likely collisions**: teammates about to work on the same files, or on files that usually change together, flagged before the first edit. The check covers what they've edited and what their briefs predict.
- **Coordination gaps**: teammates working on coupled code who never share plan items or hand off.
- **Suggested dependencies**: inferred from how files change together. Accepting or dismissing a suggestion teaches the model, and kinds of suggestions people keep dismissing stop appearing.
- **Suggested assignees** for ready, unassigned items, balancing calibrated estimates, how busy people are, and expertise.
- **Who knows what**: expertise per person and folder, from verified finished items and recent edits, fading over 30 days.
- **At-risk tasks**: a teammate's task with a high early-warning score, plus the teammate best placed to help.
- **Estimate accuracy** per person and kind of task, pooled so a new teammate's estimates are sensible from the start.

- **Handoff quality**: which parts of a handoff note save the receiver the most.

Coordination savings

Two Claudes can't claim the same plan item: the second claim is refused. The work it didn't repeat is credited as a coordination saving, at 50% of the measured tokens of that work. A picked-up handoff is credited the same way, minus the size of the note. These credits are computed on the server, because they need data from two machines. They appear on the Team dashboard.

Shared Repo Memory and fingerprints

On Team plans, `tokenbuild sync` pools Repo Memory (file paths and how often files change together, never task wording) and fingerprints (error hashes and the files that fixed them, never error text). Each teammate's brief and fix suggestions then learn from the whole team's history on the same repo. Turn shared memory off per repo with `"teamMemory": false`.

Back to top · Next: ML Validation: tests and the Verify Loop

CHAPTER 9

ML Validation: tests and the Verify Loop

The third product, **ML Validation and Economics** (`tokenbuild learn`), uses models trained on your own repo's history to check work and spend tokens where they pay off. Validation starts with tests: after every edit, the Verify Loop runs only the tests that edit can affect, ordered and filtered by what this repo's test history says.

Verify Loop

What it does. After each edit, TokenBuild works out which test files the change can affect: tests that import the changed file directly or through other files, plus tests with matching names. It runs only those, in the background, with your project's own runner, and Claude gets one line back:

```
TokenBuild verify: 12 tests in 3 files pass (src/billing/*.test.ts)
TokenBuild verify: FAIL invoice.test.ts › applies credit — expected 90, got 100 · log: .tokenbuild/l
```

Supported runners: Vitest, Jest, `node --test` , pytest and `go test` , detected from your project files. For anything else, set a command with a `{files}` placeholder:

```
{ "verify": "npx mocha {files}" }
```

When a check fails at the end of a turn, Claude is asked to keep going and fix it before handing back to you. You get working code, not a red test you discover later.

By hand. `tokenbuild verify src/file.ts` runs the tests for one file. `tokenbuild verify` with no file shows the last result. Claude can call `tb_verify` , with scope `full` for the whole suite (`verifyFull` , else `npm test` , pytest, `go test` or `cargo test`).

Verified results. A task counts as *verified* when a check passed after its last edit. That's the standard the learning layer, the team plan and the proof all use.

The sections below cover the test history, test selection, check ordering and flaky tests.

Test history

Every test the Verify Loop runs is recorded in `.tokenbuild/tests.jsonl`: which test, which files had changed, pass or fail, how long it took, and whether it passed on a rerun. Nothing about the test's content is stored, only names, paths, results and timings. Selection and ordering get better as this history grows.

Predictive test selection

For the files just changed, TokenBuild estimates how likely each affected test is to fail. The estimate combines:

- the test's overall failure rate here,
- how often it failed when these particular files changed,
- whether the test file itself was edited, and
- how recently it last failed.

Once there are about 60 runs with 6 failures, a model learns the weights. Before that, sensible defaults apply, and every estimate is capped by the test's own smoothed record.

Skipping is token-aware. A skipped test that would have failed lets a bug through, and fixing it later costs a whole task. Running a test costs Claude only its one-line report. So by default a test is skipped only when its chance of failing times that rework is smaller than the cost of running it, which in practice means every affected test runs. The learned order still applies, so failures show up first.

Skipping to save time is opt-in: with `"testSelect": "time"`, a test is skipped if it has 5 or more runs, passed last time and wasn't edited, while the combined risk of everything skipped stays under 2%. The report always says how many were skipped ("41 skipped as low-risk"). `"testSelect": false` turns off learned ordering and skipping alike.

High-risk changes widen the check (0.20.4). Before each run the Verify Loop scores the whole pending change (unpushed plus uncommitted) with the deploy risk model (Chapter 10). When the risk is high, nothing is skipped and the run widens to the tests of every file in the

change, not just the file you edited, up to 24 test files. The report says so and asks for the full suite before you ship. This runs in the background worker, never in a hook. Turn it off with `"riskExpand": false`.

> **WATCH OUT**
> Selection applies to the Verify Loop's quick checks. Before merging or releasing, run your full suite: ask Claude for `tb_verify` with scope `full`, or run it in CI.

Learned check order

The tests most likely to fail per second of run time go first, in their own batch. If any of them fail, the rest wait. Claude gets the failure sooner and doesn't pay for output from tests that would have passed. If the first batch passes, the rest run as usual.

Flaky tests

A failing test is rerun once. If it then passes with no change, that's a **flip**. After two flips, the test is marked **flaky** and reported as flaky rather than broken:

```
TokenBuild verify: 1 flaky (passes on rerun): upload.test.ts › retries on 503 — not caused by this c
```

Claude then doesn't spend a fix cycle on a failure your change didn't cause. Only the test files that failed are rerun. TokenBuild reads them from the runner's output, including the TAP format that `node --test` prints on Node 20 and 22 in CI. Turn reruns off with `"flakyRerun": false`.

> **TIP**
> Flaky tests cost real tokens and attention. The test history in `tokenbuild learn` shows where they are. Fixing or quarantining them pays back quickly.

Custom runners and monorepos

- **A different runner.** Set `"verify": "your command {files}"`. `{files}` is replaced with the affected test files, space-separated.

- **A full-suite command.** Set `"verifyFull"`, for example `"pnpm -r test"`.
- **Monorepos.** The Verify Loop follows imports across packages. If each package has its own runner, set `verify` to a script that dispatches by path.
- **Slow suites.** Selection and ordering help most here. Make sure tests and sources are linked by imports or matching names (`foo.ts` ↔ `foo.test.ts`), so the Verify Loop can find them.
- **Turning it off.** `"verify": false`.

Check spend

The Real Meter counts how many tokens of check output Claude actually read in each task. On Team plans, the dashboard shows this next to the estimate and actual tokens for each plan item, so you can see which areas have expensive or noisy tests.

Back to top · Next: ML Validation: before you ship

CHAPTER 10

ML Validation: before you ship

Before a change leaves your machine: how likely it is to need a revert or a follow-up fix, learned from what happened after past changes here; where a production incident should start; and which parts of the codebase are getting more expensive to work in.

Change risk

`tokenbuild risk` (or `tb_risk`, or just asking Claude to push, merge, deploy or ship) shows how likely the current change is to need a revert or a follow-up fix, why, and what to run first.

What it looks at. The change is everything since your upstream branch (or the last commit) plus uncommitted work. The signals:

- churn: lines changed, files and folders touched,
- past fixes in those files,
- a test gap: changed code without matching test changes, scaled by size,
- whether the change was verified,
- overlap with teammates' recent work (Team plans), and
- whether the change reaches into an area that's new to you.

How it learns. Each check is logged in `risk.jsonl`. A check is later labeled bad if a fail or fix task touched the same files within 72 hours, or an incident pointed there. After 30 labeled checks including 5 bad ones, a model trained on this repo takes over from the starting heuristics.

Learned pre-deploy checks are the commands that caught failures in these folders before. They're listed with the risk, in the order to run them.

Pre-push hook. `tokenbuild risk --install-hook` prints the risk before every `git push`. It never blocks the push. Turn the risk line in prompts off with `"risk": false`.

Production incidents

When something breaks in production, paste the error:

```
npx tokenbuild risk --incident "TypeError: Cannot read properties of undefined (reading 'amount')"
```

TokenBuild fingerprints it and shows the files that fixed this error before, plus recent work in those files, which is where a regression most likely came from. It also labels risk checks from the last 7 days on those files as bad, so the risk model learns from the incident.

Drift

Some areas of a codebase get more expensive to work in over time. `tokenbuild learn` tracks tokens per task per folder and flags folders where the cost has shifted upward. `tokenbuild shape` then suggests what to split or co-locate (Repo Shape Optimizer).

Back to top · Next: ML Economics: how TokenBuild learns

CHAPTER 11

ML Economics: how TokenBuild learns

A feature that saves tokens in one repo can cost tokens in another. A brief is great in a large backend and mostly noise in a three-file script. So TokenBuild learns, per repo, which help to give. All learning runs on your machine, from this repo's own measured tasks.

What it optimizes: tokens to a verified result

The learning layer's score for a task is the tokens it took to reach a **verified** result: a passing check after the last edit. Tasks pay a rework penalty when they:

- end on a failing check,
- edit code without ever passing a check, or
- get reopened within 48 hours.

So spending fewer tokens by giving up early, or by skipping the tests, never counts as a win.

Kinds of task

Each request is classified as a **question, fix, feature, refactor or test**. Word rules give a starting guess. After about 30 tasks, a classifier learns from what tasks here actually turned out to be (a "question" that ended with edits was really a fix) and takes over. Policies, budgets and brief sizes are learned per kind.

Which help to give

For each kind of task, TokenBuild learns whether the Task Brief and Next-Read are worth their tokens here, and how many likely files the brief should list.

- **Thompson sampling** balances trying both options with using the one that's winning. For each kind of task and feature, it keeps a belief about how much cheaper or costlier tasks are with the feature on than off, measured against each task's own forecast rather than a simple

"under the usual cost" yes/no. Comparing a task with its forecast removes much of the difference between big and small tasks, so fewer tasks are needed to see an effect.

- **Contextual decisions.** Once there's enough data (about 40 tasks with the feature on and 15 with it off), a contextual model also looks at the task itself: its size, the files involved, how clear the request is. It only takes over once it has proven better, as described below.
- **Brief size** is chosen from 3, 5 or 8 files: mostly 5, with some 3 and 8 to keep learning, until the learned sizes have proven better than always 5.
- **Brief confidence** (how strongly the top file matches the request) is part of what the contextual model sees, so it can learn when showing the brief pays.
- Every decision is logged with its probability, which is what makes the unbiased checks below possible.

Learning can switch off only the brief and Next-Read. It never turns off the Verify Loop, Edit Preflight or the Context Ledger. Tasks where learning held a feature off are left out of the proof and dashboard credit.

Unbiased comparisons

Comparing "tasks with the brief" against "tasks without" is biased if, say, the brief was more often on for hard tasks. TokenBuild uses **doubly robust off-policy evaluation**, which combines a model of task cost with the logged decision probabilities. It estimates what "always on" or "always off" would have scored, as if you had run both. `tokenbuild learn` shows these estimates per feature, and `tokenbuild replay` uses the same method.

