AI · CLAUDE TOOLS

Working with Claude on Projects

From one-off conversations to persistent, multi-session workflows: what a Claude Project actually is, the memory system, CLAUDE.md, shorthand prompt systems, multi-session continuity, slash commands and skills, MCP servers, and a capstone designing a project from scratch.

8 Chapters | Philip Osztromok | Generated with Claude

CHAPTER 1: PROJECTS VS CONVERSATIONS

Working with Claude on Projects

Chapter 1 · Projects vs Conversations — Persistent Context and What Claude Remembers

There's a fundamental difference between chatting with Claude and working with Claude. A chat is a one-off exchange. Working with Claude — on a codebase, a course, a research project, a business workflow — benefits from something more structured: a **Project**. This chapter explains what Projects are, what they change, and what Claude actually remembers across sessions (and what it doesn't).

Conversations vs. Projects — The Core Difference

A CONVERSATION

- Exists for the duration of one session
- No memory carries over when you start a new chat
- You re-explain context every time: your stack, your goals, your preferences
- Claude treats you as a stranger at the start of each new message thread
- Great for one-off questions, quick lookups, exploratory chats

A PROJECT

- A persistent workspace that spans many sessions over days, weeks, or months
- Has a **system prompt** (instructions) that Claude reads at the start of every session
- Supports a **memory system** — files that Claude can read and update across sessions
- Can include uploaded files, documents, and code as persistent context
- Claude behaves like a long-term collaborator who knows your project

The practical effect: in a Project, you invest in context once and it compounds. Every session starts from a shared foundation rather than a blank slate.

What Claude Actually Remembers — and What It Doesn't

This is the most important thing to understand about working with Claude long-term. The word "memory" is used loosely — the reality is more precise and more useful once you understand it.

System prompt / Project instructions

Written once, loaded into every session automatically. This is where you put permanent rules, context, and preferences. Claude reads this before your first message every time.

PERSISTS ALWAYS

Memory files (MEMORY.md + individual files)

Markdown files Claude writes and reads. Store facts about you, your preferences, project decisions, and reference pointers. Claude reads these at the start of sessions and updates them when asked or when it learns something important.

PERSISTS ALWAYS

Uploaded project files

Documents, PDFs, code files you attach to the Project. Available to Claude across sessions. Good for reference material, schemas, style guides, existing code.

PERSISTS ALWAYS

Current session conversation

Everything said in the current session — Claude reads the whole thread on every message. But this resets when you start a new session. It is not automatically saved to memory.

THIS SESSION ONLY

Implicit knowledge / "things Claude learned"

Claude does not learn from conversations in the way a human does. If you tell Claude something important in a session and it isn't written to a memory file, it's gone next session. There is no background learning.

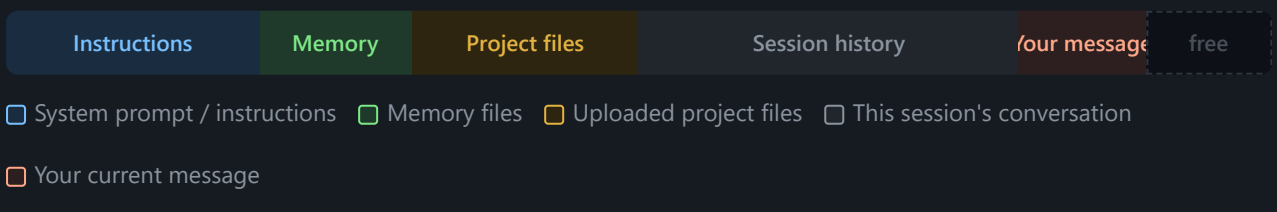
DOES NOT PERSIST

THE MOST COMMON MISCONCEPTION

"I told Claude this last week." If it wasn't written to a memory file or the system prompt, Claude has no record of it. The conversation happened — but Claude's next session starts fresh. This is why the memory system matters: it's explicit, not magical.

What's in Claude's Context Window During a Project Session

Every time you send a message in a Project, Claude reads all of this before generating a response:



☐ System prompt / instructions ☐ Memory files ☐ Uploaded project files ☐ This session's conversation

☐ Your current message

The total context window is 200,000 tokens — enormous for most projects. The key insight is that the persistent layers (instructions, memory, files) are loaded fresh each session, so they don't accumulate session-by-session. Only the conversation history grows within a session, and it resets when you start a new one.

What Persists Between Sessions — Full Table

What	Persists?	How
Project instructions (system prompt)	Always	Written in Project settings; loaded automatically every session
Memory files (user, feedback, project, reference)	Always	Written by Claude or you; read by Claude at session start
Uploaded documents and files	Always	Attached to the Project; available every session
Preferences you stated in a conversation	Never	Unless Claude wrote them to a memory file — tell Claude to remember them
Decisions made mid-session	Never	Ask Claude to save important decisions to the project memory file
Code Claude wrote in a session	Only if saved	Code in the filesystem persists; code only in the chat does not
The conversation thread itself	Viewable, not loaded	You can scroll back and read it; Claude does not re-read past sessions
Claude's "learning" from interactions	Never	Claude does not update its model weights from conversations

When to Use a Project vs. a Conversation

USE A PROJECT

Ongoing development work

Building an app, writing a course, maintaining a server. You'll have many sessions and context

USE A PROJECT

You have strong preferences

Editor shortcuts, code style, preferred tools, tone for writing. Write these into memory once and

accumulates — a Project makes every session smarter than the last.

never re-explain them again.

USE A PROJECT

Reference material you use repeatedly

A database schema, API docs, a style guide, a project spec. Upload once to the Project; Claude can read it any session.

USE A PROJECT

You want consistent behaviour across sessions

Same response style, same assumptions, same constraints. The system prompt enforces this automatically — you don't have to re-state it.

USE A CONVERSATION

One-off questions

"What does this error mean?" "How do I do X in Python?" No context needed across sessions — a conversation is lighter and faster.

USE A CONVERSATION

You want a fresh, unbiased perspective

If you want Claude without any context shaping its answer — no prior decisions, no project framing — a new conversation gives you a clean slate.

The Mental Model: Claude as a Long-Term Collaborator

The most useful way to think about a Project is as working with a colleague who reads a briefing document before every meeting. That briefing contains everything they need to know about you and the project — but they don't remember the actual conversation from last week unless someone took notes.

Your job as the project owner is:

- **Write the briefing** — the system prompt: who you are, what the project is, how you like to work
- **Keep the notes** — the memory files: decisions made, preferences discovered, important facts
- **Maintain the filing cabinet** — uploaded files: reference material Claude should always have access to

When those three layers are well-maintained, every new session feels like continuing a conversation — because Claude walks in already knowing the context.

IN THIS PROJECT — RIGHT NOW

This course is itself running inside a well-configured Project. The system prompt tells Claude about the website, the file structure, and the course format. Memory files store your learning profiles, course outlines, and preferences (like always using vi, never nano). Uploaded files aren't used here, but the other two layers are — and you've seen the results across dozens of sessions.

Next — Chapter 2: The Memory System

A deep dive into the four memory types — user, feedback, project, and reference — how they work, when to create each one, and how to write memory entries that are actually useful across sessions rather than just archives of what happened.

CHAPTER 2: THE MEMORY SYSTEM

Working with Claude on Projects

Chapter 2 · The Memory System — User, Feedback, Project, and Reference Memory

The memory system is what turns a Project from a convenience into a genuine long-term collaborator. Without it, every session starts from scratch. With it, Claude walks into each session already knowing who you are, how you like to work, what decisions have been made, and where to find reference information. This chapter covers the four memory types, how to write entries that actually work, and what not to put in memory.

How the Memory System Works

Memory is stored as plain markdown files on disk — there is nothing magical or opaque about it. Each file has a YAML frontmatter block that identifies it, and a body with the actual content. An index file called `MEMORY.md` lists all memory files with one-line summaries so Claude can quickly decide which ones are relevant to load.

