Agentic Coding for Economics Research

James Okun

MIT

March 18, 2026

Background

- Proliferation of LLMs over the past few years
- Incredibly useful for coding. Leading AI companies have developed models specifically for writing code. These are more powerful than the usual chatbots
  - e.g., codex, claude code
- These tools are not just autocomplete: they can plan multi-file changes, run tests, and iterate on errors autonomously

Plan for today

1. Setting up your project
2. The economics of tokens
3. Skills and subagents
4. Reaching beyond your codebase (MCP)
5. Safety and control
6. Putting it all together

What are agentic coding tools?

- Command-line interface (CLI) tools that use large language models to edit, write, and explain code
- They can operate inside your IDE as an extension or in your terminal as a standard CLI tool. IDE extension runs the CLI tool under the hood but might not have access to all the same settings (this may change)
- The tool lives in your project folder and has full context of your entire codebase and anything else you put there. Once installed there will be a `.claude` folder in your root directory. The model and its settings exist from within this folder

Chat tools vs. agentic coding tools

Chat tools (ChatGPT, Claude, Gemini)

- Good for brainstorming, explaining concepts, writing isolated functions
- No access to your codebase. You copy-paste snippets in and out
- Context resets every conversation
- Fine for quick one-off questions

Agentic tools (Claude Code, Codex)

- Full context of your project directory
- Can read, write, and execute code autonomously
- Persistent configuration via `.claude/` folder
- Skills and agents carry your preferences across sessions

Rule of thumb: Use chat for learning and exploration. Use agentic tools for building and maintaining a project.

Tools I recommend

1. Anthropic's Claude Code
   - Opus 4.6 is the best model in my opinion. Opus 4.6 is token hungry which makes Claude Code unusable without the $100 per month subscription. The fact that it is token hungry is what makes it the best (in my opinion)
2. OpenAI's Codex
   - You have codex if you already pay $20 per month for ChatGPT. This model is usable with only this subscription.

- I started my journey with codex in November 2025. I started using Claude Code in late January and I am now fully convinced that it is better and it's worth paying the $100 per month subscription. Overall, the models are similar and their relative performance fluctuates. You might just like the UI, flow, style of one over the other. Try both and see what works for you.

Setting Up Your Project

Project constitution, configuration, and workflow

The `.claude/` project directory

When you install Claude Code in a project, it creates a `.claude/` folder in your root directory. This is where all configuration lives.

.claude/	
`settings.json`	permissions, hooks, allowed tools
agents/	
`julia-coder.md`	subagent definition
`data-explorer.md`	subagent definition
skills/	
`econ-regression-table/`	SKILL.md + scripts/
commands/	custom slash commands
`CLAUDE.md`	project constitution (root level)

Everything the agent knows about your project is defined here.

Starting out with a project constitution

Every agentic project needs a `CLAUDE.md`: a top-level markdown file that tells the agent who it is and how you work.

- **`CLAUDE.md`** — Claude Code reads this automatically on startup. It sets global rules: coding style, preferred libraries, file conventions

What goes in a good `CLAUDE.md`?

Project purpose, key packages, directory structure, coding conventions (naming, file organization, path constants), and agent usage instructions (which subagents to prefer and when). See agents.md for a curated collection of real-world examples.

Example `CLAUDE.md` for this project

```
# NursingHomeAdmissions

## Key packages
CSV, DataFrames, GLM, FixedEffectModels, CairoMakie

## Coding conventions
- Scripts numbered 01_, 02_, ...  ; each includes 00_setup.jl
- Use DATA_RAW, DATA_PROC, OUTPUT_DIR constants (not hard-coded paths)
- snake_case for functions, PascalCase for types

## Agent usage
- julia-coder:  all code writing and refactoring
- data-explorer:  dataset profiling, EDA, missingness, merge keys
- Only fall back to general-purpose if task doesn't fit either
```

