Claude Code for Academics

An AI Agent for Empirical Research

Alessandro Spina[1]

University of Technology Sydney

3rd APSA Workshop on Quantitative Methods 2026

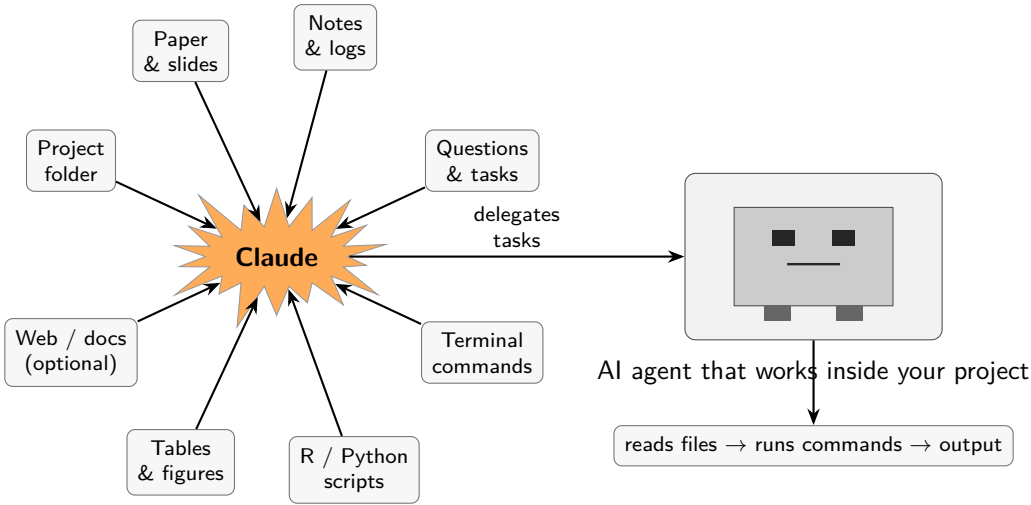
[1]This talk is in the spirit of Robin Hood: everything is stolen with the intent to share. See linked sources. All errors are mine.

Roadmap

1. What is Claude Code?
2. Setting up Claude Code
3. What can I use Claude Code for?
4. Using Claude Code effectively
5. How to stay safe
6. Where to from here?
7. Online Appendix

1
What is Claude Code?

Claude Code = LLM + Tools



Chat vs Agent

Chat-Based LLM

Stateless — no memory between messages

Copy-paste — you move text in and out

No file access — can't read your project

One-shot — answers, then forgets

Agentic AI

Persistent context — reads your whole project

Reads & writes files — edits code directly

Runs code — executes and checks output

Iterates — plans, acts, verifies, repeats

Think of it as a dedicated "RA" who reads your data, runs code, builds slides, and works with you to implement a research project.

Setup Claude Code

1. **Command Line (CLI)** — type `claude` in your terminal. Full feature set. Power-user default.
2. **Desktop App** — visual interface (Mac/Windows). Same engine, easier onboarding.
3. **IDE Extensions** — VS Code and JetBrains. Claude works inside your editor alongside your code.

Co-work vs Claude Code

Co-work (claude.ai/code) runs in a **cloud sandbox** — good for prototyping, but can't access your local files. **Claude Code** runs on **your machine**, inside **your project folder**, with full file-system access. For research workflows, you want Claude Code.

2
Setting up Claude Code

The Amnesia Problem

The Fundamental Challenge

Claude Code (any LLM) **forgets everything** between sessions. Every new terminal — even for the same project — starts from zero context.

What you need to do:

Build **external memory in markdown files** that persist across sessions:

```
CLAUDE.md  - project rules & key decisions
README.md  - what each directory contains
session_log.md - what was done & found
```

CLAUDE.md: Your Project's Ground Rules

How It Works

A **CLAUDE.md** is a short rulebook that Claude reads at the start of every session. Project overview, ground rules, key decisions, current status — everything Claude needs to hit the ground running.

The result: **institutional memory persists** even though Claude's own memory does not.

Tip: run `claude init` to auto-generate one.

"Make two markdown files. 1) README.md: documents the directory structure... 2) CLAUDE.md: our running file of stuff I want you to read first... Ground rules: 1. Under no circumstance are you to delete data or code. 2. You are never allowed to go outside this directory." — Prompt

What Belongs in CLAUDE.md?