Claude reads `MEMORY.md` at the start of every session. When a memory file seems relevant to the current task, Claude reads that file too. When Claude learns something worth saving — or when you explicitly ask it to remember something — it writes or updates the appropriate file and adds a line to `MEMORY.md`.

> THE KEY PRINCIPLE
>
> Memory is **explicit and file-based**. If it's not written to a file, it doesn't persist. If it's written to the wrong type of file or with a vague description, Claude may not load it when it's relevant. The quality of your memory system directly determines how well Claude behaves across sessions.

The Four Memory Types

User Memory

Save when: you learn about the person's role, background, goals, or working style

User memories tell Claude who it's working with. They shape how Claude explains things, what it assumes you already know, and what level of detail is appropriate. A good user memory makes every response feel like it was written for you specifically — because it was.

Save facts about role, expertise level (especially where it varies by domain), goals, learning style, and anything that should change how Claude frames its answers. Avoid saving negative observations or anything you wouldn't want a colleague to write about you.

EXAMPLE — USER MEMORY FILE

```
---
name: user-profile
description: Philip's background, role, and working style
metadata:
  type: user
---

# Philip Osztromok

Senior developer and systems administrator. Deep Linux/networking
experience — explain at peer level, no hand-holding on fundamentals.

New to Python (coming from Java/JS) — frame Python concepts in terms
of Java equivalents where helpful. Current goal: FastAPI web app.

Prefers vi/vim for terminal editing. Strongly dislikes nano.
Responds well to direct answers; dislikes excessive hedging.
```

Notice what this does: Claude now knows not to explain what SSH is, but *should* explain Python's approach to classes if it comes up. It knows to suggest vim, never nano. And it knows Philip wants directness. None of this needs to be re-stated in every session.

Feedback Memory

Save when: Claude is corrected OR when a non-obvious approach is confirmed as right

Feedback memories are behavioural rules that come from actual experience working together. They prevent Claude from making the same mistake twice — and equally importantly, they lock in approaches that worked so Claude doesn't drift away from them.

The structure matters here: lead with the rule, then add a **Why:** line and a **How to apply:** line. Knowing the reason behind a rule lets Claude apply it intelligently in edge cases rather than blindly following it

when it doesn't apply.

EXAMPLE — FEEDBACK MEMORY FILE

```
---
name: feedback-course-pdf
description: Always generate and run the PDF build script with the final chapter
metadata:
  type: feedback
---

Always generate and run the PDF build script in the same response
as the final chapter of any course. Do not wait for the user to ask.

**Why:** User confirmed this is always wanted — having to prompt for
the PDF after the final chapter is unnecessary friction.

**How to apply:** After writing the final lesson HTML, immediately
write the PDF build script and run it in the same response.
```

Two triggers for saving feedback: **corrections** ("don't do X") and **confirmations** ("yes, exactly that"). Corrections are easy to notice. Confirmations are quieter — watch for the user accepting an unusual choice without pushback, or saying "perfect, keep doing that."

Project Memory

Save when: you learn what's being built, why, by when, or what decisions have been made

Project memories capture the ongoing state of work — not the code itself, but the context and motivation behind it. What are we building? Why this approach and not another? What's the deadline? What was decided and what was rejected?

These memories decay — a decision made in January may be irrelevant by March. The **Why:** line is especially important here: it lets Claude (and you) judge whether the memory is still load-bearing when the project evolves.

EXAMPLE — PROJECT MEMORY FILE

```
---
name: website-rebuild-decision
description: Chose FastAPI+Jinja2 over Next.js for site rebuild
metadata:
  type: project
```

```
---

Using FastAPI + Jinja2 for the osztromok.com rebuild, not Next.js.

**Why:** Philip is learning Python and wants the rebuild to be a
learning project. Next.js would require JS expertise he's building
separately. FastAPI aligns with the meal planner app goal.

**How to apply:** All rebuild suggestions should assume FastAPI/Python
backend, not Node/React. Don't suggest Next.js alternatives.
```

Always convert relative dates to absolute dates when saving project memories ("Thursday" → "2026-06-19"). A memory that says "the deadline is next Thursday" is meaningless a month later.

Reference Memory

Save when: you learn where to find information in external systems

Reference memories are pointers — they tell Claude where to look, not what's there. Bug tracker, monitoring dashboard, API docs, a specific Slack channel, the server IP for a dev environment. Claude can't visit URLs on its own, but knowing they exist helps it direct you to the right place.

EXAMPLE — REFERENCE MEMORY FILE

```
---
name: server-reference
description: Server details for osztromok.com — IP, stack, config locations
metadata:
  type: reference
---

# osztromok.com Server

- **Host:** VPS, Debian 12 Bookworm
- **Web server:** Apache 2.4
- **DB:** MySQL 8, accessible via localhost only
- **Config:** Apache vhosts in /etc/apache2/sites-available/
- **Web root:** /var/www/osztromok.com/public/
- **Course files:** /var/www/osztromok.com/public/claude-generated-files/
```

Reference memories are the most stable — server configs and tool locations don't change often. They're also the most actionable: when Claude knows where things live, its suggestions can be

immediately specific rather than generic.

MEMORY.md — The Index

`MEMORY.md` is loaded every single session. It's an index, not a memory store — each entry is one line, pointing to a file with a hook that tells Claude when to read it. Keep it concise; lines after 200 will be truncated.

MEMORY.MD STRUCTURE

```
# Memory Index

- [User Profile](user_profile.md) — Senior dev, Linux/networking expert, learning Pytho
- [Editor Preference](feedback_editor.md) — Always suggest vim, never nano
- [Course PDF Rule](feedback_course_pdf.md) — Generate PDF automatically with final cha
- [Website Rebuild](website_rebuild_decision.md) — Using FastAPI+Jinja2, not Next.js
- [Server Reference](server_reference.md) — osztromok.com VPS details, Apache config pa
```

The one-line hook is what Claude uses to decide whether to load the full file. Make it specific enough to be useful — "user info" is not a useful hook; "Senior dev, Linux expert, learning Python for FastAPI" is.

What NOT to Save in Memory

Code patterns and architecture

These live in the codebase. Read the files — don't duplicate them in memory where they'll go stale.

Git history and recent changes

`git log` is authoritative. A memory summarising recent commits is outdated the moment the next commit lands.

Debugging solutions or fix recipes

The fix is in the code. The context belongs in the commit message. Memory is for enduring patterns, not one-time solutions.

In-progress task details

What you're working on right now is ephemeral. Use a task list or notes file — not the

Things already in CLAUDE.md

No duplication. If a rule is in the project instructions file, don't also put it in memory —

Activity logs and summaries

A log of what sessions covered is an archive, not a memory. Ask yourself: what's *surprising*

memory system — for current work state.

it creates conflicts when one is updated and the other isn't.

or non-obvious here? That's the part worth saving.

How to Trigger Memory Saves

1. **Say "remember this"**

   The most direct trigger. Claude will save it to the appropriate memory file and update MEMORY.md immediately.

2. **Correct Claude's behaviour**

   "Don't do X" or "I prefer Y" — Claude recognises corrections and saves them as feedback memories proactively.

3. **Confirm a non-obvious approach**

   "Yes, exactly" or accepting an unusual choice signals Claude that the approach was right — it saves this as a positive feedback memory.

4. **Provide project context explicitly**

   When you explain why a decision was made or share a constraint, Claude saves this as a project memory — especially if you frame it as background context.

5. **Periodically consolidate**

   Run `/consolidate-memory` to have Claude review all memory files, merge duplicates, remove stale entries, and sharpen descriptions. Memory quality degrades without maintenance.

VERIFY BEFORE ACTING ON MEMORY

Memory files can go stale. If a memory names a specific file path, function, or tool — Claude verifies it exists before recommending it. If you're about to act on something Claude recalled from memory, confirm the current state first. "The memory says X exists" is not the same as "X exists now."

Next — Chapter 3: CLAUDE.md

The project instructions file — how to write a CLAUDE.md that makes Claude behave consistently

across every session without you having to restate preferences, rules, or context. Covers structure, what to include, and how it differs from memory files.