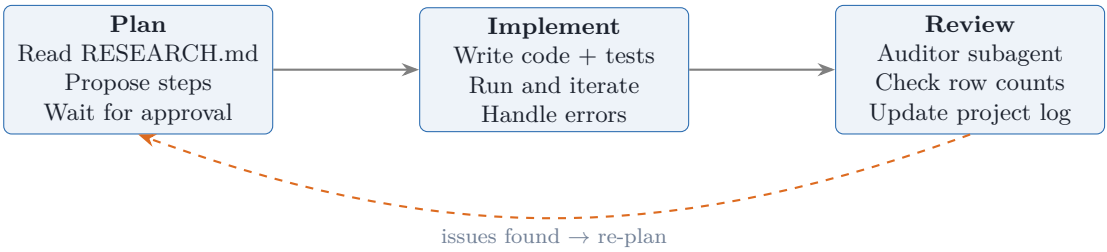
Build planning, testing, and review into your workflow

A good agentic workflow has three phases, and the agent should follow them by default:

1. **Plan** — Before writing any code, the agent reads existing context (`RESEARCH.md`, relevant data dictionaries) and proposes a step-by-step plan. You review and approve before it proceeds.
2. **Implement** — The agent writes code, runs it, and iterates on errors. It should write tests alongside the implementation, not after.
3. **Review** — The agent (or a dedicated auditor subagent) checks the output: do row counts make sense? Are merge keys unique? Does the regression output match expectations? Then it updates the project log.

A good existing plugin for doing this by default is called superpowers (usable in Claude Code and Codex; there is an extended version for Claude Code too).

Example: plan–implement–review in practice



The feedback loop is key. If the review step finds problems, it cycles back to planning rather than blindly patching.

The Economics of Tokens

Why context management matters for cost, speed, and quality

Background: What Are Tokens?

LLMs don't read raw text. They first split it into **tokens**, small chunks that are roughly $\frac{3}{4}$ of a word on average.

Example tokenization

`"Regression discontinuity"` $\longrightarrow$ [Reg] [ression] [dis] [continu] [ity] (5 tokens)

Why tokens matter:

- **Pricing** — API costs are quoted per token (e.g. $5 per million input tokens).
- **Context window** — every model has a finite token budget (e.g. 200K tokens). Prompts, tool metadata, conversation history, and the model's own reply all draw from the same budget.
- **Performance** — the more tokens in the window, the slower and more expensive each response, and the harder it is for the model to focus on what matters.

Token efficiency is therefore central to the design of skills and subagents.

The Cost of Context: Three Dimensions

Every token in the context window costs you along three dimensions:

Monetary Cost	Latency	Quality
You pay per token (API or usage limits)	More input tokens $\Rightarrow$ slower responses	Irrelevant context degrades output quality

All three compound with every additional message in a conversation.

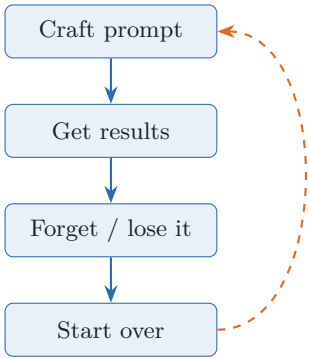
The Prompt Engineering Cycle

A familiar loop for LLM users:

1. Craft a great prompt → excellent results
2. Days later, need the same behavior again
3. Start prompting from scratch
4. Eventually save the template somewhere
5. Still need to find, paste, and tweak it each time

This workflow is fundamentally broken.

Skills and subagents are the solution: write your expertise once, and the agent applies it automatically in every future session.



Skills and Subagents

Solving the prompt engineering cycle and context bloat

Skills: Reusable, Lazy-Loaded Instruction Sets

Skills are markdown files with metadata and a body of instructions, placed in a `.claude/skills/` directory.

Minimal `skill.md` structure

```
--
name: <skill-name>
description: <short-skill-description>
--
<detailed instructions>
```

Key properties:

- **Auto-invoked** — the agent reads metadata at startup and loads the full body only when relevant.
- **Reusable** — write once, applied across every future interaction.
- **Lazy-loaded** — context enters the window progressively, not all at once.

Progressive Disclosure: Three Levels of Loading

The agent progressively discloses information *to itself*, pulling in only what the current step requires.