Keep in CLAUDE.md:

1. Project overview (1–2 sentences)
2. Ground rules (never delete, stay in directory)
3. Key decisions (estimator, clustering, sample)
4. Current status
5. File naming conventions

Move elsewhere:

1. Detailed coding rules → `.claude/rules/`
2. Data documentation → `DATA.md`
3. Session-specific notes → session logs
4. Literature / references → paper folder
5. Long prompts → skills or commands

Keep CLAUDE.md under ∼**150 lines**. If it grows past that, you've probably mixed coding rules or data documentation into it; move those into **path-scoped rule files** or DATA.md instead.

→Appendix: Path-Scoped Rules

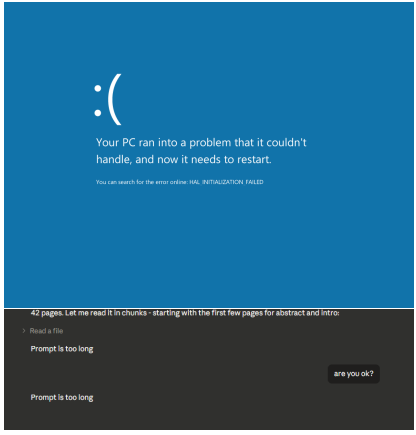
The Context Window Problem

The Challenge

Every LLM has a finite **context window**. Claude now **auto-compacts** — summarising earlier conversation to free space. You won't hit a hard wall, but quality still degrades: **context rot**.

Best practice:

- Use `/compact` manually when quality dips.
- **One task per session** — don't do data cleaning + paper editing + slides in one conversation.
- Write key decisions to CLAUDE.md or session logs *during* the session, so the next one picks up cleanly.



Context usage indicator — watch this during long sessions.

Session Logs: Your Project's Diary

Why This Matters

Future you will have **no idea** why you did something. Keep a running history by forcing Claude to keep Session Logs.

Two audiences for your logs:

Claude — reads the last log at session start. No more re-explaining.

You — searchable record of decisions, dead ends, and discoveries.

```
# 2026-05-27_session-01.md

## What was done
- Cleaned ANES panel, dropped
pre-2000 obs (codebook mismatch)
- Ran baseline DiD, 3 specs

## Key decisions
- Cluster at state level (not county)

## Open items
- Heterogeneity by race
```

One file per session, named by date. Update **continuously** — if the session crashes, the log survives. These are indispensible for keeping track of what was done.

Auto Memory

How Auto Memory Works

Claude **decides on its own** what to save — your role, preferences, corrections, project context.

Saved locally at `~/.claude/projects/.../memory/`. **Per-user, per-machine** — a co-author won't see yours.

Say "remember that..." to force a save, or "forget that..." to remove.

For Research, Build Your Own

Auto Memory is great for preferences ("I like tidyverse"). For **research projects** I want key information about the project stored in a more structured way:

1. Session logs — what and why
2. DATA.md — data decisions
3. Script registry — what each does

Personally, I do not rely on Auto Memory alone. I use the session logs, script registry, data registry, etc. to keep track of important information (for me too).

A Research Project Scaffold

Set up a **project structure** Claude can navigate:

```
project-name/
CLAUDE.md  - ground rules
README.md  - directory map
DATA.md   - source, licence,
AI-upload safety
SCRIPT REGISTRY.md - script registry
PINBOARD.md - to-dos, notes, ideas
data/
raw/ - never edit
code/   output/   paper/
.gitignore
```

Three Files You're Probably Missing

DATA.md — for each dataset, document the source, licence restrictions, and whether it is **safe to share with an API**. Document processing steps + outputs.

SCRIPT REGISTRY.md — one line per script: what it reads, what it writes, what question it answers. Keeps a growing project **navigable** for Claude and for you.

PINBOARD.md — to-dos, papers to read, ideas, data issues. The notes you'd otherwise lose between sessions.

3
What can I use Claude Code for?

Things You Can Ask Claude to Do

You've installed Claude Code — now what?

