Claude Code

Complete Installation, Configuration & User Guide

For Linux Mint 22.x (Cinnamon / MATE / XFCE)

Published at tools.ruggi.site • Author: Dario Ruggi/tools.ruggi.site

March 2026

Table of Contents

Table of Contents

Table of Contents

1. What is Claude Code?

This guide is written specifically for Linux Mint 22.x users (Cinnamon, MATE, or XFCE). It covers installation, configuration, and advanced usage of Claude Code on Linux Mint. Published at `https://tools.ruggi.site/` *by Dario Ruggi. Updated March 2026.*

Claude Code is Anthropic's official AI coding assistant that runs entirely inside your terminal. It understands your codebase through natural language, can read and write files, execute shell commands, manage git workflows, and orchestrate complex multi-step development tasks — all without leaving the command line.

1.1 Key Capabilities

- Reads, edits, and creates files across your project
- Executes shell/bash commands on your behalf
- Full git integration: commits, diffs, branches, PRs
- Multi-agent orchestration: run parallel sub-agents on different tasks
- MCP (Model Context Protocol) server support for external service integrations
- Project memory via CLAUDE.md — persistent instructions per project
- Custom slash commands and hooks for automation
- Auto-updates in the background (can be disabled)

1.2 System Requirements

Requirement	Details
OS	Linux Mint 22.x / Ubuntu 20.04+ / Debian 10+ ✓ Your system is fully supported
Architecture	x86_64 (most modern PCs and laptops) — fully supported
RAM	4 GB minimum recommended; 8 GB+ comfortable; 32 GB+ ideal for local LLMs
Internet	Required for API calls to Anthropic
Node.js	Only needed if using npm install method (native installer: NOT required)
Subscription	Claude.ai Pro, Max, or Team plan — OR — Anthropic Console API key

2. Pre-Installation Checklist

Before installing Claude Code, go through the checklist below. Most steps are one-time only.

2.1 Verify your shell

```
echo $SHELL          # Should show /bin/bash or /bin/zsh
bash --version       # Confirm bash is available
```

2.2 Install curl (already present on Linux Mint)

```
which curl           # Should print /usr/bin/curl
```

If missing:

```
sudo nala install curl -y
```

2.3 (Optional) Install Node.js — needed for some MCP servers

The native installer does NOT need Node.js. However, popular MCP servers (Context7, Playwright, GitHub) use npx, so installing Node.js is recommended.

```
curl -fsSL https://deb.nodesource.com/setup_22.x | sudo -E bash -
sudo nala install nodejs -y
node --version       # Should print v22.x.x
npm --version        # Should print 10.x.x
```

■ *Node.js 18+ is the minimum, but 22 (LTS) is recommended for 2026.*

3. Installing Claude Code

Anthropic now provides a native binary installer as the recommended method. It is self-contained, requires no Node.js runtime, and auto-updates in the background. The legacy npm method still works but is officially deprecated.

3.1 Method A — Native Installer (Recommended)

Run a single command in your terminal:

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

The script will download the correct binary for your architecture (x86_64), place it in `~/.local/bin/claude` or `~/.claude/bin/claude`, and automatically add it to your PATH.

3.2 Method B — npm Install (Legacy, deprecated)

■ *Do NOT use sudo with npm install -g. It causes permission errors and security risks.*

```
# Set up a user-level npm directory first (prevents sudo issues):
mkdir -p ~/.npm-global
npm config set prefix '~/.npm-global'
echo 'export PATH=~/.npm-global/bin:$PATH' >> ~/.bashrc
source ~/.bashrc
# Now install Claude Code:
npm install -g @anthropic-ai/claude-code
```

3.3 Verify the Installation

```
claude --version     # Should print something like: Claude Code 2.x.x
which claude         # Should print the binary path
```

■ *If 'claude' is not found, close and reopen your terminal, then try again.*

3.4 Migrating from npm to Native (if you had a previous install)

```
# Option 1: run from inside an active Claude Code session:
claude install
# Option 2: manual migration:
npm uninstall -g @anthropic-ai/claude-code
curl -fsSL https://claude.ai/install.sh | bash
```

Your MCP configurations, project settings, and session history are preserved during migration.

4. Authentication

Claude Code supports three authentication paths. Choose the one that matches your subscription.

Method	Who it's for	How to authenticate
Claude.ai Pro/Max	Individual subscribers on claude.ai	Login with your Claude.ai account via browser on first run
Console API Key	API users / developers	Set ANTHROPIC_API_KEY environment variable
Team / Enterprise	Organizations	Admin sets up OAuth; users log in via SSO

4.1 First Run — Browser Login (Pro/Max)

```
cd ~/          # go to any directory
claude         # launch Claude Code
```

On first launch, Claude Code will open your browser (Brave, in your case) to complete OAuth login with your claude.ai account. After login the session token is saved to `~/.claude/`.

4.2 API Key Authentication (Console)

If you are using an Anthropic Console API key instead of a subscription:

```
# Temporary (current session only):
export ANTHROPIC_API_KEY='sk-ant-api03-...'

# Permanent (add to your ~/.bashrc):
echo 'export ANTHROPIC_API_KEY="sk-ant-api03-..."' >> ~/.bashrc
source ~/.bashrc
```

■ *Never commit your API key to git. Add .env to your .gitignore.*

5. Configuration Files

Claude Code uses a hierarchy of configuration files. Settings at the lower levels override higher ones for the same key.