File ranking

The brief's candidate files are re-ranked by a model trained on which files tasks here really opened or edited. Its inputs are the index match score, Repo Memory edits, read history, how often a read of that file led to an edit, whether it's a test file, and the original rank. It starts after about 30 briefs and refits as new tasks finish.

Hit rate. For every task, TokenBuild records whether the brief named a file the task then edited (or, for a task with no edits, read): **hit@1** for the top file and **hit@5** for the first five. `tokenbuild learn` shows both.

Confidence gate. A brief that names the right file saves tokens, but one that names the wrong files costs more than no brief: Claude reads them first. In our measurements a brief paid off only when it was right about 75% of the time. So TokenBuild learns, from this repo's own briefs, how often a brief this confident turns out right, and holds the brief back when that chance is under 75%. It starts after 8 briefs with an outcome. One in ten held-back briefs is still shown, so the estimate keeps learning. Set the threshold with `"briefGate": 0.75`, or turn the gate off with `"briefGate": false`.

When to start fresh

Adaptive Fresh Start (0.20.1) is a separate learner with two choices: recommend `/clear` or continue normally. Every N prompts (`freshEvery`, default 5) it reads a short context: conversation size, growth since the last check, how much of the request is new to the session, whether the task kind changed, whether the prompt cache expired, the handoff note's size, and how often you ignored earlier advice. Its reward is the change in tokens per reply over the next N tasks against the N before, following a `/clear` into the new session, minus the rework penalty for failed or reopened tasks. There is also a small cost for advice you didn't follow. A Bayesian linear regression for each choice is fitted to these outcomes, and Thompson sampling picks, with "continue" slightly favoured. Every decision logs its probability. Until each choice has 8 measured outcomes, a starting rule decides and explores 10% of the time. Everything stays in `.tokenbuild/fresh.jsonl`; `tokenbuild fresh` shows it.

Budgets, ranges and early warnings

- **Forecasts and Autopilot budgets** come with an **80% conformal range**: an interval that past tasks here fell inside 80% of the time.
- **Early warning.** When a task reaches 30% of its forecast, a model trained on past tasks judges from the task so far whether it's heading for trouble: far over budget, or not reaching a verified result. If the chance is 60% or more, you're told once at the end of the reply, so you can step in early. Turn it off with `"earlyWarn": false`.

It checks itself

- **Shadow mode.** Every learned policy runs in shadow (logged but not deciding; the feature stays on) until its off-policy estimate beats the default by 1.96 standard errors (95%

confidence). The per-kind policy needs 30 or more tasks and must beat "always on". The contextual policy needs 100 or more and must beat both. Learned brief sizes must beat "always 5". Each candidate is scored only on tasks it wasn't fitted on.

- **Guardrail.** If tasks where a feature was held off verify 10 or more points less often than tasks with it on (with 20+ tasks on each side), that feature is never turned off again in that repo.
- **Drift.** When forecasts start missing, for example in a new area of the codebase, with a new teammate or after a Claude model update, exploration widens until the models catch up.

Field measures: did the work hold up?

From 0.20, TokenBuild also measures, on your own machine, whether finished work held up in real use. Each edit adds one line to `.tokenbuild/edits.jsonl` (file, first changed line, old and new line counts; never code). Everything else is computed offline, so the hooks only append and their output is unchanged.

- **V-op (operational).** A task counts as verified when every test linked to the edited files through imports passed in its latest run, within 30 minutes of the last edit. It uses the fixed import-graph mapping, not the learned test selection, so the measure can't be gamed by skipping tests.
- **V-dur (durable).** How many of the changed lines survive without a later fix task touching them, at 48 hours and at 7 days. Line positions are followed through later logged edits; edits made outside Claude Code aren't seen.
- **Turnover.** Counts of tasks per kind of task and per arm (regular sessions, and each evolved flag on trial kept apart).

`tokenbuild learn` shows them. Sync sends only these counts, one row per repo; turn that off with `"shareField": false` (or `"shareLearning": false`).

Evolved flags on trial

Evolved flags are candidate behavior changes found by TokenBuild's improvement loop. A candidate reaches you only after it passes the benchmark gate, and then first as a trial: a share of sessions (10% by default, `"trial"`; 0 = off) runs with one trial flag on, chosen per session. Control-group (holdout) sessions never run a trial, and a flag you set yourself in the config is never trialled. `tokenbuild learn` compares each flag on vs off with a doubly robust estimate

of tokens to a verified result, and a person promotes or retires it at release; a flag whose tasks verify less often is marked for retirement. TokenBuild can also switch a flag off for everyone through the signed plan answer, without a new release. **0.20.0 ships with no evolved flags**, so it behaves exactly like 0.19.1; it adds the measuring.

Starting points from everyone

A brand-new repo has no history. Instead of starting from nothing, it starts from statistics pooled across TokenBuild users, per kind of task and feature: how many tasks ran with the feature on and off, and how they did against their forecasts. These pooled figures are the main evidence, since one repo rarely has enough tasks of its own. Each machine sends only its own figures (never the pooled ones it received), and only for tasks that had their own forecast, with random noise added before sending: differential privacy, ε = 1 for each release, with the noise sized to the most one task can change. Unchanged figures are resent with the same noise and a release happens at most once a week, so repeated syncs can't be averaged to remove it; over many weeks, releases add up. Each feature's estimated effect is also noised and rounded before it's sent, though without a formal ε guarantee. Only counts and sums are shared, never tasks, files or code. A pooled figure is used only once 3 or more accounts contribute, and it counts as at most 500 tasks per side. A learned policy still acts only after this repo's own logged tasks confirm it off-policy (see "It checks itself"). Turn it off with `"shareLearning": false`.

Seeing and controlling what it learned

- `tokenbuild learn`: policies per kind of task with their off-policy checks, tips and their uptake, test history, drift, the state of the meta checks, field measures (V-op, V-dur, turnover) and any evolved flags on trial.
- `tokenbuild learn --reset`: clears what was learned. Measured tasks are kept, so it relearns quickly.
- `"learn": false`: fixed behavior, with every feature on as configured.
- `"briefSize": 5`: pins the brief size.
- `"briefGate": 0.75`: the confidence gate's threshold; `false` always shows the brief.
- `"trial": 0`: no sessions run an evolved flag on trial.

- `"holdout": 0`: no control-group sessions. This also stops the learning layer from getting new "off" data.

Back to top · Next: ML Economics: proof, strategies and repo economics

CHAPTER 12

ML Economics: proof, strategies and repo economics

Project economics means spending tokens where they pay off and proving it. This chapter covers the control-group proof of each feature's effect, Replay (which way of working costs least to a verified result), the Recipe Compiler (repeated work turned into a skill) and the Repo Shape Optimizer (what to change so your codebase is cheaper to work in).

Control groups (`tokenbuild proof`)

Some features, like the brief, save by making tasks shorter rather than by blocking a specific read. The only honest way to measure those is a control group. In about 10% of sessions, one feature is held off: the Context Ledger, Verify Loop, Task Brief, Next-Read, Edit Preflight or Project State. Comparing real metered tokens per task in those sessions with all others shows what each feature actually saves, with its own cost included.

`tokenbuild proof` shows the result once both groups are big enough. Until then it says how many tasks it still needs.

Intervals and "insufficient evidence" (0.20.4). Each effect comes with a 95% interval, computed on the log scale (token counts are skewed, and a ratio is what matters). A feature counts as a proven saving only when the whole interval is a saving; then it's credited at the low end of the interval. When the interval still includes "no effect", the line says **insufficient evidence** and nothing is credited. Small samples are for exploring, not for claims, so TokenBuild won't call a feature better on a handful of tasks.

```
Verify Loop     −14% tokens (61.2K vs 71.1K), replies 9 vs 11 (n=412/47) · 95% CI −6% to −21%
Task Brief      −5% tokens (66.8K vs 70.3K), replies 9 vs 10 (n=412/39) · insufficient evidence (9
```

The dashboard's **"Measured across all TokenBuild users"** card shows the same comparison pooled across accounts: median tokens per task with each feature on and off.

Opt out with `"holdout": 0`. You lose the proof, and features credited from it stop showing a saving.

Replay

`tokenbuild replay` (or `tb_replay`) compares ways of working on this repo's measured history, such as tasks with the brief on versus off, or with and without particular `tb_*` tools, and ranks them by **tokens to a verified result**. Comparisons use off-policy evaluation, so they correct for the fact that different tasks happened to use different strategies.

Name strategies to compare, or pass `--task "<task>"` to compare on tasks like it. Two strategies need 5 or more tasks each before they're compared at all. Even then, the cheapest is a **recommendation** only when its 95% interval against the next one excludes no difference. Otherwise Replay says "Cheapest so far … but insufficient evidence", with the interval, so a lucky streak isn't mistaken for a better way of working. Replay compares your own history, which is correlation, not a controlled test; the control-group proof is the controlled test.

Recipe Compiler

When you've done the same kind of task three or more times, such as adding an API endpoint or a database migration, TokenBuild writes a **recipe**: a Claude Code skill in `.claude/skills/tb-recipe-*`. It's built from what actually happened, not from a template: the files that change and in what order (with line numbers), files that often change too, the check that proved it worked, and errors seen before with where they were fixed.

- Claude Code loads a skill only when a task matches, so a recipe costs nothing until it's used.
- **Built only from verified work (0.20.4).** A recipe needs 3 or more tasks that ended on a passing check after the last edit and weren't reopened, and at least 60% of the similar tasks verified. A one-off success is not a recipe.
- **Versioned.** Each rewrite bumps the version, shown in the skill file and kept with its basis (tasks, confidence, cost) in `.tokenbuild/recipes.json`.
- **Regression-checked.** Each use is measured against the tasks the recipe came from. After three uses, a recipe is retired if it doesn't save tokens, if uses verify 20 or more points less often, or if two uses were reopened.

- **Retired when stale.** Also retired when its files are gone, when it hasn't been used for 90 days, or when all its source tasks are older than 180 days. Retired for disuse or age, it can come back after 90 days on new evidence; retired for cost or a regression, it doesn't.
- Recipes live in your repo, so your team gets them through git.
- To keep your own edits to a recipe, delete its marker line; TokenBuild then leaves it alone.
- List them with `tokenbuild recipes`, compile now with `--compile`, remove one with `--remove <name>`, or turn them off with `"recipes": false`.

Repo Shape Optimizer

Some files are expensive to work in: a 3,000-line module that every task reads, or two files always read together. `tokenbuild shape` (or `tb_shape`) measures what each file and each section of a file costs Claude to read, from actual reads in your sessions, and suggests:

- splitting a big file whose reads concentrate in a few sections that could stand alone,
- co-locating files that are almost always read together, and
- watching files that are growing fast.

`tokenbuild shape --plan 1` turns suggestion 1 into a step-by-step plan Claude can carry out. Once the new files are in use, `tokenbuild shape` shows the measured before and after.

Guardrails (0.20.4): fewer tokens never justify worse code.