1. **Data preparation** — clean, reshape, and construct variables from raw data
2. **Turn a paper into slides** — feed it an Overleaf doc and get a Beamer deck
3. **Audit and rerun your code base** — check existing scripts for bugs and replication
4. **Canvas / LMS management** — restructure modules, update dates, upload materials
5. **Teaching support** — convert slides, generate quizzes, manage course content
6. **Visualization & sense checks** — plot data before regression, every time
7. **Your own personal referee** — have Claude critique your draft before submission
8. **Build up results** — generate tables/figures, iterate, assemble a draft

Data Preparation

Claude can write code which reads your raw data, understands the structure, and builds the processing pipeline:

1. **Read** raw data + codebook
2. **Clean** and reshape
3. **Construct** variables
4. **Validate** with summary stats
5. **Document** everything

```
> "Read the CSV in Data/Raw/
anes_panel.csv.  Show me
variable types and missing rates."
> "Recode party_id using the
codebook in Docs/codebook.pdf."
> "Construct a binary turnout
variable.  Validate against the
CPS supplement."
```

Helps you keep a clean set of data that is well documented and easy to follow (i.e., replicable). We'll talk about keeping a script and data registry later...

Making Slide Decks with Claude

Claude can generate entire Beamer presentations. Without a **style template** to copy from, it defaults to generic Beamer themes that look like a 2008 seminar. Give it an existing deck first.

This deck was built with Claude Code. I wrote the bullet points; Claude wrote the LaTeX. We then iterated until I was satisfied.

See prompt template: `https://github.com/scunning1975/MixtapeTools/blob/main/presentations/deck_generation_prompt.md`

Tips:

- Give Claude an existing .tex file as a style reference (e.g. Rhetoric of Decks)
- It will make formatting mistakes. Use Skills to train Claude.

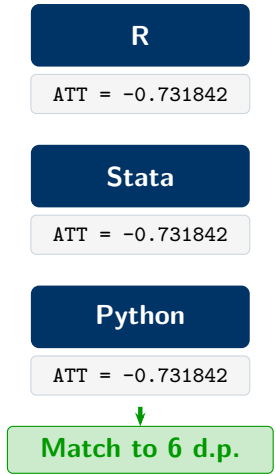
Re-running code: The Cunningham Conjecture

Checking Code

The DGP for coding errors is **orthogonal across languages**. So if you ask Claude to do the same analysis in two languages, the two error processes are essentially independent.

If Claude writes R code with a subtle bug, the Stata version will likely have a *different* bug — or none at all.

Ask Claude to replicate your R code in Python. If they produce identical results, you have higher confidence the code is correct. When they *don't* match, you've caught a bug that single-language review would miss.



Working with Canvas (LMS)

Canvas has an **API** — Claude Code can interact with it directly from your terminal.

1. **Read** course structure
2. **Update** assignments & dates
3. **Restructure** modules
4. **Upload** materials in bulk

```
> "Connect to Canvas for ECON301.
List all modules and their
current due dates."
> "Push all due dates back by
one week starting from Week 5."
> "Upload the Week 8 lecture PDF
and create a new assignment
stub for Problem Set 4."
```

Anything you do manually in the Canvas UI, Claude can do via the API. Especially useful for **bulk operations** across modules. Discuss more in the demo sessions...

Teaching with Claude

1. **PPTX → Beamer conversion** — feed Claude a PowerPoint deck, get clean LaTeX. Updating schedules each semester becomes trivial.
2. **Quiz & exam generation** — fast but often **not good enough**. I had to train it on my question style. Once trained, easy to reuse each semester.
3. **Canvas / LMS management** — restructure modules, bulk-edit due dates, upload materials via the API.

The Quality Trap

AI-generated quiz questions *look* plausible but are often of poor quality. **Invest upfront** to build a question bank of "good" questions — then Claude can replicate the pattern. Make sure to document your preferences, for replication in future semesters.

Always Plot the Data

Claude reduces the marginal cost of creating a figure to near **zero**.

I was running an event-study on a large dataset, too large to load into memory. Asked Claude to plot the key variable over time. Weird jumps in the data. Claude dug into the raw data, found the explanation, proposed a fix.

Still needed checking — but saved a lot of detective work. A regression table wouldn't have shown it.

Quick sense checks — plot raw data before regression, every time

New figure styles — describe what you want, populated with real data in minutes

Pushback — Claude sometimes questions your data or code decisions

Building Up a Paper with Claude

"Claude, write me an introduction" — you'll get an introduction that reads **like AI slop**:

1. **Generate** tables and figures
2. **Ask Claude** to explain results
3. **Add** explanation to the paper
4. **Check** — does this make sense?
5. **Iterate** on figure/table design

```
> "Here is Table 3 (DiD results).
Write 2-3 sentences explaining
the coefficients for a draft."
> "The event-study plot looks odd
pre-2015.  Remake it dropping the
pre-period outliers."
> "Compile the results .tex and
send me the PDF to share with
co-authors."
```

Iterate on a **results.tex** with figures, tables, and a few sentences of explanation next to each. By the time it's finished, you'll have a sense of the narrative of the paper. Then write the introduction **yourself**. Claude is great for final checks.