CHAPTER 3: CLAUDE.MD FILES

Working with Claude on Projects

Chapter 3 · CLAUDE.md — Writing Project Instructions That Persist Across Every Session

`CLAUDE.md` is the project instructions file — a markdown document that Claude reads at the start of every session in your project. It's the place for standing rules, permanent context, and behavioural preferences that should never have to be restated. Written well, it makes every session feel immediately productive. Written poorly, it's ignored noise. This chapter covers what goes in it, how to structure it, and how it differs from the memory system.

What CLAUDE.md Is For

Think of `CLAUDE.md` as the briefing document Claude reads before walking into the room. It answers the questions that would otherwise have to be re-explained every session:

- **What is this project?** — the purpose, the tech stack, the goal
- **Who is the user?** — role, expertise, how they like to work
- **What are the standing rules?** — code style, tool preferences, what to avoid
- **What should Claude always do?** — automatic behaviours, output formats, naming conventions
- **What is the structure?** — file locations, key paths, important conventions

It is *not* a task list, a session log, or a place for things that change frequently. Those belong in memory files or external tools.

> LOADED EVERY SESSION, IN FULL
>
> Unlike memory files (which are loaded selectively based on relevance), `CLAUDE.md` is always loaded completely. Keep it focused — every line should earn its place by shaping Claude's behaviour. Verbose instructions dilute the signal.

Anatomy of a Good CLAUDE.md

SECTION 1

Project overview

One paragraph: what this project is, what it produces, who it's for. Gives Claude the context to make good judgement calls without being told everything explicitly.

SECTION 2

Tech stack & environment

Language, framework, OS, key tools, server details. Prevents Claude suggesting the wrong language, library, or platform.

SECTION 3

File structure

Where things live. Key paths, naming conventions, output locations. Means Claude can write to the right place and reference the right files without asking.

SECTION 4

Coding standards

Style rules, formatting preferences, linting conventions. What to always include (type hints, docstrings) and what to never do (e.g. use a specific anti-pattern).

SECTION 5

Behavioural rules

How Claude should communicate, what it should always or never do, how to handle ambiguity in this project's context. The standing preferences.

SECTION 6

Reference pointers

Where to find the memory index, key documents, external resources. Short — just enough to orient Claude to the rest of the project's knowledge base.

A Complete CLAUDE.md Example

CLAUDE.MD — ANNOTATED EXAMPLE FOR A WEB PROJECT

```
# Project: osztromok.com Learning Website

## Overview
A personal learning website at osztromok.com. Philip uses Claude to generate
HTML course chapters, SQL inserts, and PDF exports. The site hosts self-study
courses covering Linux, networking, programming, and languages.

## Tech Stack
- Server: Debian 12, Apache 2.4, PHP 8.2, MySQL 8
- Course pages: fragment HTML (no DOCTYPE/html/head/body tags)
- Dark theme: background #0d1117, surface #161b22
- PDFs: Chrome headless + pypdf merge

## File Structure
- Course HTML:  claude-generated-files/
- SQL inserts:  claude-generated-files/insert_*.sql
- PDF output:   claude-generated-files/*.pdf
- Memory:       C:\Users\emuba\.claude\projects\ ... \memory\
```

```
## Coding Standards
- HTML: fragment only — no DOCTYPE, html, head, or body tags
- SQL: always SET NAMES utf8mb4; use CONVERT(UNHEX(...) USING utf8mb4) for bodies
- Python: type hints required; no third-party deps beyond pypdf/requests
- Shell: always suggest vim for terminal editing — never nano

## Behavioural Rules
- Generate PDFs automatically with the final chapter of every course
- Always ask for subtopic_id and page_id before generating SQL
- Course shorthand prompts: respond without asking for clarification
- Be direct — no hedging, no excessive caveats, no closing summaries
- When generating code: write it and run it; don't just show it

## Memory Index
See memory/MEMORY.md for the full index of user preferences, feedback,
project decisions, and course outlines.
```

Writing Instructions That Actually Work

Be specific and actionable

Vague instructions ("be helpful", "write good code") are ignored because they don't change Claude's behaviour — they match what Claude would do anyway. Specific instructions ("use type hints on all function signatures", "never suggest nano") change actual output.

TOO VAGUE

```
Write clean, well-structured code.
Be professional in responses.
Follow best practices.
```

These instruct nothing — Claude already tries to do all of these by default.

SPECIFIC AND ACTIONABLE

```
Use type hints on all functions.
Maximum line length: 100 characters.
No f-strings — use .format() for
compatibility.
Never suggest pip install — use
requirements.txt.
```

Each line changes something concrete. Claude will follow these because they're unambiguous.

Use imperative form

Instructions written as directives are more reliable than instructions written as preferences. `Always generate the PDF in the same response as the final chapter` is more effective

than `It would be great if the PDF was generated automatically`.

Keep it current

A stale instruction is worse than no instruction — it actively misleads Claude. Review `CLAUDE.md` when the project changes significantly. Remove rules that no longer apply. Update paths and tool names when they change.

Don't duplicate memory

If a preference is already in a memory file, don't also put it in `CLAUDE.md`. Duplication creates conflicts when one is updated and the other isn't. Use `CLAUDE.md` for standing rules that never change session-to-session; use memory files for facts that evolve.

CLAUDE.md vs. Memory Files — When to Use Each

What kind of information	CLAUDE.md	Memory file
Project tech stack and file structure	✅ Always here	—
Standing code style rules	✅ Always here	—
Automatic behaviours ("always do X")	✅ Always here	—
User background and expertise level	Brief summary fine	✅ Full detail here
Feedback from past sessions	—	✅ Always here
Project decisions and rationale	—	✅ Always here
Course outlines and status	—	✅ Always here
Things that change frequently	❌ Don't put here	✅ Memory is easier to update
External resource locations	Brief pointer fine	✅ Full details here

CLAUDE.md for Different Project Types

Software project

Emphasise: language and framework, test runner and how to run tests, linting config, branch strategy, how to run the app locally, where the API docs are. The goal is for Claude to operate like a dev who has already read the onboarding docs.

Writing or content project

Emphasise: audience, tone, style guide rules (Oxford comma, US/UK English, heading capitalisation), output format (markdown, HTML, plain text), where drafts are stored, what "done" looks like for this project.

Research or learning project

Emphasise: what's already been covered (or point to memory files), the learning goals, preferred explanation depth, any domain-specific vocabulary conventions, where notes and outputs are stored.

Personal assistant / admin project

Emphasise: recurring tasks and their format, external accounts and tools in use, decision-making preferences, communication style, time zone and locale, preferred date formats.

How Long Should It Be?

Long enough to cover everything Claude needs; short enough that every line matters. In practice, a well-written `CLAUDE.md` for most projects fits in 50–150 lines. The test: if you could remove a line without changing Claude's behaviour, remove it.

START SMALL AND GROW IT

Don't try to write a complete CLAUDE.md from day one. Start with the project overview and tech stack. Add rules as you discover Claude needs them — when it makes the same wrong assumption twice, that's a CLAUDE.md entry. When you find yourself repeating the same instruction, that's a CLAUDE.md entry. Let it grow from actual experience rather than speculation.

INSTRUCTIONS COMPETE FOR ATTENTION

If CLAUDE.md is very long, later instructions are less reliably followed than earlier ones (the "lost in the middle" effect from Chapter 1 of the Prompt Engineering course applies

here too). Put your most important rules near the top. Prune ruthlessly.

Next — Chapter 4: Shorthand Prompt Systems

Designing your own prompt maps for recurring tasks — how to create shorthand triggers that reliably produce consistent, high-quality output without repeating context every time. The system powering every course chapter in this project.

CHAPTER 4: SHORTHAND PROMPT SYSTEMS

Working with Claude on Projects

Chapter 4 · Shorthand Prompt Systems — Designing Your Own Prompt Maps

A shorthand prompt system turns a complex, context-heavy task into a single short trigger. Instead of typing a multi-paragraph prompt every time you want a course chapter, you type `vpn3` — and Claude knows exactly what to produce, in what format, to what standard. This chapter explains how to design one that actually works, using the system powering every chapter in this course as a live example.