- Security-sensitive files (auth, sessions, crypto, secrets, permissions, signing, licences), generated and vendored code, lock files, migrations and config files are never suggested for a split or merge. They are listed as "held back", with the reason.
- A hotspot that no test covers gets "add tests first" as the first step of its plan.
- Every plan moves code only, with re-exports so imports keep working and no logic changes. It ends with the full suite and the linter, and says to revert the split if anything fails or if it makes the code harder to follow.
- The before and after also shows the share of tasks in that area that verified, and flags a drop of more than 15 points as worse, with a suggestion to revert.

Back to top · Next: The web app and dashboard

CHAPTER 13

The web app and dashboard

The web app at app.tokenbuild.ai is where you manage your account and see your savings. This chapter goes through each page. The Team workspace has its own chapter (Chapter 8).

Getting around

Once you're signed in, the top bar has tabs for **Dashboard**, **Team**, **CLI key** and **Billing**. The **Menu** button holds the website's pages: Home, Pricing, Documentation, Commands, Help, Links, FAQ, Security and Contact. It also has **Sign out**. On phones, the tabs move into the Menu too.

When you're signed out, the top bar offers **Create account**, and the Menu has both **Sign in** and **Create account**.

Create account, sign in, reset password

- **Create account** (`/signup`): email, password, and agreement to the Terms and Privacy Policy. Starting from a "Start Team trial" link preselects the Team plan on Billing.
- **Sign in** (`/signin`): email and password. After too many failed attempts, wait a minute before trying again.
- **Forgot password** (`/forgot`): enter your email to get a reset link. For your privacy, the page says the same thing whether or not the email has an account.
- **Reset** (`/reset`): the link from the email. It works once, for 1 hour. If it has expired, ask for a new one.

Dashboard

The dashboard (`/`) shows what TokenBuild saved across all your computers and repos:

- **Headline figures**: tokens kept out of Claude's context, logs condensed by the Output Shrinker, wasteful reads blocked, and files read by Local Boost.

- **Where the saving came from**: a table of tools, with uses and tokens kept out for each. Auto-format and repo lookups show uses only, because they save turns rather than a countable number of tokens.
- **Measured across all TokenBuild users**: the control-group results, as median tokens per task with each feature on and off, once both groups are big enough.
- **Last 30 days**: daily totals.
- **Team Relay** (Team plans): your team's live Claude sessions per repo, the relay level setting, and a link to the Team dashboard.

If the numbers look out of date, run `npx tokenbuild sync` and reload. See Dashboard errors.

CLI key

The CLI key (`/keys`) connects the CLI and plugin on your computers to your plan, and lets them send counts to your dashboard.

- **Create key** asks you to confirm first, so a stray click can't replace a working key.
- The new key is shown **once**, with ready-made install lines for Windows and Mac/Linux and Copy buttons.
- **Create a new key** replaces the current one. The old key stops working right away, and every computer that used it needs the new key (`npx tokenbuild login <key>` , or the install line).

> **WATCH OUT**
> Treat the key like a password. Anyone with it can use your plan's tools and send counts to your dashboard. It can't read your code or your Claude account, but replace it if it leaks.

Billing

Before you subscribe, Billing shows the plan picker: Solo or Team, monthly or yearly, and seats for Team. Agree to the renewal terms and continue to Stripe's secure checkout.

After you subscribe, it shows your plan (Solo, Team or Complimentary) and status:

Status	Meaning
Trial	In the 30-day trial. Tools are on.
Active	Paid and current. Tools are on.
Payment due	A payment failed. Tools stay on while Stripe retries. Update your card under Manage billing.
Starting…	Checkout just finished and the plan is being set up. Reload in a few seconds.

- **Manage billing or cancel** opens Stripe's customer portal: update your card, download invoices, change seats, or cancel in one click. After you cancel, you keep access until the end of the period you've paid for, and you're not charged again.
- **Upgrade to Team** (Solo plans) switches your existing subscription to Team, prorated, and opens the Team workspace.

See Cancellation & refunds for the full policy.

Toki, the assistant

Toki, the fox in the corner of every page, answers common questions about TokenBuild instantly from a built-in knowledge base, at no cost. For questions it doesn't know, it may use an AI answer if that's enabled, with a per-visitor and daily limit. Toki can't see your account, code or dashboard. For account-specific problems, email support.

Back to top · Next: CLI command reference

CHAPTER 14

CLI command reference

Run commands as `npx tokenbuild <command>` in a terminal at your repo's root. Inside Claude Code, the plugin puts `tokenbuild` on the PATH, so `tokenbuild <command>` works too. `npx tokenbuild` with no command lists the three command groups.

Conventions: `<angle brackets>` are values you fill in; `[square brackets]` are optional. Most commands accept `--root <path>` to act on another repo. `audit` is free. Everything else, apart from `install`, `login`, `logout`, `doctor`, `init`, `sync`, `autosync` and `stats`, needs an active plan and otherwise prints how to get one.

Account and setup

`audit [--days 30] [--project name] [--json]`

Free, no account. Prices avoidable waste in your recent Claude Code session logs and shows where it came from: your costliest files, the waste categories and spend by model. `--project` matches the project folder name. Example: `npx tokenbuild audit --days 7`.

`install [key] [--yes]`

Installs the plugin into Claude Code (or the Claude desktop app), saves your key in the plugin's secure settings and confirms your plan. With no key, it asks for one, or you can add it later. `--yes` installs anything missing without asking. The one-line installers use it.

`release update`

Updates the plugin to the latest version and keeps your key. Restart Claude Code afterwards.

`login [key]`

Saves the key on this computer (`~/.tokenbuild/key`) and confirms your plan online. With no key, it re-checks the saved one. Run this after creating a new key, or when told the plan couldn't be confirmed.

`logout`

Removes the saved key and the signed plan answer from this computer.

`doctor`

Checks your plan online and reports which tools are on in this repo, where your key comes from, and the one thing to do next. It's the first command to run when something seems off.

`init [--route balanced|lean|off] [--no-format]`

Project mode: writes hooks, the MCP server, model routing and rules into this repo's `.claude` settings. Safe to rerun. See Installing per project.

How the commands are grouped

Commands are grouped under TokenBuild's three products. `npx tokenbuild` lists the groups, `npx tokenbuild <group>` lists that group's commands, and `npx tokenbuild help all` prints every command.

Group	Product	Example
`save`	Token Saver	`npx tokenbuild save run npm test`
`team`	Team Build	`npx tokenbuild team plan run`
`learn`	ML Validation and Economics	`npx tokenbuild learn risk`

The group is optional: every command also works under its single-word name (`tokenbuild run`, `tokenbuild plan`, `tokenbuild risk`), so existing scripts, hooks and habits keep working. The sections below show the grouped form and the short form together.

Token Saver: `tokenbuild save`

`save run <command>` · `run`

Runs a noisy command through the Output Shrinker. It prints errors, code frames, warnings and the summary, saves the full log to `.tokenbuild/logs`, and exits with the command's own exit code. Everything after `run` is passed through unchanged; use `run -- <command>` if the

command starts with a dash. Without a plan, it runs the command normally. Example: `npx tokenbuild save run npm run build`.

`save index` · **`index`**

Builds or refreshes the Repo Index. It runs automatically at session start, so you rarely need it by hand: after a big pull or branch switch, or in a new clone.

`save map [--budget 2500]` · **`map`**

Prints a compact map of the repo: the most-imported files and their main symbols, within the token budget.

`save find <name>` · **`find`**

Locates a symbol or file and prints `path:line`. Example: `npx tokenbuild save find createInvoice`.

`save pack "<task>" [--budget 4000]` · **`pack`**

The files and symbols a task most likely needs, with line numbers, within the budget. Example: `npx tokenbuild save pack "add rate limiting to the login route"`.

`save docs <symbol>` · **`docs`**

Docs for a library symbol at the version installed here. For JavaScript and TypeScript use `"pkg#name"`; for Python and Go, the dotted or package path. Example: `npx tokenbuild save docs "express#Router"`.

`save boost [on|off] [--model name]` · **`boost`**

Turns Local Boost on or off, or picks the Ollama model. With no argument, shows its status.

`save ask "<question>" <file>` · **`ask`**

Asks the local model about a file. Example: `npx tokenbuild save ask "which function retries on 429?" src/http.ts`.

`save handoff [--contract]` · **`handoff`**

Saves a handoff note now, so `/clear` or a new session continues from it. `team handoff` does the same.

`save fresh [on|off] [--every N] [--min T]` · **`fresh`**

Adaptive Fresh Start: its decisions, how often you followed them and the measured savings. `--every` sets how often it decides (1 to 10000 prompts, default 5); `--min` sets the smallest conversation it advises on; `on` / `off` switches it.

`save profile` · **`profile`**

What fills the context now, and how much is dead weight.

`save tune permissions|diet|claudemd [--apply]` · **`permissions`**, **`diet`**, **`claudemd`**

Each shows what it would change; add `--apply` to make the change. `permissions`: commands you always approve, turned into allow rules. `diet`: MCP servers whose tool definitions cost tokens every turn but go unused (needs 5+ sessions of data). `claudemd`: moves folder-specific CLAUDE.md sections into that folder.

`save stats [--json]` · **`stats`**

Savings in three separate tiers (measured, estimated, proven), TokenBuild's own overhead and break-even against your plan fee, then activity by tool.

`save overhead [--json]` · **`overhead`**

TokenBuild's own cost in this repo: hook latency (p50/p95 against the 50 ms rule), context added per session, and nudges against the budget.

`save meter [--json]` · **`meter`**

Real tokens and cost per task and in total, from Claude Code's usage records.

`save sync [--all]` · **`sync`**

Sends counts to the dashboard now. Counts and hashes only, plus file paths on Team plans. `--all` syncs every repo TokenBuild has seen on this computer.

`save autosync on [--every 15]|off|status` · **`autosync`**

Keeps the dashboard current with no session open: an operating-system job (a scheduled task on Windows, a crontab line elsewhere) runs `sync --all` every *N* minutes.

`save audit`, **`save init`**

The same as `audit` and `init` above.

Team Build: `tokenbuild team`

`team task [--reset]` · **`task`**

Shows the current task state: goal, plan, files changed, checks, facts. `--reset` clears it.

`team state [--decide <text>] [--forget <n>]` · **`state`**

Shows the project state. `--decide` records a decision; `--forget 2` removes decision 2.

`team preflight "<goal>" [--steps "a; b; c"]` · **`preflight`**

Plans a project: files, tests, estimate, dependencies and stage for each step.

`team autopilot "<task>" [--budget <n>]` · **`autopilot`**

A token budget split by phase (planning, coding, testing, correction), with an 80% range.

`team plan [run [--yes] [--max <n>]]` · **`plan`**

Team plans: the plan for this repo by stage. `run` works through your ready items with Claude Code.

`team relay` · **`relay`**

Teammates' live sessions in this repo.

`team relay send <name> [--note "…"] [--contract]`

Passes your current task to a teammate. Needs level 2. With `--contract`, it drafts a Verified Handoff Contract from local state for a person to review and send on the dashboard.

