An AI-Assisted Research Flow

A practical guide for empirical researchers

Dr. Jared L. Black

February 2026

Contents

An AI-Assisted Research Flow

A practical guide for empirical researchers. {: .fs-5 .fw-300 }

By Jared Black {: .fs-4 .fw-300 }

Download as PDF{: .btn .btn-primary .mr-2 }

Is this you?

You have a research project sitting in a folder somewhere on your computer. Maybe several. You started it a while ago — could be months, could be years — and life moved on. The project didn't.

The code lives in scattered files. Some of it you wrote yourself. Some you borrowed from someone else's analysis and adapted to your data. You put it together in pieces, at different times, under different assumptions, and it worked well enough to produce some results. But now those pieces are spread across folders like papers on a messy office floor. You know the work is in there. You just can't face the idea of sorting through it.

Your output is in a similar state. Maybe you have a LaTeX document, or a Word file, or some Markdown. There are figures and tables embedded in it — but they are old versions. The updated ones sit in another folder, waiting to be swapped in. The code that produced those figures may or may not be streamlined. It may be in a different directory entirely. You are not sure you could run it again and get the same results.

Your data preparation steps live in one place, grounded in one set of assumptions. Your estimation results and exhibits were created later, possibly under different assumptions, and you cannot completely trace them back. The pipeline — if you can call it that — has gaps. Dependencies are unclear. The idea of picking this project back up and bringing it to a publishable state feels overwhelming.

If this sounds familiar, this guide is for you.

What this does

By learning the approach described here, you can do what I did: take a stalled project — disorganized, daunting, collecting dust — and revive it. Organize the code. Rebuild the pipeline. Map every table and figure back to the script and data that produced it. Update your manuscript. Get the project to a state where one command reproduces everything from raw data to final output.

You do this with an AI that works directly in your project files. Not a chatbot you paste code into. A command-line tool that reads your scripts, understands your folder structure, writes new code

that follows your conventions, and maintains the documentation as you go. You make the research decisions. It handles the scaffolding.

The whole process may take a few dedicated sessions. And once the project is organized and the paper is done, you can put it down and move on to the next one — knowing that if you ever need to come back, everything is in order.

What you will learn

1. **Set up your machine** — install the tools, configure the environment, get Claude Code running.
2. **Understand the project structure** — a folder layout that enforces discipline and makes AI collaboration possible.
3. **Work the flow** — how a research session actually works when AI handles the scaffolding and you handle the thinking.
4. **Write with AI** — use AI assistance for manuscripts without losing your voice or your standards.

Why this workflow and not a chatbot

One of the most important things to understand upfront: **everything stays on your computer, in your files, in your format.**

When you paste code into ChatGPT or upload files to a web interface, your work lives in someone else's context window. You do not know exactly where it is. You cannot easily undo what was done. If the session ends or the tool changes, your work may be gone.

This workflow is different. The AI operates directly in your project folder — the same folder you see in Finder or File Explorer. Every script it writes is a file on your hard drive. Every change it makes is visible in your file system. If you are using Git, every change is tracked and reversible. Nothing is hidden from you. Nothing takes place outside of your control.

If the AI writes a bad script, you delete the file. If it makes a change you do not like, you revert it with one Git command. If you decide to stop using AI entirely, you still have a perfectly organized project folder with all your code, data, and documentation exactly where you left it.

That is the point. The AI is a tool that works in your space, on your terms, producing artifacts that belong to you.

The template

Everything in this guide maps to a concrete, reusable project template: Research-Project-Flow. You can clone it, fill in your details, and start working. The guide explains what each piece does and why.

No command-line experience required. No AI experience required. Just a willingness to follow instructions and learn as you go.

Acknowledgments

This guide builds on the work of several people:

- Scott Cunningham (*Causal Inference: The Mixtape*) — the Referee 2 audit protocol and the idea that you cannot grade your own homework. His MixtapeTools project pioneered many of the practices described here for using AI in empirical research.
- Matthew Gentzkow and Jesse Shapiro — *Code and Data for the Social Sciences*, the foundational reference for reproducible project organization.
- Deirdre McCloskey — *Economical Writing*, the source of the writing standards enforced throughout the template.
- NetworkChuck — the setup guide and video that this project's installation instructions are based on.
- AEA Data Editor — the replication standards and template README that inform the project structure and documentation requirements.

> **Tip Reading this guide** — Work through the Getting Started chapters to set up your machine. Read The Flow chapters when you start your first project. Keep the Reference section open as a cheatsheet.

Why AI-Assisted Research

The problem

Empirical research requires managing a staggering number of moving pieces: raw data, cleaning scripts, analysis code, output files, manuscripts, citations, and the documentation that ties them together. Many researchers handle this by hand. Files accumulate. Naming conventions drift. Documentation lags behind reality. By the time a paper is under review, the project folder is a maze that only the original author can navigate — and sometimes not even them.

The tools exist to solve this. Version control, reproducible pipelines, structured folder layouts — these practices are well-documented. Gentzkow and Shapiro wrote an influential guide in 2014. The TIER Protocol formalized it. The AEA Data Editor enforces it. But adoption can be slow, in part because the overhead of maintaining these systems falls on the researcher.

What changes with AI

AI does not replace the researcher's judgment. It replaces the overhead. Specifically:

Structure maintenance. An AI agent that reads your `CLAUDE.md` file knows your project layout, your naming conventions, and your pipeline order. When you add a new script, it updates the README, the master do-file, and the execution script automatically. The structure stays current because the AI maintains it.

Script writing. You describe what a script should do — its inputs, outputs, and purpose. The AI writes the code, following your existing conventions. You review and approve. The tedious translation from "I need a script that merges these files on county FIPS codes" to actual working code takes minutes instead of hours.

Documentation. Every script header, every pipeline table entry, every data dictionary update — the AI handles these as part of the workflow. Documentation is no longer a separate task you defer. It happens as you work.

Error checking. The AI reads your logs after every run. It flags warnings, identifies failures, and suggests fixes. You catch problems immediately instead of discovering them three weeks later.

What does not change

- **You** decide the research question.
- **You** choose the identification strategy.

- **You** evaluate whether results make economic sense.
- **You** write the arguments that hold a paper together.

The AI is fast at mechanical tasks and unreliable at judgment calls. The workflow described here tries to exploit the first property and guard against the second.

Where to trust it and where not to

For **data cleaning, organization, and visualization**, the AI is consistently strong. Importing, reshaping, merging, labeling, charting — this is where the workflow saves the most time.

For **estimation code**, the AI often gets things wrong — in my experience. Not just the logic — the syntax. A `reghdfe` command with the wrong absorb structure, a `felm` call with misspecified clusters. I find it helps to understand your statistical software well enough to catch these errors. If estimation is handed off to the AI without careful review, bad code can get through.

For **writing**, the AI can tighten prose and enforce style rules, but in my experience, the actual argument — the interpretation, the contribution, the narrative — needs to come from you. I recommend being hands-on for estimation and writing, even if you give the AI wide latitude over everything else.

The cost of not adopting

If you organize your project poorly, no AI can fix it. If you organize it well, AI makes every subsequent step faster. The Research Project Flow template gives you the organization. This guide shows you how to use AI within it.