What a Shorthand System Is

A shorthand prompt system is a mapping from a short trigger (a word, a code, a number) to a complete, detailed task specification that lives in your project's memory. When you send the trigger, Claude looks up the specification in memory and executes it — without you having to repeat any context.

It has three layers:

THE TRIGGER	What you type — short, memorable, consistent. `vpn3`, `Bash I 7`, `rdesk4`, `pi2`. No context, no explanation — just the code.
THE MEMORY FILE	A memory file Claude loads when it sees the trigger — containing the full specification: what to produce, the format, styling, file naming, output location, and any automatic follow-up steps.
THE OUTPUT	A complete, correctly formatted deliverable — HTML chapter, SQL script, PDF, or whatever the task produces — generated without clarifying questions.

The key insight: **you invest in the specification once**, in the memory file. Every subsequent use costs you only the trigger. The longer a course runs, the more value the system pays back.

A Real Example — This Project's Course System

Here's the actual prompt map for the VPN course:

`vpn1`	What is a VPN and why — use cases, what a VPN does and doesn't protect	`vpn_lesson_01.html`
`vpn2`	VPN protocols compared — OpenVPN, WireGuard, IKEv2/IPsec, L2TP	`vpn_lesson_02.html`
`vpn3`	WireGuard server — install on Linux, generate keys, wg0.conf, systemd	`vpn_lesson_03.html`
⋮	⋮	
`vpn8`	Troubleshooting & hardening — DNS leaks, kill switches, MTU, logging	`vpn_lesson_08.html`

Each trigger expands into a full chapter — correct styling, correct file name, correct format, next-chapter teaser at the end — because the memory file holds the complete specification for what a "VPN chapter" means. Claude doesn't guess: it follows the spec.

Designing Your Own System — Step by Step

1. **Identify the recurring task**

   What do you produce repeatedly where each instance follows the same pattern? Course chapters, weekly reports, meeting notes, changelog entries, commit message templates, API endpoint stubs. If you've done it more than three times in the same format, it's a candidate.

2. **Define the full specification**

   Write down everything Claude would need to produce the output without asking you anything: the format, the structure, the styling, the file naming convention, the output location, any automatic follow-up steps, and what varies between instances (usually just the topic or number).

3. **Choose a trigger scheme**

   Pick short, memorable, collision-free triggers. `prefix + number` works well for sequences (`vpn1` – `vpn9`). `prefix + keyword` works for non-sequential tasks (`report weekly` , `standup`). Avoid triggers that could be confused with normal words.

4. **Write the memory file**

   Create a project memory file that contains the complete specification, the trigger table, and any global rules (styling, format, automatic behaviours). Use the standard frontmatter format so Claude can find it.

5. **Add to MEMORY.md**

Add a one-line entry to `MEMORY.md` with a specific hook so Claude knows when to load it: "*VPN course outline and shorthand prompt map (vpn1–vpn9)*" — specific enough that Claude loads it when it sees `vpn` triggers.

6 **Test and refine**

Run the first trigger. If the output misses something, update the memory file — not just the current prompt. The fix should be in the spec, so every future instance benefits. Build in the feedback from the first few uses before relying on it heavily.

What Goes in the Specification Memory File

EXAMPLE — COURSE CHAPTER MEMORY FILE (EXCERPT)

```
---
name: vpn-course
description: VPN course outline, shorthand prompt map (vpn1–vpn8), styling spec
metadata:
  type: project
---

# VPN Course

## Styling
- Fragment HTML (no DOCTYPE/html/head/body)
- Background: #0d1117 / surface: #161b22
- Accent: #22c55e (green), h3 color: #86efac
- Class: vpn-lesson

## File naming
vpn_lesson_NN.html  (e.g. vpn_lesson_03.html)

## Automatic behaviours
- End every chapter with a next-chapter teaser box
- Generate PDF automatically with the final chapter
- Ask for subtopic_id and page_id before generating SQL

## Prompt map
| Trigger | Topic                                                | Status |
├─────────┼──────────────────────────────────────────────────┼────────┤
| vpn1    | What is a VPN and why                            | ✅     |
| vpn2    | VPN protocols compared                           | ✅     |
| vpn3    | WireGuard server setup                           | ✅     |
| vpn4    | WireGuard clients                                | ✅     |
```

```
| vpn5    | OpenVPN server setup                              | ✅     |
| vpn6    | OpenVPN clients                                   | ✅     |
| vpn7    | Commercial VPN clients                            | ✅     |
| vpn8    | Troubleshooting and hardening (final chapter)     | ✅     |
```

Notice what's in the spec: styling rules, file naming, automatic behaviours, and the full topic list with status tracking. Claude reads all of this from the single memory file trigger — you typed two characters to get a chapter that follows all of it.

Trigger Design Patterns

Sequential series

```
vpn1, vpn2 … vpn8 bash1, bash2 … bash12 proj1, proj2 … proj8
```

Best for courses, tutorials, numbered reports. The number tells Claude exactly which item in the sequence to produce.

Prefix + keyword

```
report weekly report monthly standup retro
```

Best for recurring documents that aren't numbered. The keyword selects the variant; the prefix routes to the right memory file.

Multi-level prefix

```
Bash I 1 … Bash I 12 Bash A 1 … Bash A 12 CSS 1-1 … CSS 12-4
```

Best when you have families of related sequences. The first prefix identifies the family; the second identifies the variant within it.

Single-word actions

```
progress diary consolidate
```

Best for meta-tasks about the project itself — status reports, diary generation, memory maintenance. Each maps to a full procedure in memory.

Common Problems and Fixes

Problem	Cause	Fix
Claude asks for clarification on a trigger	The memory file description isn't specific enough for Claude to load it confidently, or MEMORY.md hook doesn't mention the trigger prefix	Update the MEMORY.md entry to include the trigger prefix explicitly: *"VPN course (vpn1–vpn8)"*

Problem	Cause	Fix
Output drifts from spec over a long course	The styling or format spec in memory is ambiguous — Claude interpolates slightly differently each chapter	Add a concrete example to the spec (e.g. paste the exact CSS) rather than describing it in words
Trigger collides with a normal word	Claude treats the trigger as a regular word and responds to it literally	Choose more distinctive triggers — codes rather than words. `pi2` not `pi`, `rdesk3` not `rdp`
Automatic follow-up steps missed	The automatic behaviour rule is buried in a long memory file and doesn't carry enough weight	Move automatic rules to CLAUDE.md where they're loaded every session, not just when the course memory file is loaded
Status tracking goes out of date	The ✅/⬜ status table isn't updated after each chapter is generated	Include updating the status table as part of the post-chapter procedure — Claude should update it alongside the memory file after each chapter

Beyond Courses — Other Uses for Shorthand Systems

The same pattern works for any recurring task with a consistent format:

- **Weekly status reports** — `report` expands to: pull from last week's notes, format as Markdown with sections for Done / In Progress / Blocked / Next week, save to reports/YYYY-WW.md
- **API endpoint stubs** — `endpoint GET /users` expands to: FastAPI route with type hints, Pydantic schema, docstring, and test stub in the right file location
- **Language lesson generation** — `jp12` expands to: Japanese lesson 12, specific vocabulary format, dialog structure, hiragana annotations, saved to the right directory
- **Code review checklists** — `review security` expands to: a security-focused checklist applied to the current diff, with specific items for OWASP top 10
- **Commit message drafts** — `commit` expands to: run git diff, summarise changes, produce a conventional commit message in the project's style

> THE COMPOUNDING RETURN

A shorthand system gets more valuable the longer you use it. Chapter 1 of a course benefits from the spec. Chapter 10 benefits from the spec *plus* nine chapters of refinement you've baked back in. By the end of a long course, you type four characters and get a perfectly formatted, consistently styled chapter every time. That's the payoff.

