AI Agents and Prompt Engineering in Econometric Coding

Sebastian Galiani[a,b] Federico A. López[c] Raul A. Sosa[c]

[a]Tulane University [b]NBER [c]Universidad de San Andrés

August 2026

Abstract

We study how large language models write code for econometric analysis. We compare three dimensions of AI-assisted coding: statistical software (Stata, R, or Python), prompting (zero-shot versus few-shot), and the degree of agency, from a chatbot that writes a single script to an agent that executes and revises its own code. On a benchmark of applied econometric and statistical tasks, moving from the chatbot to the constrained agent raises task success from 74 to 96 percent, at about eight additional cents per run. For Claude Sonnet 4.6 and GPT-5.4 through Codex, few-shot prompting improves the chatbot far more than the constrained agent, indicating that prompting and agency act as substitutes. For these models, differences across statistical software are sizeable under the chatbot but largely disappear under the constrained agent.

Keywords: large language models; AI agents; econometric software; Stata; R; Python.
JEL: C87, C18.

Generative AI tools were used during manuscript preparation to improve grammar, writing clarity, and readability. The authors reviewed and edited all AI-generated suggestions, made all substantive scientific decisions, and take full responsibility for the final manuscript.

1 Introduction

Empirical economists pose an economic question and set up an econometric model to answer it. They estimate the parameters of interest and state identification assumptions (Garg and Fetzer, 2025). They usually write code in R, Stata, or Python to process the data and compute the estimates. Modern LLMs help with writing code, a step that is often mechanical but can also introduce errors into the estimation (Cui et al., 2026). Some errors are semantic bugs: the code executes but does not compute what was intended (Tan et al., 2014). A large-scale project reproducing analyses from 110 economics and political science articles finds coding errors in about a quarter of the studies, often alongside posted code that reproduces the published numbers exactly (Brodeur et al., 2026).

Many economists already delegate coding to research assistants and still spend time reviewing, correcting, and standardizing that code. The practical question is how much of the technical implementation a researcher should delegate to an LLM. The researcher retains the research question, the econometric method, the parameters of interest, and the identification assumptions. The model receives a defined econometric task and helps implement it in code. We study how LLMs write econometric code, measuring how task success and cost vary with the degree of agency, the prompt, and the statistical software. The agency dimension sets what the model can observe and do before it submits a result. A *chatbot* writes one script without seeing run output. A *one-repair* condition returns a structured run report when the script fails and permits one revision. A *constrained agent* uses a harness that lets it run code, read the output, revise, and decide when to submit, subject to a turn cap. The prompt dimension compares a zero-shot task statement with a few-shot variant that adds an example from a related task family (Brown et al., 2020). The software dimension compares Stata, R, and Python. We evaluate every combination on a benchmark of applied econometric tasks, measure what each agency condition costs, and repeat the experiment with models from other providers.

We describe writing and debugging code as five stages: translating the request, writing the code, executing it, inspecting the results, and revising the implementation. With a chatbot, the model completes the first two stages without observing execution. The harness provides more agency when it also lets the model execute the code, inspect the results, and revise its implementation. Agency should therefore matter when errors become visible only after execution. Econometric coding provides a useful setting for studying how agency changes the way researchers use AI. The task specifies a method and target output, and an automated scoring system can determine whether the submitted code produces the requested result. The role of agency may therefore inform other research tasks with

verifiable outputs.

Each run receives one of 21 econometric tasks. The tasks cover applied econometric methods, including linear regression and classical inference, difference-in-differences, regression discontinuity, instrumental variables, panel estimators, time series, and other statistical methods. Each task specifies its dataset, target estimand, reference output, and scoring tolerances. Every task is posed in Stata, R, and Python, under each prompt condition and agency condition. A run succeeds when it implements the requested specification, executes, matches the reference output within tolerance, and solves the task from the declared inputs. The same automated system scores every run, and the main estimates use Claude Sonnet 4.6 runs across all tasks, software, and agency conditions. We analyze whether examples improve task success, whether higher levels of agency improve it compared with a chatbot, and whether success differs across Stata, R, and Python under the zero-shot chatbot. Inference clusters at the task level with wild-cluster-bootstrap p-values.