Example skill: `econ-regression-table`

Metadata (Level 1):

```
--
name:  econ-regression-table
description:  Use when the user
  wants to create a regression
  table, format regression results
  for a paper, or export
  regressions to LaTeX...
--
```

Body (Level 2):

- Journal conventions: booktabs, no stars, SEs in parentheses
- Bottom panel: FE indicators, dep. var. mean, N

Script (Level 3):

- `scripts/regression_table.jl`
- Loaded only when actually building a table

Output:

	(1)	(2)	(3)
Medicaid	−0.142	−0.138	−0.125
	(0.031)	(0.029)	(0.028)
Private pay	0.087	0.091	0.078
	(0.024)	(0.023)	(0.022)
Facility FE	No	Yes	Yes
Year FE	No	No	Yes
Dep. var. mean	0.523	0.523	0.523
Observations	12,480	12,480	12,480

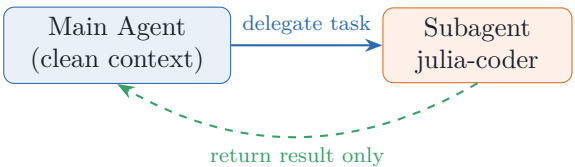
The agent produces this table automatically when you say "make a regression table". No prompt engineering needed.

Subagents: Isolated, Specialized Workers

Subagents are child agents with their own context window and tool set.

Two key properties:

1. **Isolated context** — starts with a clean window; intermediate reasoning is discarded after the task.
2. **Isolated tools** — only the skills and tools it needs are loaded; the main agent pays nothing for them.



Subagents are **lazy-loaded workers**: the main agent knows its subagents and only uses them when needed.

Why subagents matter: context isolation

When a subagent profiles a dataset, it might inspect dozens of columns, try several merge keys, and hit dead ends.

Without subagents:

- Intermediate reasoning stays in the main context
- Every subsequent message is slower and more expensive
- Irrelevant exploration degrades future output quality

With subagents:

- Dead ends and noise are discarded when the subagent finishes
- Only the final answer flows back
- Main agent's context stays clean and cheap

You still pay for the subagent's own token usage, but none of it compounds into the main agent's window.

Subagents in the nursing home project

`julia-coder` subagent

- Writes and refactors code as needed
- Writes great Julia code: type-stable, broadcasted, in-place where appropriate
- Uses `GLM.jl`, `FixedEffectModels.jl`, `CairoMakie`
- Runs on Sonnet (cheaper for routine coding tasks)

`data-explorer` subagent

- Dataset profiling: shape, types, summary stats
- Missingness analysis (counts of missing variables, patterns in missingness)
- Merge-key identification across audit and facility datasets
- Data quality flags: duplicates, mixed types

Both agents are defined in `.claude/agents/` and carry persistent memory. The main agent delegates to them by name and their intermediate work never pollutes the main context.

Concrete example: estimating payer treatment effects

Task: Estimate whether nursing homes selectively admit or reject applicants based on payer type (Medicaid vs. private-pay vs. no payer specified) using audit study data.

1. **Main agent** reads `CLAUDE.md`, understands the pipeline: `01_clean.jl` → `02_regressions.jl` → `04_tables.jl`.
2. Delegates to **data-explorer**: profile the raw audit data, check payer-type coding, facility IDs, missingness in outcome variables.
3. Delegates to **julia-coder**: write the regression specifications in `FixedEffectModels.jl` with robust SEs, produce coefficient tables via the `econ-regression-table` skill.
4. **julia-coder** generates BKY-adjusted q-values (`03_bky.jl`) for multiple-testing correction across treatment arms.
5. Main agent reviews output tables in `output/`, commits.

Each subagent works in isolation. The main agent's context never sees the intermediate data profiling or failed specification attempts.

Skills + subagents solve both problems

Recall the two problems we identified: the **prompt engineering cycle** (re-writing the same instructions every session) and the **cost of context** (tokens degrade cost, speed, and quality).

Skills fix the prompt cycle

- Write your expertise once in a `SKILL.md`
- The agent loads it automatically whenever it's relevant
- No copy-pasting
- Lazy loading means you only pay tokens when the skill is actually used

Subagents fix context bloat

- Messy intermediate work stays in the subagent's window
- Only the clean result returns to the main agent
- Main context stays small
- Each subagent loads only the skills it needs

Reaching Beyond Your Codebase

Integrating with external applications via MCP

Integrating with external applications via MCP

Skills and subagents handle *what happens inside your project.* But what if you need the agent to interact with external applications (GitHub, Slack, etc.)