In my experience, each project completed this way makes the next one faster, because the template improves and the habits stick.

Tools & Setup

Note Instructions are written for macOS with Windows/Linux notes where they differ.

Tip Feeling overwhelmed? This guide covers a lot. You do not need to absorb it all before you start. Follow the first few steps below to get an AI coding tool running on your machine. Once it is working, copy the URL for this website and paste it into your AI tool: *"Read this guide and help me set up my project."* It will walk you through the rest.

Two paths

Path A: With GitHub. Clone the template, get version control and backup. Recommended, and the rest of this guide assumes it.

Path B: Without GitHub. Download the template as a zip, unzip it, drop your files in, open an AI coding tool in that folder. No Git required. The core workflow is identical.

Step 1: Set up an AI coding tool

You need a terminal-based AI coding assistant — Claude Code, Gemini CLI, OpenAI Codex, or similar. Any of them will work with this template.

For setup instructions covering prerequisites, installation, and authentication on **Mac, Windows, and Linux**, follow the guide at:

> **ai-in-the-terminal** — Start with the prerequisites and quickstart, then follow the page for whichever tool you choose (Claude Code, Gemini CLI, Codex, etc.).

Based on NetworkChuck's companion guide to the "AI in the Terminal" video.

A note on costs

This is an AI-heavy workflow. If you are working on a real research project — reading data files, writing scripts, editing manuscripts — you will consume tokens quickly. Be realistic about that upfront.

At minimum, you need a paid subscription. Claude Code works with a Claude Pro account ($20/month), which gives you a set number of tokens per hour, per day, and per month. The free tier will not get you through a working session.

I recommend going further, at least for the first month. If you are trying to get a stalled project unstuck, treat the higher-tier subscription (Claude Max at $100/month, or equivalent for your tool) as part of the cost of getting your research moving again. Dive in, spend a few weeks working intensively, and then assess whether the ongoing cost is worth it. For me, the value was obvious after one real session — but I had to invest enough to reach that point.

You can monitor your token usage at console.anthropic.com. If you hit your limit mid-session, the tool pauses until your tokens refresh — no surprise charges. In my experience, this is not a tool you can evaluate on the free tier. Budget accordingly.

If you get stuck during setup, you can use any AI you already have access to (ChatGPT, Claude.ai, Gemini) to help troubleshoot.

Step 2: Set up Git and GitHub *(Path A only)*

Skip this if you are taking Path B (no GitHub).

The prerequisites guide above covers Git installation. Once Git is installed, create a GitHub account if you don't have one, then install the GitHub CLI:

```
brew install gh        # macOS/Linux
gh auth login
```

Step 3: Get the project template

Path A (GitHub):

```
gh repo create my-project --template Black-JL/Research-Project-Flow --private --clone
cd my-project
```

Path B (no GitHub):

1. Go to github.com/Black-JL/Research-Project-Flow.
2. Click the green **Code** button → **Download ZIP**.
3. Unzip it wherever you keep your research projects. Rename the folder to your project name.

Either way, you now have the full folder structure. Open your terminal, navigate to the folder, and launch your AI tool:

```
cd /path/to/my-project
claude                    # or gemini, codex, etc.
```

The AI reads your `CLAUDE.md`, orients itself, and is ready to work.

Step 4: Install your statistical software

Install whichever your field uses:

- **Stata:** Download from your institutional license portal. On macOS, add it to your PATH: `export PATH="/Applications/Stata/StataMP.app/Contents/MacOS:$PATH"` in `~/.zshrc`.
- **R:** `brew install r` *(Windows: r-project.org)*
- **Python:** `brew install python` *(Windows: python.org)*

Step 5: Optional tools

SuperWhisper (voice-to-text). SuperWhisper runs offline on your Mac and transcribes speech to text wherever your cursor is. Set it to offline mode, press + Space, talk, and the text appears. No typing, no cloud, no subscription. Lets you speak instructions to Claude Code instead of typing them. *(macOS only; Windows alternative: Whisper.cpp)*

Zotero + Better BibTeX (citation management). Install Zotero and the Better BibTeX plugin. Create a project collection, export with "Keep updated" to `manuscript/references.bib`. Your bibliography stays in sync automatically.

Dropbox (file sync). Place your project folder in Dropbox for backup and co-author sharing. If using Git, exclude `.git` from Dropbox sync: `xattr -w com.dropbox.ignored 1 .git` *(macOS)*. If you are on Path B with no GitHub, Dropbox becomes your primary backup — consider it strongly recommended.

Verify

```
claude --version          # or your chosen AI tool
```

If you are on Path A, also verify:

```
git --version
gh --version
```

Once everything works, move on to Project Structure.

Project Structure

Clone the template

Open Terminal, navigate to where you want the project, and clone:

```
gh repo create my-project --template Black-JL/Research-Project-Flow --private --clone
cd my-project
```

This creates a private repo on your GitHub account with the full template structure. You now have a project folder that looks like this:

```
my-project/
  README.md                  ← Project overview, pipeline, replication
  CLAUDE.md                  ← AI agent instructions
  run_all.sh                 ← Master execution script
  .gitignore

  data/
    raw/                     ← Untouched source data. NEVER modify.
      README.md              ← Source, date, access instructions
    processed/               ← Created by scripts

  scripts/
    00_run.do                ← Master do-file: globals and pipeline
    params.do                ← Research parameters
    programs/                ← Reusable helper functions

  output/
    logs/                    ← Execution logs
    figures/                 ← Plots and maps
    tables/                  ← LaTeX table fragments
    results/                 ← Stored estimation results

  manuscript/
    manuscript.tex           ← Active manuscript
    references.bib           ← Auto-exported from Zotero
    aea_style_guide.md       ← Formatting reference

  scratch/                   ← Throwaway work. Not committed.
```

Why this structure

Six principles govern the layout:

1. **Every file has one home.** Outputs in `output/`, data in `data/`, scripts in `scripts/`. No duplicates, no ambiguity.
2. **Data scripts write to `data/processed/`. Analysis scripts write to `output/`.** No script crosses this boundary.
3. **Absolute paths live in one place.** Machine-specific paths go in `00_run.do`. Everything else uses globals defined there.
4. **One command reproduces everything.** `run_all.sh` executes the full pipeline from raw data to final output.
5. **Fail loudly.** Every script logs its execution. When something breaks, the log shows where and why.
6. **Structure enforces discipline.** If a file doesn't have an obvious home, the structure needs updating — not the file.

These conventions draw from Gentzkow & Shapiro, the TIER Protocol, and the AEA Data Editor's requirements. Following them gets your project closer to replication-ready from the start.

Key files

`README.md`

The project README is the single source of truth. It contains:

- The pipeline table (every script, its inputs, its outputs)
- The parameters table (every hardcoded research value and its source)
- The table/figure map (which script produces which output)
- Replication instructions

When the AI adds a script or modifies an output, it updates the README automatically. You should verify these updates are correct.

`CLAUDE.md`

This file tells the AI how to behave in your project. It specifies:

- The project root path
- Rules (e.g., `data/raw/` is read-only)
- Key file locations
- How to execute scripts
- Writing standards

The AI reads this file at the start of every session. It is the contract between you and the AI. Edit it when you want to change the AI's behavior. If you are starting a project without the template, you can create a fresh `CLAUDE.md` by typing `/init` in Claude Code.

