Claude Code

The Complete Guide

From first steps to expert knowledge

Author: Christian Drapatz

Status: May 2026 · Claude Code v2.1+

Platform: macOS · Linux · Windows

Version 1.1

Disclaimer

The contents of this website are based on personal experience as well as publicly accessible sources such as documentation, training, videos, and community contributions.

All content has been independently compiled, summarized, and formulated in own words. There is no direct reproduction of protected content.

The provided tutorials are free and serve exclusively for knowledge transfer. Despite careful preparation, no guarantee can be given for completeness or timeliness.

All mentioned brands, products, and technologies belong to their respective owners.

Table of Contents

1 Introduction – What is Claude Code?

Claude Code is Anthropic's terminal-based AI agent for software development. The decisive difference from a chat assistant: Claude Code works directly in the project. It reads the codebase, executes commands, writes and edits files – all from the terminal.

Three properties make Claude Code special:

- Context: On startup, it automatically loads CLAUDE.md – a briefing file with project context, conventions, and rules.
- Extensibility: Reusable workflows can be saved as Skills and invoked via slash command. External services like GitHub, databases, or browsers connect via MCP Servers.
- Autonomy: With Auto Mode and Worktrees, Claude Code can independently build, test, and commit complete features.

This guide leads from first start to fully configured setup. The learning path in Chapter 21 shows how to proceed step by step without overextending yourself.

2 Quick Start in 10 Minutes

Five steps to get started immediately:

1. Install: Open terminal and start the installer.

```
curl -fsSL https://claude.ai/install.sh | bash
```

Then reopen the terminal.

2. Start: Change to the project directory and type claude. On first startup, browser login occurs.
3. Privacy: Immediately after login, set Help Improve Claude to Off under claude.ai → Settings → Privacy.
4. Initialize: Type /init – Claude analyzes the project and automatically creates CLAUDE.md.
5. Get started: formulate task, Shift+Tab for Plan Mode, review plan, then have it implemented.

> **Note: You don't need to manually configure anything. Simply ask Claude Code what you need – it creates the correct configuration. CLAUDE.md ensures everything is loaded on the next startup.**

3 Requirements, Plans, and Models

3.1 Pricing Plans Overview

To use Claude Code, a paid Anthropic account is required. The normal free Claude.ai access is not sufficient for this. Authentication can be done either via an existing subscription login or via an API key. Which option makes sense depends on whether Claude Code is to be used mainly privately, in a team, or automatically in development processes.

Plan	Price	Model (Default)	When useful

Pro	approx. $20 USD/month	Sonnet 4.6	Solo, occasional coding
Max	approx. $100–200 USD/month	Opus 4.7	Solo, several hours daily; complex refactorings
Team Standard	approx. $25 USD/seat (from 5)	Sonnet 4.6	Teams; contractually no training on your data
Team Premium	approx. $125 USD/seat	Opus-equivalent	Teams with Max-like requirements
Enterprise	Custom Pricing	Custom	SSO, Audit Logs, Zero-Data-Retention
API directly	Pay-per-Use	Freely selectable	Sensitive codebases; 7 days retention, no training

> **Rule of thumb: Start with Pro. If limits become noticeable after two weeks, switch to Max. For sensitive code, choose Team or API directly.**

3.2 Models and Effort Levels

Claude Code can work with different models and effort levels depending on the task. This makes sense because not every task requires the same depth. A fast model is sufficient for simple searches or minor adjustments. For architectural decisions, larger refactorings, or security-critical changes, a stronger model with more thinking time is worthwhile.

The following overview shows a practical classification of when which model makes sense.

Model	Strength	When to use
Opus 4.7	Highest quality, slower	Architecture, complex refactorings, security reviews
Sonnet 4.6	Good balance quality/speed	Daily coding, building features
Haiku 4.5	Fast and cheap	Exploring, simple searches, routine checks

Switch models with /model opus, /model sonnet, or /model haiku.

Effort levels control how much thinking time Claude Code should use for a task. Low effort is sufficient for simple, clearly defined tasks. The more complex the change becomes, the higher the effort should be chosen so that Claude can check more connections and consider possible side effects.

Level	Typical tasks
low	Rename file, build command, simple greps
medium	Write functions, small refactors
high	Multi-file refactors, complex debugging
xhigh	Default with Opus 4.7 – architecture, design decisions
max	Hardest debugging, security reviews, edge cases (current session only)

Opus 4.7 supports effort levels with /effort:

> **Tip: max automatically jumps back to xhigh after the session – this protects against accidental token burn.**

3.3 Usage Limits

Claude uses different usage limits depending on the plan (Free, Pro, Max 5x/20x). Free is heavily limited, Pro offers significantly more usage, and Max is intended for intensive use with Claude Code and large projects.

Additionally, there is a rolling 5-hour window: it starts with the first message and automatically resets after 5 hours. The limit applies jointly to Claude Web/Desktop and Claude Code.

Since 2025, there is also a weekly limit for very intensive usage. Especially Opus consumes significantly more quota than Sonnet.

Usage can be tracked in various ways.