`team relay share`

Shares your plan with the team. Needs level 2.

`team relay claim <n>`, **`team relay done <n>`**

Claims plan step *n*, or marks it done.

`team relay name <you>`

Sets the name teammates see.

ML Validation and Economics: `tokenbuild learn`

`learn [--reset]`

What's been learned in this repo: policies, off-policy checks, tips, test history, drift, field measures (V-op, V-dur, turnover) and evolved flags on trial. `--reset` clears learned state and keeps measured tasks.

`learn verify [file]` · `verify`

Runs only the tests a change to `file` can affect, with test selection and ordering, and prints one line. With no file, shows the last result.

`learn risk [--incident "<error>"] [--install-hook]` · `risk`

Change risk before you push, with reasons and checks to run first. `--incident` starts from a production error. `--install-hook` adds a pre-push hook that prints the risk and never blocks.

`learn proof` · `proof`

Each feature's measured effect, from control-group sessions, with its 95% interval: a proven saving, or "insufficient evidence".

`learn replay [strategy...] [--task "<task>"]` · `replay`

Compares strategies on measured history, ranked by tokens to a verified result.

`learn recipes [--compile] [--remove <name>]` · `recipes`

Lists recipes with their measured uses; compiles now; removes one.

`learn shape [--plan <n>]` · `shape`

The costliest files to read, and a step-by-step plan for suggestion *n*.

Commands Claude Code runs for you

You don't need to type these. They're listed so you recognize them in logs and settings.

`hook start|prompt|post|stop|precompact|end`

One process per Claude Code event (Chapter 3).

`guard [--reset]`

The PreToolUse hook: Waste Guard, Context Ledger, Edit Preflight and command wrapping. `--reset` clears this session's guard and ledger state.

`format [--track]`

Formats edited files with your formatter.

`mcp`

Starts the MCP server that provides the `tb_*` tools.

`bg tune|outlines`

Background jobs: the weekly Setup Check and Local Boost outlines.

Environment variables

Variable	Effect
`TOKENBUILD_KEY`	CLI key to use, ahead of the plugin settings and saved key. Use it for CI.
`TOKENBUILD_HOME`	Moves the computer-wide folder from `~/.tokenbuild`.
`TOKENBUILD_CLAUDE`	The Claude Code binary that `plan run` starts, if `claude` isn't on the PATH.

Back to top · Next: MCP tools, skills and the scout

CHAPTER 15

MCP tools, skills and the scout

The plugin gives Claude a set of tools it calls on its own, skills you can run with a slash command, and a helper subagent. Claude usually picks the right tool by itself. You can also ask by name: "use tb_pack for this", "send the scout to find where emails are sent".

The tb_ tools

Tool	What Claude uses it for	Instead of
TOKEN SAVER [save]		
`tb_brief`	A short brief for a task: likely files with lines, related files, tests, past fixes. Useful mid-session when the task changes.	Exploring from scratch
`tb_map`	A compact map of the repo	Glob, LS and reading entry files
`tb_find`	Where a symbol or file is: `path:line`	Grep, then reading whole files
`tb_pack`	The files and symbols a task needs, under a budget	Several rounds of search and read
`tb_edit`	Replace a line range without writing the old lines	The standard Edit tool's old/new text
`tb_apply`	One find/replace across many files, applied locally	Dozens of separate edits
`tb_docs`	Library docs at the installed version	Reading library source, or memory

`tb_memo`	Saved answers about the repo, while still valid	Repeating an investigation
`tb_profile`	What fills the context, and the dead weight	Guessing
`tb_digest`	Ask the local model about a big file or log (Local Boost only)	Reading the whole thing
TEAM BUILD [team]		
`tb_task`	Read or update task state: get, claim, done, fact, question, decide, after, state	Re-deriving where things stand
`tb_preflight`	Plan a project; optionally share it to the team plan	Planning in conversation
`tb_autopilot`	A phase-split token budget for the task	No budget
ML VALIDATION AND ECONOMICS [learn]		
`tb_verify`	Run affected tests (or scope `full`) and get one line	Running the whole suite and reading its output
`tb_risk`	Change risk before a push, or where an incident starts	Pushing blind
`tb_replay`	Compare strategies on history before choosing	Trial and error
`tb_shape`	Costliest files to read, and what to split	Guessing

Each tool's description starts with its product group, `[save]`, `[team]` or `[learn]`, so Claude sees which product it belongs to. The names stay the same. `tb_digest` appears only when Local Boost is on and a model is installed, after you restart Claude Code.

tb_task in more detail

`tb_task` is the tool subagents and new sessions use most. Its main operations:

- **get**: the goal, plan, files changed, checks, facts and open questions.

- **fact**, **question**: record a verified fact or an open question, so the next session doesn't rediscover it.
- **state**: the project state (decisions, recent work, outstanding).
- **claim**: claim a plan item before working on it, so parallel agents or teammates don't duplicate it. A claim on an item someone else holds, or one whose dependencies aren't done, is refused with the reason.
- **done**: finish an item or the task. It reports whether the result is verified.
- **decide**: record a project decision.
- **after**: set a dependency between team plan items.

Skills

Type	Does
`/tokenbuild:setup`	Checks TokenBuild is ready in this project and says the one thing to do next
`/tokenbuild:audit`	The free waste audit of your recent sessions
`/tokenbuild:stats`	Tokens kept out of context in this repo
`/tokenbuild:handoff`	Saves a handoff note before a fresh session
`/tokenbuild:index`	Builds or refreshes the repo index
`/tokenbuild:boost`	Turns Local Boost on or off
`/tokenbuild:rules`	TokenBuild's token-saving way of working. Claude loads it by itself before tests, builds, installs and searches.

Recipes compiled from your work (`tb-recipe-*`) appear as project skills and load only when a task matches.

The scout subagent

The scout is a fast, low-cost, read-only explorer. Claude sends it to answer "where", "what" and "which" questions, to read logs or large files, and to report back paths with line numbers and short facts rather than code dumps. It runs in its own context, so the exploration doesn't fill your main conversation, and on a lighter model (Haiku under the balanced and lean routes).

> **TIP**
> For broad questions ("how is caching done across this repo?"), ask Claude to "send the scout". You get a summary in a few hundred tokens instead of ten file reads in your main context.

Back to top · Next: Configuration reference

CHAPTER 16

Configuration reference

Settings live in `.tokenbuild/config.json` at the root of each repo. Every setting is optional, and the defaults suit most projects. Create the file if it doesn't exist. It's plain JSON:

```
{
  "verify": "npx mocha {files}",
  "handoffAt": 80000,
  "holdout": 10,
  "relay": true
}
```

Changes take effect on the next event. A few, marked below, need a Claude Code restart.

Core features

Setting	Default	What it does
`brief`	true	`false` turns off the Task Brief.
`briefSize`	learned	Pins the number of likely files in the brief, for example 5.
`predict`	true	`false` turns off Next-Read.
`memory`	true	`false` turns off Repo Memory, and with it the memory-based parts of the brief.
`state`	true	`false` stops injecting the project state at session start.
`preflight`	true	`false` turns off Edit Preflight.

`autoWrap`	true	`false` stops the guard from wrapping noisy commands with the Output Shrinker automatically. Claude is still told the wrapped form.
`format`	true	`false` stops formatting edited files at the end of each reply. `init --no-format` sets this.
`recipes`	true	`false` stops the Recipe Compiler.

Verify Loop and tests

Setting	Default	What it does
`verify`	auto-detected	A test command with `{files}`, or `false` to turn the Verify Loop off.
`verifyFull`	auto	The full-suite command for `tb_verify` scope `full`.
`testSelect`	true	`false` runs every affected test, with no risk-based skipping.
`riskExpand`	true	A high-risk pending change runs the tests of the whole change with nothing skipped. `false` turns it off.
`nudgeBudget`	5	Messages shown to you per session from the prompt, stop and edit hooks; more are held back. 0 = none, `false` = no limit.
`contextBudget`	12000	Tokens of hook context per session that `tokenbuild overhead` reports against.
`flakyRerun`	true	`false` stops rerunning failures to detect flaky tests.

Context size

Setting	Default	What it does
`handoff`	true	`false` turns off Fresh Start suggestions.

`handoffAt`	100000	The conversation size, in tokens, at which a fresh start is offered.
`shareDecisions`	false	`true` lets this machine share its stated decisions with your team as Team Knowledge drafts (Team plans, Relay level 2). Off by default: sharing needs consent.
`experiments`	true	`false` opts this machine out of team workflow experiments. A key you set yourself always wins over an experiment's setting.
`teamKnowledge`	true	`false` stops approved Team Knowledge from being given to Claude in this repo (Team plans, Relay level 2).
`adaptiveFresh`	true	`false` turns off Adaptive Fresh Start (the learned clear-or-continue advice). It's also off when `handoff` or `learn` is `false`.
`freshEvery`	5	How often Adaptive Fresh Start decides, in prompts: a whole number from 1 to 10000.
`freshMin`	20000	Conversations smaller than this (tokens) get no Adaptive Fresh Start advice.
`cacheKeeper`	true	`false` turns off the Cache Keeper's offer after a break.
`gc`	true	`false` turns off Context GC suggestions. These appear for conversations between 50K tokens and `handoffAt` that are mostly dead weight.

Learning, budgets and risk

Setting	Default	What it does
`learn`	true	`false` gives fixed behavior: no learned policies, tips or brief size.
`holdout`	10	Percent of sessions that hold one feature off for the proof. 0 turns it off.

`autopilot`	true	`false` turns off Autopilot budgets and their warnings.
`earlyWarn`	true	`false` turns off the early warning.
`risk`	true	`false` removes the change-risk line from prompts about pushing or shipping.
`shareLearning`	true	`false` stops sending and using cross-user learning counts (and the field counts).
`shareField`	true	`false` stops sending the field measure counts (V-op, V-dur, turnover). They are still computed locally.
`trial`	10	Percent of sessions that run one evolved flag on trial (only when a release lists some). 0 turns it off.

Sync, sharing and team

Setting	Default	What it does
`syncEvery`	30	Minutes between dashboard syncs during a session. 0 turns them off.
`syncOnEnd`	true	`false` stops the sync at session end.
`shareFingerprints`	true	`false` stops sharing hashed fingerprints with config file names.
`teamMemory`	true	Team plans: `false` stops pooling Repo Memory for this repo.
`relay`	true	Team plans: `false` turns Team Relay off for this repo.
`planRunMode`	acceptEdits	The Claude Code permission mode `plan run` uses.

Setup tuning and Local Boost

Setting	Default	What it does
`autoTune`	false	`true` lets the weekly Setup Check apply its findings instead of only suggesting them.
`boost`	false	Local Boost on or off. Set it with `tokenbuild boost on`. Restart Claude Code after changing it.
`model`	auto	The Ollama model for Local Boost. Set it with `boost --model`.
`outlines`	true	`false` turns off Boost Outlines.