> **Note Other tools have their own equivalents.** Codex uses `AGENTS.md`. Gemini CLI uses `GEMINI.md`. The content is the same idea — project instructions the AI reads at startup. If you use multiple tools on the same project, create the appropriate file for each. The template ships with `CLAUDE.md`; adapt it for your tool of choice.

`scripts/00_run.do`

The master do-file. It defines path globals for every collaborator's machine and calls each pipeline step in order. When you or the AI add a new script, it gets registered here.

`scripts/params.do`

All hardcoded research parameters — treatment dates, sample restrictions, outcome definitions. Centralizing them here means a parameter change propagates everywhere automatically. The values must match the Parameters table in the README.

`run_all.sh`

The shell script that executes the pipeline. It calls Stata, R, and Python scripts in sequence, captures logs, and reports success or failure. Run a single step:

```
./run_all.sh "01_import"
```

Run everything:

```
./run_all.sh --all
```

First steps after cloning

1. **Set your machine path** in `scripts/00_run.do`. Replace the placeholder path with your actual project directory.
2. **Edit the README.** Replace placeholder content with your project details: data sources, computational requirements, parameters.
3. **Configure CLAUDE.md.** Update the project root path. Review the rules and adjust if needed.
4. **Set up `.gitignore`.** The template includes sensible defaults. Add any large data files or sensitive content.
5. **If using Dropbox:** Move the project folder into Dropbox and exclude `.git`:

```
xattr -w com.dropbox.ignored 1 .git
```

6. **Launch Claude Code** and start your first session:

```
claude
```

The AI will read your `CLAUDE.md`, orient itself, and tell you if anything needs attention. You're ready to work.

Your First Session

You have the tools installed. You have the template cloned. You are staring at a terminal. Now what?

This chapter walks through what actually happens when you launch an AI coding tool in your project for the first time — and how to use it to rescue a stalled project.

The first 30 minutes

Open your terminal. Navigate to your project folder. Type `claude` (or `gemini`, or `codex`) and press Enter.

Here is what happens:

```
$ cd ~/Dropbox/my-project
$ claude

> Claude Code is reading your project...
> Found CLAUDE.md. Loading project instructions.
```

The AI reads your `CLAUDE.md` and silently runs `/status` to orient itself. If everything is clean, it says nothing. If something needs attention — uncommitted changes, missing files, stale logs — it tells you.

Now you are in a conversation. The AI can see your files. You can see your files. Everything it does happens in the folder in front of you.

Your first message should be simple. Try:

```
> Look at my project and tell me what you see. What's here,
> what's missing, and what needs attention?
```

The AI will scan your folder structure, read your scripts, check your README, and give you a status report. It might say: "You have 6 scripts in `scripts/` but none of them are registered in the pipeline table. Your `data/raw/` folder has 3 CSV files but no README documenting their source."

That is your starting point. You now know what the AI sees, and you can start giving it tasks.

A few things to try in your first session:

- *"Read `scripts/05_merge.do` and explain what it does."* — Test whether the AI understands your code.

- *"Create a README for `data/raw/` documenting the three CSV files in there."* — Low-risk, high-value. You can see exactly what it writes.
- *"Run `/check` and show me everything that's inconsistent."* — Let the AI audit the project so you can see where the gaps are.

Do not try to rebuild your entire pipeline in the first session. Get comfortable with the conversation. Watch how the AI reads files, proposes changes, and asks for confirmation. Build trust gradually.

Rescuing a messy project

This is the core use case. You have a folder — maybe several folders — with scattered scripts, old data files, half-finished analysis, and a manuscript that references outputs you cannot reproduce. Here is how to turn that into an organized project.

Step 1: Drop everything into the template

Clone the template (or download the zip). Copy your existing files into it:

- Data files → `data/raw/`
- Scripts → `scripts/` (keep their original names for now)
- Output files → `output/figures/`, `output/tables/`, or `output/results/`
- Manuscript → `manuscript/`
- Everything else → `scratch/`

Do not worry about getting it perfect. The point is to get everything into the structure so the AI can see it.

Step 2: Tell the AI what you have

```
> I just dropped my existing project files into this template.
> Here's what I have:
> - 6 Stata do-files in scripts/. They were written at different
>   times and I'm not sure of the correct run order.
> - 3 CSV files in data/raw/ from CDC WONDER.
> - A half-finished manuscript in manuscript/.
> - Some old figures in output/figures/ that may or may not match
>   the current code.
>
> Please read through the scripts, figure out the dependency
> order, and tell me what you find.
```

The AI will read every script, trace what each one reads and writes, and map the dependencies. It will come back with something like: "Here is the order I think these run in, based on inputs and outputs. Script 03 reads a file that script 01 produces. Script 06 references a variable that doesn't exist in any upstream data — this may be a bug or a missing step."

Step 3: Let the AI build the pipeline

Once you agree on the order:

```
> That looks right. Please:
> 1. Rename the scripts to follow the numbering convention
>    (01, 05, 10, etc.)
> 2. Add structured headers to each one
> 3. Register them in the README pipeline table
> 4. Set them up in 00_run.do and run_all.sh
```

The AI does the mechanical work. You review the result. In one session, your scattered folder becomes a documented, reproducible pipeline.

Step 4: Test it

```
> /run --all
```

Run the full pipeline. The AI reads the logs and reports what succeeded and what failed. Fix the failures. Run again. Repeat until everything is clean.

This process — drop, describe, organize, test — can take a single long session or a few shorter ones depending on how messy the project is. The point is that you supply the knowledge of what each piece does, and the AI handles the reorganization.

How to talk to the AI

When the AI is operating on your actual files, specificity matters more than it does in a chat interface. The AI is operating on your actual files, and it needs specificity.

Be explicit about inputs and outputs

Bad: "Write a script to clean the data."

Good: "Write a Stata do-file that reads `data/raw/cdc_wonder_2015_2022.csv`, keeps only county FIPS, year, and death count columns, drops rows where the Notes column contains 'Unreliable', and saves to `data/processed/cdc_clean.dta`."

The second prompt gives the AI everything it needs to write working code on the first try. The first prompt forces it to guess — and in my experience, it usually guesses wrong.

Name your files

The AI can see your file system, but it helps to be explicit. Instead of "merge the datasets," say "merge `data/processed/cdc_clean.dta` with `data/processed/acs_demographics.dta` on county FIPS code." This eliminates ambiguity and prevents the AI from merging the wrong files.

State what you do NOT want changed

This matters more than you think. If you ask the AI to "clean up this script," it may rewrite your estimation logic along with the formatting. Be specific:

```
> Edit scripts/20_estimate.do. Tighten the formatting and add
> a proper header. Do NOT change any estimation commands, variable
> definitions, or sample restrictions.
```

Give context about your research

The AI does not know your field. If you say "run a diff-in-diff," it will write generic code. If you say "run a difference-in-differences where treatment is city-level decriminalization, the treatment date varies by city, and we need to account for staggered adoption using Callaway and Sant'Anna (2021)," it will write much better code — and you will still need to verify it.

