Context, memory and costs ◆

How the context window really works, how the cache behaves, where memory lives, and how not to burn through tokens and plan limits.

Updated: July 2026 · Source: official Anthropic documentation (code.claude.com/docs) · Vol. 3 of the Claude Code cheatsheet series.

1. The context window and the cache

why some actions cost and others don't

The model remembers nothing between requests: on every message you send, Claude Code re-sends **everything** — system prompt, project context, the entire conversation, the new message. **Prompt caching** is what avoids reprocessing the identical part: the API compares the start of the request (the *prefix*) with what it has processed recently. The match is **exact**: a change anywhere in the prefix recomputes everything that follows. That's why Claude Code orders the request with the rarely-changing parts first:

Layer	Contents	Changes when...
System prompt	Core instructions, tool definitions, output style	An MCP connects/disconnects, or Claude Code gets updated
Project context	CLAUDE.md, auto memory, non-scoped rules	Session start, or after /clear and /compact
Conversation	Messages, responses, tool results	Every turn (it **appends** at the end: the cache holds)

✗ Actions that invalidate the cache (slow, expensive turn)

- **Switching models** (/model): each model has its own cache; the next request re-reads the whole history with no hits. Note: `opusplan` switches models on every plan mode toggle.
- **An MCP connecting/disconnecting** mid-session (even on its own: stdio process exiting, expired HTTP session, automatic reconnection).
- **Denying an entire tool** (bare rule like `Bash`): removes the tool from context. Scoped denies like `Bash(rm *)` don't touch the cache.
- **/compact**: by design replaces the history with a summary (but the summarization request reuses the cache: the real cost is the generation, not the miss).
- **Updating Claude Code**: changes the system prompt. Resuming a long session after an upgrade reprocesses the whole history: it can be the most expensive request you send.

✓ Actions that preserve it

- **Editing repo files**: contents only enter the context when Claude reads them; the edit is flagged with an appended reminder.
- **Editing CLAUDE.md mid-session**: the cache holds... but **the change doesn't apply**! The file is read once at startup; the new content arrives at the next /clear, /compact or restart. Same for the output style.
- **Invoking skills and commands**: they inject instructions as appended messages.
- **Switching permission mode** (Shift+Tab) and **changing effort**: cache-safe.
- **/rewind**: truncates to a prefix **already in cache** — if you've gone down the wrong path, rewinding costs less than compacting.
- **Subagents**: they work in a separate window; yours stays intact.

▶ The practical rule

Pick your model and connect MCP servers **at the start of the session**, then leave them alone. Save `/compact` for natural pauses between tasks (with instructions: `/compact keep the error handling patterns`) instead of waiting for auto-compaction mid-work. Fewer mid-task changes = more cache hits = a faster, cheaper session. Check what's filling the window with `/context`.

2. CLAUDE.md, rules and auto memory

the project's persistent memory

Every session starts from scratch; two mechanisms carry knowledge across sessions, both loaded at startup. Caution: they are **context, not forcibly applied configuration** — to block an action regardless of what Claude decides, you need a PreToolUse hook or a permission rule.

	CLAUDE.md	Auto memory
Who writes it	You	Claude (learns from corrections and preferences)
Contents	Instructions and rules	Learnings and discovered patterns

Scope	Project, user or organization	Per repository, shared across worktrees
Loading	Every session, in full	Every session (first 200 lines or 25KB)
Use for	Conventions, build commands, architecture	Debug insights, discovered commands, preferences

◆ Where CLAUDE.md files live (in loading order)

Scope	Location	What for
Managed (org)	`/etc/claude-code/CLAUDE.md (Linux)`	Company standards via MDM/Ansible; individuals can't opt out
User	`~/.claude/CLAUDE.md`	Personal preferences across all projects
Project	`./CLAUDE.md or ./.claude/CLAUDE.md`	Shared with the team via git
Local	`./CLAUDE.local.md`	Personal per-project preferences; add it to .gitignore

Claude Code **walks up the directory tree** from the working directory collecting every CLAUDE.md; contents are **concatenated** (root downward, so the ones closest to you are read last). Those in subdirectories load on demand when Claude touches files there. In a monorepo, `claudeMdExcludes` skips other teams' files. HTML comments `<!-- -->` are stripped before injection: maintainer notes at zero token cost.

Writing effective instructions

- **Under 200 lines**: longer files consume context and reduce adherence.
- **Specific and verifiable**: "use 2-space indentation", not "format nicely"; "run npm test before committing", not "test your changes".
- **Consistent**: two conflicting rules = Claude picks one at random. Review periodically.
- **Imports** with `@path/to/file` (up to 4 levels deep; inside backticks it stays literal text). `@AGENTS.md` at the top reuses instructions from other agent tools.
- Add a line when Claude gets the same thing wrong **twice** — multi-step procedures belong in a skill instead.

.claude/rules/ — modular instructions

One markdown file per topic (`testing.md`, `security.md`...), discovered recursively, even via symlinks shared across projects. The superpower is the `paths` frontmatter:

```
---
paths:
  - "src/api/**/*.ts"
---
# API rules
- Every endpoint validates input
- Standard error format
```

Rules with `paths` load **only when Claude works on matching files**: clean context and tokens saved. User rules in `~/.claude/rules/` apply everywhere (lower priority than project rules).

3. Cutting token usage

the levers, most effective first

Strategy	How
Clean up between tasks	`/clear` when switching to unrelated work: stale context costs on **every** subsequent message. Run `/rename` first, so you can find the session again with /resume.
Right model for the task	Sonnet handles most coding well and costs less than Opus; save Opus for architecture and multi-step reasoning. For simple subagents: `model: haiku`.
Effort and thinking	Extended thinking is billed as output and the default budget can be worth tens of thousands of tokens per request: for simple tasks lower it with `/effort` or turn thinking off in /config.
Subagents for the noise	Tests, logs, documentation fetches: delegate them to a subagent; only the summary comes back to main.
CLI > MCP where possible	`gh`, `aws`, `gcloud`, `sentry-cli` add no tool definitions to the context. Disable unused servers from /mcp; MCP definitions are deferred by default anyway (tool search).
Preprocessing hooks	Instead of having Claude read 10,000 lines of logs, a hook filters with grep and returns only the ERROR lines: from tens of thousands of tokens down to hundreds.
From CLAUDE.md to skills	Workflow-specific instructions (PR reviews, DB migrations) loaded every session are wasted tokens whenever you're doing something else: move them into on-demand skills.
Specific prompts	

	"Improve this codebase" triggers broad scans; "add input validation to the login function in auth.ts" works with minimal reads.
Plan mode and early corrections	Shift+Tab to plan before implementing; if Claude heads the wrong way, **Esc immediately** and /rewind: rework is the most avoidable expense.

What it costs, in practice

On enterprise API deployments the average is ~$13 per developer per active day ($150-250/month), with 90% of users under $30/day. On subscription (Pro/Max) the dollar figure in /usage is informational only: what matters are the plan-limit bars, with breakdowns per skill, subagents, plugins and MCP servers (keys `d`/`w` for 24h/7d). There's also a small background spend (<$0.04/session) for summaries and status commands.

Watch out for agent teams

Each teammate is a full Claude instance with its own window: in plan mode consumption reaches ~7× a normal session. If you use them: Sonnet for teammates, small teams, lean spawn prompts, and shut teammates down when they're done.

Sources: code.claude.com/docs — prompt-caching, memory, costs, context-window · /context and /usage are your everyday diagnostic tools.