File	Scope	Purpose
~/.claude/settings.json	Global (all projects)	Your personal defaults: preferred model, permissions, env vars
.claude/settings.json	Project (team-shared)	Project-specific config — commit this to git
.claude/settings.local.json	Project (local only)	Personal overrides for a project — add to .gitignore
~/.claude.json	Global legacy	Auth tokens, history, stats — managed by Claude Code automatically
CLAUDE.md	Project memory	Natural-language instructions Claude reads at session start

5.1 Global settings.json — Recommended starter config

Create this file:

```
mkdir -p ~/.claude && xed ~/.claude/settings.json
```

Paste the following content:

```
{
  "model": "claude-opus-4-6",
  "env": {
    "DISABLE_AUTOUPDATER": "0",
    "DISABLE_TELEMETRY": "0",
    "CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC": "0"
  },
  "permissions": {
    "allowedTools": ["Read", "Write", "Bash", "Edit"],
    "deny": [
      "Read(./.env)",
      "Read(./.env.*)",
      "Write(./production.*)"
    ]
  },
  "attribution": {
    "commits": true,
    "pullRequests": true
  }
}
```

■ *DISABLE_AUTOUPDATER: "1" — control when Claude Code updates manually. DISABLE_TELEMETRY: "1" — opt out of usage telemetry sent to Anthropic. CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC: "1" — disables ALL non-critical network calls (tips, surveys, update checks), leaving only actual API calls to Claude. Set any of these to "0" (or remove the line) to keep the default behaviour.*

5.2 Project CLAUDE.md — Persistent project memory

The CLAUDE.md file lives at the root of each project. Claude Code reads it automatically at session start. Use it to record project conventions, build commands, coding standards, and anything Claude should always know.

Auto-generate one with:

```
cd /path/to/your/project
claude          # start a session
/init           # Claude scans the project and writes CLAUDE.md
```

Example CLAUDE.md for a Python project:

```
# MyProject — Python / FastAPI

## Build
pip install -r requirements.txt && python -m pytest

## Test
python -m pytest tests/ -v

## Code conventions
- Python naming: snake_case for functions, PascalCase for classes
- One class per file, filename matches class name
- No JavaScript frameworks in this project

## Important paths
- Source: src/
- Tests: tests/
- Output: dist/
```

6. Your First Session — Step by Step

This walkthrough gets you from zero to your first productive session.

1. **Open a terminal**

Press Ctrl+Alt+T or open a terminal from your desktop menu.

2. **Navigate to your project**

```
cd ~/Projects/MyFPCProject
```

3. **Launch Claude Code**

```
claude
```

4. **Wait for authentication**

On first run, your browser opens. Log in with your claude.ai account.

5. **Initialize project memory**

Type `/init` and press Enter. Claude scans your project and creates CLAUDE.md.

6. **Ask Claude something**

Type a natural language request, e.g.: "Explain what this project does" or "List all Python files that define a class".

7. **Exit the session**

Type `/exit` or press Ctrl+C.

> ■ *Use the up-arrow key in a Claude Code session to navigate your prompt history, even from previous sessions.*

7. Slash Commands Reference

Slash commands are typed directly in the Claude Code prompt. They control session behavior, context, and configuration.

7.1 Essential Commands

Command	What it does	When to use
/help	List all available commands	Any time you're lost
/init	Generate CLAUDE.md from your project	First time in a new project
/clear	Clear conversation history (free token space)	When switching to a new task
/compact	Summarize history, keep key context	When context > 80% full — mid long session
/compact [focus]	Compact, keeping specific info	/compact "retain the auth module changes"
/exit	Exit Claude Code	End of session
/cost	Show token usage and session cost	Check spending during session
/status	Show current model, auth, and session info	Diagnose connection issues
/model	Switch model interactively	Change between Opus, Sonnet, Haiku
/config	View/edit settings interactively	Adjust settings without editing JSON
/memory	View and edit CLAUDE.md files	Update project memory mid-session
/add-dir	Add another directory to current session	Multi-repo workflows
/bug	Report a bug to Anthropic	When something goes wrong
/doctor	Diagnose installation issues	Troubleshooting

7.2 Keyboard Shortcuts (Interactive Mode)

Shortcut	Action
Shift+Tab	Cycle through modes: Normal → Auto-accept edits → Plan mode
Ctrl+C	Interrupt current Claude action
Ctrl+D	Exit session

Shortcut	Action
↑ / ↓	Navigate prompt history (across sessions)
Ctrl+L	Clear terminal screen
! command	Run a shell command directly (bypasses Claude, uses fewer tokens)
@filename	Reference/include a file in your prompt

7.3 Session Modes

Press Shift+Tab to cycle between three modes. The current mode is shown below the prompt:

- **Normal mode** — Claude asks permission before each file edit or command
- **Auto-accept mode** — Claude applies edits without asking (faster, less interruption)
- **Plan mode** — Claude describes what it plans to do before doing anything

■ *Use Plan mode when starting a large task so you can review the approach before Claude modifies any files.*

8. Choosing a Model

Claude Code supports multiple Claude models. Use `/model` inside a session to switch, or set a default in `settings.json`.

Model	Best for	Cost / Speed
claude-opus-4-6	Complex architecture, deep analysis, hard bugs	Highest quality — slower, higher cost
claude-sonnet-4-6	Everyday coding, refactoring, explanations	Great balance — recommended default
claude-haiku-4-5	Quick lookups, simple edits, fast turnaround	Fastest, lowest cost

Switch model mid-session:

```
/model                      # Opens interactive model selector
# or use CLI flag at launch:
claude --model claude-sonnet-4-6
```

■ *Pro/Max subscribers: Claude Code starts on Opus until you hit 50% usage limit, then automatically drops to Sonnet for cost efficiency.*

8.1 What happens when you switch model mid-session?

This is an important question. When you use `/model` to switch during an active session, the following happens:

- **Conversation history is fully preserved.** Everything Claude has read, written, and discussed in the current session remains in context. The new model receives the complete history.
- **The new model takes over immediately.** The next message you send is processed by the newly selected model. There is no transition message or gap.
- **File and tool permissions are unchanged.** The allow/deny rules in settings.json apply regardless of which model is active.
- **Token costs change from that point forward.** Messages sent before the switch are already billed at the old model's rate. Only new messages use the new model's pricing.
- **CLAUDE.md and custom commands remain active.** Project memory and any slash commands you have defined are model-agnostic — they work the same with any model.

