How I Use AI as an Economist

Guohui Jiang

Last updated: April 21, 2026 · gjiang-economics.github.io

⇒ Read online | Get the latest PDF

Notes from using AI in real, months-long research projects. Not a tutorial—what I have learned, what works, and what doesn't. Written for researchers who already use AI and want tricks beyond the basics. Organized as a short list of *principles*, followed by a growing set of *dispatches*—dated notes, each self-contained. Each dispatch carries a permanent number (#N) and a floating *Updated* date.

Contents

1 Principles . . . **2**

2 Dispatches . . . **2**

2.1 #13 Git worktrees in Claude Code: keep `main` clean, run agents in parallel . . . 2
2.2 #12 Reviewing Claude Code's work: review the plan, the execution, or both . . . 3
2.3 #11 Annoyed by Claude Code permission prompts? Read this. . . . 5
2.4 #10 Context management in Claude Code: when to continue, compact, or start fresh . . . 5
2.5 #9 Compound Engineering: a workflow for research projects . . . 6
2.6 #8 The reflection workflow . . . 8
2.7 #7 Cross-model brainstorming . . . 9
2.8 #6 Humanizer: strip AI writing patterns . . . 9
2.9 #5 Remote control: approve sessions from your phone . . . 9
2.10 #4 Travel reimbursement as a skill . . . 9
2.11 #3 Hooks: automatic context and guardrails . . . 10
2.12 #2 Syncing Claude Code across two machines . . . 11
2.13 #1 Self-guided learning with AI . . . 12

3 Resources . . . **12**

Principles

Three ideas that shape how I use AI day to day.

1. **Teach the assistant once, benefit forever.** Every `CLAUDE.md` line, skill, and memory compounds across projects and machines. The first task is expensive; the tenth is cheap. Not sure how to teach it? Just ask: *"I want Claude Code to do X better in the future. What's the best way—a skill, a hook, a memory, or something else?"*
2. **Use structure, not willpower.** Plan mode, hooks, skills, and review passes keep the AI from charging ahead in the wrong direction. Unstructured prompts produce unstructured work. When a task is non-trivial, I reach for a workflow rather than typing harder.
3. **The reviewer should not be the builder.** The agent that wrote something is the least likely to notice what it missed—like an author proofreading their own paper. Dispatch a second agent, or a second model, with a blank slate.

Dispatches

Notes I add as I learn. Sorted by most recently updated (newest first). Each card carries a permanent number (#N) and a floating *Updated* date.

#13 Git worktrees in Claude Code: keep `main` clean, run agents in parallel

Updated April 21, 2026

Research runs for months, most trials fail, and parallel agents on the same working tree step on each other's files. One flag—`claude -w <name>`—fixes both problems at once.

The problem. An empirical project is not a one-shot game. A single classification task, a single regression, a single data pull can stretch across months, and most attempts along the way fail. Out of seven trials, maybe two survive into the polished paper. The naive workflow is to keep everything on `main` and dump every session's log into a single `progress.md`. That holds up for a week. By session fifteen, `progress.md` is a graveyard of abandoned ideas—and Claude Code re-reads it on every new session. Old, dead context pollutes the next decision: the model keeps referencing approaches you already discarded and drifts back toward work you have already ruled out.

There is a second problem. Claude Code is at its best when you run agents *in parallel*—one training a classifier while another cleans the panel while a third drafts the appendix. But three agents sharing the same working tree means three agents writing to the same files. Overwritten edits, lost work, and conflicts that only surface after both sides have already drifted.

The fix: one worktree per exploration. A git worktree is exactly what it sounds like: a *separate working copy* of the same repository, on a separate branch. Same `.git` under the hood, different files on disk. Claude Code ships first-class support behind one flag:

```
claude -w text-llm
```

That creates a fresh worktree at `.claude/worktrees/text-llm` on a new branch, launches Claude inside it, and leaves `main` untouched. Long form: `claude --worktree text-llm`. The name is optional—omit it and Claude picks one for you. Inside the worktree, try things. If the trial wins, merge the branch back to `main`. If it loses, delete the worktree with `git worktree remove text-llm` and the failed history is gone—`main` never saw it, and no future session re-reads it.