Three prompts, from worst to best

1. *"Analyze the data"* — The AI has no idea what you want. It will produce something generic and probably wrong.
2. *"Run a regression of death_rate on treatment with county and year fixed effects"* — Better. The AI can write this. But it might pick the wrong command, the wrong standard errors, or the wrong sample.
3. *"Write a Stata do-file that estimates a two-way fixed effects regression:* `reghdfe death_rate treatment, absorb(county year) cluster(state)`*. Use the analysis file at* `data/processed/analysis_panel.dta`*. Save the estimates to* `output/results/main_fe.ster` *and produce a formatted table to* `output/tables/main_results.tex` *using* `esttab`*."* — The AI can execute this exactly. You still check the output, but you are far less likely to get something wrong.

The pattern is always the same: tell the AI what goes in, what comes out, and what the constraints are.

The AI-Assisted Workflow

This chapter shows how a research session works in practice. Not theory — the actual sequence of actions when you sit down to work.

Your workspace

You can work in a plain terminal window — navigate to your project folder, type `claude`, and go. That works fine. But a better setup is to use **VS Code** or **Cursor** (they work almost identically) so you can see everything at once.

Recommended layout

1. Open VS Code (or Cursor).
2. **File → Open Folder** and select your project folder.
3. Press Ctrl + ‘ (backtick) to open the integrated terminal.
4. *(Optional)* Move the terminal panel to the side: press Cmd + Shift + P (macOS) or Ctrl + Shift + P (Windows/Linux), type `View: Move Panel Left`, and hit Enter.

Now you have:

- **Left panel:** Terminal running Claude Code (or your AI tool of choice)
- **Center/right:** Your script or manuscript file open for editing
- **Additional tabs:** A compiled PDF preview, data files, logs — whatever you need

Everything in one window. You talk to the AI on the left, watch the code change in the center, and preview your compiled manuscript on the right.

Alternatives

- **Terminal only.** Open a terminal, `cd` to your project, type `claude`. Open a separate file browser and your PDF viewer alongside it.
- **VS Code / Cursor extensions.** Both editors offer Claude and Codex extensions that embed the AI directly in the editor sidebar. These work well for code editing. The terminal approach (typing `claude` in the integrated terminal) gives you the full Claude Code experience with slash commands and project management.

Use whatever feels comfortable. The workflow is the same either way.

Starting a session

In your terminal (standalone or inside VS Code), navigate to your project and launch:

```
cd ~/Dropbox/my-project
claude
```

The AI reads your `CLAUDE.md` file and runs `/status` automatically. It scans the project and reports anything that needs attention: stale logs, Dropbox conflicts, uncommitted changes. If everything is clean, it stays quiet.

You talk to the AI by typing — or speaking, if you set up SuperWhisper — in the terminal. It responds, reads files, writes code, and executes commands, all within your project directory.

How the AI runs your scripts

When you tell the AI to run a Stata do-file, an R script, or a Python script, it does not fire the command blindly. The template includes `run_all.sh`, a shell script that wraps every execution with structured logging and feedback.

Here is what happens when the AI runs a script:

1. You say "run Step 3" (or use the `/run` command).
2. The AI calls `./run_all.sh "Step_3_setup.do"`.
3. `run_all.sh` detects the file type (`.do` → Stata, `.R` → R, `.py` → Python) and launches the appropriate tool in batch mode.
4. All output is captured to a timestamped log in `output/logs/`.
5. The log opens automatically so you can see what happened.
6. The AI reads the same log, checks for errors and warnings, and reports what it finds.

You see the tool running. The AI sees the same output you do. This is how it can tell you "row 4,312 has a missing FIPS code" instead of just "the script ran."

What this looks like with Stata

The `run_all.sh` script creates a wrapper do-file that sets up proper logging, runs your script, and saves the log to `output/logs/`. When the run completes, the log opens in your default text editor. If you are working in VS Code with the terminal on the left, you can watch the log appear in a new tab while the AI reads it and reports results in the terminal beside it.

The `CLAUDE.md` file enforces this workflow. The AI is instructed to always run scripts through `run_all.sh`, always read the log afterward, and never assume success without checking. If a script fails, the AI reads the error from the log and proposes a fix.

Pipeline tracing

Before the AI modifies any script, it traces dependencies in both directions:

- **Upstream:** What data does this script read? What created that data?
- **Downstream:** What does this script produce? What consumes it — other scripts, tables, the manuscript?

This prevents the common problem of fixing one script and breaking another. The dependency chain is documented in the README pipeline table, and the AI checks it before making changes.

The core loop

Every research task follows the same pattern:

1. **You describe what you need.** Plain English. "Write a script that merges the treatment and control datasets on county FIPS codes." Be specific about inputs and outputs.
2. **The AI proposes a plan.** It reads your existing scripts, checks the pipeline table, and tells you what it intends to create or modify. If the change affects the project's structure, the AI is required to stop and warn you before proceeding. See "Break the glass" below.
3. **The AI writes the code.** It follows your project conventions: script headers, naming patterns, parameter references. It creates the script, updates the README pipeline table, and registers the step in `00_run.do` and `run_all.sh`.
4. **You run the script.** Either directly or through the AI:

   ```
   /run 05_merge
   ```

5. **The AI reads the log.** It checks for errors, warnings, and unexpected output. It reports what it finds.
6. **You review and iterate.** If the output is wrong, you describe the problem. The AI fixes the code and you run again.

Break the glass

The `CLAUDE.md` file includes a safety mechanism for high-impact changes. If you ask the AI to do anything that would alter:

- **The pipeline** — adding, removing, or reordering scripts; changing what a script reads or writes
- **Core parameters** — modifying `params.do` or the Parameters table in the README
- **The master scripts** — changing `00_run.do` or `run_all.sh`
- **The AI's own instructions** — editing `CLAUDE.md` itself

…the AI will stop and warn you before doing it. It will tell you exactly what it plans to change and what will be affected downstream. It will not proceed until you confirm.

This exists because pipeline changes cascade. If you rename a data file that three scripts depend on, those scripts will break. If you change a parameter that feeds into your estimation, every table and figure downstream may need to be regenerated. The AI knows this and will flag it.

For routine work — writing a new script, editing prose, fixing a bug, generating a table — the AI just does it. The warning only fires when the change touches something structural.

Long tasks and sub-agents

Some tasks — rebuilding an entire pipeline, reorganizing a messy project, processing a dozen scripts — take longer than a single conversation. AI tools have a finite context window, and a sprawling session can lose focus.

When you have a large task, you can ask the AI to break it into pieces and spawn separate agents for each one. For example: *"I need to clean and standardize all six raw data files. Spin up an agent for each one."* The AI will launch parallel sub-tasks that work independently and report back. You stay in the main session, reviewing results as they come in.

This is not something you need to learn the mechanics of right now — the tools handle the details. Just know the capability exists. When a task feels too big for one session, tell the AI to decompose it.