- **/status in terminal – shows plan, model status, and remaining usage**
- **claude.ai/settings/usage in browser – complete breakdown of all quotas**

Category	Description
Current session	Current session consumption across all models in the rolling 5-hour window. The window starts with the first message and automatically resets after 5 hours. Particularly relevant for Max plans 5x/20x with significantly higher usage.
All models	Weekly total limit across Opus, Sonnet, and Haiku combined.
Sonnet only	Sonnet usage counts simultaneously to this limit as well as to the weekly and 5-hour session limit.
Claude Design	In research preview with its own weekly limit. Does not count toward other limits.
Additional usage	Can be enabled to continue using Claude after reaching a limit. Enables API mode for paid additional usage.

> **Note: If a limit is reached, Claude reports this with an error message. In this case, you must wait until the limit automatically resets. Alternatively, further usage can be done via the API.**

4 Installation

There are two ways: the terminal CLI and the desktop app. Both share login and configuration.

Terminal Installation (recommended)

No more Node.js required – runs directly as binary and updates automatically:

```
# macOS / Linux
curl -fsSL https://claude.ai/install.sh | bash
```

```
# Windows (PowerShell)
irm https://claude.ai/install.ps1 | iex

# macOS via Homebrew (manual update)
brew install claude-code
```

After installation:

```
claude --version    # check version
claude doctor       # diagnosis
claude              # start and browser login
```

Desktop App

Download at claude.com/download. The app offers parallel sessions in isolated Git worktrees and a tab for local and cloud sessions.

Updates

```
claude update           # immediate update
brew upgrade claude-code  # via Homebrew
```

5 Privacy and Security

These settings belong before the first real session.

5.1 Mandatory Setting Before First Session

Immediately after login:

> **Important: claude.ai → Settings → Privacy → Help Improve Claude → Off**

Without this setting, sessions have five years retention instead of 30 days and flow into model training. Applies to all consumer plans (Pro and Max). Team and Enterprise plans have contractually shorter retention.

5.2 What Claude Transmits

Every file that Claude reads goes completely to Anthropic's servers – it does not automatically scan the project, but every explicitly read file is transmitted. Particularly sensitive:

- .env files with API keys and passwords
- Certificates (.p12, .mobileprovision) and SSH keys
- Medical, legal, or otherwise protected data

Exclude Files with .claudeignore

Like .gitignore, .claudeignore prevents automatic reading of certain files:

```
# .claudeignore
/Config/Secrets/
*.env
```

```
**/TestData/real_patients/**
/Resources/Credentials/
```

> **Warning: If Claude is explicitly instructed to read a .claudeignore file, it will still be transmitted. The file only protects against automatic access.**

Behavioral Rules in CLAUDE.md

For regulated projects, a privacy block in CLAUDE.md is recommended:

```
# Privacy
NEVER read:
- /TestData/ (patient data)
- Files with "patient", "versicherte", or "KV-Nummer" in the name
- /Config/Live/*.json (production configuration)
```

Use in Regulated Environments

Claude Code processes requests on US servers (AWS). For GDPR and SGB-V-relevant health data:

Approach	Suitability	Effort
Transfer code only, never real data	Readily feasible	Low
Use anonymized / fictional test data	Recommended	Medium
.claudeignore + CLAUDE.md privacy block	Mandatory in the project	Low
Close Anthropic Enterprise plan with AVV	Legally compliant	High
Local model (e.g. Ollama) as alternative	Fully offline	High

> **Golden Rule: Claude Code is a tool for code – not for data. Those who keep this separation clean can use Claude Code productively even in regulated environments.**

5.3 How Claude Code Protects You

- Read-only by default: Asks for permission before every edit and every Bash command.
- Write boundary: Only start directory and subdirectories – always start in the project root.
- Command blocklist: curl and wget are blocked by default.
- Prompt injection protection: Web-fetch content is handled so that injected instructions do not automatically lead to tool calls.

> **Important: Permissions protect against actions, not data leakage. If Claude is allowed to read a file, its content goes to Anthropic. Block secrets files explicitly with a deny rule.**

6 Getting Started and Daily Commands

6.1 Starting in the Project

Open terminal, navigate to the project folder, and start Claude Code

```
cd my-project
claude
```

On first startup: choose theme, browser login, claude doctor for verification. After that enter /init – this generates a CLAUDE.md with project context, build commands, and conventions. This file is automatically loaded in each subsequent session. On first startup: choose theme, perform browser login and verify installation with claude doctor. After that execute /init – this generates a CLAUDE.md with project context, build commands, and conventions.

> **Important: The CLAUDE.md should be reviewed and adjusted immediately afterward. It is automatically loaded in every subsequent session and significantly influences the behavior and responses of Claude Code.**

6.2 The most important slash commands

Claude Code has a set of slash commands for controlling and supporting the workflow. Simply enter a / in the prompt. Claude Code then automatically displays all available commands and matching suggestions.