Task success increases across the three levels of agency. In the full benchmark, task success is 74.4 percent under the chatbot, 83.2 percent under one-repair, and 95.7 percent under the constrained agent. The gains come primarily from executability and creating the required result file. Examples improve task success most for the chatbot and have much smaller effects under the constrained agent. This negative interaction indicates that prompt engineering and agency are substitutes in the main Claude Sonnet 4.6 analysis. For Claude Sonnet 4.6, the average few-shot effect is 6.6 percentage points (wild-bootstrap $p = 0.009$). Under the zero-shot chatbot, Stata is an estimated 33.3 percentage points below Python ($p = 0.005$). Across the full benchmark, task success is 88.7 percent in Python, 88.9 percent in R, and 75.7 percent in Stata. These software differences narrow under the constrained agent. The remaining failures implement a related estimator, follow a different data convention, or misread the requested calculation, and execution alone does not reveal them.

We also analyze the cost of the agency conditions. Additional model turns and tool calls raise the cost of a run. We report mean cost per run, cost per successful run, and the additional cost of moving from the chatbot to a higher level of agency. Across the full benchmark, the constrained agent raises mean cost per run from \$0.18 to \$0.26 and costs \$0.27 per successful run, compared with \$0.22 for the chatbot. The incremental cost is \$0.36 per additional successful run.

The rest of the paper proceeds as follows. Section 2 reviews the related literature. Section 3 describes the benchmark, the agency conditions, and the evaluation rule. Section 4 reports the results. Section 5 analyzes the cost of agency. Section 6 compares the main

Claude Sonnet 4.6 results with GPT-5.4 through Codex and DeepSeek V4 Flash. Section 7 concludes.

2 Related Literature

Empirical research depends on code, and implementation errors can change reported results. Practitioner guides therefore treat code as a production input and set engineering standards for empirical work (Gentzkow and Shapiro, 2014). Research-code audits document widespread failures of computational reproducibility and frequent coding errors (Galiani et al., 2017; Gertler et al., 2018; Brodeur et al., 2026). This code concentrates in a few statistical software environments. Stata accounts for 73 percent of author-provided replication packages across American Economic Association journals, with R, Python, and MATLAB covering most of the remainder (Vilhuber et al., 2020). Econometric packages can return different numbers for the same estimator, and each software environment carries its own programming conventions (McCullough and Vinod, 1999; Cox, 2005).

Text-as-data methods established machine learning as a measurement tool in empirical economics (Gentzkow et al., 2019). AI systems now also support literature review, coding, data collection, and the construction of research datasets (Korinek, 2025; Afonso et al., 2026). Recent work uses LLM agents to assess replication packages or reimplement published analyses from methods descriptions and original data (Hu et al., 2025; Kohler et al., 2026; Xu and Yang, 2026).

Executable-code benchmarks score generated programs by running them against tests. This approach covers general programming, data science, multilingual code generation, and statistics (Chen et al., 2021; Lai et al., 2023; Zhuo et al., 2025; Cassano et al., 2023; Orlanski et al., 2023; Cassano et al., 2024). StatLLM releases statistical analysis tasks with datasets, human-verified SAS solutions, and expert scores (Song et al., 2026, 2025). MetricsAI evaluates an econometrics agent that plans, executes code, and revises after errors, alongside direct Python and Stata code-generation controls (Chen et al., 2025). Eltokhy (2026) maintains an executable benchmark of Stata coding tasks that reports accuracy and cost. Full-agent benchmarks extend these evaluations with file inspection, editing, command execution, and model-directed submission (Jimenez et al., 2024; Merrill et al., 2026; Kwa et al., 2025).