Example session

Here is what an actual session looks like. You type the lines after `>`. Everything else is the AI responding.

```
> I need to import the raw CDC WONDER data and clean it. The file
> is data/raw/cdc_wonder_2015_2022.txt. It's tab-delimited with
> a header row. I want to keep county FIPS, year, and death count.
> Drop any rows with "Unreliable" in the notes column.
```

The AI reads the raw file to understand its format. It writes `scripts/01_import_cdc.do` with a proper header, the import logic, and a save to `data/processed/cdc_clean.dta`. It updates the README pipeline table, adds the step to `00_run.do`, and registers it in `run_all.sh`.

```
> /run 01_import_cdc
```

The script runs. The AI reads the log and reports: "Imported 15,847 rows. Dropped 312 with 'Unreliable' flag. Saved 15,535 observations to `data/processed/cdc_clean.dta`."

You check the numbers. They make sense. You move on.

Commands

The template includes built-in commands that handle common tasks:

Command	What it does
`/init`	Initializes a new `CLAUDE.md` in the current directory. Use for non-template projects.
`/status`	Scans the project. Reports pipeline state, last run dates, conflicts, uncommitted work.
`/run`	Runs a pipeline step. Validates the script exists and logs the output.
`/run --all`	Runs the full pipeline from start to finish.
`/check`	Full integrity audit. Verifies scripts, data, params, and manuscript references all match.
`/add-step`	Scaffolds a new pipeline step. Creates the script, updates README, `00_run.do`, and `run_all.sh`.
`/git`	Stages, commits, and pushes all changes to GitHub.
`/handoff`	Writes a session summary to `session_logs/`. Useful when ending a work session or handing off to a co-author.

Working with data

Protect your raw data from the AI

> **Important If you have proprietary, restricted-use, or individually identifiable data, the AI should never see it.**

This matters more than anything else in this guide. If your data is covered by a data use agreement (DUA), HIPAA, or any other access restriction, you must do your data cleaning, de-identification, and aggregation **before** the AI has access to any of it. That means:

1. **Write your data cleaning and collapsing code offline.** The AI can help you write the code — describe what you need, let it draft the script — but you run that code yourself, outside of the AI session, on your own machine. Do not give the AI a path to read the raw files.
2. **Only give the AI access to aggregated or de-identified data.** Once your data is collapsed to the level where it is safe — no individual records, no identifiers, within the bounds of your DUA — then you can let the AI read it, work with it, and help you analyze it.
3. **Consider physical separation.** One approach: keep your raw disaggregated data on an external drive. Do not even have the drive plugged in while working with the AI. This makes it impossible for the AI to access raw data, even accidentally.

The template's `data/raw/` folder is marked read-only in `CLAUDE.md`, and the AI is instructed never to modify it. But "read-only" still means the AI can read the files if they are on a connected drive. For truly sensitive data, physical separation is the safest approach.

> **Warning `data/raw/` is sacred** — Never modify raw data. The AI knows this rule (it's in `CLAUDE.md`). If you ask it to edit a file in `data/raw/`, it will refuse and explain why.

The typical data workflow

1. Place raw data in `data/raw/` with a README documenting its source and access date.
2. Write import/cleaning scripts that read from `data/raw/` and save to `data/processed/`. If your raw data is restricted, **run these scripts yourself offline** — not through the AI.
3. Once your processed data is safe for the AI to see, write analysis scripts that read from `data/processed/` and save results to `output/`. The AI can run these.

Each script has a structured header that documents its purpose, inputs, outputs, and dependencies. The AI writes these headers and reads them before modifying any script.

Working with parameters

Research parameters — treatment dates, sample restrictions, outcome definitions — live in `scripts/params.do`. They also appear in the README's Parameters table. The two must match.

When you need to change a parameter:

```
> Change the sample start year from 2015 to 2016. Update params.do
> and the README.
```

The AI modifies both files and verifies consistency.

Git is your undo button

If you have never let software modify your research files before, this is the most important thing to understand: **every change the AI makes is reversible.**

When the AI edits a script, creates a file, or modifies your README, that change exists in your file system — visible, inspectable, and undoable. If you are using Git (Path A), you have a complete history of every change and can undo any of them:

```
git diff                          # see what changed
git checkout -- scripts/05_merge.do   # revert one file
git stash                         # undo all uncommitted changes (keeps them saved)
```

This is why the workflow uses `/git` to commit after each working session. Each commit is a snapshot. If something goes wrong three sessions later, you can go back to any previous snapshot.

Even if you are not using Git (Path B), the AI is working on files in your Dropbox folder. Dropbox keeps version history. You can restore any file to a previous version through the Dropbox website.

The point: **you are not handing control to the AI.** You are letting it make changes that you can see, review, and reverse. The common fear is "the AI will destroy my project." In practice, the usual bad outcome is "the AI made a change I don't like" — and if you are using Git, reverting it is straightforward.

Session hygiene

AI sessions degrade over time. The longer a session runs, the more context the AI accumulates, and eventually it starts losing track of earlier decisions, repeating itself, or making mistakes it would not have made at the start. This is a property of how context windows work, and it affects every AI tool.

Keep sessions focused. One task per session is ideal. "Reorganize the pipeline" is a session. "Write the estimation code" is a different session. Do not try to do both in one sitting — the quality of the second task will suffer because the AI's context is full of the first.

Watch for signs the session is going sideways: - The AI proposes changes you already discussed and rejected - It forgets file names or paths it was using earlier - It starts writing code that contradicts its own earlier output - Responses get slower or more generic

When you see these signs, it is time to reset. You have two options:

- **Clear the context without leaving.** Type `/clear` in the same terminal. This wipes the AI's context window and starts a fresh conversation — but you stay in the same terminal, in the same folder. The AI re-reads `CLAUDE.md` and orients itself. This is the fastest way to reset.

- **Run `/handoff` first.** If you want a record of what happened before clearing, run `/handoff` to save the session state, then `/clear` to start fresh. The new conversation will read `CLAUDE.md` and the session log, and pick up where you left off with a clean context window.

You can run multiple sessions at once. Open two or three terminal tabs in the same project folder, each running its own AI session. One tab cleans data while another writes documentation. They are working on the same files, so coordinate — do not have two sessions editing the same script simultaneously. But for independent tasks, parallel sessions can save real time.

Use `/handoff` liberally. Even if you are not done for the day, a handoff captures what happened so the next session (or the next day) starts with a clear summary instead of a vague memory. Think of it as saving your game.

Commit before and after. Run `/git` at the start of a session (to capture the baseline) and at the end (to capture your work). If a session goes badly, you can revert to the start-of-session commit and try again.

Ending a session

When you are done working:

```
> /handoff
```

This writes a summary of what changed during the session: files modified, outputs regenerated, open questions. The summary is saved to `session_logs/` and is useful for:

- Picking up where you left off in a future session
- Briefing a co-author on recent changes
- Maintaining a record of project evolution

Then save your work:

```
> /git
```

This stages, commits, and pushes to GitHub.

Writing with AI

The role of AI in academic writing

AI writes fast and clean. It does not write with insight. Here is how I divide the labor:

- **You** supply the argument, the interpretation, the economic intuition.
- **The AI** handles structure, formatting, style enforcement, and mechanical editing.

This chapter covers how to use AI for manuscript work without producing prose that reads like it was generated by a machine.

