Token-Saving Tips for Agentic Coding

Best practices to reduce token usage and keep sessions efficient.

General Best Practices (Any Coding Agent)

1. **Write specific prompts.** A vague request triggers broad exploration — the agent reads many files to guess what you mean. Include four things in every prompt (source):
   - **Goal:** What are you trying to build or change?
   - **Context:** Which files, folders, errors, or examples are relevant?
   - **Constraints:** What standards, architecture rules, or conventions apply?
   - **Done when:** What should be true when the task is complete — tests pass, bug no longer reproduces, behavior changes?
2. **Keep sessions focused.** Avoid long sessions that mix unrelated tasks. Context accumulates and performance degrades as the window fills. Start a fresh session when switching to unrelated work, and carry over only the context that matters.
3. **Break work into focused, bounded tasks.** Complete and verify one task before starting the next. Mixing multiple partially-done tasks in one session fills context with conflicting state and makes it harder for the agent to stay on track.
4. **Design your project structure upfront.** A well-organized project means the agent reads fewer files to understand where things belong. Structural ambiguity leads to unnecessary file reads and more iterations.
5. **Limit what goes into context.** Avoid loading your entire codebase at once. A tool like Repomix compresses a codebase into a compact, AI-readable format (strips comments, unused files, etc.) to reduce context size. Point the agent at only the files it needs.
6. **Explore before implementing.** Let the agent read the relevant parts of the codebase and propose a plan before it touches any files. Reviewing and approving the plan first prevents expensive re-work when the initial direction is wrong.
7. **Match model capability to task complexity.** Most tools offer multiple model tiers. Use a higher-capability model for complex, ambiguous, or long-running problems (architecture decisions, deep debugging). Use a lighter model for everyday coding tasks and simple, repetitive work. Choosing the right tier is one of the highest-impact ways to reduce token cost.

Claude Code-Specific Tips

1. **Keep CLAUDE.md short and specific.** The CLAUDE.md file is loaded into context at the start of every session. Keep it under 200 lines. Include only what Claude cannot infer from the code: conventions, build commands, non-obvious gotchas. A bloated file wastes tokens and causes Claude to miss the rules that matter.
2. **Use /compact to manage session length.** When a session grows long, run /compact to summarize the conversation history. Then start a fresh session using that summary as context. Also available in Codex.
3. **Use skills for repeated tasks.** If you repeat the same type of task across sessions (e.g., following a project-specific API convention), create a skill for it. Skills load on demand and avoid bloating every session with instructions that only apply sometimes.
4. **Choose the right Claude model.** Use **Fable or Opus** for complex, long-running, or ambiguous problems: architecture decisions, root-cause debugging, multi-step reasoning. Use **Sonnet** for everyday coding tasks. Use **Haiku** for fast, simple, or repetitive work. Switch mid-session with `/model`.
5. **Monitor costs with /usage.** Run /usage during a session to see a token and cost breakdown by model. Useful for spotting expensive patterns (long context, cache misses, high-cost models) before they add up.
6. **Use plan mode before implementing.** Switch to plan mode (Shift+Tab) before making changes. Claude reads the codebase and proposes an approach for your approval — no files are modified. This is the Claude Code implementation of the general "explore before implementing" practice.
7. **Prefer CLI tools over MCP for common platform operations.** MCP servers are powerful for connecting to external services (Figma, databases, documentation). But for common platform tools — GitHub, AWS, GCloud — CLI tools (`gh`, `aws`, `gcloud`) are often more token-efficient because they don't add per-tool overhead to the context. Use MCP when the integration is non-trivial; use CLI when a command will do the job.

For the Web Interface

- **Use the web interface for Q&A, research, and document drafting** — not for direct codebase work. It has no access to your files.
- **Include examples in your prompts.** Showing the expected output is faster than describing it.
- **Verify before trusting.** Review output critically before using it — this applies equally to the web and CLI.