Practical use: A common workflow is to start a session on `claude-opus-4-6` for the complex planning and architecture phase, then switch to `claude-sonnet-4-6` for the routine implementation work. You keep the full context of the planning conversation, but pay less for the mechanical coding that follows.

■ *Switching to a smaller model mid-session does NOT summarise or truncate the context. If your context window is large, the smaller model must still process it all — which can be slow or hit the smaller model's context limit. Use /compact before switching if the session history is very long.*

9. Custom Slash Commands

You can create your own slash commands as simple Markdown files. These become shortcuts for common prompts.

9.1 Where to put command files

Location	Scope
~/.claude/commands/	Personal — available in ALL your projects
.claude/commands/	Project — available only in this project

9.2 Creating a command — Example: /review

```
mkdir -p ~/.claude/commands
xed ~/.claude/commands/review.md
```

File contents:

```
Review the file(s) at $ARGUMENTS for:
- Logic errors and off-by-one bugs
- Memory leaks or resource management issues
- Code style consistency with the rest of the project
- Anything that could cause runtime exceptions
Summarize findings as a bullet list with severity: [HIGH], [MED], [LOW]
```

Use it in a session:

```
/review src/main.py
```

■ *The $ARGUMENTS placeholder is replaced by whatever you type after the command name.*

9.3 Example commands for common development tasks

Create these files in `~/.claude/commands/` to have them available in every project. Examples below use Python, but adapt them to any language.

explain.md — explain a Pascal unit in plain English

```
Read the Pascal unit at $ARGUMENTS carefully.
Explain it in plain English:
1. What is the purpose of this unit?
2. What are the main types, classes, or records it defines?
3. What does each public procedure or function do?
4. Are there any non-obvious design decisions worth noting?
Keep the explanation concise but complete. Assume the reader knows
programming but is unfamiliar with this specific unit.
```

Usage: `/explain src/mymodule.py`

refactor.md — modernise FPC code style

```
Refactor the file at $ARGUMENTS to follow modern clean-code conventions:
- Replace repetitive code with well-named helper functions
- Use descriptive variable and function names throughout
- Replace magic numbers and strings with named constants
- Simplify complex conditionals for readability
- Ensure all resources are properly closed or released
- Fix any inconsistent naming to follow language conventions
Show a summary of changes made. Do not change logic, only style.
```

Usage: `/refactor src/utils.py`

document.md — add XML doc comments

```
Add Google-style docstrings to all public functions, classes,
and methods in $ARGUMENTS.
Format:
  """Short one-line description.
  Args:
      param_name: Description of parameter.
  Returns:
Do not modify any existing code. Only add comments.
If a comment already exists, improve it but keep the format.
```

Usage: `/document src/main.py`

buildfix.md — diagnose and fix a build error

```
Run the build/test command for this project on $ARGUMENTS.
Analyse the build output.
For each error or warning:
1. Identify the file and line number
2. Explain what is wrong in plain language
3. Show the fix
Apply all fixes directly to the source files.
Apply all fixes. Then run the build again to confirm it succeeds.
```

Usage: `/buildfix MyProject.lpi`

commit.md — write a git commit message and commit

```
Review the current git diff (run: git diff --staged).
Write a conventional commit message following this format:
  <type>(<scope>): <short summary>

  <body: what changed and why, wrapped at 72 chars>
Types: feat, fix, refactor, docs, style, test, chore
Example: fix(scoring): correct misere penalty calculation
Then stage all modified files and commit with that message.
Show the final commit hash when done.
```

Usage: `/commit`

10. Hooks — Automate Actions

Hooks let you run shell commands automatically before or after Claude performs certain actions (like writing a file).

10.1 Hook lifecycle events

Event	Fires when…
PreToolUse	Before Claude uses any tool (Read, Write, Bash, Edit…)
PostToolUse	After Claude uses a tool successfully
Stop	When Claude finishes a turn
Notification	When Claude sends a notification

10.2 Example: auto-format Python files after edit

Add this to your project's `.claude/settings.json`:

```
{
  "hooks": {
    "PostToolUse": [
      {
        "matcher": "Write(*.py)",
        "hooks": [
          {
            "type": "command",
            "command": "ptop $file"
          }
        ]
      }
    ]
  }
}
```

■ *You can replace ptop with any formatter: black (Python), prettier (JS), gofmt (Go), etc.*

10.3 Example: run tests after any Python edit

```
"PostToolUse": [
  {
    "matcher": "Edit(*.py)|Write(*.py)",
    "hooks": [{ "type": "command",
                "command": "make build 2>&1 | tail -5" }]
  }
]
```

11. MCP Servers — Extending Claude Code

Model Context Protocol (MCP) is an open standard created by Anthropic that lets Claude Code connect to external tools and data sources through a unified interface. Think of MCP servers as plugins: each one gives Claude a new set of capabilities beyond what it can do natively.

Without MCP, Claude Code can only work with files and run shell commands on your local machine. With MCP servers, it can query live documentation, open GitHub issues, search your company's internal tools, control a browser, and more — all through the same natural language interface you already use.

Each MCP server runs as a separate process alongside Claude Code. When you ask Claude to do something that requires an MCP tool, it calls the server, gets the result, and incorporates it into its response — transparently. You don't need to change how you interact with Claude; the tools simply become available.

