7 ways to reduce token costs when using Claude Code

A detailed guide on how to optimize tokens when using Claude Code, reducing costs and increasing work efficiency.

Claude Code is an incredibly useful tool, but the cost of using it can add up much faster than many people realize. The reason is quite simple: you're not just paying for the prompt you enter, but also for the entire context of the session. This includes previous messages, files read, output from the tool, memory files like CLAUDE.md, and many other background instructions.

Therefore, when the token starts to surge, the problem is often not a poor prompt, but rather an uncontrolled "inflation" of the context.

Many common pieces of advice like 'keep conversations short' sound right, but they don't pinpoint exactly what actually makes a difference. More importantly, it's about understanding how Claude Code builds context, which parts are sent back repeatedly, and which workflow habits are silently causing waste.

Choose a model that matches the complexity of the task.

Not every task requires the most powerful model. This is one of the simplest yet most overlooked approaches.

In Claude's system, the models have significantly different costs. For example, Opus can be many times more expensive than Sonnet in terms of tokens. Therefore, using the wrong model for simple tasks will lead to unnecessary resource waste.

A sensible approach is to start with Sonnet for everyday tasks like writing tests, minor edits, or explaining code. When encountering more complex problems such as designing multi-file systems or debugging errors involving multiple components, then switch to Opus. For mechanical tasks like renaming, formatting, or quick lookups, Haiku is a more economical choice.

```
/model sonnet # Day-to-day: writing tests, simple edits, # explaining code, refac
```

Additionally, you can adjust the 'effort' level to control the amount of inference the model performs. For simple tasks, reducing effort will save a significant amount of tokens.

Keep the CLAUDE.md file clean and functional.

One of the most effective ways to save tokens is to avoid repeating project rules in every chat. That's where the CLAUDE.md file comes in.

This file is loaded from the start and always exists within the context of the entire session. This also means: if CLAUDE.md is 5,000 tokens long, you will have to 'pay' 5,000 tokens for each interaction, even if you only send a very short message.

Therefore, CLAUDE.md should contain stable information such as how to run tests, the package manager, formatting rules, architectural constraints, or directories that should not be modified. At the same time, this file should be kept as compact as possible.

Content such as meeting notes, design history, or lengthy instructions should not be included here. The most effective approach is to use CLAUDE.md as a quick lookup table, rather than a 'giant brain'.

Use subagent only when absolutely necessary.

Subagents are separate instances of Claude that operate in independent contexts. When a subagent is run, lengthy output such as logs, file searches, or multi-step reasoning will not "contaminate" the main context; only a summary will be returned.

This helps keep the main workflow cleaner. However, subagents aren't always cost-effective.

For small tasks like running simple shell commands or performing quick git operations, using subagents can sometimes incur additional token costs due to overhead from the architecture, prompts, and tool calls.

Therefore, the practical principle is to only use subagents when the context you 'avoid' is greater than its initial cost.

Specify the file and scope to be processed.

One of the quickest ways to waste tokens is to ask Claude to 'view the entire repo' when the actual problem lies in just a few files.

When the request is too vague, Claude will have to open multiple files, try different approaches, and rebuild the context himself—all of which costs tokens.

A better approach is to specify the file and even the relevant line of code. This helps narrow down the processing scope and significantly reduce the amount of wasted tokens.

Additionally, you can use the plan mode (Shift + Tab) to request Claude to provide a plan before execution. This allows you to eliminate unnecessary steps before they consume resources.

Use /compact at the right time.

Claude can automatically or allow you to manually compress sessions. However, the timing of use is the crucial factor.

After Claude has read many files, run commands, and tried a few wrong approaches, the session often contains a lot of unnecessary information. This is where using /compact to 'clean' the context comes in handy.

If you wait until the system reports that the context is full or starts 'forgetting' information, it's too late. At that point, the summary will be less accurate.

Conversely, if you compact early while the session is still "healthy," you'll retain important information and eliminate unnecessary elements, making subsequent steps significantly lighter.

Check the context before optimizing.

A common mistake is blind optimization, without knowing where tokens are being spent.

The /context command is a tool that helps you see this clearly. Often, the most token-consuming part isn't the current prompt, but a large file that was loaded earlier, output from a tool, or a memory-heavy file.

Instead of guessing, check the facts. Simply eliminating a 'silent culprit' in context can lead to significant improvement.

Keep the tool system simple.

Claude Code can connect with many different tools and data sources, but the more integrations, the more context can become overloaded.

If you add too many tools, the model may have to bear unnecessary overhead for each task.

The solution is to keep the system clean. Only use integrations that actually solve recurring problems, instead of turning everything on just because 'it might'.

The most effective way to reduce token costs is not to control each prompt, but to redesign the workflow.

When you control the context—knowing what to keep, what to leave out, and what shouldn't appear in the first place—you'll save more than any little tricks.

In other words, don't just think about the prompt. Think about how the entire context is built and operates.