Don't Trust Me. Trust Your AI.

How to audit any token-savings claim in 15 minutes using the Claude stack.

Why this playbook exists

LinkedIn is full of "I saved 75% of my tokens with this tool" posts. Every week there's a new one. Every claim sounds plausible. Almost none of them survive a proper investigation.

I wasted two afternoons on two of them (RTK and Caveman). Both promised 60-90%. Both delivered single digits in real sessions. The marketing wasn't wrong on purpose, but it was measuring the wrong thing.

So instead of investigating each new tool one by one, I built a 5-step method you can run yourself on any future claim. It takes about 15 minutes. You don't need to trust me. You don't need to trust the repo. You let your AI investigate it for you and then measure the result against your own data.

Don't trust me. Trust your AI.

The 5-step method

Step 1: Feed the repo URL to Cowork

Open Cowork in your desktop app. Paste the investigation prompt from this playbook, with the tool's GitHub URL at the top. Hit send.

Step 2: Let Cowork read the source and audit the benchmark

Cowork will fetch the README, the benchmark script (if one exists), and the core implementation. It reports back with:

- The exact claimed savings percentage
- The baseline used in their benchmark (this is where most tools get caught)
- Whether the tool actually touches the part of a real session where tokens live
- Security or privacy concerns in the install
- A verdict: INSTALL, EXTRACT, or SKIP

This alone filters out 80% of the hype. Most tools fail at the baseline check.

Step 3: Let Cowork generate a Claude Code audit prompt

If the tool survives Step 2, Cowork writes a Claude Code prompt tailored to the specific claim. Different tools compress different slices of your session. The audit prompt matches the tool's actual scope.

Step 4: Run it in Claude Code

Open Claude Code. Paste the generated prompt. Let it parse your last 20 session logs. It reports back with a clean table showing:

1. Where your tokens actually go (tool results, file reads, assistant text, system)
2. What percentage of your session the tool can touch
3. The realistic session-wide savings if you installed it

This is the number that matters. Not the headline, not the benchmark. What YOUR workflow would actually save.

Step 5: Decide

Three possible outcomes:

- **INSTALL:** savings are real, scope is wide, security is clean. Rare.
- **EXTRACT:** useful idea, tiny implementation, don't trust the binary. Most common.
- **SKIP:** marketing without substance, or security risks, or both.

Then move on with your day.

Prompt 1: Cowork investigation

Copy this into Cowork. Replace the URL on the first line.

```
Investigate this repo: https://github.com/[USER]/[REPO]

Do NOT install anything. Read-only analysis.

Your task:

1. Fetch the README. Pull the exact claims: savings percentage, methodology references, any "X% faster / smaller / cheaper" numbers.

2. Fetch the benchmark source if one exists (usually benchmarks/, eval/, or tests/). Tell me:
   - What is the baseline / control group? Quote it verbatim.
   - How many test cases? Are they isolated one-shot prompts or multi-turn agentic sessions with tool calls?
   - What is being measured (input tokens, output tokens, wall clock)?
   - If the baseline is weak (e.g. "You are a helpful assistant"), flag it. That's a rigged benchmark.

3. Fetch the core implementation. Tell me:
```

```
   - Does it run as a binary, shell proxy, prompt injector, or
Claude skill?
   - Does it touch data on disk or over the network?
   - Any telemetry calls? Any subprocess spawning? Any shell
injection risks?
   - Any known CVEs or security flags from the issue tracker?

4. Put the claim in context against a real Claude Code session.
In a typical session:
   - Read, Grep, Glob, Edit, Write bypass Bash entirely and run
directly as Claude's built-in tools (~60% of operations)
   - Tool results (file contents, grep matches, Bash output)
dominate the token budget
   - Claude's own text output to the user is 5-15% of session
tokens
   - Code blocks are usually exempt from any "be terse"
instruction

   Which slice does this tool actually touch? What's the
realistic session-wide impact?

5. Give a verdict:
   - INSTALL — savings are real, scope is meaningful, install
risk is clean
   - EXTRACT — useful idea, you can rebuild the valuable part in
under 200 lines, don't trust the binary
   - SKIP — marketing without substance, or security risk, or
both

6. Generate a Claude Code prompt I can paste to measure the
tool's actual impact against my own session data. The prompt
must be:
   - Read-only (investigation, not fix)
   - Parse ~/.claude/projects/ JSONL session logs
   - Bucket tokens by category (tool_result, tool_use,
assistant_text, other)
   - Apply the tool's claim ONLY to the bucket it actually
touches
   - Output a clean markdown table I can screenshot
```

```
Save findings to OUTPUTS/[tool-name]-second-opinion.md and STOP.
```