11.1 Configuring an MCP server

Add servers to `~/.claude/settings.json`:

```
{
  "mcpServers": {
    "context7": {
      "command": "npx",
      "args": ["-y", "@upstash/context7-mcp"]
    },
    "github": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-github"],
      "env": { "GITHUB_PERSONAL_ACCESS_TOKEN": "ghp_..." }
    }
  }
}
```

11.2 Useful MCP servers for a developer

The following servers cover the most common developer workflows. Each one is installed automatically the first time it is used via `npx`.

@upstash/context7-mcp — Live library documentation

One of the most practically useful servers. Claude's training data has a cutoff date, so it can produce outdated API calls for fast-moving libraries. Context7 fetches the real, current documentation for any library you are using and injects it directly into Claude's context. The result: no more hallucinated method names or deprecated patterns. Particularly valuable for fast-moving libraries where online resources can be sparse or outdated.

@modelcontextprotocol/server-github — GitHub integration

Gives Claude direct access to your GitHub account. You can ask Claude to list open issues, read the description of a specific issue and start working on it, create a pull request from the current branch, or search across all your repositories. Requires a GitHub Personal Access Token set as an environment variable in the server config.

@modelcontextprotocol/server-filesystem — Controlled filesystem access

Provides structured read/write access to directories you explicitly allow. Unlike Claude Code's native file tools — which operate relative to your current project — this server lets you grant access to specific paths anywhere on your system (e.g. a shared snippets library or a documentation folder) while keeping everything else blocked. Useful when you want Claude to reach across projects without giving it unrestricted access to your home directory.

@modelcontextprotocol/server-git — Advanced git operations

Extends Claude's git capabilities beyond what it can do through simple shell commands. Enables structured operations like reading full commit histories, comparing branches with rich diff output, searching commit messages, and understanding the repository's graph — all in a form Claude can reason about rather than just parsing raw text output.

@playwright/mcp — Browser automation

Lets Claude control a real browser (Chromium) to navigate pages, click elements, fill forms, and take screenshots. Useful for end-to-end testing of web UIs, scraping content from sites that require JavaScript, or automating repetitive browser tasks. Claude can write a test scenario in plain English and then execute it, verifying results step by step.

■ *MCP servers that use npx require Node.js 18+ to be installed. See Section 2.3 for Node.js installation.*

12. Security & Permissions

Claude Code can read files and execute shell commands. Understanding the permission model keeps you in control.

12.1 Permission model

- By default, Claude asks before every file edit, creation, or shell command
- Auto-accept mode (Shift+Tab) lets Claude act without asking — use cautiously
- deny rules in settings.json block Claude from touching sensitive files
- allowedTools limits which tool categories Claude can use

12.2 Recommended deny rules

```
"permissions": {
  "deny": [
    "Read(./.env)",        // never read .env files
    "Read(./.env.*)",      // never read .env.* variants
    "Write(./production.*)", // never touch production configs
    "Write(~/.ssh/*)",     // never touch SSH keys
    "Bash(rm -rf *)"       // never run destructive rm -rf
  ]
}
```

■ *deny rules for Bash commands only block Claude's built-in tools. Enable /sandbox mode for OS-level enforcement (uses Linux bubblewrap).*

12.3 Sandbox mode — and how it differs from deny rules

It is important to understand the difference between **deny rules** and **sandbox mode**, because they operate at completely different levels and protect against different things.

Deny rules — what they are

Deny rules live in `settings.json` and tell Claude Code's own internal logic what it is not allowed to do. When Claude decides to call a tool (Read, Write, Edit, Bash…), it checks the deny list first and refuses if there is a match. This works well for Claude's built-in tools.

The critical limitation: deny rules **do not apply to raw Bash commands**. If Claude runs `bash -c 'rm -rf ~/important'` as a shell command, the deny rules are bypassed entirely — they only see the Bash tool call, not what is inside it. So deny rules are a guardrail against accidental mistakes, not a security boundary.

Sandbox mode — what it is

Sandbox mode wraps Claude Code's entire execution inside a Linux `bubblewrap` container. Bubblewrap is the same low-level sandboxing technology used by Flatpak. It enforces restrictions at the **operating system level** — the kernel itself — so there is no way for any command, script, or tool call to bypass them,

regardless of how it is invoked.

In sandbox mode, your deny rules are also enforced for Bash commands. A shell command that tries to read `~/.ssh/id_rsa` will be blocked by the OS before it even runs, not just flagged by Claude's internal logic.

When to use each

Situation	Recommended approach
Day-to-day coding on your own projects	Deny rules are sufficient — fast, no overhead
Running Claude on unfamiliar or third-party code	Enable sandbox mode for the session
Automated / unattended Claude runs (scripts, CI)	Sandbox mode strongly recommended
Protecting credentials and SSH keys	Both: deny rules + sandbox for defence in depth

Enable sandbox for a single session:

```
/sandbox            # toggle sandbox on for the current session
```

Enable sandbox permanently in settings.json:

```
{ "sandbox": true }
```

■ *Sandbox mode requires bubblewrap to be installed. On Linux Mint: sudo nala install bubblewrap -y*

13. Updates & Maintenance

13.1 Auto-updates

Claude Code checks for updates on startup and installs them in the background. New versions take effect on your next session start.