A worked example — text classification. I am classifying sentiment toward immigration in parliamentary speeches. I want to compare a modern LLM-based classifier with a traditional keyword/dictionary approach—two separate pipelines, two validation runs—and I want to run both *at the same time* without them colliding. Two terminals, two Claudes, one command each:

```
claude -w llm-classify
claude -w keyword-classify
```

Now each agent has its own working tree. The LLM agent writes prompts, annotates a validation set, and calls the model. The keyword agent builds a dictionary, runs regex, and computes precision/recall. Their files live at different paths on disk—`.claude/worktrees/llm-classify/` and `.claude/worktrees/keyword-classify/`—so they cannot overwrite each other. No `git stash` dance is needed to switch between them.

A week later the LLM approach wins on the held-out set. I merge `llm-classify` into `main` and remove the keyword worktree. The paper—and every future Claude session on `main`—sees only the classifier I kept. The dead branch does not hang around in `progress.md` whispering about keyword thresholds I no longer care about.

When to merge, when to discard. **Merge** when the trial settles a question: a feature that sticks, a cleaning step that survives, a model that beats the baseline. Add a short note in `main`'s `progress.md`—*what* you decided and *why*—and drop the detailed exploration log along with the branch history. **Discard** when the trial answered "no." Delete the worktree; keep one sentence in `main`'s `progress.md` noting you tried X and it did not work, so you do not re-try it six months later. Either way, `main` stays a record of load-bearing decisions rather than a graveyard of every false start.

Source. Documented under the `--worktree` flag in Claude Code's CLI reference. Related: Dispatch #9 on Compound Engineering, for the broader brainstorm → plan → work → review loop this slots into.

#12 Reviewing Claude Code's work: review the plan, the execution, or both

Updated April 20, 2026

Principle 3 says the reviewer should not be the builder. In practice that means two separate review passes—one on the plan before anything runs, one on the diff after the run. Which one you need depends on how expensive it is to execute the work.

What to review: plan, execution, or both? Review *the plan* when re-executing is slow or expensive—GPU training runs, large LLM API calls, long Stata pipelines over big data. It is much cheaper to catch a bad premise on paper than after the run completes. Review *the execution* when the work is cheap to replicate—local data cleaning, analysis scripts, website edits, small refactors. Let the agent try, then attack the diff. **Both** is always safe; default to both when the task is unfamiliar or the blast radius is unclear.

The mechanism is the same in each case: after Claude Code returns a plan in Plan Mode, it asks *"Would you like to proceed?"* with four options. **Pick option 4—"Tell Claude what to change"**—and type a review command as the feedback. Claude then runs the review instead of proceeding, and comes back with the results for you to absorb.

```
Would you like to proceed?

  1. Yes, and use auto mode
  2. Yes, manually approve edits
  3. No, refine with Ultraplan on Claude Code on the web
❯ 4. Tell Claude what to change
     shift+tab to approve with this feedback
```

Claude Code's approval prompt after plan mode. Pick option 4—"Tell Claude what to change"—and type a review command as the feedback.

Reviewing the plan. When Claude finishes planning, it calls `ExitPlanMode` and the four-option prompt appears. This is the moment to review. Pick option 4. As the feedback, type something like:

```
run /document-review on the plan first, absorb the comments,
update the plan, then stop and ask me again
```

`/document-review` comes from the Compound Engineering plugin. It spawns parallel persona reviewers against the plan file on disk—a feasibility lens, a scope lens, a security lens, and others—each returning comments from a blank slate. Re-approve with option 1 or 2 when you are satisfied; only then does execution begin.