Ready-made configurations

Maximum privacy (nothing leaves the machine beyond plan checks and your own dashboard's counts):

```
{ "shareLearning": false, "shareFingerprints": false, "teamMemory": false, "relay": false }
```

Fixed, predictable behavior (for demos or benchmarks):

```
{ "learn": false, "holdout": 0, "briefSize": 5 }
```

Big, slow test suite:

```
{ "verify": "pnpm vitest run {files}", "verifyFull": "pnpm -r test", "testSelect": true }
```

Long, focused sessions (offer a fresh start earlier):

```
{ "handoffAt": 70000 }
```

Back to top · Next: Privacy, security and your data

CHAPTER 17

Privacy, security and your data

TokenBuild is built so it doesn't need your code. This chapter lists exactly what stays on your machine, what's sent, and how the system is protected. The legal versions are the Privacy Policy, Security and Subprocessors.

What never leaves your machine

- Source code and file contents
- Your prompts and Claude's replies (session transcripts)
- Command output and logs, including test failures
- Error message text. Only one-way hashes of normalized errors are shared.
- Task descriptions. The exception is Team Relay level 2, where task goals and plan titles are shared with your own team only.
- Local Boost traffic, which only goes to a model on localhost

What's sent, and why

Data	When	Why
Your CLI key	Plan checks	To confirm your plan and sign the answer
Per-tool daily counts (uses, tokens kept out)	Sync	Your dashboard
Per-task token totals, kind of task, which feature was held off	Sync	The control-group proof

Noised task counts and forecast-relative sums per kind of task and feature	Sync, unless `shareLearning: false`	Starting points for new repos
Field measure counts per kind of task and arm (V-op, V-dur, turnover), one row per repo	Sync, unless `shareField: false` or `shareLearning: false`	Checking whether released changes hold up in real use
Error hashes plus well-known config file names that fixed them	Sync, unless `shareFingerprints: false`	Shared fingerprints
File paths and co-change counts; fingerprint hashes with fix paths	Team plans, sync	Shared Repo Memory and fingerprints, for your team only
Name, repo folder name, branch, edited paths	Team plans, heartbeat	Team Relay level 1, for your team only
Task goals, plan titles, memo answers, handoff notes	Team plans at level 2	Plans, handoffs and the board, for your team only
Stated decisions (as knowledge drafts), only with `"shareDecisions": true`	Team plans at level 2	Team Knowledge drafts for your team only
Which knowledge records a session was given, and the result of a handoff's resumption check (pass/fail)	Team plans at level 2	Retrieval audit and handoff measures, for your team only
Handoff contract drafts (tests carry the repository revision they ran at): goal, file paths, test commands with pass/fail and times, risks, next steps	Team plans at level 2	Team Knowledge and handoff contracts, for your team only; approved knowledge reaches sessions in the same repo only

GitHub webhook metadata: PR numbers, branches, authors, review and CI states, changed file paths (PR titles at level 2)	Only if an admin connects GitHub	Pull requests, review blockers and the Conflict Radar. Commit messages and PR bodies are dropped on arrival; kept for the team's retention period

Your Claude account

TokenBuild never asks for, stores or uses your Claude login or subscription. It doesn't offer Claude sign-in, share subscriptions, or route your usage through its servers. You run official Claude Code, and your requests go from Claude Code to Anthropic as they always do. TokenBuild is independent and not affiliated with Anthropic.

How the plan check is protected

When the CLI confirms your plan, the server answers with a token signed using an Ed25519 private key that never leaves the server. The CLI checks the signature with a public key built into TokenBuild. A hand-written or edited plan file fails the check. The token is tied to your CLI key and lasts 14 days, which is what makes offline work possible.

Account security

- Passwords are stored hashed, never in plain text. Sign-in is rate-limited.
- Password reset links work once, for 1 hour.
- Payments are handled by Stripe. TokenBuild never sees your card details.
- CLI keys are shown once. Replace a key at any time on the CLI key page; the old one stops working immediately.

Deleting your data

- **Local data:** delete `.tokenbuild` in a repo, and `~/.tokenbuild`.
- **Learned state only:** `tokenbuild learn --reset`.
- **Your account and server-side counts:** email support@tokenbuild.ai from your account's address. See the Privacy Policy for retention and your rights.

Back to top · Next: Errors and fixes

CHAPTER 18

Errors and fixes

Every message below is one TokenBuild can actually show, grouped by where you'd see it. Search this page for the exact words of the message you got. If nothing here helps, run `npx tokenbuild doctor` and email the output to support@tokenbuild.ai.

TIP

The golden rule: TokenBuild is designed to step aside rather than break anything. If something seems wrong in the middle of work, Claude Code keeps working normally without it. Fix it when convenient.

Installation

The installer stops at "1/3 Node.js" with "Please install Node.js (the LTS button) from https://nodejs.org"

Cause: Node.js is missing or older than 20, and it couldn't be installed automatically. On Windows this usually means winget isn't available, which is common on managed work laptops.

FIX

Install Node.js LTS from nodejs.org, close and reopen PowerShell or Terminal so the new PATH loads, then paste the install line again.

"Node.js didn't install"

Cause: winget ran but Node isn't on the PATH yet, or a company policy blocked it.

FIX

Open a new terminal and run `node -v`. If it prints v20 or higher, paste the install line again. If not, install Node.js by hand from nodejs.org, or ask IT to allow it.

The installer stops at "2/3 Claude Code's command-line tool"

Cause: Claude Code's installer failed. This is usually a network proxy, or a corporate policy that blocks downloads.

> **FIX**
> Install Claude Code by hand with Anthropic's instructions (see Links), check that `claude --version` works in a new terminal, then paste the install line again.

PowerShell says "running scripts is disabled on this system"

Cause: your execution policy blocks scripts.

> **FIX**
> Run `Set-ExecutionPolicy -Scope CurrentUser RemoteSigned` in PowerShell, answer Y, and paste the line again. Or use `npx tokenbuild install <key>` if Node is already installed.

"curl: command not found" (Linux)

> **FIX**
> Install curl with your package manager (for example `sudo apt install curl`), or use `npx tokenbuild install <key>`.

"Couldn't save the key in the plugin settings: run /plugin configure tokenbuild@tokenbuild"

Cause: the plugin installed, but Claude Code didn't accept the key option. This sometimes happens with older Claude Code versions or the desktop app.

> **FIX**
> In Claude Code, run `/plugin configure tokenbuild@tokenbuild` and paste your key, or run `npx tokenbuild login <key>`. Either one works.

"No CLI key on this computer yet: run npx tokenbuild login <key>"

> **FIX**
> Get your key from the CLI key page (create a new one if you didn't save it) and run that command.

Something failed and I want to send the log

The installer saves `tokenbuild-install.log` in your temp folder (`%TEMP%` on Windows, `$TMPDIR` or `/tmp` on Mac and Linux). It contains no keys or code. Attach it to an email to support.

Plan and key messages

"TokenBuild tools need an active trial or plan. Start one at app.tokenbuild.ai/billing, create a CLI key at app.tokenbuild.ai/keys, then run: tokenbuild login <key>"

Cause: no key on this computer and no confirmed plan.

> **FIX**
> Start a trial on Billing, create a key on CLI key, and run `npx tokenbuild login <key>`.

"TokenBuild: couldn't confirm your plan yet. Run: tokenbuild login"

Cause: there's a key, but no confirmed answer from the server yet, often because the first check happened offline.

> **FIX**
> Connect to the internet and run `npx tokenbuild login`.

"TokenBuild: your plan hasn't been confirmed for 14 days. Connect to the internet and run: tokenbuild login"

Cause: the signed plan answer expired because the machine has been offline for 14 days, or a firewall blocks `app.tokenbuild.ai`.

> **FIX**

> Run `npx tokenbuild login` while online. On a locked-down network, ask IT to allow HTTPS to `app.tokenbuild.ai`.

"TokenBuild: your saved plan could not be verified. Run: tokenbuild login"

Cause: the plan file in `~/.tokenbuild` was edited, corrupted, or copied from another key.

> **FIX**
> Run `npx tokenbuild login`. It replaces the file with a fresh signed answer.

"TokenBuild: this CLI key is no longer valid. Create a new one at app.tokenbuild.ai/keys..." or "This CLI key is not valid..."

Cause: the key was replaced, perhaps by you on another computer, or mistyped.

> **FIX**
> Use your latest key, or create a new one on the CLI key page, then run `npx tokenbuild login <key>` on every computer. Also update the plugin with `/plugin configure tokenbuild@tokenbuild`, or clear its key, so the old key doesn't linger there.

"TokenBuild: no active trial or plan (canceled). Renew at app.tokenbuild.ai/billing"

Cause: the subscription ended. The status in brackets says why, for example `canceled` or `unpaid`.

> **FIX**
> Restart a plan on Billing. Your local data is still there, and everything picks up where it left off.

"Missing CLI key", "Missing or invalid CLI key", "Invalid key"

These are server replies to the CLI. They mean the same as the messages above: run `npx tokenbuild login <key>` with your current key.

I paid, but the tools still say there's no plan

> **FIX**

> Billing can take a few seconds to update after checkout. Reload Billing until it shows Trial or Active, then run `npx tokenbuild login`.

Plugin not loading

The tb_ tools or /tokenbuild: skills don't appear

> **FIX**
>
> 1. Restart Claude Code fully. Plugins load at start.
> 2. Run `claude plugin list` and check that `tokenbuild@tokenbuild` is installed and enabled. If it isn't, run `claude plugin enable tokenbuild@tokenbuild` or reinstall.
> 3. In Claude Code, run `/mcp` and check that the tokenbuild server is connected. If it failed, the error is usually Node: run `node -v` (20+ is needed).
> 4. Run `/tokenbuild:setup`. It names what's missing.

Hooks don't seem to run in one repo

Cause: usually project mode. If the repo's `.claude/settings.json` already runs TokenBuild hooks from `tokenbuild init`, the plugin steps aside there, by design. If those project hooks point at an old path, nothing runs.

> **FIX**
>
> Run `npx tokenbuild init` again in that repo to refresh the entries, or remove TokenBuild's entries from `.claude/settings.json` to use the plugin.

Everything counts twice

That shouldn't happen, because the plugin checks for project hooks. If you've copied hooks by hand under a different command, remove one set.

"tokenbuild: command not found" in my own terminal

The plugin puts `tokenbuild` on Claude Code's PATH, not your shell's. Use `npx tokenbuild <command>`, or install globally with `npm i -g tokenbuild`.

The wrong folder is treated as the project

The root is the nearest folder above your working directory that contains `.git` or `.tokenbuild`. If you open Claude Code in a subfolder of a monorepo with its own `.git`, that subfolder is the root. Open Claude Code at the root you want, or run `git init` there.

Messages Claude gets during a session

These are advice to Claude, not errors. You'll see them in tool results. Each says how to proceed.

"TokenBuild: <file> is generated/vendored (~N tokens). Read the source file or package.json instead. Retry only if you truly need it."

Waste Guard. If the file really is needed, for example when debugging a lockfile conflict, Claude can retry and the read goes through.