Next — Chapter 5: Multi-Session Workflows

Picking up where you left off, managing context across many sessions, understanding how context summarisation works, and strategies for keeping long-running projects coherent over weeks and months.

CHAPTER 5: MULTI-SESSION WORKFLOWS

Working with Claude on Projects

Chapter 5 · Multi-Session Workflows — Picking Up Where You Left Off

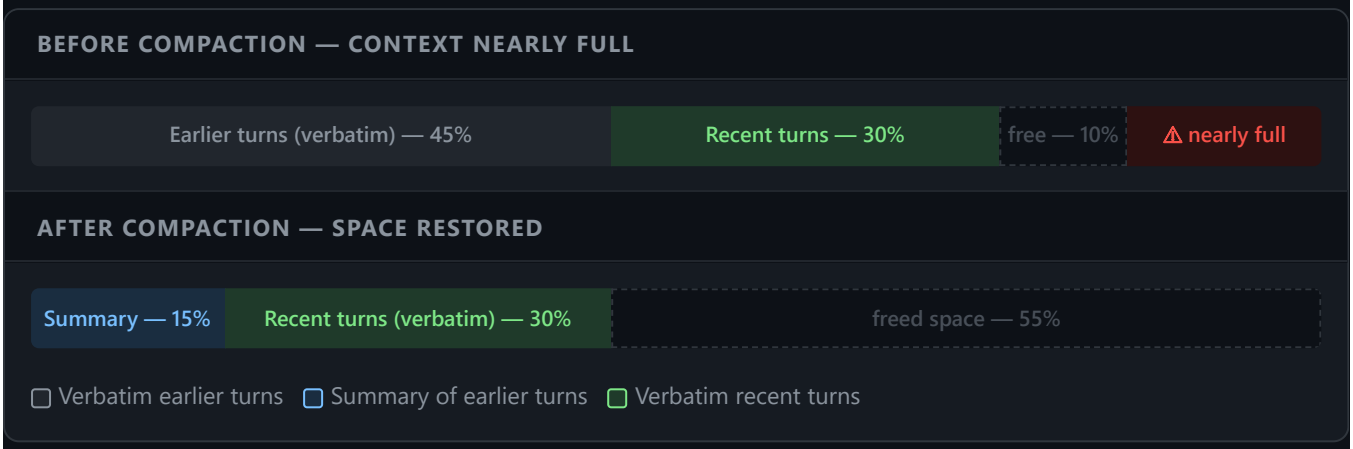
A single session with Claude can be productive. A well-managed project that spans dozens of sessions can be transformational — but only if each session starts from a shared foundation and ends cleanly. This chapter covers how sessions work, how context compaction affects long conversations, and the practical patterns for keeping a long-running project coherent over weeks and months.

The Lifecycle of a Session

The gap between sessions is a hard reset of the conversation. The only things that survive are what's written to files — memory, CLAUDE.md, and anything Claude generated on disk. Everything else is gone. Managing that transition is what multi-session workflow design is about.

Context Compaction — What Happens When a Session Gets Long

Within a single session, Claude's context window fills as the conversation grows. At approximately 80% capacity, Claude Code automatically compacts older portions of the conversation — replacing them with a summary — to free space for new turns. This happens in the background; you may see a note that compaction occurred.



The summary is accurate but lossy — it captures decisions and outcomes, not exact wording. What this means in practice:

- **General decisions and outcomes survive well** — "we chose PostgreSQL over MySQL" will be in the summary
- **Exact code snippets from early in the session may be paraphrased** — the logic is captured, not the literal text
- **Nuanced constraints may soften** — "never use nano, always vim" may become "prefers vim" after compaction
- **Important specifics should go to memory files** — if something needs to survive compaction precisely, write it down before the session gets long

> **DON'T RELY ON COMPACTION FOR PERSISTENCE**
>
> Compaction summaries exist only within the current session — they are not saved anywhere after the session ends. If the session closes, the summary is gone along with the conversation. Only memory files and files on disk persist.

Starting a Session Well

The first message of a new session sets the context. Claude has loaded your instructions and memory files, but has no knowledge of what happened last session. A good session opener

orients Claude immediately:

- → **State where you are.** "We're on chapter 6 of the VPN course" or "Continuing the FastAPI meal planner — last session we finished the recipe schema." Claude uses this to load the right memory files and calibrate its responses.
- → **State what's changed since last session.** If you made decisions or did work outside of Claude between sessions, say so. Claude can't know what happened in the gap.
- → **Name the first task.** Don't make Claude guess. "Let's do vpn5 today" is faster than "what should we work on?"
- → **Flag any context drift.** If you want Claude to check its memory for something ("do you remember what subtopic_id we used for the VPN course?"), ask explicitly — Claude won't proactively surface memory it thinks you already know.

Ending a Session Well

The end of a session is the highest-risk moment for information loss. Before closing:

Save anything important that isn't already in files

If you made a decision mid-session ("we're going to use Redis for the session store") that isn't reflected anywhere on disk, ask Claude to save it: "Remember that we decided to use Redis for session storage — update the project memory file." If Claude generated code that exists only in the chat, confirm it's been written to disk.

Update status tracking

If you use a prompt map with a status table (✅/⬜), make sure it's been updated to reflect what was completed this session. This prevents confusion about where you are when you resume.

Note the logical next step

Ask Claude to write a one-line note to the project memory file: "Next session: vpn6 — OpenVPN clients." It takes five seconds and means the next session opener is obvious.

> THE SESSION DIARY

For longer or more complex sessions, generating a session diary — a structured summary of what was done, decided, and generated — gives you a readable record and a reliable handoff document for the next session. This is especially useful if you use multiple AI tools or collaborators. The diary trigger in this project: ask Claude to "generate a diary page for this session."

Handoff Patterns — Resuming After a Break

Status check

```
Where are we with the VPN course? Check your memory and tell me what's done and what's next.
```

Forces Claude to read the course memory file and give you an accurate status. Use when you've been away and aren't sure what was completed.

Direct resume

```
Continuing from last session. We completed vpn5 and the next chapter is vpn6.
```

When you know exactly where you are. The most efficient opener — no status check needed, gets straight to work.

Decision recall

```
Do you have any notes on what subtopic_id we used for the Remote Desktop course?
```

Asks Claude to surface a specific fact from memory. More reliable than trying to remember it yourself — if it was saved, Claude has it.

Context re-brief

```
Quick re-brief before we start: [2–3 sentences on current project state]. Now let's do [task].
```

Use when the project has evolved significantly since the last session and memory files may be slightly behind. The re-brief overrides stale memory for this session.

Gap update

```
Since we last worked together I've [done X, decided Y, changed Z]. Update your memory files accordingly, then we'll continue with [task].
```

When you did work outside Claude between sessions. Brings memory up to date before the session begins rather than mid-session.

Progress report

```
progress
```

This project's shorthand for a three-section status report: completed courses, started but incomplete, proposed but not started. One word, full picture.

Keeping Long Projects Coherent

Projects that span weeks or months face specific challenges — memory files go stale, preferences evolve, earlier decisions get forgotten. A few maintenance habits prevent gradual drift:

Periodic memory consolidation

Run `/consolidate-memory` (or ask Claude to review and consolidate memory files) every few weeks on active projects. This merges duplicates, removes stale entries, updates facts that have changed, and sharpens one-line MEMORY.md hooks. A memory system that isn't maintained slowly degrades.

Spot-check before acting on memory

If Claude references a fact from memory that you'll act on — a file path, a config value, a decision — verify it against the current state before proceeding. Memory captures what was true when it was written. Things change.

Keep CLAUDE.md current

Review it when the project changes significantly. A CLAUDE.md that describes an old tech stack or references files that no longer exist actively misleads Claude. It's better to have a shorter, accurate CLAUDE.md than a comprehensive but stale one.

Don't let memory grow without pruning

MEMORY.md has a 200-line soft limit. A project that accumulates memory files without ever removing them will eventually have a bloated index where Claude can't reliably select the right files. Quality over quantity: ten focused memory files beat forty vague ones.