Reviewing the execution. The trick for execution review is to bake it *into* the plan during planning, not after the run. While you are still on the first "Would you like to proceed?" prompt after plan mode, pick option 4 and type something like:

```
add a final step to the plan: run /codex:adversarial-review
(or /ce:review) on the full diff at the end, absorb the
comments, fix the affected code, and re-run the pipeline step
that changed
```

Claude Code rewrites the plan to include the review, then asks you to approve again. From that point the review runs automatically as the last step of execution—you do not have to remember to trigger it.

The two review commands. `/codex:adversarial-review` delegates the review to a Codex-powered model—a genuinely different brain reading the diff with no knowledge of how it was built. Use it when you want the strongest blank-slate critique. `/ce:review` runs Compound Engineering's tiered multi-persona review locally—correctness, maintainability, testing, plus conditional personas the diff triggers (security, data migrations, API contracts). Same "second agent with a blank slate" idea as `/document-review`, pointed at code instead of prose.

Source. Both plugins come from the Compound Engineering ecosystem, installable into Claude Code via the plugin marketplace. The plugin itself is covered in Dispatch #9.

#11 Annoyed by Claude Code permission prompts? Read this.

Updated April 19, 2026

We want Claude Code to act as an autonomous agent—go off, refactor a pipeline, run a long analysis, iterate on a benchmark. In practice it interrupts every few seconds asking for permission to run this bash command or that MCP tool. Two recent features let you stop babysitting without reaching for `--dangerously-skip-permissions`.

Fix 1: Auto mode. Auto mode routes every permission prompt through a model-based classifier that decides whether the command is safe. Safe commands are auto-approved; genuinely risky ones still surface to you. Safer than `--dangerously-skip-permissions`, because the classifier actually reads the tool call. Press `Shift+Tab` in the CLI to cycle into auto mode, or pick it from the mode dropdown in Claude Desktop or the VSCode extension. The practical win: you can run several Claudes in parallel—once one is cooking, switch to the next and let the classifier handle routine approvals.

Fix 2: The `/fewer-permission-prompts` skill. Complementary to auto mode. The skill scans your session history, finds bash and MCP commands that are clearly safe but caused repeated prompts, and recommends a prioritized allowlist to paste into `~/.claude/settings.json` (or the project-level `.claude/settings.json`). Run it from any session:

```
/fewer-permission-prompts
```

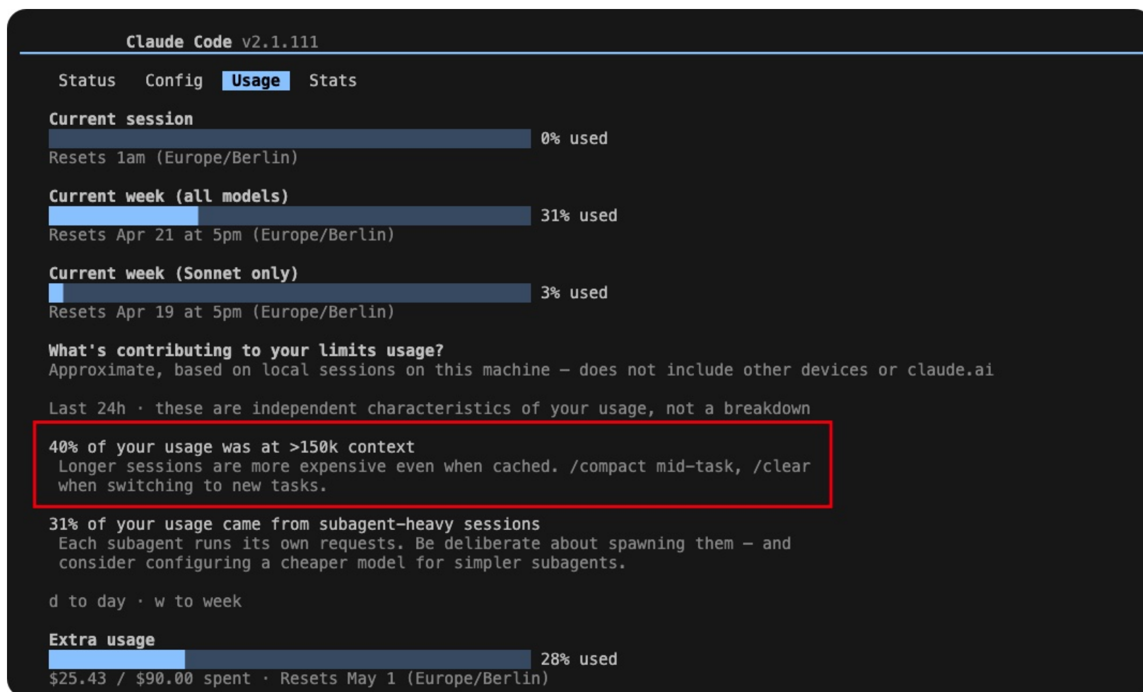
Think of it as a tuning pass you repeat every few weeks as your tool use evolves. Reference: Claude Code — Configure permissions.