Prompt 2: Claude Code session audit

Cowork generates a tailored version of this in Step 3, but here's the template so you know what to expect. It's also useful standalone when you already know which slice a tool touches.

```
PHASE 1 — INVESTIGATION ONLY. Do not change files. Report and STOP.

Context: I'm validating a token-savings claim. The tool claims to [CLAIM — e.g. "reduce output tokens by 75%" or "compress Bash output by 60-90%"]. I want to know what that means against my REAL Claude Code sessions, not against their benchmark.

Your task:

1. Locate my Claude Code session logs at ~/.claude/projects/
2. Pick the most recent 20 session JSONL files.
3. For each session, parse every message and categorize every token into one of four buckets:
   - tool_result — file reads, grep matches, Bash output, anything fed back to the model from tool calls
   - tool_use — JSON blocks for Read, Grep, Glob, Edit, Write, Bash (the function call args)
   - assistant_text — your natural-language text output to the user
   - other — system prompts, user messages, anything else
4. Sum per category across all 20 sessions. Report totals and percentages.
5. Apply the tool's claim to the specific slice it can touch:
   - If the tool compresses BASH OUTPUT: apply savings % to the Bash portion of tool_result only
   - If the tool compresses CLAUDE'S TEXT OUTPUT: apply savings % to assistant_text only
   - If the tool compresses something else: apply to the correct bucket, ignore the rest

Output a clean markdown table I can screenshot:
```

```
| Bucket | Tokens | % of session | Tool touches? |
|---|---|---|---|
| Tool results | X | X% | [yes/no] |
| Tool use blocks | X | X% | [yes/no] |
| Assistant text | X | X% | [yes/no] |
| Other | X | X% | [yes/no] |

Bottom Line:
- Headline claim: X%
- Real session-wide impact: Y%
- Gap: (X - Y)%

Do NOT install anything. Do NOT change files. Read-only. STOP after reporting.
```

How to read the results

Three numbers matter:

1. **Headline claim:** what the repo says on the README
2. **Claimed-slice impact:** the savings within the slice the tool actually touches
3. **Session-wide impact:** the savings as a percentage of your entire session

Most tools report #1 but deliver #3. The gap is where the hype lives.

Honest thresholds for a Claude Code session:

- Any tool that touches only Bash output caps out around 1-4% session-wide
- Any tool that touches only Claude's text output caps out around 2-8% session-wide
- Any tool touching both might reach 5-12% session-wide
- Anything claiming 30%+ session-wide savings is either lying or measuring something else

These numbers assume you run real Claude Code work: reading files, running commands, editing code. If your workflow is pure chat without tool use, the math shifts, but then you probably shouldn't be using Claude Code in the first place.

Red flags checklist

Use this while reading a tool's README. Three or more means SKIP.

- [ ] Benchmark baseline is `"You are a helpful assistant"` or similarly weak
- [ ] Test cases are isolated one-shot prompts, not agentic sessions

- [ ] Measurement excludes tool results, file reads, or system messages
- [ ] Tool ships as a binary with no source you can audit
- [ ] Installation modifies your shell, your PATH, or your git hooks
- [ ] README mentions "telemetry" or "anonymous usage data" with no opt-out
- [ ] Security scanner results are dismissed as "false positives" without explanation
- [ ] Savings percentage is a range wider than 30 points (e.g. "60-90%")
- [ ] The tool intercepts anything between your AI and your terminal
- [ ] The repo is less than 4 weeks old and has no real adoption

What to do after the audit

If the tool passed: install it, measure again, move on.

If the tool failed but the idea was useful: extract the concept. Most of these tools compress to 3-10 lines of logic hidden behind 10,000 lines of scaffolding. You can rebuild the useful part in Python, add it to your Claude Code hooks, and own it end to end.

If the tool failed and the idea was also useless: delete the tab, save the time, thank your AI.

The anchor line

Every time a new token-savings tool lands in your feed:

Don't trust me. Trust your AI.

Run the method. Measure the result. Then decide.

About this playbook

This is the investigation method we use inside **Agent-J+** — a community for people who want to stop guessing and start measuring when it comes to AI tooling. We teach how to orchestrate Claude Code, Cowork, and the full Claude stack to audit claims, build your own tools, and ship fast.

If this method saved you from installing one bad tool, it already paid for itself.

agent-j-plus.com