To disable auto-updates:

```
# In ~/.claude/settings.json:
{ "env": { "DISABLE_AUTOUPDATER": "1" } }
# Or in ~/.bashrc:
export DISABLE_AUTOUPDATER=1
```

13.2 Manual update

```
# Native install — re-run the installer:
curl -fsSL https://claude.ai/install.sh | bash
# npm install — update the global package:
npm update -g @anthropic-ai/claude-code
```

13.3 Diagnose problems

```
claude --version     # check installed version
/doctor              # run built-in diagnostics inside a session
```

13.4 Full uninstall

```
# Native install:
rm -f ~/.local/bin/claude
rm -rf ~/.local/share/claude
# npm install:
npm uninstall -g @anthropic-ai/claude-code
# Remove config and history (optional):
rm -rf ~/.claude ~/.claude.json
```

14. Troubleshooting

'claude: command not found' after install

- Close and reopen the terminal (PATH is updated in .bashrc)
- Run: source ~/.bashrc
- Check: echo $PATH — look for ~/.local/bin or ~/.claude/bin
- If missing, add manually: echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc

Authentication fails / browser doesn't open

- Make sure Brave is your default browser (it should open automatically)
- Try: BROWSER=brave claude
- Check internet connectivity
- If using API key: verify export ANTHROPIC_API_KEY is set in your terminal

Permission errors during npm install

- Never use sudo npm install -g
- Set up user-level npm prefix: npm config set prefix '~/.npm-global'
- Add to PATH: echo 'export PATH=~/.npm-global/bin:$PATH' >> ~/.bashrc

Claude stops mid-task asking for permission (annoying)

- Press Shift+Tab to enable Auto-accept mode
- Or add the tools to allowedTools in settings.json

Context window full / slow responses

- Run /compact to summarize history and free space
- Run /clear when switching to a completely new task
- Use /cost to check token usage
- Switch to Sonnet or Haiku for less context-heavy tasks

MCP server not working

- Verify Node.js 18+ is installed: node --version
- Check the server name/args in settings.json match the npm package exactly
- Test the npx command manually: npx -y @upstash/context7-mcp
- Run /doctor for diagnostics

15. Tutorial — Practical Examples

These worked examples show how to use Claude Code for real tasks. All examples use generic project names — adapt them to your own stack.

Tutorial 1 — Understand an unfamiliar codebase

```
cd ~/Projects/MyProject
claude
> Explain the overall architecture of this project. What are the main units and
> what role does each play? List the key data structures.
```

Tutorial 2 — Find and fix a bug

```
> In mymodule.py, the calculate_score function seems to return wrong results
> for edge case inputs. Find the logic error and fix it.
```

Tutorial 3 — Add documentation to a unit

```
> Add XML doc comments to all public procedures and functions in
> mymodule.py. Follow Google-style docstring format.
```

Tutorial 4 — Refactor with git safety

```
> Before making changes, commit the current state of legacy_api.py to git.
> Then replace all the stub functions with real implementations.
> After, show me a diff of what changed.
```

Tutorial 5 — Multi-file search

```
> Find all places in the project where process_data is called. Show the file
> name, line number, and context for each occurrence.
```

Tutorial 6 — Generate a test suite

```
> Create a pytest test suite for the core logic in mymodule.py.
> Cover: normal inputs, boundary conditions, and edge cases.
> Put the tests in tests/test_mymodule.py.
```

Tutorial 7 — Shell scripting assistance

```
> Look at deploy.sh in the project root. Add a new menu option
> that rolls back the last deployment, with a confirmation prompt before running.
```

■ *For long tasks, start with Plan mode (Shift+Tab twice) so Claude describes its plan before touching any files. You can then approve, redirect, or abort.*

16. Backup, Restore & Migration

Claude Code stores all its configuration, MCP server definitions, custom commands, session history, and authentication tokens in a small set of files and directories. Knowing where these are makes it trivial to back them up, restore them after a reinstall, or move your setup to a new machine.

16.1 What to back up

Path	What it contains
~/.claude/settings.json	Your global config: model, permissions, env vars, MCP servers
~/.claude/CLAUDE.md	Your global memory / instructions for Claude
~/.claude/commands/	All your personal custom slash commands
~/.claude.json	Auth tokens, approved API keys, session stats (auto-managed)
.claude/ (per project)	Project-specific settings, commands, local CLAUDE.md
~/ClaudeLibrary/	Your personal knowledge library (if you created one — Section 17)

16.2 Backup script

Run this whenever you want a snapshot — before a system update, before reinstalling, or on a schedule via cron:

```
#!/bin/bash
# backup-claude.sh — backs up Claude Code config to a dated archive
DEST=~/Backups/claude-backup-$(date +%Y%m%d).tar.gz
mkdir -p ~/Backups
tar -czf "$DEST" \
    ~/.claude/ \
    ~/.claude.json \
    ~/ClaudeLibrary/   # remove this line if you don't use it
echo "Backup saved to: $DEST"
```

Make it executable and run:

```
chmod +x ~/backup-claude.sh
~/backup-claude.sh
```

16.3 Restore after a fresh install

After installing Claude Code on a fresh system (see Section 3), restore your backup:

```
# 1. Install Claude Code (native installer)
curl -fsSL https://claude.ai/install.sh | bash

# 2. Extract your backup
```

```
tar -xzf ~/Backups/claude-backup-YYYYMMDD.tar.gz -C /

# 3. Verify
claude --version
cat ~/.claude/settings.json
```

■ *If you use git to version your ~/.claude/ directory, restoration is just a git clone followed by the Claude Code installer. Much cleaner than tar archives.*

16.4 Git-based versioning (recommended)

A better long-term approach than manual backups:

```
cd ~/.claude
git init
# Create a .gitignore to exclude sensitive files
echo "*.log" > .gitignore
echo "projects/"  >> .gitignore   # session history — often large
git add settings.json CLAUDE.md commands/
git commit -m "Initial Claude Code config"

# Push to a private repo for off-site backup:
git remote add origin git@github.com:youruser/claude-config.git
git push -u origin main
```

■ *Never commit ~/.claude.json to a public repository — it contains your authentication tokens.*

16.5 Migrating to a new machine

- Install Claude Code on the new machine (Section 3)
- Clone your private config repo: `git clone git@github.com:youruser/claude-config.git ~/.claude`
- Copy or restore ~/.claude.json from backup (or just log in fresh — it regenerates automatically)
- If you use MCP servers that need Node.js, install Node.js (Section 2.3)
- Run `claude` and verify everything works with `/doctor`

17. Claude Code on the Web

In addition to the CLI, Claude Code is available through a browser interface. This opens up new workflows — particularly for running tasks in the cloud while you continue working locally, or for collaborating with others.