"TokenBuild: <file> is ~N tokens. Use tb_find to get line numbers, then Read with offset/limit. Retry to read it all anyway."

A whole-file read of a big file. Usually the right response is a targeted read. If Claude keeps needing whole big files, consider `tokenbuild shape`.

"TokenBuild: this command prints a long log. Run it as: tokenbuild run ..."

The Output Shrinker's suggestion for a command that would need a permission prompt to wrap. Approve the wrapped command, or add an allow rule so it's wrapped automatically next time (`tokenbuild permissions`).

"TokenBuild preflight: this Edit would leave <file> with a syntax error..."

Edit Preflight. Claude fixes the edit. If the intermediate broken state is intentional, resending the same edit lets it through.

"TokenBuild loop check: `<command>` has failed 3 times with the same error" or "...you have reverted your own change in <file> 2 times..."

Loop Breaker. Claude should stop, name the root cause, then make one deliberate change or ask you. A hint from you helps most here.

"TokenBuild Cache Keeper: you were away N min..."

Type `/clear` and send your message again for a small fresh start, or just resend it to continue the big conversation.

"TokenBuild Team Relay: <teammate and branch> ... Keep this change small and in a different part of the file, or take another plan item"

Conflict radar. Keep the change small and in a different part of the file, or take another plan item. Sending the same edit again goes through.

"Already in your context" when Claude really needs to see the file

That's the Context Ledger. Asking again gets the full text, and the saving is subtracted. If it happens often in one session, for example after a big summarization, run `tokenbuild guard --reset`.

Verify Loop and tests

The Verify Loop never reports

Causes: no supported runner was detected, no test file links to the changed file, or `verify` is set to `false`.

> **FIX**
> Check with `npx tokenbuild verify src/that-file.ts`. If it finds no tests, link them by import or matching name (`x.ts` ↔ `x.test.ts`), or set `"verify": "<command> {files}"`.

It reports a failure that I can't reproduce

Run the exact command from the log path it gives. Different environment variables (in CI or `.env`) are the usual cause. If the test passes on its own, it may be order-dependent or flaky. Reruns mark it flaky after two flips.

It skipped a test that would have failed

Tests are skipped only with `"testSelect": "time"`, which keeps the combined risk of skipped tests under 2%, so this can occasionally happen. The default never skips in practice. Run the full suite before merging (`tb_verify` scope `full`, or CI). To never skip, set `"testSelect": false`.

A test is reported as flaky but it's really broken

It passed on rerun twice. Fix its nondeterminism (timing, shared state, network). Set `"flakyRerun": false` if you'd rather see every failure as a failure.

Tests run too slowly in the background and slow my machine

Point `verify` at a faster command (for example with `--bail`), or narrow it. Selection and ordering already put the riskiest tests first.

Index, find and brief

`find` returns nothing, or old line numbers

> **FIX**
>
> Run `npx tokenbuild index`. Check the language is supported (Repo Index) and the file isn't in an ignored folder such as `dist` or `vendor`.

The brief lists the wrong files

Early on it has only the index to go on. It improves as Repo Memory and the ranking model learn from finished tasks, usually within 30 tasks. Naming files or symbols in your prompt helps right away. See Tips.

The brief or Next-Read stopped appearing

The learning layer may be holding it off for this kind of task because it didn't pay here, or this may be a control-group session. `tokenbuild learn` shows which.

Local Boost

"...Install Ollama (https://ollama.com), then: ollama pull qwen2.5-coder:7b"

Ollama isn't running or isn't installed. Install it, start it, then pull the model.

"Ollama is running but has no model..."

Run the `ollama pull` command it suggests, or choose an installed model with `tokenbuild boost on --model <name>`.

`tb_digest` doesn't appear

Restart Claude Code after `boost on`. The tool is only offered when Boost is on and a model is found.

Answers are slow or poor

Use a smaller model on slower machines, or a larger coding model if you have a GPU. Local Boost is for reading and locating. Leave real code changes to Claude.

Dashboard

The dashboard shows zeros or looks out of date

FIX

1. Run `npx tokenbuild sync` in a repo where you've worked, then reload.
2. Make sure you're signed in to the same account the key belongs to.
3. Check that `syncEvery` isn't 0 and `syncOnEnd` isn't `false`.
4. Run `npx tokenbuild doctor` to confirm the key and plan.

The "Measured across all TokenBuild users" card says it's loading or has no data

It needs enough control-group sessions across accounts. It fills in over time. Your own proof is in `tokenbuild proof`.

The Team Relay card is missing

It appears only on Team plans, or for people who joined a team.

Sign-in and account

"Email or password is incorrect."

Check for typos and Caps Lock, or use **Forgot password**.

"Too many attempts. Try again in a minute."

Sign-in is rate-limited for security. Wait a minute.

"An account with this email already exists. Sign in instead."

Sign in, or reset the password if you've forgotten it.

"Enter a valid email."

Check the address format.

"Please agree to the Terms and Privacy Policy."

Tick the box on the sign-up form.

"This reset link has expired or was already used. Ask for a new one."

Links work once, for 1 hour. Request a new one on Forgot password.

The reset email never arrives

Check spam and promotions folders for mail from hello@tokenbuild.ai. Corporate filters sometimes hold it for a few minutes. Still nothing? Email support from that address.

"Sign in first."

Your session expired. Sign in again.

Billing

"Please agree to the auto-renewal terms."

Tick the renewal agreement on Billing before continuing to payment.

"You already have a plan. Use Manage billing."

You can't start a second subscription. Change plan or seats in Manage billing.

"No billing yet. Start your trial first." / "No active subscription found. Use Manage billing."

Start a trial first, or, if you had one, use Manage billing to see its state.

"No subscription to upgrade yet: start a Team plan instead."

Upgrading needs an existing Solo subscription. Without one, choose Team on Billing.

"You are already on the Team plan."

Nothing to upgrade. Change seats in Manage billing.

"Checkout is not open yet." / "Upgrades are not open yet."

Payments are temporarily unavailable on the server. Try again later, or contact support.

Status shows "Payment due"

A charge failed. Your tools stay on while Stripe retries. Update your card under Manage billing.

"Checkout was cancelled. Nothing was charged."

You left the Stripe page before paying. Start again when you're ready.

Team workspace and Relay

"The Team dashboard is part of the Team plan." / "Team Relay is part of the Team plan." / "Shared memory is part of the Team plan."

Upgrade to Team on Billing, or ask to join an existing team.

"Turn on Team Relay level 2 first (an admin can, on the Overview)."
"Passing a task needs Team Relay level 2 (dashboard)."
"The shared plan needs Team Relay level 2 (dashboard)."

Plans, the board and handoffs share task goals, which level 2 allows. An admin turns it on.

"Item N is already claimed by <name>. Pick another."

Another teammate's Claude is on it. That's the duplicate-work protection working. Take another item.

A claim is refused because the item is assigned to someone else, or its dependencies aren't done

Pick a ready item that's assigned to you. Your prompt says when one becomes ready.

"A plan holds 40 items. Finish or remove some first."

Split the work across projects, or remove done items.

"That task is already on the plan."

Item ids come from their text, so identical items are refused. Reword the item if it really is different work.

"Everything on this plan is done."

Nothing is left to claim or deploy. Add the next items.

"Who to? tokenbuild relay send <name>" / "Pick who gets it." / "No such person on the team."

Name a teammate as they appear in `tokenbuild relay` or on the People view.

"Say what the task is." / "Write the task first."

A handoff or plan item needs text.

"No team found for that email. Ask the team owner for an invite link."

Use the owner's account email exactly, or an invite link.

"That invite code is not valid any more. Ask your team admin for a new one."

Invites can be replaced. Get a fresh link.

"You are on (or have asked to join) another team. Leave it first."

A person can be on one team at a time. Leave on People, or cancel the pending request.

"You already run a team on your own Team plan." / "That is your own team."

Owners can't join another team while they own one.

"Approve or decline the request first." / "Approve as coder or viewer (or admin)."

Pending requests need a decision before their role can change.

"The owner is always an admin." / "A team needs an admin."

The owner's role can't be lowered, and the last admin can't be removed.

"That repo is already a project here."

Open the existing project instead of adding a new one.

"The code repo link should start with https:// (or be a git@host:org/repo address)." / "The project files link should start with https://."

Paste the full URL.

"Give the project a name."

Projects need a name.

The board is empty

Relay starts when a teammate opens Claude Code in a repo with a git remote, with TokenBuild installed and signed in to an account on the team.

A teammate's handoff didn't arrive

It's picked up on their next Claude prompt *in that repo*. Check the Handoffs view to see whether it's waiting, and that they're working in the same repository (the same remote).

Toki

"Daily question limit reached." / "AI answers are not enabled." / "AI unavailable."

Toki's AI answers have per-visitor and daily limits, and may be off. Knowledge-base answers always work. For anything else, this manual and support are always available.

Generic server replies

"Server error", "Not found", "Bad JSON", "Send JSON.", "Unknown action."

Usually a temporary problem or an out-of-date CLI. Update with `npx tokenbuild release update`, and try again. If it persists, email support with the time it happened.

"Network problem. Please try again."

The web app couldn't reach the server. Check your connection and reload.

Back to top · Next: Tips and tricks

CHAPTER 19

Tips and tricks

TokenBuild works without any change to how you work. These habits, collected from heavy users, make it work noticeably better.

Writing prompts that brief well

- **Name something concrete.** A file, function, route, component, table or the exact error text. The brief matches on it, and Claude starts in the right place.
- **Paste the error, not a description of it.** "TypeError: Cannot read properties of undefined (reading 'amount')" matches a fingerprint. "It crashes on invoices" matches nothing.
- **Say what "done" means.** "...and the invoice tests pass" tells Claude which check proves it, and the Verify Loop will confirm it.
- **One task per prompt** when you can. Tasks are measured and learned individually, and smaller tasks get better briefs and budgets.
- **Answer the clarifying question.** When the brief has nothing to match, Claude asks one question. Ten seconds of your time saves a long search.

Managing sessions

- **Clear at milestones.** After a verified, committed step, `/clear`. The handoff note and project state carry everything important, and the next turn costs a fraction as much.
- **Take Fresh Start seriously.** When it says a fresh start is cheaper, it is: every turn past 100K tokens re-sends all of it.
- **Before a long break,** run `/tokenbuild:handoff`, then `/clear` when you're back. It avoids re-caching a huge conversation.
- **Use the scout for broad questions.** "Send the scout to map how emails are sent" keeps the exploration out of your main context.

- **Check** `tokenbuild profile` when a session feels sluggish. If most of it is dead weight, clear.
- **Record decisions** with `tokenbuild state --decide "…"`. They load in every new session for about 20 tokens each and stop Claude re-arguing settled questions.