Skills: Reusable Workflows

Skills are instruction files (**pre-written workflows**) that teach Claude Code how to do a specific task.

A skill is one markdown file:

```
---
name: paper-review
description: summarise a paper
---
# Paper Review
Extract: question, data,
identification, main finding.
```

Without a skill:

"Read `paper.pdf`. Tell me the research question, the data, the identification strategy, and the main finding..."

Every. Paper. You. Read.

With a skill:

```
/paper-review paper.pdf
```

Done.

You can also create **slash commands** (`/newscript`, `/reviewcode`) — one-shot verbs stored in `.claude/commands/`. → Appendix: Skills vs. Commands

A Few Skills I've Built*

Research & Code

`/spin-up` — brief Claude on project state at session start

`/wrap-up` — save the session log before context fades

`/code-sweep` — end-of-milestone `Code/` audit vs. paper

`/data-profiler` — systematic dataset profiling protocol

Writing & Project Memory

`/paper-editor` — seven-audit editorial review of drafts

`/script-registry` — table of what each script does

`/glossary` — per-project terms & command-phrases

`/pinboard` — quick notes, to-dos, ideas, data issues

A skill is just a markdown file — **copy mine, write your own** at `github.com/aspi6246/Claude-Code-Skills-for-Academics`.

Harnessing your own personal referee

1. **Generate report** — "The Editor" SKILL triages issues, offers solutions, produces a report
2. **Plan fixes** — in a fresh session, feed the report to Claude. It creates an action plan for each comment
3. **Implement** — work through each issue one by one.
4. **Re-review** — generate a 2nd report.

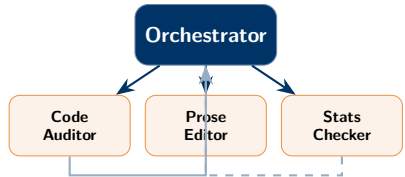
Paid services are now available — see `https://www.refine.ink/` (at significant cost). While nowhere near as capable, the Claude Code workflow above lets you quickly pick up the 'low-hanging fruit' before a conference or journal submission.

Agents and Agent Teams

Agents are autonomous workers Claude spawns for subtasks. Each gets its own context window, works independently, and reports back.

Agents can adopt personas:

1. Referee 2 — code auditor
2. The Editor — prose auditor
3. Custom — your own protocols



Multiple specialists in parallel, results merged by the orchestrator.

Agents have their own context window. Hand off a 5,000-line audit; your main session keeps its space. Agent Teams run *multiple specialists in parallel*.

4
Using Claude Code effectively

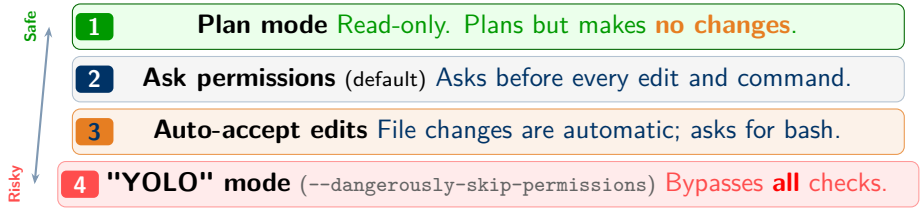
Four Habits for Working with AI

1. **Start in Plan Mode** — ask Claude clarifying questions before it acts
2. **Define terms** — tell Claude to explain jargon with examples; catches misunderstandings early
3. **Be specific** — concrete details beat vague commands; clever $\neq$ clear
4. **Separate action from judgment** — AI drafts; you decide what is true

Together they prevent most of the errors I see — and most of the awkward moments when a co-author asks "**wait, what did you actually do here?**"

Li & Liu, *AI for Econ/Finance Research Handbook*, 2026

Start in Plan mode



The Pattern for Research

Start in **Plan mode** (no longer the default), read the plan, then switch to **Ask permissions** to execute. Reading the plan takes thirty seconds; reconstructing what a **YOLO-mode** session actually changed can take an hour.