The writing standard

The template follows McCloskey's *Economical Writing* principles. These are baked into `CLAUDE.md`, so the AI follows them automatically:

1. **Active verbs.** "Prices increased" — not "an increase in prices occurred."
2. **Concrete language.** "Machines and workers" — not "capital and labor inputs."
3. **Plain words.** "Use" — not "utilize."
4. **Delete ruthlessly.** Cut "very," "basically," "actually," "it is important to note that."
5. **One word, one meaning.** Pick a term and keep it throughout the paper.
6. **End strong.** The last word in a sentence carries the emphasis.
7. **No boilerplate.** Never open with "This paper discusses…" or "The remainder of this paper is organized as follows."
8. **Tables and graphs are writing.** Same rules apply to labels, titles, and notes.

AEA style

The template includes `manuscript/aea_style_guide.md` with detailed AEA formatting rules. The AI reads this file before making any edits to the manuscript. Key points:

- Equations: numbered, referenced as "equation 1" (lowercase, not abbreviated)
- Tables: all in the body (not an appendix), landscape if needed, with descriptive notes
- Figures: high-resolution, clear labels, source notes
- Citations: author-date format via `natbib`
- Numbers: spelled out below 10 in text, digits in tables

How to use AI for writing tasks

Drafting sections

Give the AI your argument and let it draft:

```
> Write the data section. We use CDC WONDER mortality data
> from 2015-2022, county-level, restricted to accidental drug
> poisoning deaths (ICD-10 X40-X44). We merge with ACS county
> demographics. Explain the sample construction and any
> exclusions. Keep it under 400 words.
```

Review the draft. The AI will follow your style guide, but you need to verify that the substance is correct. Add your interpretation. Cut anything generic.

Editing existing prose

```
> Edit section 3.2. Tighten the language. Apply McCloskey
> rules. Don't change the argument, just the writing.
```

The AI will shorten sentences, activate passive voice, remove filler words, and improve flow. Review every change — occasionally the AI will alter meaning while trying to improve style.

Tables and figures

```
> Write the LaTeX for Table 1. It should show summary statistics
> for treatment and control counties: mean, sd, min, max for
> population, income, death rate. Add a note explaining the
> sample and time period.
```

The AI writes the `.tex` fragment and saves it to `output/tables/`. The manuscript `\inputs` it directly.

Citation management

If your Zotero auto-export is configured, the AI can add citations:

```
> Cite Ruhm (2019) for the claim about economic conditions
> and drug mortality. Use the BibTeX key from references.bib.
```

The AI searches your `.bib` file, finds the key, and inserts the `\cite{}` command.

What to watch for

- **Bland hedging.** AI defaults to "may," "could," "potentially." Replace with direct claims supported by your evidence.
- **False precision.** AI sometimes invents specific numbers or citations. Verify every factual claim.
- **Generic framing.** Opening paragraphs that could describe any paper. Delete and write your own.
- **Over-signposting.** "In this section, we will discuss…" — just discuss it.

Tip A useful test — Read the AI's draft aloud. If it sounds like a committee wrote it, rewrite it. Good academic writing has a voice. The AI can help you find yours by cleaning up your rough drafts, but it should not replace your voice entirely.

Auditing your draft: Referee 2

Once you have a working draft — organized, coded, with tables and figures in place — you need to stress-test it before submitting. You cannot grade your own homework, and the AI that helped you write it should not be the one reviewing it.

Referee 2 is an open-source audit protocol created by Scott Cunningham (of *Causal Inference: The Mixtape*) that uses AI to perform five systematic audits on your research. For Cunningham's own account of how he uses Claude Code for empirical research and where Referee 2 fits, see his Claude Code for Empirical Research post.

The five audits:

1. **Code audit.** Identifies coding errors, missing value problems, and logic gaps in your scripts.
2. **Cross-language replication.** Creates independent implementations of your analysis in R, Stata, and Python. The key insight: hallucination errors are likely *orthogonal* across AI-produced code in different languages. If your Stata script has a subtle bug, an independent R implementation will likely produce a *different* bug — making discrepancies easy to catch by comparing outputs to six or more decimal places. You cannot find errors by re-reading the same code that produced them.
3. **Directory audit.** Checks that your replication package is submission-ready: every file accounted for, every path working, every dependency documented.
4. **Output automation audit.** Verifies that all tables and figures are programmatically generated from code, not manually edited.
5. **Econometrics audit.** Reviews specification coherence, identification strategy, standard error choices, and sample definitions.

The key principle is independence. Referee 2 operates as a separate session — it reads your project but never modifies your code. It produces a formal referee report and saves replication scripts to `code/replication/` for your review.

How to use it

After your draft is complete and your pipeline runs cleanly:

1. Open a new AI session in your project folder.
2. Load the Referee 2 persona from the MixtapeTools personas directory.
3. Point it at your manuscript, scripts, and output.
4. Review the report. Fix what needs fixing. Repeat.

Think of it as a pre-submission peer review — rigorous, structured, and designed to catch the things you are too close to the project to see.

Commands Cheatsheet

Quick reference for every command available in the Research Project Flow template.

Project commands

These commands work inside any project built from the template.

`/init`

Initializes a new `CLAUDE.md` file in the current directory. Use this to set up the AI's project instructions from scratch.

```
> /init
```

When to use: If you are starting a project without the template, or if you want to create a fresh `CLAUDE.md` tailored to your project. The template already includes a `CLAUDE.md`, so you only need this for new or non-template projects. You can also borrow the `CLAUDE.md` from this template and adapt it.

`/status`

Scans the project and reports its current state.

```
> /status
```

Shows: pipeline steps and last run dates, uncommitted changes, Dropbox conflicts, stale logs, scratch files.

When to use: Start of every session. The AI runs this automatically when you launch Claude Code in a project with `CLAUDE.md`.

`/run`

Executes a pipeline step with logging and validation.

```
> /run                    # prompts for which step
> /run 05_merge           # runs a specific step
```

```
> /run --all              # runs the full pipeline
> /run --from 05          # runs from step 05 forward
```

What it does:

1. Validates the script exists
2. Executes it via `run_all.sh`
3. Captures the log to `output/logs/`
4. Reads the log and reports results

`/check`

Full integrity audit of the project.

```
> /check
```

Verifies:

- Every script in the pipeline table exists and has a valid header
- Script inputs/outputs match the pipeline table in the README
- `params.do` values match the README Parameters table
- Referenced data files exist
- Manuscript `\input` references point to existing files
- Table/figure map matches actual output files

`/add-step`

Scaffolds a new pipeline step end-to-end.

```
> /add-step
```

Prompts for: step number, language (Stata/R/Python), description, inputs, outputs.

Creates:

- The script file with a structured header
- An entry in the README pipeline table
- A registration in `00_run.do`
- A registration in `run_all.sh`

`/handoff`

Writes a session summary for future reference or co-author communication.

```
> /handoff
```

Produces: a markdown file in `session_logs/` with:

- Files created or modified
- Outputs regenerated

- Open questions or next steps
- Git status at time of handoff

`/git`

Stages, commits, and pushes all changes.

```
> /git
```

What it does:

1. Stages all tracked and new files
2. Writes a descriptive commit message based on changes
3. Commits
4. Pushes to the remote

No confirmation prompt — it runs immediately. Use when your project is in a clean, working state.