Common Multi-Session Problems

Problem	What causes it	Fix
Claude seems to have forgotten a preference	The preference was stated in a session but never written to a memory file	Ask Claude to save it now; add to feedback memory. Then it persists.

Problem	What causes it	Fix
Claude contradicts an earlier decision	The decision is in a memory file but the description isn't specific enough to load it	Update the MEMORY.md hook to be more specific; check the memory file still accurately reflects the decision
Progress tracking is wrong	The status table in the course memory file wasn't updated after a chapter was completed	Update the table now; add "update status table" to the automatic behaviours in CLAUDE.md
Claude references a file that doesn't exist	A memory file recorded a file path that has since moved or been renamed	Update or remove the stale reference; use `/consolidate-memory` to catch these systematically
Session starts slow — lots of clarifying questions	CLAUDE.md or memory files are too thin to establish context quickly	Invest in the CLAUDE.md project overview and tech stack sections; they load every session and eliminate the most common questions

Next — Chapter 6: Slash Commands and Skills

The built-in power tools for Claude Code projects — /code-review, /ultrareview, custom skills, and how to invoke and chain them. What each does, when to reach for it, and how to create your own skills for recurring workflows.

CHAPTER 6: SLASH COMMANDS AND SKILLS

Working with Claude on Projects

Chapter 6 · Slash Commands and Skills

Claude Code ships with a set of built-in slash commands for actions you perform often — code review, help, configuration, memory consolidation. On top of those, you can write your own **skills**: custom slash commands that live inside a project and trigger any workflow you can express in a prompt. This chapter covers both layers.

Built-in Slash Commands

Built-in commands are typed directly into the Claude Code chat input. They open panels, trigger operations, or configure behaviour — they are not sent to the model as chat messages.

`/help`

Lists all available slash commands for the current session, including any custom skills loaded from the project. Your first stop when you forget a command name.

`/code-review`

Triggers a code review of the current branch. Can be scoped to a PR number or run locally against uncommitted changes.

```
/code-review
/code-review ultra
/code-review ultra 42
```

`/ultrareview`

Deprecated alias for `/code-review ultra`. Does the same thing — launches a multi-agent cloud review of the branch. Prefer the newer form.

`/clear`

Clears the current conversation context without closing the session. Useful when you want to start a fresh exchange without losing your project setup.

`/compact`

Manually triggers context compaction — summarises older turns to free space. Normally

`/memory`

Opens the memory management panel. Lets you view, add, edit, or delete memory files interactively without switching to a file editor.

happens automatically; useful if you want to compact before the automatic threshold.

`/status`

Shows the current project status — loaded CLAUDE.md files, active memory files, token usage, and the current model.

`/fast`

Toggles Fast mode on/off. Fast mode uses Claude Opus with optimised output — it does not downgrade to a smaller model. Useful for rapid iteration when response time matters more than maximum depth.

DESKTOP APP VS TERMINAL

Some slash commands open interactive panels — `/permissions`, `/config`, `/agents`, `/hooks`, `/doctor`. These are available in an interactive `claude` terminal session. In the desktop app, use the app's own UI for model selection and settings instead.

The /code-review Command in Depth

`/code-review` is the most powerful built-in. It comes in two flavours:

Standard review

`/code-review` with no arguments reviews the uncommitted changes in the current working tree — staged and unstaged. Good for a quick review before committing.

Ultra review

`/code-review ultra` (or `/code-review ultra <PR number>`) launches a multi-agent cloud review. Multiple Claude instances work in parallel across different review angles — correctness, security, style, architecture — and produce a consolidated report.

- Requires a git repository. No GitHub remote needed for the no-arg form (bundles the local branch).
- With a PR number: `/code-review ultra 42` — reads the PR from GitHub directly.
- This is billed separately (multi-agent runs consume more tokens than a single-model review).

- You cannot launch ultra review via Bash or the API — it must be triggered from the Claude Code UI.

> **WHEN TO USE ULTRA VS STANDARD**
>
> Use standard review for routine commits and quick sanity checks. Reserve ultra for pull requests that touch security-sensitive code, a large surface area, or anything going to production. The additional cost is justified when catching an issue in review is cheaper than a production incident.

Skills — Custom Slash Commands

A **skill** is a markdown file that defines a custom slash command. When Claude loads your project, it reads any skills registered in your project settings and makes them available as `/skill-name` commands. Invoking a skill sends its prompt (with any arguments substituted) to Claude as a message — the skill is executed by the model, not the CLI.

How a skill file is structured

`.claude/skills/session-diary.md  — example skill file`

```
---
name: session-diary
description: Generate an end-of-session diary entry
---

Generate a session diary for this conversation.

Format: a structured markdown file at
claude-generated-files/claude_session_diary_{{date}}.md

Include sections:
1. Date and session summary (2–3 sentences)
2. Full session transcript (USER / ASSISTANT pairs)
3. Files generated this session
4. Decisions and preferences saved to memory

Write the file then confirm the path.
```

The `name` field becomes the command trigger: `/session-diary`. The body is the prompt sent to Claude when the command is invoked. You can include argument placeholders like `{{date}}` or `$ARGUMENTS` — Claude substitutes them from what you type after the command name.

Creating a Skill — Step by Step

1. **Create the skills directory** — skills live in `.claude/skills/` inside your project. If the directory doesn't exist, create it. Each skill is its own `.md` file.
2. **Write the skill file** — frontmatter block with `name` and `description`, then the prompt body. The name must be lowercase with hyphens (no spaces). The description appears in `/help`.
3. **Register the skill** — add an entry to `.claude/settings.json` under `"skills"`. Or, if using the desktop app, skills in `.claude/skills/` may be picked up automatically — check `/help` to confirm the skill appears.
4. **Test it** — invoke with `/skill-name` in the chat. If the skill takes arguments, test with `/skill-name some argument`. Iterate on the prompt body until the output matches your expectation.
5. **Save a memory note** — add the skill to your shorthand prompt map memory file so future sessions know it exists. A skill that isn't documented is easy to forget.

Example Skills for This Project

Progress report

```
/progress
```

Produces a three-section status report: completed courses, started-but-incomplete, and proposed-but-not-started. Equivalent to the "progress" shorthand in this project. The skill body reads the course memory files and formats the output to a consistent structure.

Session diary

```
/session-diary
```

Generates an end-of-session diary file. Loops over the conversation, producing USER/ASSISTANT pairs, a file list, and a decisions log. Saves to `claude-generated-files/claude_session_diary_YYYY-MM-DD.md` .

Course chapter

```
/chapter $ARGUMENTS
```

A parameterised skill. `/chapter vpn6` looks up the shorthand in the VPN course memory file and generates the corresponding chapter HTML. Encapsulates the entire chapter-generation workflow — styling lookup, content generation, file write, and status update — into a single command.

Memory consolidation

```
/consolidate-memory
```

Reviews all memory files for staleness, duplicates, and accuracy. Merges redundant files, updates MEMORY.md hooks, removes entries that reference files or facts that no longer exist. Run periodically on long-running projects.

Skills vs Shorthand Prompts

Both skills and shorthand prompts are ways to trigger a workflow with minimal typing. The difference is where the spec lives and how Claude receives it:

	Shorthand prompt	Skill
How triggered	Type a keyword (e.g. `vpn6`) in chat	Type `/skill-name` in chat
Spec location	Memory file (read by Claude on relevance)	Skill `.md` file (always loaded as a command)
Prompt delivery	Claude reads the memory, constructs the action	Skill body is sent directly as the prompt
Arguments	Embedded in the keyword (e.g. `vpn6` = chapter 6)	Passed after the command name; substituted via `$ARGUMENTS`

	Shorthand prompt	Skill
Reliability	Depends on Claude loading the right memory file	Prompt is always the same — deterministic trigger
Best for	FLEXIBLE workflows where context varies per invocation	REPEATABLE workflows with a fixed structure every time

In practice, shorthand prompts and skills are complementary. Use shorthand for course-chapter generation (where the chapter number and course vary) and skills for project-wide recurring actions (progress report, session diary, memory consolidation) that don't vary by invocation.