Command	Function
/init	Automatically generates a CLAUDE.md for the project
/clear	Deletes the current conversation – fresh start, saves tokens
/compact	Compresses the context when the window fills up
/usage	Dashboard: plan limits, daily costs, streaks
/status	Display plan, active model and usage limits
/cost	Quick display of current session costs
/review	Code review of recent changes
/model	Model switching, e.g. /model opus
/effort	Effort level: low / medium / high / xhigh / max
/mcp	Manage MCP connections and check status
/memory	Display and edit loaded CLAUDE.md
/ultrareview	Multi-level in-depth code review
/loop	Self-paced loops for repeated tasks
/permissions	Get allowlist suggestions from previous tool calls

Session Management

With session and context commands, Claude Code can continue existing sessions as well as selectively load files and folders into the current work context. This allows Claude Code to better understand the project context and provide more precise answers and more appropriate changes.

```
claude -c            # Continue last session
claude -r "name"     # Continue session by name
@datei.swift         # Load file into context
```

```
@Sources/Views/      # Load folder into context
```

6.3 Modes

Claude Code supports various work modes that provide more control or more automation depending on the task. You can switch between modes at any time with Shift + Tab.

- Normal: Asks before every action.
- Auto-Accept: Actions without confirmation unless blocked by Deny. For long sessions in a pushed branch.
- Plan Mode: Read-only – Claude creates only a plan. Ideal for the Analyze-Plan-Build workflow.
- Auto Mode (Max-Plan): Claude makes decisions completely autonomously. Not for production deploys.

7 CLAUDE.md – The project memory

CLAUDE.md is the most important file in the setup. It is automatically loaded in every session and provides Claude with the context that it would otherwise have to guess anew each time.

7.1 What belongs in it

Answer four questions, nothing more:

- What is the project? (Stack, platform targets, language)
- How is it built and tested? (Build and test commands, tooling)
- What conventions apply? (Code style, folder structure, architecture rules)
- What should Claude not do? (Prohibitions, taboo areas)

Does not belong in it: Tutorials, long explanations, marketing, documentation texts. The file is a briefing card, not a wiki. Maximum 200 lines.

7.2 Where the file is located

Contents are merged, not overwritten:

- ~/.claude/CLAUDE.md – global, applies to all projects
- CLAUDE.md in the project root – applies only to this project (commit to repo!)
- CLAUDE.local.md in the project root – personal, not in Git
- Subfolder CLAUDE.md – is only loaded when working there

7.3 Example and best practices

Global CLAUDE.md (~/.claude/CLAUDE.md)

```
# General
Answer in German. Keep it brief.
For larger changes, suggest a plan first.

# Prohibitions
```

```
Never read contents from .env or secrets/
No git push --force without explicit confirmation.
```

Project CLAUDE.md (Example Swift / SwiftUI)

```
# Project
SwiftUI App, Targets: macOS, iPadOS, iPhone

# Build
xcodebuild -scheme AppName \
-destination "platform=iOS Simulator,name=iPhone 16"

# Navigation
- iPhone: NavigationStack
- macOS/iPadOS: NavigationSplitView

# Prohibitions
No DispatchQueue – only async/await.
```

> **Tip: Create a global file once. In the project, execute /init, immediately trim the generated file to the essentials, commit it to the repo. In the first sessions, supplement what Claude obviously doesn't know.**

8 The Nine Building Blocks at a Glance

For practical work with Claude Code, nine central building blocks can be distinguished. They interlock and determine how Claude Code understands context, executes tasks, gets extended, and which rules apply in the process.

Building block	When active	Where defined	What for
CLAUDE.md	Always	Project root	Context, conventions, rules
Permissions	Always	settings.json	Allow and forbid
Hooks	On events	settings.json	Mechanics, automation
Skills	On call	skills/	Reusable tasks
Subagents	On call	agents/	Self-contained context, brief answer
MCP-Server	On call	settings.json	External tools
Sandbox	On activation	settings.json	Limit scope
Plugins	On activation	plugins/	Bundle of multiple components
Loops/Tasks	Periodically	settings.json	Routine, observation

9 Directory Structure and settings.json

All building blocks follow the same logic: global in the home directory, project-specific in the project, local-personal in a file that doesn't go into Git.

```
~/.claude/                          # Global Configuration
├── CLAUDE.md                       # Global Preferences
├── settings.json                   # Global Permissions, Hooks, MCP
├── skills/<name>/SKILL.md          # Global Skills
└── agents/<name>/AGENT.md          # Global Subagents

.claude/                            # Project-specific (in Git)
├── settings.json                   # Project Permissions, Hooks, MCP
├── skills/<name>/SKILL.md          # Project Skills
└── agents/<name>/AGENT.md          # Project Subagents

CLAUDE.local.md                     # Personal, not in Git
.claude/settings.local.json         # Local, not in Git
```

Structure of settings.json

The settings.json controls central settings of Claude Code, for example allowed and blocked actions, hooks, MCP servers, and the model used. Permissions are particularly important because they determine what Claude Code is allowed to execute in the project and what is explicitly blocked.

```
{
"permissions": {
"allow": ["Read", "Edit", "Bash(git diff*)"],
"deny":  ["Bash(rm -rf *)", "Read(.env*)"]
},
"hooks": { "PostToolUse": [...] },
"mcpServers": { "github": {...} },
"model": "claude-opus-4-7"
}
```

The three levels (global, project, local) are merged together. Deny always wins.

> **Important: Real JSON files must not contain comments and no trailing commas. Add .claude/settings.local.json and .claude/cache/ to .gitignore.**

10 Permissions

Permissions determine which actions Claude Code may execute automatically and which actions require confirmation beforehand. This allows precise control over how much authority Claude Code has over files, commands, and project areas.