Setting up a repo for best results

- **Allow your test, build and lint commands** (`tokenbuild permissions --apply`). Then the Output Shrinker wraps them automatically.
- **Name tests after their sources** (`billing.ts` ↔ `billing.test.ts`) and import what they test. The Verify Loop then finds them every time.
- **Keep CLAUDE.md short at the root**, and run `tokenbuild claudemd` to move folder-specific rules down.
- **Run** `tokenbuild diet` after a couple of weeks. Unused MCP servers are a silent per-turn cost.
- **Commit** `.claude/skills/tb-recipe-*` so teammates get your recipes, and ignore `.tokenbuild/`.
- **Split the file everyone reads.** `tokenbuild shape` shows which one, and it usually saves more than any setting.

Tools worth asking for by name

Say	When
"Start with tb_map"	A new or unfamiliar repo
"Use tb_pack for: …"	A cross-cutting task where you don't know the files
"Use tb_apply to rename X to Y everywhere"	Renames and API migrations
"Check tb_docs for zod#object first"	A library that changed recently

"Run tb_risk before we push"	Before merging anything non-trivial
"tb_verify with scope full"	Before you call it finished
"Record that as a decision"	After settling an approach

Handy command-line tricks

- `npx tokenbuild run <cmd>` works great for you, not only for Claude: a one-screen summary of a 5,000-line build failure.
- `npx tokenbuild find <name>` is a fast "go to definition" across languages.
- `npx tokenbuild meter --json` pipes into `jq` or a spreadsheet for cost reports.
- `npx tokenbuild audit --days 7 --project <name>` compares one project's waste week over week.
- `npx tokenbuild risk --incident "<error>"` is the fastest first step in an incident.
- `npx tokenbuild autopilot "<task>"` before you start tells you whether to break the task up.

Team tips

- **Write specs.** "Done when", constraints and starting files are the cheapest tokens you'll ever spend: they replace the receiving Claude's exploration.
- **Use dependencies,** not chat messages, for ordering. The plan then refuses out-of-order claims and announces when items become ready.
- **Deploy by stage,** with auto-advance on for well-specified work.
- **Act on collision warnings** before the first edit. Merge conflicts in AI-written code are expensive to untangle.
- **Hand off instead of re-explaining.** `relay send` carries the state, and the receiver starts at your last verified step.
- **Keep names consistent.** Set `relay name` to match what's on People, so handoffs and assignments find the right person.

Keeping costs predictable

- Watch the forecast line. A task forecast at 200K+ tokens is worth splitting.
- When the early warning fires, step in. It's right more often than not by design: it only speaks at 60%+ confidence.
- Use the `balanced` route. `lean` saves more on subagents, but can cost extra turns on hard subtasks.
- Compare `tokenbuild meter` week over week, not session by session. Tasks vary a lot.

Back to top · Next: Improving your results

CHAPTER 20

Improving your results

If the savings are lower than you expected, or you want to get more from TokenBuild, work through this chapter. It starts with diagnosis, then the improvements that usually matter most, then the known limits.

Diagnose first

1. `npx tokenbuild doctor`: is everything actually on? A missing key, a stale plan or a project-mode conflict explains most "no savings" reports.
2. `npx tokenbuild audit`: how much waste is left? If the audit finds little, your sessions were already lean, and savings will be modest.
3. `npx tokenbuild stats`: which tools are working? A zero for the Output Shrinker usually means commands aren't being wrapped (no allow rules). A zero for the Verify Loop means no runner was found.
4. `npx tokenbuild proof`: which features help here, measured? You need a few weeks of sessions for clear results.
5. `npx tokenbuild learn`: what has learning switched off, and why?

The improvement checklist

Symptom	Likely cause	Improvement
Output Shrinker rarely used	Test and build commands need a permission prompt, so they aren't auto-wrapped	`tokenbuild permissions --apply`
Verify Loop silent	Unsupported runner, or tests not linked to sources	Set `verify` with `{files}`; name tests after sources

Brief often wrong	Vague prompts; new repo with little history	Name files, symbols and errors; give it 30 tasks
Sessions regularly over 100K tokens	Never clearing	`/clear` at milestones; lower `handoffAt`
High cost per turn even in short sessions	Many MCP servers, or a huge root CLAUDE.md	`tokenbuild diet`, `tokenbuild claudemd`
Same file read in most tasks	A "god file"	`tokenbuild shape --plan 1`
Many fix cycles	Flaky tests; Claude guessing at library APIs	Fix flaky tests; ask for `tb_docs`
Loops on the same failure	Missing context only you have	Give a hint when the Loop Breaker speaks
Subagent spend high	Helpers running on your main model	`init --route balanced`, or check plugin routing
Team members duplicating work	Relay level 1 only, no shared plan	Level 2, a shared plan with assignments

The three habits that matter most

1. **Clear often.** Context size is the biggest single driver of cost. A disciplined `/clear` after each verified step can save more than every other feature combined, and TokenBuild's handoff notes make it painless.
2. **Be specific.** Concrete prompts let the brief and memo work, and cut exploration.
3. **Keep tests fast and linked.** The Verify Loop turns "edit and hope" into "edit and know", which is where fewer turns come from.

Known limits

- **Edit Preflight** checks syntax only, for JavaScript, TypeScript, JSON and Python. It doesn't type-check, and other languages aren't checked.
- **The Repo Index** covers 11 languages. Others get Waste Guard and the Context Ledger, but no symbol map.
- **The Verify Loop** auto-detects five runners. Others need a `verify` command.
- **Learning needs history.** In the first weeks, defaults and pooled priors decide most things.
- **Control-group proof** needs volume. Light users may wait a while for confident numbers.
- **Very small repos and tasks** have little waste to remove.
- **Team features** rely on a shared git remote to match teammates to a project.

Suggesting improvements

Ideas, missing languages, runners you'd like detected, and confusing messages are all welcome at support@tokenbuild.ai. The most useful reports include the output of `npx tokenbuild doctor` and, for a specific message, its exact text. Never send code or keys. They're not needed.

Back to top · Next: Glossary

CHAPTER 21

Glossary

Baton

A task passed to a teammate with Team Relay: task state plus handoff note.

Brief (Task Brief)

The short context TokenBuild adds to your prompt: likely files, related files, tests, past fixes.

Cache (prompt cache)

Claude Code's store of the conversation so far. Reading it costs a fraction of normal input. It expires after a few idle minutes.

CLI key

The secret that connects your computers to your plan. Created on the CLI key page, shown once.

Codemod

One find-and-replace applied across many files (`tb_apply`).

Command group

One of the three CLI groups, `save` , `team` and `learn` , one per product. `tokenbuild <group>` lists its commands; the single-word names still work.

Compaction

Claude Code summarizing a long conversation to make room. TokenBuild saves a handoff note first.

Conformal range

An interval that past tasks fell inside 80% of the time. Shown with forecasts and budgets.

Context window

Everything Claude can see in the current conversation. All of it is re-sent on every turn.

Control group (holdout)

The roughly 10% of sessions with one feature held off, used to measure that feature's effect.

Dead weight

Content in the context that nothing has used since it arrived.

Doubly robust estimate

An unbiased way to estimate what a policy would have scored, from logged decisions and their probabilities.

Fingerprint

A one-way hash of a normalized error, linked to the files that fixed it.

Flaky test

A test that failed, then passed on rerun with no change, twice.

Handoff note

A compact summary of the current work that a fresh session starts from.

Hook

A program Claude Code runs at a fixed moment in a session.

Kind (of task)

Question, fix, feature, refactor or test. Learned from outcomes.

Ledger

Either the savings ledger (counted events) or the Context Ledger (what Claude has read this session).

MCP

Model Context Protocol: how tools like `tb_find` are offered to Claude.

ML Validation and Economics

The product that uses models trained on your repo's history to check work (Verify Loop, test selection, change risk) and to spend tokens where they pay off (learning, budgets, proof, Replay, recipes, repo shape). Commands: `tokenbuild learn`.

Policy

The learned rule for whether to give the brief and Next-Read for a kind of task.

Project State

Durable repo facts (decisions, recent work, outstanding items) injected at session start.

Recipe

A skill TokenBuild writes from tasks you repeat. Retired if it doesn't save.

Relay level

How much Team Relay shares: level 1 (names, branches, paths, hashes) or level 2 (also goals, plans, memos, handoffs).

Scout

TokenBuild's low-cost, read-only explorer subagent.

Stage

A group of team plan items whose dependencies are all in earlier stages.

Task State

The live record of the current task: goal, plan, files, checks, facts.

Team Build

The product that keeps task and project state outside the conversation and coordinates a team's Claude sessions: Task State, Project State, Preflight, Autopilot, Team Relay and the team plan. Commands: `tokenbuild team`.

Thompson sampling

A method of choosing between options that balances trying each one with using the best so far.

Token

The unit Claude reads and writes, about four characters of text.

Token Saver

The product that keeps what wouldn't help out of Claude's context, cuts what Claude writes and counts every token saved. Commands: `tokenbuild save`.

Undo

A held-back read or answer that Claude re-requested. Its saving is subtracted.

Verified

A task with a passing check after its last edit.

Back to top · Next: Worked examples

APPENDIX A

Worked examples

These walkthroughs show TokenBuild in real situations, step by step, with what you type, what TokenBuild adds, and why it saves. The repos and numbers are illustrative. Your exact output will differ.

Example 1: Fixing a bug from an error message

Situation. A customer reports a 500 error when applying a discount code. Your server log shows:

```
TypeError: Cannot read properties of undefined (reading 'percent')
    at applyDiscount (src/checkout/discount.ts:57:31)
```

1. Prompt with the real error. Paste it into Claude Code: "Fix this: TypeError: Cannot read properties of undefined (reading 'percent') at applyDiscount (src/checkout/discount.ts:57). It happens when the code has expired."

2. What TokenBuild adds. The brief matches `applyDiscount` and the file path in the index. It lists `src/checkout/discount.ts` (applyDiscount:52, loadCode:18), the related `src/checkout/codes.ts` that usually changes with it, and the test file `discount.test.ts`. If this error was fixed before, the fingerprint points at the files that fixed it last time. The forecast says something like "similar fixes here: ~20K tokens".

3. Claude reads only what it needs. It reads lines 40–80 of `discount.ts` rather than the whole 900-line file. Waste Guard would have suggested that anyway.

4. The edit. Claude adds a guard for expired codes. Edit Preflight checks the syntax before it lands. The Verify Loop runs the 9 tests in `discount.test.ts` and the 3 in `checkout.test.ts` that import it, skipping unrelated suites, and reports "12 tests pass".

5. The proof of done. Ask Claude to add a test for an expired code. The Verify Loop runs it, and it passes. The task is now *verified*, and the fingerprint records `discount.ts` as the fix for this error.

Why it was cheaper. No exploration rounds, a partial read instead of a whole file, a targeted test run with one line of output, and no failed fix cycle. Next time this error appears, on any branch, Claude starts at the fix.

Example 2: Adding a feature in an unfamiliar repo

Situation. You've just joined a project and need to add CSV export to the reports page.