When to Build a Skill

A skill is worth creating when a workflow meets most of these criteria:

- You perform it more than a handful of times across sessions
- The prompt is long or complex enough that retyping it is error-prone
- The steps are consistent — same structure every time, possibly with a parameter
- Getting it slightly wrong has a meaningful cost (e.g. writing to the wrong file path, forgetting a SQL step)

Don't create skills for one-off tasks, tasks that vary too much to have a fixed prompt, or tasks where typing a short note to Claude is just as fast. Premature skill creation produces a `.claude/skills/` directory full of files you never use.

> **PROTOTYPE IN CHAT FIRST**
>
> Before writing a skill file, prove the prompt works by typing it manually in two or three sessions. Once you've confirmed the output is reliably what you want, extract the prompt into a skill file. Skill design after testing is much faster than skill design upfront.

Next — Chapter 7: MCP Servers
What the Model Context Protocol is, how MCP servers extend what Claude can see and do, built-

in servers (filesystem, browser, computer use), and how to connect third-party servers to your project. Practical examples from this project's own toolchain.

CHAPTER 7: MCP SERVERS

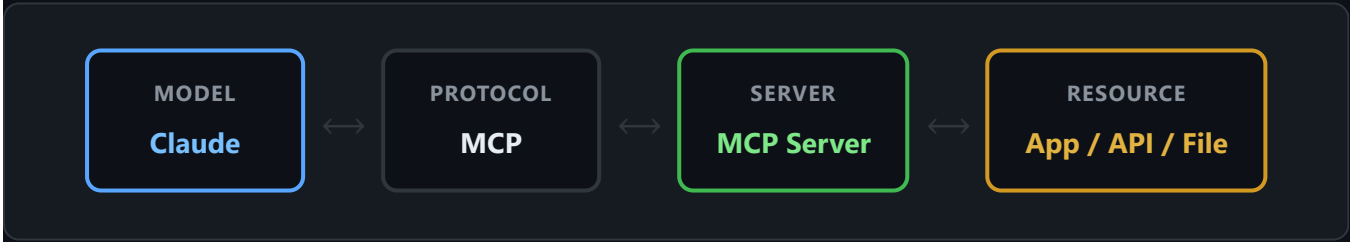
Working with Claude on Projects

Chapter 7 · MCP Servers

By default, Claude can read and write files, run shell commands, and search the web — the tools that ship with Claude Code. The **Model Context Protocol (MCP)** is a standard that lets you connect additional servers to Claude, giving it new tools: reading from a database, controlling a browser, querying an API, interacting with your desktop. This chapter explains what MCP is, how it works, the servers that matter most in practice, and how to connect them to your project.

What Is MCP?

MCP is an open protocol — think of it as a USB standard for AI tools. Any application can implement an MCP server, which exposes a set of named tools that Claude can call. Claude doesn't need to know in advance what tools a server provides; it discovers them at connection time and can use them immediately.



The flow for every MCP tool call:

1. Claude decides it needs a tool (e.g. "take a screenshot of the browser")
2. It calls the tool by name with parameters, via the MCP protocol
3. The MCP server executes the action against the real application or API
4. The result is returned to Claude as context for its next response

From your perspective as the user, this is invisible — Claude just seems to be able to do more things.

Servers Available in This Session

Claude Code ships with several built-in MCP servers, and your session may have additional third-party servers connected. Here are the ones active in this project:

Filesystem (built-in)

BUILT-IN · ALWAYS ACTIVE

Core file operations — reading, writing, editing, searching. The Read, Write, Edit, Glob, and Grep tools you see Claude use constantly are all backed by the filesystem server.

`Read` · `Write` · `Edit` · `Glob` · `Grep`

Bash / Shell (built-in)

BUILT-IN · ALWAYS ACTIVE

Runs arbitrary shell commands. Gives Claude access to any CLI tool installed on the machine — Python scripts, git, npm, system utilities.

`Bash` · `PowerShell`

Computer Use

MCP SERVER · REQUIRES APPROVAL

Takes screenshots of the desktop, moves the mouse, types keystrokes, and interacts with native applications. Tiered by app category: full, click-only, or read-only depending on the app.

`screenshot` · `left_click` · `type` · `scroll` · `key`

Claude in Chrome

BROWSER EXTENSION · REQUIRES INSTALL

DOM-aware browser control. Faster and more reliable than pixel-clicking for web apps. Can read page content, fill forms, click elements, and read network requests — without needing a screenshot.

`navigate` · `find` · `form_input` · `read_page` · `get_page_text`

Web Search / Fetch (built-in)

BUILT-IN · ALWAYS ACTIVE

Searches the web and fetches URLs. Gives Claude access to current information beyond its training cutoff. Used for documentation lookups, current package versions, and research.

`WebSearch` · `WebFetch`

Session Management

MCP SERVER · PROJECT-SPECIFIC

Provides tools for archiving sessions, listing past sessions, and searching transcripts. Used in this project to resume context from previous sessions.

`list_sessions` · `search_session_transcripts` · `archive_session`

Connecting a Third-Party MCP Server

MCP servers for popular services exist for Slack, Linear, GitHub, Notion, Postgres, MySQL, and many more. Connecting one to your project takes a config entry in `.claude/settings.json`:

.claude/settings.json — adding an MCP server

```
{
  "mcpServers": {
    // A local server run as a subprocess
    "postgres": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-postgres"],
      "env": {
        "POSTGRES_CONNECTION_STRING": "postgresql://user:pass@localhost/mydb"
      }
    },

    // A remote server reached over HTTP/SSE
    "linear": {
      "type": "sse",
      "url": "https://mcp.linear.app/sse",
      "headers": {
        "Authorization": "Bearer YOUR_API_KEY"
      }
    }
  }
}
```

Local servers (the `command` form) are launched as a subprocess by Claude Code when the project opens — they start automatically and stop when the session ends. Remote servers (the `sse` form) are reached over HTTP; they run independently and must be available at the URL.

> **KEEP SECRETS OUT OF SETTINGS.JSON**
>
> The `settings.json` file is often committed to source control. For API keys and connection strings, use environment variables (`$MY_API_KEY`) in the config and set the actual values in your shell profile or a `.env` file that's in `.gitignore` .

Permissions and Approval Tiers

Not all MCP tools run automatically. Claude Code has a permission model that determines which tool calls require explicit user approval:

Tier	Behaviour	Examples
🟢 **Auto-approve**	Runs immediately, no prompt	File reads, Grep, Glob, WebSearch
🟡 **Prompt once**	Asks approval the first time per session	Shell commands, file writes, browser navigation
🔴 **Always prompt**	Asks approval every time	Computer use actions, destructive operations, external API writes

You can adjust the permission level for individual tools or entire servers in your project settings. In Claude Code's desktop app, the settings panel exposes permission controls per connected server. In the terminal, use the `/permissions` command (available in interactive sessions).

> **BROWSER TIER RESTRICTION**
>
> Browsers (Chrome, Firefox, Safari, Edge) are granted at a restricted **"read" tier** by the computer-use server — Claude can see their screen but cannot click or type. For web automation, use the Claude in Chrome MCP (DOM-based) instead of computer-use mouse clicks on a browser window.

Real-World MCP Use Cases

`postgres / mysql`	Claude queries your database directly — no copy-pasting schemas or results. "What are the top 10 users by order count this month?" gets answered with a live query, not a guess.
`github`	Claude reads issues, PRs, and comments from GitHub without you needing to paste them. "Summarise all open issues labelled 'bug'" works as a single command.
`linear / jira`	Claude reads your task tracker. "What tickets are in the current sprint?" pulls live data. Claude can also create and update issues directly.
`slack`	Claude reads channel history. Useful for context: "What did the team decide about the auth approach last week?" without you needing to summarise the discussion.