17.1 The official web interface — claude.ai/code

Anthropic's official web interface is available at `claude.ai/code` (currently in beta). It lets you connect your GitHub repositories, describe a task in plain language, and have Claude execute it on Anthropic-managed cloud infrastructure — no terminal required.

- **Connect GitHub:** Link your repositories directly. Claude can clone, edit, test, and open PRs.
- **Run tasks in parallel:** Each task runs in its own isolated cloud environment. Start multiple tasks simultaneously.
- **Real-time steering:** Monitor progress and redirect Claude mid-task, just like in the CLI.
- **Mobile access:** Sessions are also accessible via the Claude iOS app.
- **Session sharing:** Share session links with teammates (Pro/Max: public or private; Teams/Enterprise: team-visible).
- **Open in CLI:** Any web session can be handed off to your local terminal with one click.

■ *Best use case for the web interface: kicking off long-running or parallel tasks that you want to run in the cloud while you do other things locally.*

17.2 Remote sessions from the CLI

You can also start a remote (cloud) session directly from your terminal, which appears in your claude.ai/code dashboard:

```
# Start a task that runs in the cloud:
claude --remote 'Fix all failing tests in the auth module'

# Check on running remote tasks:
/tasks

# Teleport a remote session back to your terminal:
# (Open In CLI button on claude.ai/code, or use the session URL)
```

A powerful pattern is to plan locally and execute remotely:

```
# 1. Start locally in plan mode to design the approach:
claude  # then press Shift+Tab twice for Plan mode
> Plan how to refactor the scoring module. Don't make changes yet.

# 2. Once satisfied with the plan, hand off to the cloud:
claude --remote 'Execute the refactoring plan we just discussed'
```

17.3 Community web UIs — local browser interfaces

Several community-built tools add a browser interface on top of your local Claude Code installation. These are not Anthropic products but are widely used:

Claudia GUI — claudia.so

An open-source desktop app (Tauri + React) built by a YC-backed team. Adds a visual project browser, custom agent creation, cost analytics, session history with branching, and MCP server management — all as a native desktop app on Linux. Free and open-source (AGPL).

Claude Code WebUI — github.com/sugyan/claude-code-webui

A lightweight local web server that wraps your existing Claude Code installation and exposes it as a browser chat UI. Install globally with `npm install -g claude-code-webui`, then run `claude-code-webui` and open `http://localhost:8080`. Your auth, files, and git repos are all local — nothing goes to a third-party server.

CloudCLI — github.com/siteboon/claudecodeui

A more full-featured web UI with file explorer, git integration, session management, and a plugin system. Reads from and writes to the same `~/.claude` config that Claude Code uses natively, so any MCP servers you add via the UI show up in the CLI immediately. Available self-hosted (run on your own machine) or as a paid cloud service.

17.4 CLI + Web — working together

Workflow	Best interface
Quick code edits, interactive debugging	CLI — fastest, lowest overhead
Long autonomous tasks (run while you do other things)	Web interface / --remote
Code review with non-developer teammates	Web interface (shareable sessions)
Visual file browsing and session management	Claudia GUI or CloudCLI
Working from a phone or tablet	claude.ai/code (mobile browser or iOS app)
Full control, privacy, offline work	CLI only

18. Using Local LLMs with Claude Code

Yes — Claude Code can use a local LLM running on your own machine instead of Anthropic's API. This means zero API cost, complete privacy (nothing leaves your machine), and the ability to work fully offline.

The key: Claude Code speaks the Anthropic Messages API format. As of late 2025, both **Ollama** (v0.14.0+) and **LM Studio** (v0.4.1+) implement an Anthropic-compatible endpoint, so Claude Code can talk to them directly — no adapter needed.

■ *Local LLMs are significantly less capable than Claude Opus or Sonnet for complex reasoning and large codebases. They work well for simple edits, quick lookups, and offline work. Expect slower responses and occasional quality gaps. Minimum recommended context window: 32K tokens (64K+ preferred).*

18.1 Method A — Ollama (recommended, CLI-native)

Ollama is the easiest path. It runs as a local server and supports the Anthropic Messages API natively since v0.14.0.

Install Ollama

```
curl -fsSL https://ollama.com/install.sh | sh
ollama --version   # verify
```

Pull a coding-optimised model

```
# Good choices for coding tasks (need at least 32K context):
ollama pull qwen2.5-coder:7b    # 4.7GB — fast, good for most tasks
ollama pull qwen2.5-coder:32b   # 19GB  — much better quality, needs ~24GB RAM
ollama pull devstral:24b        # 14GB  — Mistral's coding model

# Check what you have:
ollama list
```

Start Ollama and connect Claude Code

```
# Start the Ollama server (auto-starts on install, but manual if needed):
ollama serve

# In a new terminal, set env vars and launch Claude Code:
export ANTHROPIC_AUTH_TOKEN=ollama
export ANTHROPIC_API_KEY=''
export ANTHROPIC_BASE_URL=http://localhost:11434
claude --model qwen2.5-coder:7b
```

To make this permanent, add to `~/.claude/settings.json`:

```
{
  "model": "qwen2.5-coder:7b",
  "env": {
    "ANTHROPIC_AUTH_TOKEN": "ollama",
    "ANTHROPIC_API_KEY": "",
```

```
    "ANTHROPIC_BASE_URL": "http://localhost:11434"
  }
}
```

18.2 Method B — LM Studio (GUI-based model manager)

LM Studio provides a graphical interface for downloading and running local models. It added Anthropic API compatibility in v0.4.1.