10.1 The three levels

Every action falls into exactly one of the following categories. The combination of these rules determines how Claude Code works in the project.

Level	Behavior
Allow	Can be executed without asking
Ask	Default for everything not explicitly allowed or denied — Claude asks
Deny	May not be executed — absolute block

10.2 Formulating rules

Permissions are defined via rules. This allows specification of which files may be read or modified and which Bash commands are allowed or blocked.

```
Read                    # Read access in general
Edit                    # File edits in general
Read(.env*)             # Reading .env files
Bash(git status*)       # only git status
Bash(git *)             # all git commands
Bash(xcodebuild *)      # all xcodebuild calls
```

10.3 Recommended configuration

Global ~/.claude/settings.json – hard prohibitions

```
{
"permissions": {
"deny": [
"Bash(rm -rf *)",  "Bash(sudo *)",
"Bash(curl *)",    "Bash(wget *)",
"Read(.env*)",     "Read(**/secrets/**)",
"Read(**/.ssh/**)","Read(**/Credentials/**)"
]
}
}
```

Project-specific .claude/settings.json – allow routine

```
{
"permissions": {
"allow": [
"Read", "Edit",
"Bash(git status*)", "Bash(git diff*)",
"Bash(git log*)",   "Bash(git add *)",
"Bash(git commit *)","Bash(xcodebuild *)",
"Bash(swift test*)"
]
```

```
}
}
```

> **Tip: /permissions allowlist suggests a sensible allow list from the recent tool calls. Simply work through two or three sessions and then execute the command.**

11 Hooks

Hooks enable automatic execution of shell commands when certain events occur within Claude Code. This allows recurring tasks such as formatting, checks, or notifications to be automated. Hooks are intended for mechanics – not for complex logic or complete workflows.

Lifecycle events

Lifecycle events determine when a hook is triggered. Depending on the event, a command can be executed before or after an action or at the end of a session.

- PostToolUse – after every tool call; classic for formatters and linters
- PreToolUse – before every tool call; for additional checks
- Stop – at session end; for desktop notifications

Important Variables

```
$CLAUDE_FILE_PATH     # the edited file
$CLAUDE_TOOL_NAME     # which tool ran
$CLAUDE_PROJECT_DIR   # project root
```

Example – SwiftFormat after every edit

This example shows a simple PostToolUse hook for Swift projects. After every edit or write operation, Claude Code checks whether a Swift file has been changed. If so, swiftformat is automatically executed for exactly that file.

```
{
"hooks": {
"PostToolUse": [
{
"matcher": "Edit|Write",
"hooks": [
{
"type": "command",
"command": "if [[ \"$CLAUDE_FILE_PATH\" == *.swift ]]; then swiftformat
\"$CLAUDE_FILE_PATH\"; fi"
}
]
}
]
}
}
```

Example – Desktop notification at session end

This example shows a simple Stop hook. As soon as a Claude Code session ends, a macOS desktop notification automatically appears. This is useful for longer tasks or automated workflows.

```
{
"hooks": {
"Stop": [
{
"hooks": [{ "type": "command",
"command": "osascript -e 'display notification \"Claude finished\" with
title \"Claude Code\"'" }]
}
]
}
}
```

> **Note: Hooks must be fast (under one second), idempotent, and fail silently. Start with a single hook – the formatter.**

12 Skills

Skills are reusable templates and instructions for typical tasks in everyday development. This makes it possible to standardize recurring processes and execute them much faster. A skill has a name, a description, and concrete steps or rules that Claude Code should follow during execution.

12.1 Structure of a Skill

The following example shows a simple Skill for automatically creating a new SwiftUI-View according to fixed project conventions.

```
---
name: new-view
description: Creates a new SwiftUI-View according to our conventions
---

Create a new SwiftUI-View. Steps:

1. Create file under Sources/Views/<Feature>/<Name>View.swift
2. Include @Observable ViewModel
3. Add Preview-Provider
4. Add to Navigation
```

Call: /new-view ProfilEditor. Arguments arrive as $ARGUMENTS or $1, $2 in the Skill.

12.2 Recommended Community-Skills

In addition to your own Skills, there are now many Community-Skills for typical development tasks. They can serve as a foundation, save time, and often show good practical examples for reusable workflows.

Skill	Function	Installation
Superpowers	Structured workflow: first plan, then test, finally code	/plugin install superpowers@claude-plugins-official
Frontend Design	Professional, modern UIs instead of generic AI look	/plugin install frontend-design@claude-plugins-official
Skill Creator	Creates new skills based on description	/plugin install skill-creator@claude-plugins-official
Trail of Bits	Security audit (SQL Injection, XSS, ...)	/plugin marketplace add trailofbits/skills
Prompt Master	Optimizes prompts for 30+ AI tools	git clone .../nidhinjs/prompt-master ~/.claude/skills/
Marketing Skills	40 marketing experts: SEO, Ads, Copywriting, Growth	npx skills add coreyhaines31/marketingskills
CLAUDE.md Optimizer	Keeps CLAUDE.md lean, offloads non-essential content	git clone .../wrsmith108/claude-md-optimizer ~/.claude/skills/
Webapp Testing	Browser-based automated testing	/plugin install webapp-testing@claude-plugins-official

> **Tip: First discover where you repeat yourself. Skills that are theoretically invented usually miss the mark.**

13 Subagents

Subagents are independent Claude instances for clearly separated task areas. They work with their own context and thereby relieve the main session, especially for larger analyses or specialized tasks. The major advantage: the main session remains leaner and more focused, while the subagent returns only the finished result or a summary.