Terminal commands

These are standard terminal commands you will use alongside Claude Code.

Command	Purpose
`claude`	Launch Claude Code in the current directory
`cd ~/Dropbox/my-project`	Navigate to your project
`git status`	Check for uncommitted changes
`git log --oneline -10`	View recent commits
`./run_all.sh "script_name"`	Run a pipeline step directly
`./run_all.sh --all`	Run the full pipeline directly

Keyboard shortcuts in Claude Code

Shortcut	Action
`Enter`	Send message
`Esc`	Cancel current operation
`Ctrl+C`	Exit Claude Code
`Up arrow`	Scroll through message history

FAQ

Getting started

Do I need to know how to code?

Not to get started. The AI writes code based on your descriptions. You need to understand what a script should do — what data goes in, what results come out — but you do not need to write it from scratch.

That said, **learning to write code is worth the effort** — not just reading it. The more you understand the syntax and logic of your statistical software, the better you can evaluate what the AI produces. This matters most where it matters most: estimation. See the next question.

Where is the AI good and where is it not?

Be honest with yourself about this. The AI is not equally reliable across all parts of the research workflow.

Where it excels:

- **Data cleaning and organization.** Importing, reshaping, merging, labeling — the AI does this very well. Not perfect, but consistently strong. This is the bread and butter of the workflow.
- **Visualization.** Charts, graphs, maps. The AI writes clean plotting code and iterates quickly on formatting. Post-estimation, once you know what the output means, the AI is a great tool for turning results into publication-quality figures.
- **Project scaffolding.** Folder structure, script headers, documentation, pipeline management — everything this template automates.

Where you must be hands-on:

- **Estimation code.** The AI often gets this wrong. Not just the logic — the *syntax*. A `reghdfe` command with the wrong absorb structure, a `felm` call with misspecified clusters, a `linearmodels` panel estimator with the wrong entity effects. You need to understand the software you are using well enough to catch these errors. If you hand estimation off to the AI without careful review, you risk accelerating bad code. This is where learning to write code pays off.
- **Writing.** The AI is good at telling you what it thinks results mean, and it can tighten prose and enforce style rules. But in my experience, the actual argument — the interpretation, the contribution, the narrative that holds a paper together — needs to come from you. No AI I

have used writes well enough to produce publishable academic prose without heavy revision. I recommend being hands-on for the writing that matters.

This workflow is designed for **resurrecting stalled projects** — organizing them, rebuilding the pipeline, getting the scaffolding right. It is not a replacement for critical thinking, and your mileage may vary.

Which statistical software should I use?

The template supports Stata, R, and Python. Use whatever your field and co-authors expect. Most economics departments use Stata. The template handles mixed-language pipelines, so you can use R for data visualization and Stata for estimation in the same project.

What if I want to use Word instead of LaTeX?

The template ships with a LaTeX manuscript setup, but the entire data pipeline — scripts, `run_all.sh`, logging, parameters, version control — has nothing to do with your manuscript format. If you never touch LaTeX, all of that still works.

The limitation is the manuscript integration. The AI can read and edit `.tex` files the same way it reads and edits code — they are plain text. A `.docx` file is a binary format. The AI cannot open your Word document and tighten a paragraph the way it can with LaTeX or markdown. You also lose the live-compile workflow where your PDF updates in a VS Code tab as the AI edits the source.

My recommended workaround: write in markdown. Markdown has almost no learning curve — if you can write an email, you can write markdown. The AI can read and edit markdown files natively, so you keep the full "AI helps with prose" workflow from the Writing with AI chapter. When you are happy with the content, convert to Word as a final step:

```
pandoc manuscript.md -o manuscript.docx
```

If your journal or co-author requires a specific Word template, pandoc supports that too:

```
pandoc manuscript.md -o manuscript.docx --reference-doc=journal_template.docx
```

This way you draft and iterate with the AI in markdown, and producing the `.docx` is a one-command conversion at the end. You skip the upfront investment of learning LaTeX, and you keep the AI in the loop for writing tasks.

The other option: just split the labor. Let the AI handle the pipeline and code. Do your writing in Word yourself. Skip the "Writing with AI" chapter and treat the manuscript as your domain. That is a perfectly valid way to use this workflow — you are still getting the bulk of the value from the data pipeline side.

How much does this cost?

This is an AI-heavy workflow. You will use tokens quickly when the AI reads data files, writes scripts, edits manuscripts, and runs your pipeline. Be realistic about that upfront.

At minimum, you need a paid subscription. Claude Code works with a Claude Pro account ($20/month), which comes with a set number of tokens per hour, per day, and per month. The free

tier will not get you through a working session. Other tools (Gemini CLI, Codex) have their own subscription tiers — check their pricing.

I recommend starting with a higher tier. If you are trying to get a stalled project unstuck, treat the higher-tier subscription (Claude Max at $100/month, or equivalent) as part of the cost of getting your research moving. Dive in, work intensively for a few weeks, then decide whether the ongoing cost is worth it. For me, the value was obvious after one real session — but I had to invest enough to reach that point.

You can monitor your token usage at console.anthropic.com to see how quickly you burn through your allocation. If you hit your limit mid-session, the tool pauses until your tokens refresh — you do not get surprise charges.

Can I use this with ChatGPT or other AI tools?

The template ships with `CLAUDE.md` and slash commands designed for Claude Code. Other terminal-based AI tools (Gemini CLI, Codex) work with this template too — they read and write files directly in your project the same way. Chat-based tools like ChatGPT operate differently and cannot interact with your file system.

What about auto-accept / "YOLO" mode?

All three tools have a mode that auto-approves every action without asking you first:

Tool	Command
Claude Code	`claude --dangerously-skip-permissions`
Codex	`codex --yolo`
Gemini CLI	`gemini --yolo`

I do not recommend this, especially for anything involving estimation code. In auto-accept mode, the AI may decide it needs to do extensive web research and disappear down a rabbit hole for a long time. It may wander into other folders on your machine and write itself helper scripts you did not ask for. It may modify files outside your project. You lose the ability to review each action before it happens — which is the whole point of the guardrails in this workflow.

If you do use it, you can confine the AI to specific folders:

- **Claude Code:** Set `permissions.deny` rules in `.claude/settings.json` (e.g., `"Edit(data/raw/**)"` to block raw data edits), or use `--disallowedTools` to block specific tool categories.
- **Codex:** Use `--sandbox workspace-write` to restrict writes to your project directory.
- **Gemini CLI:** Use `--sandbox` for OS-level isolation, or `--allowed-tools` to restrict which tools auto-approve.

These commands exist. Use them as you see fit. But for the parts of the workflow that matter most — estimation and writing — stay hands-on.

Working with the template

How do I add a co-author?

1. Share the Dropbox folder with them.
2. Add their machine path to `scripts/00_run.do`.
3. Point them to the Co-Author Instructions section in the README.

They do not need Claude Code to contribute. They edit files in Dropbox, and you commit the changes to Git on their behalf.

I have a project in Overleaf. How do I bring it over?