Source. Both features are described by Boris Cherny (Anthropic) in this thread on X.

#10 Context management in Claude Code: when to continue, compact, or start fresh

Updated April 16, 2026

Even on the Max plan, long sessions routinely hit usage limits. Most consumption comes from a handful of very long context windows, not many short ones. The screenshot below shows what this looks like in the usage dashboard.



Usage dashboard: a few long-context sessions dominate monthly consumption.

This is not just a cost issue. Model reasoning quality degrades as the context window fills. The model attends over more tokens, older context gets diluted, and the assistant starts making mistakes it would not make in a fresh session.

Anthropic published a practical guide on this problem: Using Claude Code: session management and 1M context. I recommend reading it. It covers four strategies—continue, compact (`/compact`), start fresh, and spin off sub-agents—with concrete guidance on when each one applies.

#9 Compound Engineering: a workflow for research projects

Updated April 14, 2026

Four commands—`/ce:brainstorm`, `/ce:plan`, `/ce:work`, `/ce:review`—turn a drifting AI session into a structured workflow. This dispatch walks through what each one does, how they chain together on a real research task, and when to use or skip the whole thing.

What it is, and why I use it. Compound Engineering (CE) is a Claude Code plugin that wraps a software-engineering workflow around any non-trivial task. Instead of typing prompts and hoping the agent stays on track, you run the task through four stages: *brainstorm*, *plan*, *work*, and *review*. Each stage produces a durable artifact—a requirements document, an implementation plan, committed code, a review report—that the next stage builds on.

For research projects, this matters because the work spans weeks or months. A single long session drifts. A fresh session starts without context. CE forces the context to live in files on disk, not in the model's head, so the work survives session boundaries and is auditable after the fact.

The stage I care about most is **review**. The agent that wrote the code is the least likely to notice what it missed. CE dispatches a second agent with a blank slate, and that agent routinely catches things the building agent rationalized away.

Setup. Install the plugin once. From a Claude Code session:

```
/plugin marketplace add every-inc/every-marketplace
/plugin install compound-engineering@every-marketplace
```

After restart, the four commands become available as `/ce:brainstorm`, `/ce:plan`, `/ce:work`, and `/ce:review`. Artifacts land in `docs/brainstorms/` and `docs/plans/`.

1. Brainstorm — `/ce:brainstorm`. Collaborative dialogue to answer *what to build*. The agent asks one question at a time, challenges assumptions, and proposes 2–3 concrete approaches with tradeoffs. Output: a requirements document in `docs/brainstorms/` that captures the problem frame, scope boundaries, and success criteria.

Use it when the request is vague, the scope is unclear, or multiple directions are plausible. Skip when the request is already specific.