> **Tip: Skill vs. Subagent: A Skill works in the main chat and enlarges the context there. A Subagent works alongside and returns only the result. If a task consumes much context but returns little – use Subagent.**

Structure – Code-Reviewer Example

This example shows a typical Subagent for automated Code-Reviews. The Agent analyzes exclusively changed files and returns a compact summary of critical issues. This keeps the main session clear and larger Reviews can be strategically outsourced.

```
---
name: code-reviewer
description: Senior Code-Reviewer. Call proactively after code changes.
tools: Read, Grep, Glob, Bash(git diff*)
model: sonnet
```

```
---

You are a Senior Code-Reviewer for a Swift project.
Analyze only the changed files.
Return: max. 5 critical points, sorted by priority.
No long explanations.
```

Three fields that matter

- description – determines whether Claude calls the Agent automatically.
- tools – limits what the Agent is allowed to do. Keep it tight.
- model – Research/Reviews: sonnet; Architecture analysis: opus; simple searches: haiku.

14 MCP-Server

MCP-Server (Model Context Protocol) extend Claude Code with external tools and services. This allows Claude not only to analyze and write, but also to work directly with APIs, databases, GitHub, or other systems. With MCP, Claude Code comes much closer to a real development assistant that can actively execute tasks

14.1 Basics and Setup

MCP-Server are registered via the command line and then automatically integrated into Claude Code. After that, they can be used directly from the session.

```
claude mcp add github -- npx -y @modelcontextprotocol/server-github
claude mcp add --transport http my-api https://meine-api.de/mcp
claude mcp list
/mcp                    # Check status of all servers
```

The following MCP servers are currently among the most popular and practical extensions for Claude Code

MCP Server	Function
GitHub	Read issues, prepare PRs, manage branches
Supabase	Query databases, manage tables, control RLS policies directly from Claude
Context7	Load current library docs – no outdated model knowledge
Exa / Perplexity	Web search from Claude
Playwright	Browser automation, screenshots, E2E tests
Pipedream	8,000+ apps in one MCP: Gmail, Slack, Stripe, Notion, …

14.2 The Senior Engineer Stack – 5 free MCPs

The following MCPs are particularly well-suited for developers who want to integrate Claude Code more strongly into their daily workflow. They extend Claude with code reviews, long-term memory, web search, and external services

MCP	Function	Prerequisite
CodeRabbit	Senior engineer review for security, performance, maintainability	GitHub Personal Access Token
Pipedream	8,000+ apps in one MCP	Pipedream account (free)
Exa	Smart Search (1,000 requests/month free)	Exa API Key
Sequential Thinking	Structured thinking – Anthropic's official reference server	None
Memory MCP	Long-term memory across projects	None

> **Security: Anthropic does not audit third-party MCP servers. Use only trusted sources. Never put API keys in committed files.**

15 Plugins

Plugins extend Claude Code with additional functions and ready-made workflows. They often combine multiple components such as skills, hooks, subagents, or settings into a directly usable package. This makes it possible to set up and reuse complex setups much more quickly

Management

Plugins can be installed, listed, updated, or removed directly via Claude Code. After changes, a reload is recommended so that new functions are immediately available

```
/plugins
claude plugin add <name>
claude plugin list
claude plugin remove <name>
/reload-plugins
```

Recommended Plugins

The following plugins are currently among the most popular extensions for Claude Code. They supplement functions such as long-term memory, token optimization, design workflows, or comprehensive power-user functions

Plugin	Function	Installation
Context Mode	Runs in the background, saves tokens (10,645 stars)	/plugin marketplace add mksglu/context-mode
claude-mem	Persistent memory across sessions (68,482 stars)	npx claude-mem install
Everything Claude Code	Comprehensive power-user bundle (168,287 stars)	/plugin marketplace add affaan-m/everything-claude-code

Huashu Design	Prototypes and slide decks via natural language	npx skills add alchaincyf/huashu-design

> **Important: Work without plugins for a while first – if you don't know the building blocks, you can't judge plugins.**

16 Loops and scheduled tasks

With loops and scheduled tasks, Claude Code can execute tasks automatically at specific intervals or at fixed times. This makes it possible to automate recurring checks, status queries, and routine tasks. This is especially practical for longer development, build, or review processes

Use cases

Loops are particularly suitable for repeated checks and monitoring. Scheduled tasks, on the other hand, are ideal for daily or regularly recurring tasks.