Clarify first

> *"Claude is more or less like a reasonably trained Labrador retriever. It can rush ahead, off its leash, and even though it will come back, it can get into trouble in the meantime."*
> — Cunningham

The technique: Ask Claude to explain its understanding *before* it writes code:

> > "Do you see the issue with this specification?"
>
> > "Before you run anything, explain what this regression identifies."

Why this matters for research:

If Claude guesses wrong, that reveals a **misunderstanding** that needs correcting *before* you proceed.

Cunningham, *Claude Code* series, 2026

Define Terms Early

The problem:

Claude will confidently use jargon — even when it **misunderstands** what you mean. Terms like "resid," "instrument," or "fixed effects" can mean different things.

The fix:

Ask Claude to **define key terms** in its own words before proceeding.

```
> "Before we start:  define what
you mean by 'panel data' and
'fixed effects' in this context."

> "Explain back to me what this
regression identifies."
```

Next level: build a **glossary** skill that captures your project's vocabulary.

I call these 'translation errors' rather than hallucinations — Claude isn't making things up, it's just misunderstanding. Catching these early is crucial to saving time and effort later on.

Be Specific

You don't need to be a "prompt engineer." You need to be **clear** about what you actually want.

Vague	Specific
"Summarise this literature."	"Using only these 12 papers, build a table: question, data, identification, finding, limitation, link to my project. Mark missing facts."
"Clean my data."	"Inspect the folder and propose a raw-to-processed pipeline; don't edit yet. Identify raw, derived, code, privacy risks, and checks I should approve. Document all changes in the script and data registry."
"Make my slides."	"Build a 25-minute seminar outline for an internal Brownbag presentation. For each slide: the claim, the evidence, figure/table needed, risk of overclaiming, and a speaker note."

If you're not sure you've been clear enough, ask Claude to **ask you questions back**.

Separate Action from Judgment

We need to be honest about what AI is good at / not good at*:

Claude does well:

Draft code and text

Clean and reshape data

Generate figures and tables

Suggest specifications

Flag inconsistencies

Can't do well:

Is this specification identified?

Does this result make sense?

Which robustness checks matter?

What's the narrative of the paper?

Is this ready for co-authors?

Claude is a *very* fast research assistant. But you need to cast a critical eye over anything AI-generated. Unsupervised, Claude will generate a lot of figures/tables — it doesn't mean they are all useful or correctly set up.

5
How to stay safe

What I Do to Protect Myself

1. **Scope access** — only open Claude within a specific project folder
2. **Write the rules down** — CLAUDE.md specifies what it can and cannot do
3. **Start with Plan mode** — never YOLO mode
4. **Global settings** — allow/deny rules in `settings.json`
5. **Backups** — version control via Git/Dropbox

```
// ~/.claude/settings.json
"permissions": {
"allow": [
"Read(.../Projects/**)",
"Edit(.../Projects/**)"],
"deny": [
"Read(.../Dropbox/**)",
"Read(**/.env)",
"Bash(rm -rf *)"]
}
```

Deny takes precedence over **allow**.

Additional security: Running Claude in a Secure Sandbox

The Concern

An AI agent with file-system access can run destructive commands.

The Solution: Containers

Run Claude in an isolated container. It can install packages, run arbitrary commands, and **delete files inside the container** — **none of which reach your real machine**.

```
$ claude-container start
```

Goldsmith-Pinkham's one-command setup

`github.com/paulgp/claude-container`

Your Data Leaves the Machine

The Risk

When you ask Claude to read `XX_panel.csv` and run summary statistics, the **file contents are sent to Anthropic's API** as part of the conversation context.

Does this breach your data licence or IRB agreement?

Only files Claude actively reads enter the API context. Files it never opens are not transmitted.