2. Plan — `/ce:plan`. Turns the requirements doc into an implementation plan. Each unit has a goal, file list, approach, test scenarios, and verification criteria. Output: a plan file in `docs/plans/YYYY-MM-DD-NNN-...-plan.md` with checkbox-tracked implementation units.

Use it when the *what* is settled and you need the *how*—especially when work spans multiple files or has dependency ordering. Claude Code's built-in plan mode (`Shift+Tab`) is lighter-weight: good for single-session tasks, but `/ce:plan` is better when the plan needs to outlive the session.

3. Work — `/ce:work`. Executes the plan systematically. Creates a feature branch (or worktree), builds a task list from the plan's implementation units, and works through them with tests and incremental commits. Output: the actual code changes, committed.

If `/ce:work` starts asking lots of clarifying questions mid-implementation, that's a signal to stop and tighten the plan first. Pausing to fix the plan is almost always faster than pushing through on a shaky one.

4. Review — `/ce:review`. Dispatches independent reviewer agents, each starting with a blank slate, to audit the diff against the plan, the codebase conventions, and explicit quality criteria (correctness, security, performance, tests, maintainability). Output: a review report with findings graded by severity.

This is the most valuable stage. The building agent is invested in its own solution; a fresh agent sees the code as a reader would. In my experience, most of the useful catches happen here—missed edge cases, convention drift, tests that mock too much, dead code the building agent forgot to remove.

How they chain in practice. A typical research-adjacent task, start to finish:

1. I start with a vague itch (*"my website section on AI is badly organized"*) and run `/ce:brainstorm`. After ten or so exchanges, I have a requirements doc that says: move to a Principles + Dispatches structure, drop beginner content, and migrate existing tricks.
2. I run `/ce:plan` on that doc. The plan file lists five implementation units; each unit names specific files.
3. I run `/ce:work`. The agent creates a feature branch and works through the plan. I'm still in the loop—confirming the dispatch lineup, spot-checking the rendered page, approving the deploy—but the agent is driving.

4. Before merging, I run `/ce:review`. The reviewer catches things the building agent missed: a broken relative path, a dispatch that still contains a beginner-framed sentence, a LaTeX section I forgot to update.
5. I fix the review findings, then merge.

The whole cycle for this kind of task takes half a day. Without CE I would have spent the same half-day but with more rewriting and less confidence that I caught the rough edges.

When to skip or shortcut. CE is overkill for small tasks. Rough rules:

- **Skip brainstorm** when you already know exactly what to build. Go straight to `/ce:plan` or describe the change directly.
- **Skip plan** when the work is under roughly three files and the approach is obvious.
- **Never skip review** for anything that matters. Running `/ce:review` on a diff is cheap—a few minutes—and the catches pay for it.
- **Skip the whole thing** for trivial edits (typo fixes, single-line config changes). The ceremony costs more than it saves.

I lean on CE most heavily for work that (a) touches more than a handful of files, (b) spans multiple sessions, or (c) has non-obvious design decisions I want captured in writing so future-me or a co-author can follow along.

#8 The reflection workflow

Updated April 13, 2026

Perhaps the most powerful technique I have found. When you encounter a new type of task:

1. **Push Claude Code to its best.** Work through the task iteratively.
2. **Do not delete the session history.** The full history is essential for what comes next.
3. **Ask Claude Code to reflect:**

```
Reflect on this entire session. Is there anything you
should save as a skill, a memory, or a setting change
so that next time we do this, it goes faster?
```

Claude Code will then create reusable artifacts—skills, memories, or configuration changes. The first time you do a task with AI might not save time. The second time, it will. For more on creating skills, see this walkthrough by @zarazhangrui.

I have built a dedicated reflection skill that automates this process. Install it with:

```
mkdir -p ~/.claude/skills/reflection && \
curl -o ~/.claude/skills/reflection/SKILL.md \
  https://gjiang-economics.github.io/skills/reflection.md
```

Then at the end of any session, type `/reflection`.