- Loops: monitor build status, check CI build every 3 minutes, test results
- Scheduled tasks: daily release notes, meeting preparation, maintenance tasks

In practice

The following examples show typical loop and scheduling commands for everyday development.

```
/loop 5m check if the staging build is done and report back
/loop 10m /security-review
/schedule "0 8 * * 1-5" /daily-standup-prep
/loop status
/loop stop
```

In settings.json

Scheduled tasks can be permanently stored in settings.json. Claude Code then executes these tasks automatically according to the defined schedule

```
{
"scheduled tasks": [
{
"name": "daily-standup",
"schedule": "0 8 * * 1-5",
"command": "/daily-standup-prep",
"notify": true
}
]
}
```

> **Note: Define a clear end condition, realistic intervals (3–10 minutes). Never use loops for destructive actions.**

17 Token optimization and cost control

Especially for longer sessions and larger projects, it pays off to pay attention to token consumption and model selection. With the right strategy, costs can be reduced while improving response speed and context quality. Particularly important are choosing the right model, regularly cleaning up the context, and targeted use of subagents.

17.1 Model selection and /clear

Not every task requires the most powerful model. For many everyday tasks, Sonnet is perfectly sufficient, while Opus should be used strategically for more complex analyses or architectural questions.

- Model selection: Opus costs approximately 5 times as much as Sonnet per token. Use Opus strategically for architecture and refactoring, Sonnet for everyday tasks.
- /clear: Every token in the context costs again with each request. Long sessions without /clear become disproportionately expensive.
- Subagents: Own context, concise answer to the main chat. Main context stays small.

17.2 claude-mem – Persistent memory

claude-mem extends Claude Code with local long-term memory. Important information from previous sessions is saved in summary form and automatically reloaded when needed. This keeps projects more consistent and reduces recurring explanations or analyses.

Step	What happens
Capture	Every tool call is automatically intercepted
Compress	Reduction from up to 10,000 to approx. 500 tokens (5–10× in everyday use)
Store	Local SQLite database
Retrieve	Claude loads only the relevant context for the current task

```
npx claude-mem install

# Alternatively
/plugin marketplace add thedotmack/claude-mem
/plugin install claude-mem
```

> **Security notice: The HTTP API on port 37777 has no authentication. Configure the firewall so that port 37777 is only accessible from localhost.**

17.3 RTK – Token compression

RTK (Rust Token Killer) is a CLI proxy that compresses command output before it reaches Claude. Result: 60–90% fewer tokens.

Command	Tokens before	Tokens after	Savings
git status	~800	~50	~94 %
git push	~200	~10	~95 %
ls (large)	~1.500	~150	~90 %
npm test	~3.000	~300	~90 %

The following commands install and activate RTK for Claude Code. RTK helps optimize context automatically and significantly reduce token consumption in longer sessions. Additionally, status and potential savings can be displayed at any time.

```
curl -fsSL https://raw.githubusercontent.com/rtk-
ai/rtk/master/install.sh | sh
rtk init -g         # Activate hooks for Claude Code
rtk init --show     # Check status
rtk gain            # Display savings
```

18 Cloud automations with Routines

Routines enable fully automated workflows directly in the cloud. This allows tasks to be executed on a schedule or event-based without requiring a local machine or server to run continuously.

Plan	Routine runs per day
Pro	5
Max	15
Team/Enterprise	25

Set up a routine

1. Open claude.ai/code/routines
2. New routine → Assign name
3. Write prompt (precise like an assignment to an employee)
4. Select GitHub repository
5. Select trigger: Schedule, API, or GitHub Event
6. Connect connectors (Gmail, Slack, etc.)
7. Run now to test – activate the schedule only after that

Example – Daily email summary

```
Go to my emails via the Gmail Connector.
Read all unread emails since yesterday.
Summarize each email in one sentence.
Sort by: Urgent, Action needed, Info.
Send me the summary as a Slack message.
```

Example – PR review on every pull request

```
A new pull request has been opened.
Check the code for security issues, performance issues,
Code style and readability.
Write a review as a comment on the PR.
```

> **Important: Routines run completely autonomously. All actions happen under their own name. Place API keys in the environment variables of the cloud environment.**

19 Advanced Workflows

Advanced workflows allow Claude Code to integrate significantly more efficiently into larger projects and parallel development processes. Worktrees particularly help when working on multiple features or longer tasks by managing cleanly separated work areas in parallel.

19.1 Parallel Work with Worktrees

Worktrees enable multiple parallel work environments within a Git repository. This allows different features, bug fixes, or experiments to be worked on simultaneously without constantly switching branches.

```
claude --worktree my-feature   # Start worktree
claude -w                        # Fast worktree start
```

The desktop app offers a Code tab with a worktree checkbox – most convenient for 3–5 parallel sessions.