- Download LM Studio from `lmstudio.ai/download` and install it
- Open LM Studio, go to the Models tab, and download a coding model
- Go to the Server tab and click Start Server (default port: 1234)

```
# Connect Claude Code to LM Studio:
export ANTHROPIC_AUTH_TOKEN=lmstudio
export ANTHROPIC_BASE_URL=http://localhost:1234
claude --model your-model-name
```

18.3 Switching between local and cloud

You don't have to choose permanently. Use shell aliases to switch quickly:

```
# Add to ~/.bashrc:
# Cloud mode (default Anthropic API):
alias claude-cloud='unset ANTHROPIC_BASE_URL ANTHROPIC_AUTH_TOKEN; claude'
# Local mode (Ollama):
alias claude-local='ANTHROPIC_AUTH_TOKEN=ollama \
  ANTHROPIC_API_KEY= \
  ANTHROPIC_BASE_URL=http://localhost:11434 \
  claude --model qwen2.5-coder:7b'
```

■ *A typical modern Linux machine with 16GB+ RAM can run 7B models comfortably in CPU mode. GPU acceleration requires 6GB+ VRAM, but Ollama's CPU inference with AVX2 support is still usable for development tasks. For better quality, try qwen2.5-coder:7b.*

18.4 Recommended local models for your system

Model	Size	Notes
qwen2.5-coder:7b	4.7 GB	Good default — fast, requires ~8GB RAM
qwen2.5-coder:14b	9 GB	Better quality, still runs well in CPU mode
devstral:24b	14 GB	Mistral's coding model — excellent but slower
granite3.3:8b	5 GB	IBM model, 128K context, strong tool-use support

All models above support tool-use (required for Claude Code's file operations) and have at least 32K context windows. Pull any of them with `ollama pull` .

18.5 Using local and cloud models together

You are not locked to a single provider per session. By switching the active endpoint you can combine local and cloud models within the same project, using each where it makes the most sense.

Practical workflow

- **Start local** - read files, explore, understand the codebase. Free, instant, private. Use qwen2.5-coder:7b or similar.
- **Switch to cloud** - when you need deep reasoning: complex algorithms, subtle bugs, architectural decisions. Full Claude Sonnet or Opus quality.
- **Switch back to local** - mechanical follow-up: boilerplate, tests, formatting, once the hard thinking is done.

How to switch mid-session

```
# 1 - Explore with local model:
claude-local
> Read the main module and explain how the data pipeline works.
> Summarise key findings in 5 bullet points for handoff.
  [copy the output]
# 2 - Switch to cloud for deep reasoning:
/exit
claude-cloud
> Context from prior session: [paste summary here].
> Redesign the pipeline to handle edge cases X, Y, Z.
# 3 - Switch back to local for mechanical work:
/exit
claude-local
> Write unit tests for the new misere scoring function.
```

Preserving context across switches

- Before exiting the local session, ask for a summary of key findings and paste it as your first message in the next session.
- Save important decisions to `CLAUDE.md` so they auto-load each session.
- Keep a `HANDOFF.md` in the project root that you update at the end of each local session: what was done, what is next, open questions.

■ *Use qwen2.5-coder:7b locally for all file reading and exploration (free, instant), then switch to claude-sonnet-4-6 only for the parts that need deep reasoning. This can reduce API spend by 60-80% on large projects.*

19. Building a Personal Knowledge Library

One of the most powerful ways to use Claude Code is to give it a persistent, organised library of your own code — reusable projects, snippets, reference implementations, and even technical books or documentation — that it can draw from in any session. This section explains how to set that up.

19.1 The concept

Claude Code has no long-term memory on its own. Each session starts fresh. But it **can read files** — so if you maintain a well-organised directory of reusable material and tell Claude where it is, that directory effectively becomes Claude's external memory for your work.

The library lives on your disk. You control what goes in it. Claude reads it when you want it to, either automatically at session start (via CLAUDE.md) or on demand (via `@path/to/file` references or the filesystem MCP server).

19.2 Recommended directory structure

```
~/ClaudeLibrary/
■■■ snippets/          # short reusable code fragments
■   ■■■ python/        # Python snippets
■   ■■■ bash/          # shell script patterns
■   ■■■ php/           # PHP utilities
■■■ projects/          # complete reference projects
■   ■■■ template-python-app/    # base Python application skeleton
■   ■■■ template-php-api/    # base PHP API skeleton
■■■ books/             # PDFs or extracted text of technical references
■   ■■■ fpc-reference.txt
■   ■■■ lazarus-components.txt
■■■ patterns/          # architectural patterns and design notes
■   ■■■ api-auth-pattern.md  # your preferred API auth pattern
■■■ LIBRARY.md         # master index — Claude reads this first
```

■ *Keep books as plain text or Markdown rather than PDF — Claude reads text directly and efficiently. Use pdftotext to extract from PDFs: pdftotext book.pdf book.txt*

19.3 LIBRARY.md — the master index

This file is the entry point. Claude reads it to understand what the library contains and where to find things, without having to read every file upfront.

```
# Claude Library — Your Name

## Snippets
- python/base-class.py   — base class template with logging
- python/db-connect.py   — database connection wrapper
- bash/menu-template.sh  — interactive bash menu pattern
```

```
- php/secure-get-post.php — custom encrypted GET/POST handler

## Reference Projects
- projects/template-python-app/  — complete Python app skeleton with i18n

## My Patterns & Conventions
- patterns/api-auth.md — preferred API auth and security pattern
- patterns/ui-guidelines.md    — dark mode color rules and UI conventions

## Books / Docs
- books/python-reference.txt — Python reference (text extracted from docs)
```