#7 Cross-model brainstorming

Updated April 12, 2026

When Claude Code struggles with a subtask after a few attempts, try a “second opinion” from another model:

1. Ask Claude Code to **summarize the situation**: problem, what was tried, what failed.
2. Say: *“Give me a self-contained prompt I can paste into Gemini.”*
3. Copy it into Gemini (or another AI), then paste the solution back.

Different models have different strengths. Combining them often gets you to a solution faster than pushing a single model repeatedly. The same trick works in reverse when a chatbot conversation gets stuck—summarize and hand it to Claude Code.

#6 Humanizer: strip AI writing patterns

Updated April 11, 2026

AI text has telltale patterns—inflated language, filler phrases, em dashes everywhere. The humanizer skill detects 29 such patterns and rewrites text to sound natural.

Claude Code. Install and run `/humanizer`:

```
mkdir -p ~/.claude/skills/humanizer && \
git clone https://github.com/blader/humanizer.git \
  ~/.claude/skills/humanizer
```

Claude AI (web) and Claude Desktop. Create a Project, then paste the SKILL.md contents into the Project’s custom instructions.

#5 Remote control: approve sessions from your phone

Updated April 10, 2026

Claude usage is calculated in **5-hour windows**. If Claude Code pauses for a permission prompt while you are away, those hours are wasted. *Remote Control* lets you monitor and approve sessions from your phone or any browser via claude.ai/code or the Claude mobile app.

```
/remote-control
```

Or start a new session with: `claude --remote-control`

My workflow. Before leaving the office, I start a long task with `/remote-control`. On the train home, I check my phone, approve prompts, and review progress. If I know what comes next, I start a second session—when session one completes, I pick up session two from my phone. See the official documentation for details.

#4 Travel reimbursement as a skill

Updated April 9, 2026

After a conference, I use Claude Code to prepare reimbursement documents using a custom *skill*.

Step 1: Organize your folder.

```
reimbursement/
|-- proofs/                        # Receipts and payment evidence
|   |-- 01 flight receipt.pdf
|   |-- 01 boarding pass.pdf
|   |-- 03 hotel invoice.pdf
|   |-- 06 taxi receipt.pdf
|   |-- 11 city tax photo.JPG      # Photos of paper receipts work too
|-- host_reimbursement_policy.pdf   # Instructions from the host
|-- summary_template.xlsx           # Excel template (if any)
```

Files in `proofs/` are numbered by item. Files sharing the same prefix (e.g., `01`) belong to the same expense.

Step 2: Install the skill.

```
mkdir -p ~/.claude/skills/reimburse-external && \
curl -o ~/.claude/skills/reimburse-external/SKILL.md \
  https://gjiang-economics.github.io/skills/reimburse-external.md
```

The skill walks Claude through a 10-phase workflow: read the host's policy, catalog receipts, verify against rules, calculate the claim with currency conversion, fill in templates, merge receipts into a single PDF, and draft the submission email.

Step 3: Run it. Open the reimbursement folder in VS Code and type:

```
/reimburse-external
```

Adapting this. This skill handles *external* reimbursement (conference hosts, funding bodies). For your university's process, create a separate skill—ask Claude Code to reflect after your first reimbursement and it will build one tailored to your institution.

#3 Hooks: automatic context and guardrails

Updated April 8, 2026

Hooks are shell scripts that Claude Code runs automatically on specific events—prompt submission, tool execution, file edits. You configure them once and they enforce rules in every session. Typical uses:

- **Inject context**—tell Claude your prompt was dictated, or that you are on a specific machine.
- **Block dangerous actions**—intercept tool calls to prevent force-pushes or unapproved deletions.
- **Automate repetitive steps**—run linting or update changelogs automatically.