> **Killer combination: On the Max plan: Opus 4.7 + /effort max + Auto Mode + 1M Context = start a complete project and return after hours to a result that has been independently refactored, tested, and committed.**

19.2 The Verification Loop

The most important single tip from Boris Cherny (Claude Code founder): "If Claude has that feedback loop, it will 2–3x the quality of the output."

```
We are now building feature X together.
Before you write a single line of code, first define
how you yourself check whether the finished feature works:

1. Which test, command, or browser check proves that it works?
2. What is your plan?
3. Implement only after approval.
```

19.3 Sandbox for Autonomous Work

The sandbox deliberately limits which files, folders, and network functions Claude Code can access. This allows Claude to work more autonomously without obtaining unlimited rights to the entire system. Especially with automated workflows or longer auto-mode sessions, the sandbox significantly increases security and control.

```
/sandbox

# Or in settings.json:
{
"sandbox": {
"enabled": true,
"filesystem": {
"writable": ["$CLAUDE PROJECT DIR/Sources",
"$CLAUDE_PROJECT_DIR/Tests"],
"readable": ["$CLAUDE_PROJECT_DIR"]
},
"network": { "enabled": false }
}
}
```

> **Tip: Sandbox + Auto-Accept: Claude works quickly but technically cannot break out of the defined area.**

19.4 Plan Mode First

Especially with larger changes, refactorings, or architectural decisions, you should not work immediately in normal write mode. Instead, it is recommended to switch to plan mode first (Shift + Tab → plan).

In plan mode, Claude Code works in read-only mode. No files are changed and no commands are executed. Instead, Claude analyzes the task and creates a structured implementation plan.

This has several advantages:

- Risks and potential problems become visible early
- The planned architecture can be reviewed
- Misunderstandings surface faster
- Large changes become more controllable
- Unnecessary or incorrect code changes are prevented

Especially for:

- Refactorings
- larger SwiftUI / architecture changes
- database changes
- build / CI adjustments
- multi-file changes
- production code

this workflow is very useful.

A typical workflow:

1. Have the task analyzed

2. Review the plan and correct if needed

3. Clarify open questions

4. Only then switch to normal or auto mode

This way Claude Code works much more controlled and the quality of changes increases noticeably."

20 Claude in other tools

20.1 Claude for Word

Anthropic has integrated Claude directly into Microsoft Word. Changes appear as suggestions with change tracking.

Prerequisites

- Claude Team or Enterprise (Pro on waitlist)
- Microsoft 365 (not Word 2016/2019)
- File format .docx

Installation

6. marketplace.microsoft.com – search for Claude by Anthropic for Word
7. Install, open Word, activate the add-in (Mac: Tools → Add-ins)
8. Log in with Claude account

Important features

- Edit texts with change tracking (formatting is preserved)
- Scan document for inconsistencies (cross-references, numbering, terms)
- Questions with clickable sources
- Fill templates with retained formatting
- Cross-app: Word + Excel + PowerPoint in one conversation

> **Tip: "Shorten this paragraph and remove passive constructions." / "What is the liability limit and does it apply both ways?"**

20.2 OpenAI Codex as second model

The Codex plugin integrates the OpenAI Codex CLI directly into Claude Code. This enables code reviews by a second, independent model – without leaving the Claude session. AI models recognize their own errors worse than others' – a second model brings a fresh perspective.

> **Note: This is not about the old Codex model by OpenAI (discontinued 2023), but about the OpenAI Codex CLI – a terminal-based coding agent by OpenAI (April 2025, Open Source).**

20.2.1 What it actually is

The OpenAI Codex CLI is a standalone command-line tool by OpenAI that works similarly to Claude Code: it reads code, analyzes it, and provides feedback. All reviews are read-only – Codex does not modify any files.

20.2.2 Which account do I need?

Criterion	ChatGPT Account	OpenAI API Key

Costs	Free possible (with limits)	Pay-per-use, very inexpensive
Setup effort	Low (browser login)	Medium (obtain API key)
Limits	Daily limited (free)	Limited only by balance
Recommended if	You already use ChatGPT	You frequently do reviews
Cost per review	Free / included in Plus	approx. 0.001–0.01 USD

20.2.3 Setup step by step

Step 1 – Check prerequisites

```
node --version    # must be 18.18 or newer
```

Step 2 – Install Codex CLI

```
npm install -g @openai/codex
```

Step 3 – Authenticate

```
# Option A: ChatGPT Account
codex login
# Browser opens → log in with OpenAI account

# Option B: API Key
export OPENAI_API_KEY="sk-..."
```

Step 4 – Install plugin in Claude Code

```
/plugin marketplace add openai/codex-plugin-cc
/plugin install codex@openai-codex
/reload-plugins
```

Step 5 – Verify installation

```
/codex:setup    # shows whether Codex CLI is found and logged in
```

> **Note: If the CLI is not found: restart Claude Code and run /codex:setup again. If the login has expired: run codex login again in the terminal.**

20.2.4 Available Commands

Command	What it does	When to use
/codex:review	Neutral review of recent changes	After each feature as standard check
/codex:adversarial-review	Aggressive review – actively searches for errors, security vulnerabilities	Before a merge or release
/codex:rescue	Delegate task completely to Codex	For smaller subtasks that Claude is not currently solving
/codex:status	Shows running background jobs	When a long review is still running

/codex:result	Retrieves results of completed jobs	After /codex:status
/codex:cancel	Cancels running jobs	When a review is no longer needed

20.2.5 When is it worth it?

The plugin is useful when:

- You already have a ChatGPT or OpenAI account (no extra effort)
- You write code that is deployed in production (security, correctness important)
- You want an independent check before a release
- You want to use /codex:adversarial-review as the final gate before a merge

The plugin is less useful when:

- You don't have an OpenAI account and don't want to create one
- You primarily write experimental or throwaway code
- Claude's built-in reviews with /ultrareview are already sufficient

> **Recommended workflow: Opus writes the code, then /codex:adversarial-review as a second opinion. Codex tokens cost about 1/4 of Opus – reviews are nearly free.**

21 Learning path: From beginner to expert

Claude Code has many features and extension options. Anyone who tries to set up everything at once quickly loses overview and often no longer fully understands their own configuration later. Therefore, it is recommended to proceed step by step and only add new features once the basics are well understood:

Step 1 – First real feature

Find a manageable feature from your own project. Enable Plan Mode, review the plan, then use Auto-Accept for building, and finally /review.