The Workaround

Use Claude Code for **code**, not **data**:

- Write scripts that reference file paths — don't ask Claude to read the raw data
- Use synthetic / anonymised data for prototyping
- Run the final pipeline locally, outside Claude
- Check your data licence terms before sharing any file with an API
- Document each dataset's AI-upload status in **DATA.md** — decide *before* you start

Three Things That Will Go Wrong

The Failure Modes

1. **Fake citations** — plausible but nonexistent references
2. **Wrong estimator** — clean code that answers the wrong question
3. **Sycophancy** — reinforcing your priors because it's optimised to be agreeable

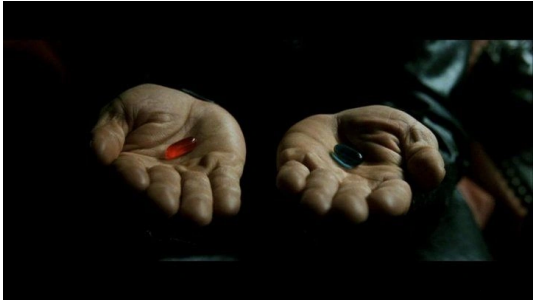
The Defences

1. **Local papers only** — save PDFs to a folder; only cite what Claude can verify
2. **Cross-language verification** — R = Stata = Python to 6 d.p.
3. **Personas & Agents** — adversarial review catches what you miss
4. **Your judgement** — the paper goes out with your name on it, not Claude's

6
Where to from here?

Where to from Here?

1. Skilled researcher + AI > AI
2. Startup costs (and actual $ costs) are non-trivial. The benefits take time.
3. Beware of complexity. Academics have ~~physics~~ computer science envy.



When Should You Use AI?

Use AI on a task when:

$$\underbrace{\text{Labour saved}}_{\text{time, tedium}} > \underbrace{\text{Setup}}_{\text{learning, prompting}} + \underbrace{\text{Verification}}_{\text{checking output}} + \underbrace{\text{Privacy risk}}_{\text{data exposure}} + \underbrace{\text{Maintenance}}_{\text{future cost}}$$

High ROI tasks:

Data cleaning, figure generation, code auditing, slide formatting, boilerplate writing

Low ROI tasks:

Novel theoretical arguments, creative framing, tasks requiring deep domain judgment, one-off tasks faster to do by hand

Li & Liu, *AI for Econ/Finance Research Handbook*, 2026

Honest Downsides

1. **The bottleneck is you** — Claude generates figures and tables faster than you can process them. You have to slow down and read what was produced. Almost every error I've caught was caught at this step, not by re-prompting.
2. **Cognitive switching costs** — easy to run multiple Claude sessions in parallel, hard to actually read carefully across sessions. I now do one at a time when the work matters.
3. **Workflow optimisation is a trap** — if you spend more time tinkering with your setup than the time you save, what have you achieved? I think I'm net more productive, but this is something to watch out for.

The mechanical work gets faster. The **thinking** takes the same time it always did, and you still have to do it. When I find myself juggling **five sessions**, it's usually because I've stopped doing the second part. Creating lots of output can give the illusion of making progress.

*"The deeper goal is to make scholars **better at research**, not merely faster at producing research-looking artifacts."*

— Li & Liu, *AI for Econ/Finance Research Handbook*, 2026

`https://github.com/SuperJayLiu/AI-for-Economics-and-Finance-Research`

Resources

Reading & Philosophy

Cunningham's Blog (12+ parts)
causalinf.substack.com

Causal Inference: The Mixtape
mixtape.scunning.com

Kustov: Wake Up on AI
alexanderkustov.substack.com

Claude Code Safety
producttalk.org/how-to-use-claude-code-safely/

AI Agents Gone Wrong (video)
youtu.be/JiA4fvoeUfI

Li & Liu: Econ/Finance Handbook
github.com/SuperJayLiu/AI-for-Economics-and-Finance-Research

Tools & Repositories

MixtapeTools Skills
github.com/scunning1975/MixtapeTools

Sant'Anna Workflow
github.com/pedrohcgs/claude-code-my-workflow

Claude Container (Goldsmith-Pinkham)
github.com/paulgp/claude-container

ChernyCode (Mele, LSE)
github.com/meleantonio/ChernyCode

APE Project
ape.socialcatalystlab.org

Frank Lee: Research Skills
github.com/franklee16/academic-research-skills

Thank you.

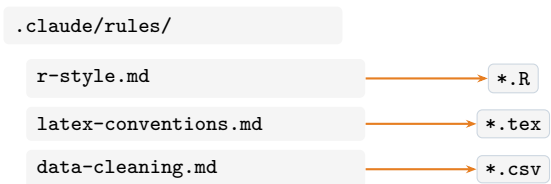
Find slides, code, and resources at:

https://www.alessandro-spina.com/claude-code/

7
Online Appendix

Rules: Path-Scoped Instructions

Rules are markdown files that Claude loads **automatically**. The clever trick is **path-scoping**:



Each rule file loads *only* when you're working on matching file types. R rules don't clutter your LaTeX sessions.