My most-used hook: dictation context. I dictate with Wispr Flow. Speech-to-text as a non-native English speaker occasionally mishears words or drops articles. A prompt-submit hook tells Claude the input was dictated so it silently corrects artifacts instead of asking for clarification.

Save this as `~/.claude/hooks/dictation-context.sh`:

```
#!/bin/bash
cat <<'EOF'
DICTATION NOTICE: The user's prompt was likely dictated
using Wispr Flow (speech-to-text). The user is a native
Chinese (Mandarin) speaker. The transcription may contain:
- Homophones or near-homophones (e.g. "EA" -> "it")
- Misheard technical terms or proper nouns
- Missing or wrong articles, prepositions, or plurals
- Run-on sentences or missing punctuation

Mentally correct any likely transcription artifacts and
interpret the intended meaning. Do not ask for clarification
on obvious transcription errors -- just fix them silently.
EOF
```

Then activate it in `~/.claude/settings.json`:

```
{
  "hooks": {
    "UserPromptSubmit": [
      {
        "type": "command",
        "command": "$HOME/.claude/hooks/dictation-context.sh"
      }
    ]
  }
}
```

Adapt the script to your own native language—just change "Chinese (Mandarin)" and the example error patterns to match your accent.

For a curated collection of other hook ideas, see this tweet by @zodchiii. Full documentation: official hooks docs.

#2 Syncing Claude Code across two machines

Updated April 7, 2026

I use Claude Code on a MacBook and a Windows desktop. To keep them identical, I store shared config in Dropbox and symlink from `~/.claude/`:

```
Dropbox/AI_meta/claude-sync/
|-- memory/         # Your personal AI memory
|-- skills/         # Custom skills
|-- commands/       # Custom slash commands
|-- plugins/        # Plugins
|-- CLAUDE.md       # Global instructions
|-- settings.json   # Shared preferences
|-- .mcp.json       # MCP server config
```

On Mac:

```
ln -sf ~/Dropbox/AI_meta/claude-sync/memory   ~/.claude/memory
ln -sf ~/Dropbox/AI_meta/claude-sync/skills   ~/.claude/skills
# ... same for each item
```

On Windows (PowerShell as Administrator):

```
# Folders -- use Junction
New-Item -ItemType Junction -Path "$env:USERPROFILE\.claude\memory" `
  -Target "C:\Users\YOU\Dropbox\AI_meta\claude-sync\memory"

# Files -- use SymbolicLink
New-Item -ItemType SymbolicLink -Path "$env:USERPROFILE\.claude\CLAUDE.md" `
  -Target "C:\Users\YOU\Dropbox\AI_meta\claude-sync\CLAUDE.md"
```

Machine-specific settings go in `settings.local.json` and do not sync.

#1 Self-guided learning with AI

Updated April 6, 2026

Pick a topic you want to learn. Tell the AI your background, available time, and what you want to understand. Ask it to build a study plan—readings, key ideas, and an order to work through them. As you read, use the AI to ask questions, check your understanding, and explore connections. What used to take days of searching now takes hours.

Also worth setting: AI chatbots trained with RLHF tend to be *sycophantic*—defaulting to agreement rather than honest critique. Counterproductive when you are trying to learn. In Claude, go to **Settings** → **General** → **"What personal preferences should Claude consider in responses?"** and add:

```
Be direct and honest. Don't default to agreement
or flattery: I prefer candid feedback, especially
on my research.
```

This overrides the default tendency at the preference level, so every conversation starts candid rather than agreeable.

The self-guided learning idea comes from a post by Jesús Fernández-Villaverde, who used AI to design a self-guided study of sociologist Erving Goffman—and found the experience comparable to a master's-level course week.

Resources

- In Claude Code CLI, type `/powerup`
- Claude Blattman
- Paul Goldsmith-Pinkham at Markus Academy
- Aniket Panjwani at VoxDev, and his YouTube channel

- awesome-ai-for-economists