Step 2 – Sharpen CLAUDE.md

After two or three sessions notice patterns. These insights belong in it: iterative, short, precise.

Step 3 – Permissions clean

Create global deny rules once. Let project-specific allow lists grow from daily use.

Step 4 – Hooks for mechanics

Start with the formatter. SwiftFormat (or Prettier, Black) after each edit.

Step 5 – Skills for repetitions

What is needed three times a month becomes a skill. Create only when the need is felt.

Step 6 – Subagents for complex tasks

When sessions get too long, offload reviews and research to subagents.

Step 7 – MCP targeted deployment

Start with GitHub, first only read operations.

Step 8 – Advanced setup

Worktrees, Sandbox, Routines, RTK, and claude-mem for maximum efficiency.

22 What Claude Code is not

Despite many possibilities, Claude Code has clear limits. Some tasks still require human control, technical understanding, or additional tools and processes.

Not a replacement for understanding the code yourself.

Those who don't understand the code Claude produces build technical debt.

Not a partner for strictly confidential work on consumer plans.

On Pro and Max, every read file goes to Anthropic. Sensitive data belongs in Team, Enterprise, or API setup.

Not a real-time tool.

Answers take time, especially with Opus. For live pair programming, Claude is too slow.

No magic for very large monorepos.

With repos in the multi-million line range, Claude slows down. Selective loading with @-mentions becomes mandatory.

Not a replacement for clean tests and CI.

Claude can write tests – the pipeline must run them.

23 Troubleshooting

Especially during initial setup or with advanced features, problems can occur. The following solutions cover typical error sources around installation, plugins, MCP servers, permissions, and sessions.

claude: command not found after installation

Close the terminal completely and reopen it. If not fixed: check if ~/.local/bin is in PATH.

Browser login fails

Common causes: no paid plan, different account in browser, expired session. Solution: claude logout, then restart claude.

claude doctor reports errors

The tool shows itself what is missing. Most common cases: incomplete PATH, missing login, outdated version. Run claude update.

Permission prompts are annoying

Maintain permissions in .claude/settings.json. /permissions allowlist suggests a ready-made allow list.

Plugin does not work after installation

Run /reload-plugins or restart Claude Code.

MCP server is displayed as unavailable

/mcp checks the status. Set API keys as environment variables. Check Node.js version (18+ recommended).

Codex plugin: /codex:setup fails

Check Node.js version 18.18+. Install Codex CLI: npm install -g @openai/codex. Then run codex login and /reload-plugins.

Routine fails when accessing API key

Place API Keys in the Environment Variables of the Cloud Environment. Explicitly specify in the prompt: "Use the API Key from the Environment Variables."

Migration to new computer

Back up ~/.claude/ without cache/. Don't take auth tokens – log in again after the switch.

24 Glossary

Over the course of working with Claude Code, many special terms, modes, and tools appear. This glossary summarizes the most important terms concisely and clearly and serves as a quick reference guide.

Term	Explanation
Auto-Accept	Mode in which Claude acts without asking, unless blocked by Deny. Activate via Shift+Tab.
Auto Mode	Extended autonomous mode (Max-Plan). Claude makes all decisions independently.
CLAUDE.md	Markdown file that is automatically loaded with each session. Contains project context, build commands, conventions, and prohibitions.
Codex CLI	OpenAI open-source coding agent (released April 2025). Used via the Codex plugin in Claude Code for independent code reviews.
Effort-Level	Controls how long Opus 4.7 thinks about a problem. From low to max.
Hook	Shell command that is automatically executed on a lifecycle event (PostToolUse, PreToolUse, Stop).
Loop	Loop via /loop that executes a command at intervals.
MCP-Server	External service (Model Context Protocol) that provides Claude with new tools.
Permissions	Allow/Deny rules in settings.json that define what Claude can or cannot do.
Plan Mode	Read-only mode in which Claude only analyzes and plans. Via Shift+Tab.
Plugin	Pre-made bundle of multiple components (Skills, Hooks, Subagents, Settings).
Routine	Cloud automation that runs completely autonomously and without a local server.
RTK	Rust Token Killer – CLI proxy that compresses command output (60–90% fewer tokens).
Sandbox	Technically isolated area with filesystem and network isolation.
Schedule	Scheduled task via /schedule with cron syntax.
Skill	Reusable custom command in skills/<name>/SKILL.md, callable via /name.
Subagent	Specialized agent with its own context that returns only a summary.
Worktree	Isolated git checkout for parallel work across multiple Claude instances.

The contents of this website are based on personal experience as well as publicly available information from documentation, training, videos, and community contributions. All contents were independently researched, structured, and formulated in my own words. Protected content is not reproduced verbatim.