Why This Matters

Aim to keep your CLAUDE.md file to **100–150 lines**. When you accumulate too many rules, path-scoping keeps the active instruction set small and relevant, avoiding **information overload** that degrades Claude's performance.

← Back to: What Belongs in CLAUDE.md?

Skills vs. Commands

The difference is about **who triggers it**.

Commands = Verbs

You invoke them **explicitly**.

Type `/newscript` → Claude creates a new `.Rmd` with the right header, naming convention, and location. One command, one action, done.

Files in `.claude/commands/`

Skills = Knowledge

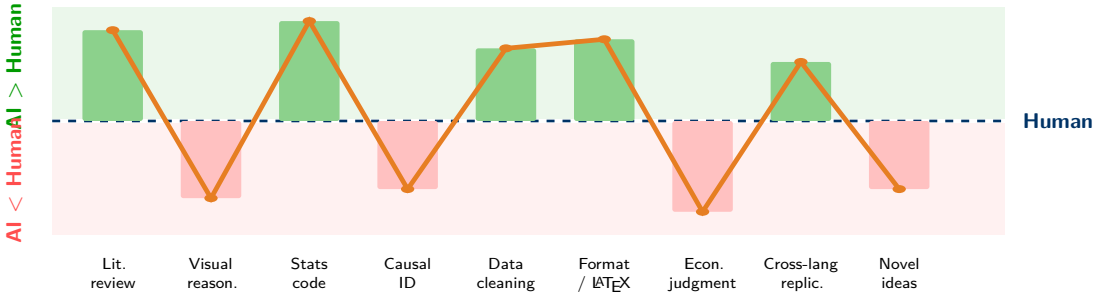
Claude discovers them **on its own**.

Say "make me a regression table" → Claude scans `.claude/skills/`, finds your conventions (fixest, clustering, LaTeX output), and applies them automatically.

Files in `.claude/skills/`

Commands say **"do this thing now."** Skills say **"whenever this topic comes up, here's how we do it."**

The Jagged Frontier



The implication: AI skeptics and AI evangelists are both right.

https://www.oneusefulthing.org/p/the-shape-of-ai-jaggedness-bottlenecks

Glossary I: Computing Fundamentals

- **LLM** (Large Language Model) — a statistical model trained on text to predict the next token; the engine behind Claude, ChatGPT, etc.
- **CLI / Terminal** (Command-Line Interface) — text interface to your computer: you type commands instead of clicking.
- **IDE** (Integrated Development Environment) — a code editor with debugger, autocomplete, and tooling (VS Code, JetBrains, RStudio).
- **API** (Application Programming Interface) — a structured way for one program to send requests to another, often over the internet.
- **Markdown** — a lightweight plain-text formatting syntax (`.md` files); human-readable as-is, renders to HTML/PDF.
- **Git** — distributed version-control system that tracks every change to a project's files so you can roll back.
- **GitHub** — web platform that hosts Git repositories for sharing and collaboration.
- **Repository (repo)** — a project folder placed under Git's version control.
- **JSON** (JavaScript Object Notation) — structured plain-text format (e.g. `{"key": "value"}`) widely used for config files like `settings.json`.
- **Container / Sandbox** — an isolated execution environment (e.g. Docker); programs inside cannot see or change files outside.

→ More: Claude Code specifics ← Back to talk

Glossary II: Claude Code Specifics

- **Anthropic** — the company that develops Claude.
- **Context window** — the LLM's working memory: everything it can "see" in one conversation. Finite (∼200k tokens for Claude).
- **Auto-compact** — Claude's automatic summarisation of earlier turns to free up space when the context window fills.
- **Context rot** — response-quality degradation as the context window fills up.
- **Plan mode** — Claude Code mode where the model reads and reasons but cannot edit files. The safe default for research.
- **Skill** — a pre-written instruction file (markdown) Claude loads when invoked; encodes a reusable workflow (e.g. `/compile-latex`).
- **Slash command** — a shortcut starting with / (e.g. `/newscript`) that triggers a skill or one-shot action.
- **Agent / Subagent** — an autonomous AI worker Claude spawns for a subtask; has its own context window and reports back when done.
- **Persona** — instructions that make Claude behave as a specific role (e.g. Referee 2, The Editor).

← Previous: Computing fundamentals ← Back to talk