1. **Orient.** "Start with tb_map, then tell me how reports are generated." Claude gets a 2,500-token map of the most-imported files and their main symbols, then sends the scout to read the report pipeline. You get a summary with file paths and line numbers, and your main context stays small.
2. **Plan.** "Use tb_preflight for: add CSV export to reports." TokenBuild proposes steps (add a serializer, add an endpoint, add a button, add tests), with the files, tests, an estimate and the stage for each.
3. **Budget.** The Autopilot line in the brief shows the budget split by phase. If the planning phase runs more than twice its budget, you're told.
4. **Build step by step.** For each step, Claude edits, and the Verify Loop confirms. Ask Claude to record decisions as you make them, such as "decide: CSV uses RFC 4180 quoting, UTF-8 with BOM for Excel".
5. **Check docs at your versions.** "Check tb_docs for csv-stringify#stringify." Claude uses the API of the version you installed, not the one it remembers.
6. **Clear between steps.** After step 2 passes and you commit, `/clear`. The project state and handoff note carry the plan, so step 3 starts at about 2K tokens of context instead of 60K.
7. **Before pushing.** "Run tb_risk." It flags a test gap in the endpoint and suggests the integration test that caught failures in that folder before.

Example 3: A large rename with Codemod Mode

Situation. `getUser` is being renamed to `fetchUser` across 140 files, with the import path changing from `lib/users` to `lib/accounts`.

1. Commit your current work so the change is easy to review.
2. "Use tb_apply to rename getUser to fetchUser as a whole word, and change imports from 'lib/users' to 'lib/accounts'. Show the counts first."
3. Claude runs the codemod locally and gets a summary: files changed and matches per file. It doesn't get 140 diffs. With standard edits, each file would have cost a read plus old-and-new text.
4. The Verify Loop runs the affected tests. Because the change touches many files, selection and ordering matter: the riskiest tests run first.
5. Review with `git diff --stat`, then run the full suite with "tb_verify scope full".

> **WATCH OUT**
> Text codemods don't understand scope. Check for matches in strings, comments and unrelated identifiers. A whole-word match and a careful pattern avoid most surprises.

Example 4: Rescuing a long, expensive session

Situation. You've been debugging for two hours. Each reply is slow, and the meter shows the conversation is at 160K tokens.

1. TokenBuild has already said, once, that a fresh start is cheaper, and has written a handoff note.
2. Run `tokenbuild profile` or ask "tb_profile". It shows 110K of the 160K is dead weight: old logs, files from an abandoned approach and search results.
3. Ask Claude to record what it has established as facts: "tb_task fact: the race is in the cache invalidation, not the DB." Facts carry into the handoff note.
4. Type `/clear`. The new session starts from the note: the goal, the plan, the files that matter, the facts, and the failing check. It's about 2K tokens.
5. Continue. Each reply now re-sends a few thousand tokens instead of 160K.

Example 5: Production incident

Situation. Alerts fire after a deploy: `Error: connect ETIMEDOUT` from the payment worker.

1. Run `npx tokenbuild risk --incident "Error: connect ETIMEDOUT 10.0.3.7:5432"`. The fingerprint ignores the IP and port. It shows the files that fixed this error before (the pool config) and recent work there, including yesterday's change to the pool size.
2. Start Claude with that context: "Incident: ETIMEDOUT from the payment worker. TokenBuild points at src/db/pool.ts, changed yesterday."
3. Claude reads the change, finds the pool size dropped from 20 to 2, and proposes a revert. The Verify Loop confirms.
4. The risk model labels yesterday's risk check on that file as bad, so similar changes will be flagged higher in future.

Example 6: A team sprint with staged plans

Situation. Three developers ship a "teams" feature: schema, API, UI and docs.

1. **Plan.** The lead runs `tb_preflight` for the goal and shares it to the team plan. On the Team dashboard, the steps become items in the project.
2. **Organize.** The lead sets **after #** dependencies (API after schema, UI after API), writes a spec for each item ("done when: POST /teams returns 201 with the new team; constraints: no breaking change to /users; files: src/api/teams.ts"), and assigns people. Insights suggest assignees for the rest, based on who knows those folders.
3. **Deploy stage 1.** The schema item goes to its owner's next prompt with the spec. Auto-advance is on.
4. **Work.** When the schema item is done and verified, stage 2 is sent automatically. The API developer's prompt says the item is ready. A coder can also run `tokenbuild plan run` to work through their ready items with Claude Code.
5. **Avoid collisions.** Insights flag that the UI and API developers will both touch `src/types/team.ts`. They agree who owns it before either edits.

6. **Hand off.** At the end of the day, one developer runs `tokenbuild relay send sam --note "endpoint done, pagination left"`. Sam's next prompt starts from that state.
7. **Review.** The dashboard shows estimate versus actual for each item, check spend, and the coordination savings from a refused duplicate claim.

Example 7: Using TokenBuild in CI and scripts

The Output Shrinker and change risk are useful outside interactive sessions too.

```
# CI step: condensed test output in the job log, full log as an artifact
- run: npx tokenbuild run npm test
  env: { TOKENBUILD_KEY: ${{ secrets.TOKENBUILD_KEY }} }
- uses: actions/upload-artifact@v4
  if: failure()
  with: { name: test-logs, path: .tokenbuild/logs }
```

Locally, `npx tokenbuild risk --install-hook` prints the risk before every push. It never blocks the push.

Example 8: Onboarding a teammate in ten minutes

1. An admin sends an invite from **People** → **Invite people**.
2. The teammate creates an account, opens the invite link, and asks to join. The admin approves them as coder.
3. The teammate creates a CLI key and runs the install line.
4. They open Claude Code in the repo. Relay connects, and the project state, shared Repo Memory, fingerprints and committed recipes mean their very first brief already knows where past tasks like theirs ended up.
5. They run `/tokenbuild:setup` to confirm, and pick a ready item assigned to them.

Back to top · Next: More questions answered

APPENDIX B

More questions answered

Questions that come up in support and don't fit neatly elsewhere. For the short list, see the FAQ.

General

Does TokenBuild make Claude worse at anything?

It's designed not to. It never changes your main model and never hides anything Claude asks for twice. It's also measured: control groups compare tasks with and without each feature, on tokens to a verified result. If a feature hurt in your repo, learning would turn it down, and the guardrail stops it from turning off anything whose absence lowers verification.

Will it work with the Claude desktop app, or only the terminal?

Both. The installer puts the plugin into Claude Code, which the desktop app's coding sessions use too. IDE integrations that run Claude Code also load it.

Does it work with other AI coding tools?

No. TokenBuild is built on Claude Code's hooks, MCP server, skills and subagents. The CLI commands such as `run`, `find`, `audit` and `risk` work on their own in any terminal.

Does it change my code?

Only in ways you can see: it formats files Claude edited with your own formatter (turn it off with `"format": false`), and with `--apply` the tuning commands change Claude Code settings. It writes recipes under `.claude/skills`. Claude makes all code changes, through tools you can review.

What happens during an outage of TokenBuild's servers?

Nothing visible. Hooks never call the network, plan checks fall back to the signed answer (valid 14 days), and sync retries later. Team Relay pauses until the server is back.

Is the CLI open source?

The CLI is published on npm, so you can inspect the exact package that runs on your machine. The plugin's marketplace entry points at that npm package.

Using it day to day

Should I turn off auto-compact now that Fresh Start exists?

You don't need to. Fresh Start usually offers a clean start well before compaction would trigger, and if compaction does happen, TokenBuild saves a handoff note first. Many people prefer `/clear` at milestones because the result is more predictable than a summary.

Can I edit the handoff note before clearing?

Yes. It's `.tokenbuild/handoff.md`, plain Markdown. Edit it and then `/clear`. The edited note is what the new session gets.

Can I see exactly what the brief added?

Yes. It appears as hook context in Claude Code's transcript view. On the CLI, `tokenbuild pack "<task>"` shows the same kind of selection.

Claude ignores the tb_ tools. Why?

Claude chooses tools by itself. After a session or two in a repo, learned tips nudge it towards the ones that pay. You can always ask by name. Make sure the MCP server is connected (`/mcp`).

Can I use TokenBuild on a repo without git?

Yes. Create a `.tokenbuild` folder at the project root so TokenBuild knows where the project starts. Change risk, Team Relay matching and some memory features need git.

Does it slow Claude Code down?

Hooks are built to finish within 50 milliseconds, and heavy work runs in the background. Your formatter's run time at the end of a reply is the main visible cost, and you can turn formatting off.

How do I pause TokenBuild for one session?

Disable the plugin with `claude plugin disable tokenbuild@tokenbuild`, then enable it again later. To turn off individual features in one repo, use the settings in Chapter 16.

Numbers and billing

Why does the dashboard's total differ from the meter's?

They measure different things. The dashboard counts tokens kept *out* of context, tool by tool. The meter counts tokens actually *spent*, from Claude Code's records. The proof connects the two: spend per task with and without each feature.

Why are costs shown at API prices on my Max plan?

To give a consistent measure. On a subscription, the same tokens count toward your usage limits, so the figure shows how much of your allowance the work used.

Do seats need to match the people on the team?

Seats are what you're billed for (2–100). Add or remove them in Manage billing; changes are prorated.

Can I switch between monthly and yearly?

Manage billing shows the changes available for your subscription. If switching isn't offered there, email support and we'll arrange it.

Do you offer refunds?

See Cancellation & refunds for the policy. Cancelling stops all future charges immediately, and you keep access until the end of the period you paid for.

Is there an education or open-source discount?

Ask at support@tokenbuild.ai.

Teams and organizations

Can a teammate see my code through Relay?

No. Relay never sends code, file contents or prompts. At level 1, teammates see your name, branch, repo folder name and the paths of files you edited. At level 2, they also see task goals, plan titles, saved answers and the notes of tasks you pass on.

Can I be on a team and keep my own Solo plan?

You can be on one team at a time. A team owner's seats cover the team's members. Talk to your team owner about whether you need your own plan as well.

We have several repos. One project each?

Yes. Each project maps to one code repo. Sessions in that repo line up with it automatically.

Can viewers use the TokenBuild tools?

The viewer role is about the Team workspace: viewers see but don't change plans. Their own Claude Code tools depend on their plan or seat.

Do you support SSO or on-premises hosting?

Talk to us about Enterprise at Contact.

Security reviews

What do I send our security team?

The Security page, the Privacy Policy, the Subprocessors list and Chapter 17 of this manual. The short version: code, contents, prompts and error text stay local, and sync sends counts, hashes and, on Team plans, file paths.

Which network destinations does TokenBuild use?

`app.tokenbuild.ai` over HTTPS for plan checks, sync and Team Relay. The installer also downloads from npm, Node.js and Anthropic's official sources. Local Boost uses localhost only.

Can we audit what's sent?

Yes. Run `npx tokenbuild sync` behind an inspecting proxy, or read the published CLI on npm.

Still stuck? Email support@tokenbuild.ai with the output of `npx tokenbuild doctor` (it contains no code or keys), or ask Toki, the assistant in the corner of this page.



TokenBuild

Build more. Use less.

www.tokenbuild.ai · support@tokenbuild.ai

© 2026 TokenBuild LLC. Claude and Claude Code are products of Anthropic. TokenBuild is independent and not affiliated with Anthropic.