Out of the box, the agent can only read your files, write code, and run shell commands. **MCP (Model Context Protocol)** gives it new capabilities through a standardized plug-in system.

Without MCP:

- Agent can only touch local files
- You manually copy data between tools
- "Create a PR" $\rightarrow$ you do it yourself

With MCP:

- Agent calls GitHub API directly
- Agent queries databases
- Agent searches documentation sites
- Agent posts to Slack, reads emails, etc.

Safety and Control

Permissions, hooks, and slash commands

Guardrails: permissions vs. hooks

The agent needs two kinds of constraints.

Permissions = access control

"Can the agent use this tool at all?"

- Binary gate
- Evaluated *before* anything runs
- Examples: allow `Read` but block `Write(.env)`; whitelist `Bash(git *)`

Hooks = behavioral guardrails

"When the agent uses a tool, what else should happen?"

- Logic that fires before/after a tool call
- Can block, modify, or add steps

Use both together: permissions decide *what the agent can touch*; hooks decide *what happens when it does.*

Hooks: deterministic guardrails for agent behavior

Hooks are shell commands that fire automatically at specific points. Unlike `CLAUDE.md` rules (which are advisory), hooks are *enforced gates*.

Key hook events:

- `PreToolUse` — fires *before* a tool runs; can **block** the action
- `PostToolUse` — fires *after* a tool runs; good for formatting and tests
- `Stop` / `SubagentStop` — fires when agent finishes; can force continuation
- `SessionStart` — inject environment variables or context at startup

Defined in `settings.json`:

```
{
  "hooks": {
    "PreToolUse": [{
      "matcher": "Bash",
      "hooks": [{
        "type": "command",
        "command":
          "./scripts/security-check.sh"
      }]
    }]
  }
}
```

Using hooks to enforce banned behaviors

Hooks turn "please don't" instructions into hard blocks. These are all implemented in this project's `.claude/settings.json`:

- **Block `rm -rf`** — a `PreToolUse` hook on `Bash` parses the command; if it matches `rm -rf`, exit 2 blocks the action.
- **Protect raw data** — a `PreToolUse` hook on `Write|Edit` checks if the target path is inside `data/raw/`; if so, block the write.
- **Restrict directory access** — a `PreToolUse` hook on `Bash|Write|Edit|Read` blocks any file access outside the project directory.
- **Run tests after code changes** — a `PostToolUse` hook runs `pytest` on modified files.
- **Force auditor review before stopping** — a `Stop` hook checks whether the auditor subagent has run; if not, exit 2 forces continuation.

Hook types: `command` (shell script), `prompt` (LLM evaluation), `agent` (spawns a verifier subagent with tool access).

Permissions and safety settings

Claude Code has a granular permissions system that controls what the agent can do without asking.

Permission modes:

- **Default** — asks permission for every file write, shell command, etc. (safe but tedious)
- **`allowedTools`** — whitelist specific tools: `"Read", "Grep"`
- **`deny`** — blacklist sensitive paths: `"Read(./.env)"`, `"Write(./output.*)"`
- **`-dangerously-skip- permissions`** — full autonomy (use with caution)

Practical advice:

- Start with defaults to understand what the agent tries to do
- Gradually whitelist common safe operations (reading files, running tests, git commands)
- Use `deny` rules to protect sensitive files (`.env`, credentials, raw data)
- Combine with hooks for defense in depth: permissions control *what* the agent can access; hooks control *how* it uses that access

Slash commands and context management

Built-in slash commands help you manage sessions efficiently:

Session management:

- `/clear` — reset conversation (use when switching tasks)
- `/compact` — summarize history to free token space (use at >80% context usage)
- `/cost` — check current token spend
- `/model` — switch models mid-session (Opus for hard problems, Sonnet for routine work)

Other useful commands:

- `/review` — code review of recent git changes
- `/init` — generate a starter `CLAUDE.md`
- `/hooks` — browse configured hooks
- Custom: any `.md` file in `.claude/commands/` becomes a slash command

Rule of thumb: Use `/clear` often when switching tasks. Use `/compact` when staying on the same task but running low on context. Both are much cheaper than letting the agent auto-compact mid-thought.