1. In Overleaf, go to **Menu → Download → Source**. This gives you a zip file with all your `.tex`, `.bib`, figures, and style files.
2. Unzip and sort into the template structure:
   - `.tex` files → `manuscript/`
   - `.bib` file → `manuscript/references.bib`
   - Figures → `output/figures/`
   - Style files (`.cls`, `.sty`, `.bst`) → `manuscript/`
3. Update the file paths in your `.tex` file. Overleaf keeps everything flat in one folder, but the template separates manuscripts, figures, and tables into different directories. The `\includegraphics` and `\input` paths will need to change.

The AI is good at that third step. Once you have the files in place, tell it: *"I pulled these files from Overleaf. Update all the paths in `manuscript.tex` to match the template structure."* It will find every `\includegraphics`, `\input`, and `\bibliography` command and fix the paths.

What if the AI modifies something it shouldn't?

The `CLAUDE.md` file includes rules that prevent the AI from modifying raw data or making unauthorized structural changes. If the AI proposes something you do not want, say no. It will adjust.

If the AI does make an unwanted change, Git makes it trivial to undo:

```
git diff              # see what changed
git checkout -- path/to/file   # revert a specific file
```

How do I handle sensitive or restricted data?

- Never commit sensitive data to Git. Add the file paths to `.gitignore`.
- Document access instructions in `data/raw/README.md`.
- Use the Data Availability Statement in your README to describe restrictions.
- The AI operates locally on your machine — it does not upload your data anywhere.

What if the pipeline breaks?

Run `/check` to identify the problem. The AI will report which scripts, data files, or references are inconsistent. Common issues:

- A script references a file that was renamed → update the script or rename back
- `params.do` and README disagree → reconcile the values

- A log shows an error → read the log, fix the script, re-run

---

Git and GitHub

I have never used Git. Is that a problem?

No. The `/git` command handles staging, committing, and pushing. You do not need to learn Git commands to use the template. But understanding what Git does — tracking changes, enabling undo, syncing with GitHub — will help you appreciate why the template uses it.

Can I use a private repository?

Yes. When you clone the template, use `--private`:

```
gh repo create my-project --template Black-JL/Research-Project-Flow --private --clone
```

Your code and data descriptions stay private. Only people you explicitly invite can see the repo.

What goes on GitHub vs. Dropbox?

- **GitHub:** Code, scripts, documentation, manuscript source, configuration files. Everything text-based and version-controlled.
- **Dropbox:** Large data files, binary outputs, anything too big for Git. Dropbox provides the backup and sharing layer.

The `.gitignore` file controls what Git tracks. Large data files should be excluded from Git and shared via Dropbox.

Common mistakes

Things I wish I had known

These are patterns that tripped me up, and that I have seen others run into. Learn from them before you repeat them.

Giving the AI too many tasks at once. "Reorganize all my scripts, rebuild the pipeline, update the README, and fix the estimation code" will produce mediocre results across the board. One task per request. Let it finish, review the output, then move on.

Letting sessions run too long. After an hour or two of active work, the AI's context window is full. It starts forgetting what it did earlier, contradicting itself, or getting slower. Run `/handoff`, close the session, and start fresh. A clean session with a good handoff note is more productive than a marathon session that degrades.

Trusting the log summary instead of the log. When the AI says "the script ran successfully and produced 15,000 observations," read the actual log. The AI is usually right, but "usually" is not good enough for research. The log is in `output/logs/`. Read it.

Not stating what should stay the same. When you ask the AI to edit a script, it may "improve" things you did not ask it to touch. Always say what should not change: "Edit the formatting. Do not change any estimation commands or sample restrictions."

Skipping the first `/check`. Before you start real work in a session, run `/check`. It takes a few seconds and tells you whether the pipeline is consistent. Finding out that something is broken *after* you have been building on top of it for an hour is painful.

Using AI for the parts that require judgment. Data cleaning and visualization? Let the AI drive. Estimation code and manuscript writing? You drive. In my experience, the AI works best for the mechanical parts of research. The intellectual parts are where your judgment matters most. See Where is the AI good and where is it not? above.

Working with co-authors

My co-author does not use AI. Is that a problem?

No. Some co-authors use AI, some do not. Some write code, some do not. The workflow accommodates all of these.

Your co-author sees the same project folder you do — scripts, data, output, manuscript. Every file is a normal file in a normal folder. Nothing about the project depends on AI to function. If your co-author wants to open `scripts/10_balance.R` and edit it by hand, they can. If they want to run the pipeline themselves, `./run_all.sh --all` works without any AI tool installed.

The question your co-author will have is: *"How do I know what the AI changed?"*

The answer is in three places:

1. **Session logs** (`session_logs/`). Every working session produces a dated log that says what the AI did and why. Your co-author reads these the same way they would read your lab notebook.
2. **Git history.** Every commit has a message describing the changes. `git log --oneline` shows the history. `git diff` shows exactly what changed in any file.
3. **The code itself.** Every script has a structured header documenting its purpose, inputs, outputs, and dependencies. The code is readable, conventional, and follows the same patterns whether you or the AI wrote it.

If your co-author is skeptical — and healthy skepticism is appropriate — point them to the session logs and the git history. The audit trail is the whole point of this workflow. You can explain and defend every change because every change is documented.

How do I leave notes for my co-author?

The `/handoff` command is designed for this. When you finish a working session, `/handoff` writes a date-labeled summary to `session_logs/` that lists everything you did: files created, scripts modified, outputs regenerated, open questions. Tell your co-author: "Go look at the session logs." They are clean, dated, and live in one place in the project structure — no stray files accumulating in random folders.

If you want something more tailored, you can also just ask the AI directly: *"I'm wrapping up. Create a markdown file called* ***notes_for_sarah.md*** *in the project root. Summarize everything we changed today, plus these additional items I'm about to list."* The AI will create the file with whatever you tell it, and you can text your co-author a link or just say "everything I did is in `notes_for_sarah.md`."

Both approaches work. I recommend the session logs as your default because they are consistent — same format, same location, same naming convention, every session. They build up into a clean record of the project's evolution rather than a collection of one-off notes. But for a quick "here's what changed today" message to a specific person, a custom markdown file takes ten seconds to create.

How do I bring this up with a co-author?

Be direct: *"I've been using an AI coding tool to organize the project and write data cleaning scripts. Every change is tracked in version control and documented in session logs. The estimation code and the writing are mine. Here's the session log from the last working session if you want to see what it did."*

The audit trail is there for exactly this conversation. Anyone who wants to verify what the AI did can read the session logs, check the git history, and inspect the code. That is the point of the workflow.

Troubleshooting

Claude Code says "command not found"

Your PATH is not set up correctly. Check that Node.js and Claude Code are installed:

```
node --version
which claude
```

If `claude` is not found, reinstall:

```
npm install -g @anthropic-ai/claude-code
```

The AI seems confused about my project

It probably has not read `CLAUDE.md`. Make sure the file exists in your project root and contains the correct project path. Launch Claude Code from the project directory:

```
cd ~/Dropbox/my-project
claude
```

Scripts run in Terminal but not through `run_all.sh`

Check that `run_all.sh` is executable:

```
chmod +x run_all.sh
```

Also verify that the statistical software is accessible from the command line. Stata in particular may need a PATH addition (see Tools & Setup).