19.4 Making the library globally available

Add the library to Claude Code's global `~/.claude/settings.json` so it can be accessed from any project, using the filesystem MCP server:

```
{
  "mcpServers": {
    "library": {
      "command": "npx",
      "args": [
        "-y",
        "@modelcontextprotocol/server-filesystem",
        "/home/dario/ClaudeLibrary"
      ]
    }
  }
}
```

Alternatively — and more simply — add a line to your global `~/.claude/CLAUDE.md`:

```
## My Code Library
I have a personal code library at ~/ClaudeLibrary/.
Read ~/ClaudeLibrary/LIBRARY.md to understand its contents.
Use it as a reference when writing new code.
Ask me before adding anything to it.
```

■ *The CLAUDE.md approach is simpler and works without Node.js. The MCP filesystem server is better if you want Claude to be able to write new material to the library, not just read it.*

19.5 Per-project library subset

Not every project needs the whole library. You can give each project its own `CLAUDE.md` that points to only the relevant parts:

```
# MyProject — CLAUDE.md

## Relevant library entries
- ~/ClaudeLibrary/snippets/python/base-class.py
- ~/ClaudeLibrary/snippets/python/db-connect.py
- ~/ClaudeLibrary/patterns/api-auth.md
```

```
## Do NOT use from library
- anything in snippets/ not relevant to this project
```

This way Claude gets the right context for each project without being distracted by unrelated material.

19.6 Adding new material — strategy

The library only has value if it grows. Here are two complementary strategies:

Strategy A — User-triggered: save on demand

At any point in a session, you can ask Claude to save something to the library. Example prompts:

- "Save this function to ~/ClaudeLibrary/snippets/python/string-utils.py"
- "Add a summary of this encryption pattern to ~/ClaudeLibrary/patterns/encryption-comms.md"
- "Update LIBRARY.md to include the new file"

Because Claude can write files (with your permission), it will create or append to the file, then update the index. You review the result before anything is committed.

Strategy B — Claude-suggested: proactive recommendations

Add this instruction to your global `~/.claude/CLAUDE.md`:

```
## Library policy
When you write a reusable function, pattern, or solve a non-trivial problem,
suggest at the end of the task whether the result would be worth saving to
~/ClaudeLibrary/. Describe what it is and where it would go. Wait for my
approval before writing anything.
```

With this in place, Claude will proactively notice when something is worth keeping and suggest it — but never act without your explicit go-ahead. You stay in control of what enters the library.

Keeping the library clean

- Review additions periodically — delete outdated or superseded snippets
- Keep LIBRARY.md accurate — an outdated index is worse than no index
- Use git to version the library: `cd ~/ClaudeLibrary && git init`
- One file per snippet — small focused files are easier for Claude to use than large mixed ones

20. Quick Reference Card

Install / Update

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

Launch

```
claude
claude --model claude-sonnet-4-6
claude -p "fix the build error"
claude --add-dir ../other-project
```

Inside a session — Slash Commands

Command	Action
/help	List all commands
/init	Create CLAUDE.md
/clear	Clear history
/compact	Summarize history
/cost	Token usage & cost
/model	Switch model
/config	View/change settings
/memory	Edit CLAUDE.md
/exit	Exit session
/doctor	Diagnose issues
/bug	Report a bug

Keyboard Shortcuts

Key	Action
Shift+Tab	Cycle: Normal→Auto-accept→Plan
↑ / ↓	Navigate prompt history
Ctrl+C	Stop current action
Ctrl+D	Exit session
@filename	Include file in prompt
!command	Run shell command directly

Key Config Files

File	Scope
~/.claude/settings.json	Global personal
.claude/settings.json	Project (commit to git)
CLAUDE.md	Project memory
~/.claude/commands/	Personal commands
.claude/commands/	Project commands

Key env vars (in settings.json)

Variable	Effect
DISABLE_AUTOUPDATER=1	Stop auto-updates
DISABLE_TELEMETRY=1	Opt out of telemetry
CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC=1	API calls only

docs.claude.com/en/docs/claude-code/overview

Install / Update

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

Launch

```
claude
claude --model claude-sonnet-4-6
claude -p "fix the build error"
claude --add-dir ../other-project
```

Slash Commands

Command	Action
/help	List all commands
/init	Create CLAUDE.md
/clear	Clear history
/compact	Summarize history
/cost	Token usage & cost
/model	Switch model
/config	View/change settings
/memory	Edit CLAUDE.md
/exit	Exit session
/doctor	Diagnose issues
/bug	Report a bug

Keyboard Shortcuts

Key	Action
Shift+Tab	Cycle: Normal → Auto-accept → Plan
↑ / ↓	Navigate prompt history
Ctrl+C	Stop current action
Ctrl+D	Exit session
@filename	Include file in prompt
!command	Run shell command directly

Key Config Files

File	Scope
~/.claude/settings.json	Global personal
.claude/settings.json	Project (commit to git)
CLAUDE.md	Project memory
~/.claude/commands/	Personal commands
.claude/commands/	Project commands

Environment Variables

Variable	Effect
DISABLE_AUTOUPDATER=1	Stop auto-updates
DISABLE_TELEMETRY=1	Opt out of telemetry
CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC=1	API calls only

Session Modes (Shift+Tab to cycle)

Mode	Behaviour
Normal	Claude asks before every edit/command
Auto-accept	Claude acts without asking (faster)
Plan	Claude describes plan before acting

Print-friendly version. Page 19 has the same content in dark theme for on-screen use.