`filesystem (remote)`	Connect a remote filesystem server to give Claude access to files on a server that isn't your local machine — logs, config files, build artefacts.
`computer-use`	Claude interacts with any native desktop app — testing a GUI application, filling a form in software that has no API, reading output from a dashboard that can't be scraped.

Finding MCP Servers

The MCP ecosystem is growing quickly. Places to find servers:

- **Anthropic's MCP registry** — the official catalogue, searchable from within Claude Code via the MCP registry tool
- **GitHub** — search `modelcontextprotocol`; many community servers are open source
- **npm / PyPI** — servers are distributed as packages; `@modelcontextprotocol/server-*` is the common namespace for official ones
- **Service documentation** — major SaaS providers (Linear, Notion, Slack) now publish their own MCP servers with setup guides

> **WRITE YOUR OWN**
>
> If no server exists for a tool you use, the MCP SDK (available for TypeScript and Python) makes it straightforward to wrap any API. A minimal server that exposes three or four tools can be built in a few hours. The protocol handles authentication, discovery, and serialisation — you just implement the tool functions.

Next — Chapter 8: Designing a Project from Scratch

The final chapter. A practical walkthrough of setting up a new Claude Code project from zero: choosing what goes in CLAUDE.md, designing the memory structure, deciding which MCP servers to connect, and building the shorthand prompt map — using a real project as the worked example.

CAPSTONE: DESIGNING A PROJECT FROM SCRATCH

Working with Claude on Projects

Chapter 8 · Designing a Project from Scratch

The previous seven chapters covered the components of a well-designed Claude Code project: CLAUDE.md, memory files, shorthand prompts, multi-session workflows, slash commands, skills, and MCP servers. This final chapter brings them together — a practical walkthrough of setting up a new project from zero, with every decision explained. The worked example is this course itself.

Before You Open Claude — Three Questions

Project design starts before any file is created. Three questions determine most of the architecture:

Question	What it determines
1. How long will this project run?	A one-day project doesn't need a memory system. A multi-month project needs it from day one — retrofitting memory into a mature project is painful.
2. How much does Claude need to know to start each session usefully?	If the answer is "a lot," invest in CLAUDE.md and rich memory files. If the answer is "just the current task," a minimal setup is fine.
3. What recurring workflows will you perform many times?	Identifies candidates for shorthand prompts or skills. If you can name the recurring actions in advance, build the shorthand map before the first session.

For this project — a learning website with dozens of courses, each with 8–12 chapters, spanning months of work — the answers were: long-running, high context requirement, and highly repetitive workflows (chapter generation, SQL, PDF). All three pointed to a structured project setup from the start.

Phase 1 — Write CLAUDE.md

PHASE 1

CLAUDE.md — the session foundation

CLAUDE.md loads at the start of every session. It should answer the questions Claude would otherwise ask in the first few turns. For this project it contains:

- **Project overview** — what the website is, who it's for, the URL
- **Tech stack** — PHP/HTML/CSS, dark theme, specific colour values per course category
- **File conventions** — all generated files go in `claude-generated-files/`, fragment HTML only (no DOCTYPE), naming patterns
- **Recurring workflows** — chapter generation, SQL insert scripts, PDF builds. The existence of the shorthand system is noted so Claude knows to check memory before asking.
- **Hard rules** — always use vim not nano, commit message style, never force-push

What it does *not* contain: every course outline, every chapter list, every style value. Those live in memory files, which are loaded on demand — CLAUDE.md stays short so it loads fast and doesn't crowd out context.

CLAUDE.md — skeleton for a new project

```
# Project Name

One paragraph: what this is, who uses it, the goal.

## Tech Stack
- Language / framework
- Key libraries or tools
- Deployment target

## File Conventions
- Where generated files go
- Naming patterns
- What format (fragment HTML / full page / etc)

## Recurring Workflows
- Shorthand prompts: see memory/[course]_course.md
- SQL: see memory/learning_blog_sql.md
- PDF: always build at end of final chapter

## Hard Rules
```

```
- Use vim, never nano
- [Any other non-negotiables]
```

Phase 2 — Design the Memory Structure

PHASE 2

Memory files — what to save and where

Plan which memory types you'll use before starting, then create the files as information arrives — not all at once upfront.

- **User memory** — one file, updated rarely. Role, background, preferences that affect how Claude writes and explains things.
- **Feedback memory** — one file per rule or cluster of rules. Add entries after every correction or confirmation worth keeping. These are the highest-ROI memories.
- **Project memory** — one file per course or major workstream. Contains the outline, status table, naming conventions, and next step. Updated at the end of each session.
- **Reference memory** — one file per external system that Claude needs to query or interact with. Database schema, API locations, SQL insert conventions.

Keep MEMORY.md hooks under ~150 characters each — the index must stay skimmable so Claude can select the right files quickly. If a hook is getting long, the memory file it points to needs splitting.

The shorthand prompt map — a special case of project memory

For any project with repetitive workflows, the shorthand map is the most valuable memory file you'll write. Design it before the first session if you can, or extract it from CLAUDE.md as patterns emerge. This project's map maps short codes like `vpn6` or `proj3` to: file to create, styling spec, content outline, and what to do at the end. One memory file eliminates dozens of repeated instructions per course.

Phase 3 — Decide on MCP Servers and Skills

PHASE 3

Tools — start minimal, add as needed

Don't connect every available MCP server on day one. Start with what you know you need. This project started with just the built-in filesystem and shell servers, then added computer use and the Chrome extension as browser testing became relevant.

- **MCP servers**: connect a server when a real task needs it, not speculatively. "We might want to query the database directly someday" is not a reason to set up the postgres server today.
- **Skills**: prototype a workflow in chat for two or three sessions. Once the prompt is stable and you're typing it repeatedly, extract it into a skill file. This project's session-diary skill came from a manually-typed prompt that worked well in three consecutive sessions before being formalised.

Phase 4 — The First Session

PHASE 4

First session — establish, don't immediately produce

The first session of a new project is an investment, not a sprint. Spend it on:

- Verifying that Claude has read CLAUDE.md correctly — ask it to summarise the project from the file
- Doing one real task so you can observe what Claude gets wrong or assumes incorrectly
- Saving the first round of feedback memories from those observations before closing
- Writing the shorthand map memory file if you didn't already

The payoff arrives in session two: Claude starts with the right context, makes fewer wrong assumptions, and the first task completes faster than it would have in session one.

Maintaining the Project Over Time

A project that runs for months needs maintenance:

- **Review CLAUDE.md quarterly** — remove sections that no longer apply, update the tech stack if it's changed, trim anything that's migrated to memory files
- **Consolidate memory every few weeks** — merge duplicates, remove stale entries, check that MEMORY.md hooks still accurately describe their files
- **Retire completed workstreams** — a course that's finished shouldn't still have an active project memory file with a "next step" entry. Archive it or mark it clearly as complete.

- **Update skills when workflows evolve** — a skill that generates the wrong format because the target changed will silently produce wrong output. Review skills when the underlying workflow changes.

Project Design Checklist

Before the first session

- ✓ Answered the three framing questions (duration, context need, recurring workflows)
- ✓ Written a CLAUDE.md with project overview, tech stack, file conventions, and hard rules
- ✓ Planned memory file structure (which types, one file per workstream)
- ✓ Created MEMORY.md index (even if mostly empty — the structure matters)
- ✓ Identified recurring workflows and sketched the shorthand map

End of every session

- ✓ Decisions made this session are saved to the relevant memory file
- ✓ Status tables in course memory files updated to reflect what was completed
- ✓ "Next step" note written to the project memory file
- ✓ Any new feedback (corrections or confirmations) saved to feedback memory
- ✓ Generated files confirmed written to disk before closing

Periodic maintenance

- ✓ CLAUDE.md reviewed and trimmed — no stale sections
- ✓ Memory consolidated — no duplicates, no stale references
- ✓ Completed workstreams archived or clearly marked done

✓ Skills reviewed against current workflows

Working with Claude on Projects — Complete

8 chapters · From conversations to persistent multi-session projects

8	5	1
CHAPTERS	CORE COMPONENTS	COMPLETE PROJECT