Putting It All Together

A worked example from the nursing home audit study

Example

- Amy and coauthors have run an audit study of nursing homes in the US to detect selective admissions practices on the basis of race, payer, and ADRD status.
- We want to estimate treatment effects now that we have the audit study results
- Let's estimate payer treatment effects and write the code with claude code

What skills and agents do we use here?

1. **Skill:** `econ-regression-table` — generates publication-quality LaTeX tables from fitted `GLM.jl` and `FixedEffectModels.jl` objects. Booktabs formatting, no significance stars, FE indicators, dep. var. means. Write once, reuse across every project.
2. **Agent:** `julia-coder` — handles all code writing and refactoring. Knows Julia coding well. Runs on Sonnet to keep costs low.
3. **Agent:** `data-explorer` — profiles datasets, checks missingness, identifies merge keys, flags data quality issues. Also runs on Sonnet.

Workflow

1. Write out the econometrics and detailed study design in a .md or .tex file that will live in your project folder. In this project it's saved in `docs/econometrics.md`.
2. Enter plan mode and discuss what you want with the claude code referencing any documentation you've written (superpowers is a good plugin for this). All plans are then saved to `docs/superpowers/plans/`.
3. Read and approve plan
4. Implement (should automatically use subagents and skills but in your first few sessions you might need to instruct it to do this and remember it)
5. Code review (human + AI)
6. Commit and push

Results

- With very detailed instructions, works perfectly in one shot modulo some differences in how q-values are calculated
- Once given access to our analysis code's version of the q-value calculation, it works perfectly

Common mistakes to avoid

- Exclusively using chat tools (chatGPT, gemini, claude, chipotle's free chat bot)
- Providing code snippets and error messages without full context of your project
- No systematic workflow to plan out code, implement, and review it (using AI and human review)
- Not setting proper permissions and banned behaviors
- Not implementing specific skills and agents that best suit your project (saving resources and improving performance)

Getting started: practical steps

1. **Install Claude Code into an existing project.** Pick a project you already know well (you need to be able to judge the output).
2. **Run `/init`** to generate a draft `CLAUDE.md`. Tweak it to match your preferences. SWEs have assembled good examples of these online (take a look).
3. **Browse plugins for skills others have built.** Two must-haves: claude-automation-recommender (auto-generates skills and agents from your codebase) and `skill-creator` (helps you write and test new skills).
4. **Run `claude-automation-recommender`.** It will scan your project and suggest skills and agents tailored to your existing code. Plugin is called `claude-code-setup`
5. **Start with a task where you know the answer.** Experiment with how detailed you are in the planning step and see how it affects quality. Then add skills, agents, and hooks incrementally and observe how results improve.

Key takeaways

1. **Use agentic tools, not just chat.** Give the model full project context, not copy-pasted snippets.
2. **Start with a constitution.** `CLAUDE.md` and `agents.md` encode your preferences so every session inherits them.
3. **Plan–implement–review by default.** Bake the workflow into your configuration, not your memory.
4. **Skills** give agents reusable expertise; **MCP** gives them capabilities. They are complementary.
5. **Subagents** isolate context and tools, keeping your main agent lightweight and your token bill low.
6. **The prompt engineering cycle is optional.** Write your expertise once and let the infrastructure apply it.

Looking forward

- Given their promise, we should all try these tools and figure out how they can help us
- They will excel in some areas and not others. In my experience they excel when
  - You provide detailed prompts
  - You leverage the full power of Claude with skills, agents, etc
  - You review code and avoid the self-driving car problem
- Areas where they excel
  - Project scaffolding (i.e., setting up SWE-level project structure, extremely organized, self-contained, and easily replicable)
  - Answering quick questions about code, refactoring code, etc.
  - Basic data exploration (summary stats, missingness, merge keys)
  - Making tables and figures based on an existing analysis dataset
  - Basic econometrics (can do harder stuff in my experience, e.g., GMM for a pretty complicated model, monte carlo sims, etc., but requires very detailed prompts and iterating. Won't be able to one-shot harder stuff).

Things I want to experiment with in the near future

- MCP
- Agent teams
- Creating adversarial agents that horserace different approaches