Harnesses expand a model's agency while coding. ReAct interleaves model reasoning with actions that return observations (Yao et al., 2023). InterCode formalizes interactive coding as code actions and observations from reproducible execution environments (Yang et al., 2023). SWE-agent introduces the agent-computer interface, the layer that defines

the commands available to the agent and the observations it receives (Yang et al., 2024). Self-debugging, self-refinement, Reflexion, and self-repair study iterative revision after tests, observations, or critiques (Chen et al., 2024; Madaan et al., 2023; Shinn et al., 2023; Olausson et al., 2024). Li et al. (2026) describe the surrounding engineering layer as an agent harness. On research code, Kim et al. (2026) mask functions from the codebases of replicated machine-learning papers. Letting the agent choose its actions and debug after failed runs raises one model's pass rate more than fourfold relative to a predefined pipeline. The closest design, Shah et al. (2026), holds the model fixed and varies the degree of agency in the repair of existing R code.

We use *prompt engineering* for changes to the task statement and its example. We use *agency* for the model's ability to act through the harness, which sets the tools, observations, revision opportunities, and stopping rules. Our design holds the model fixed and compares prompt engineering with three agency conditions implemented through the harness. Unlike the repair setting, every run generates new econometric code from a task statement, in Stata, R, or Python, with recorded per-run cost. The software comparison shows how the value of agency depends on the coding environment.

3 Design

3.1 Design dimensions

The design crosses two prompt conditions (zero-shot, few-shot), three agency conditions (chatbot, one-repair, constrained agent), three statistical software environments (Stata, R, Python), and 21 econometric tasks. The prompt conditions implement prompt engineering, and the harness implements the three agency conditions. Each cell combines one task, one software environment, one prompt, and one agency condition. The analysis uses five runs in every cell, and each run creates a record. The shared task registry records the tasks, prompts, reference outputs, scoring tolerances, model identity, and execution limits. Each task uses the same dataset across repetitions, software, prompts, and agency conditions.

A zero-shot prompt states the task, variables, dataset path, and required output. A few-shot prompt adds one example in the target software, drawn from the task registry and matched to a related task family (Brown et al., 2020). The example shows a short task statement, complete code for the related task, and the required result-file structure.

3.2 Degree of agency

The agency conditions differ in what the model can see and do between receiving the task and submitting a result. The harness implements each condition by defining the context, tools, execution process, stopping rule, and traces (Li et al., 2026). The *chatbot* condition asks for one complete script and shows the model no run output. The *one-repair* condition executes the submitted script and, if it fails or the required result file is missing, returns one structured run report and accepts one revised script. The *constrained agent* runs a self-directed loop under two deliberate restrictions: shell execution is its only pre-authorized tool, and the loop is capped at twelve model turns. Within those limits it writes, runs, reads errors, revises, and decides when to submit. It works in a dedicated task folder containing the input data. Requests for any other tool are denied and recorded, so only the shell tool can execute. The three agency conditions allow the model to complete different stages of the coding process. The chatbot translates the request and writes code. One-repair adds execution, a structured report, and one revision. The constrained agent executes, inspects, and revises within the loop until it submits or reaches the turn limit. Figure 1 summarizes the design, the three agency conditions, and the evaluation rule.

3.3 Tasks, references, and evaluation

Each task specifies a dataset, a target estimand, a required result-file structure, reference scripts, method verification checks, and per-quantity tolerances. The scoring system runs generated code on the task inputs and compares it with stored reference outputs (Chen et al., 2021; Lai et al., 2023; Zhuo et al., 2025; Song et al., 2026). The tasks combine newly simulated datasets with adapted statistical-programming datasets in the MetricsAI and StatLLM style (Chen et al., 2025; Song et al., 2026). For simulated tasks, the registry records the DGP, seed, baseline parameters, and finite-sample reference output. The families span core regression and inference, diagnostics, classical tests, GLMs, survival, panels and endogeneity, treatment effects, time-series methods, and other tasks in statistics.[1]

The primary outcome is task success. A run succeeds when it satisfies four criteria. *Specification correctness* requires the intended estimator, sample restriction, controls or fixed effects, target quantity, and inference procedure. *Executability* requires the script to run without timeout or fatal failure. *Output correctness* requires the submitted result file to match the reference output within the declared tolerance. *Submission validity* requires the script to use the declared input file and forbids access to stored reference outputs or

[1]The full registry is in Appendix A, with concrete examples of scored targets and evaluation checks in Table 3; Appendix C states the primary-estimand protocol.

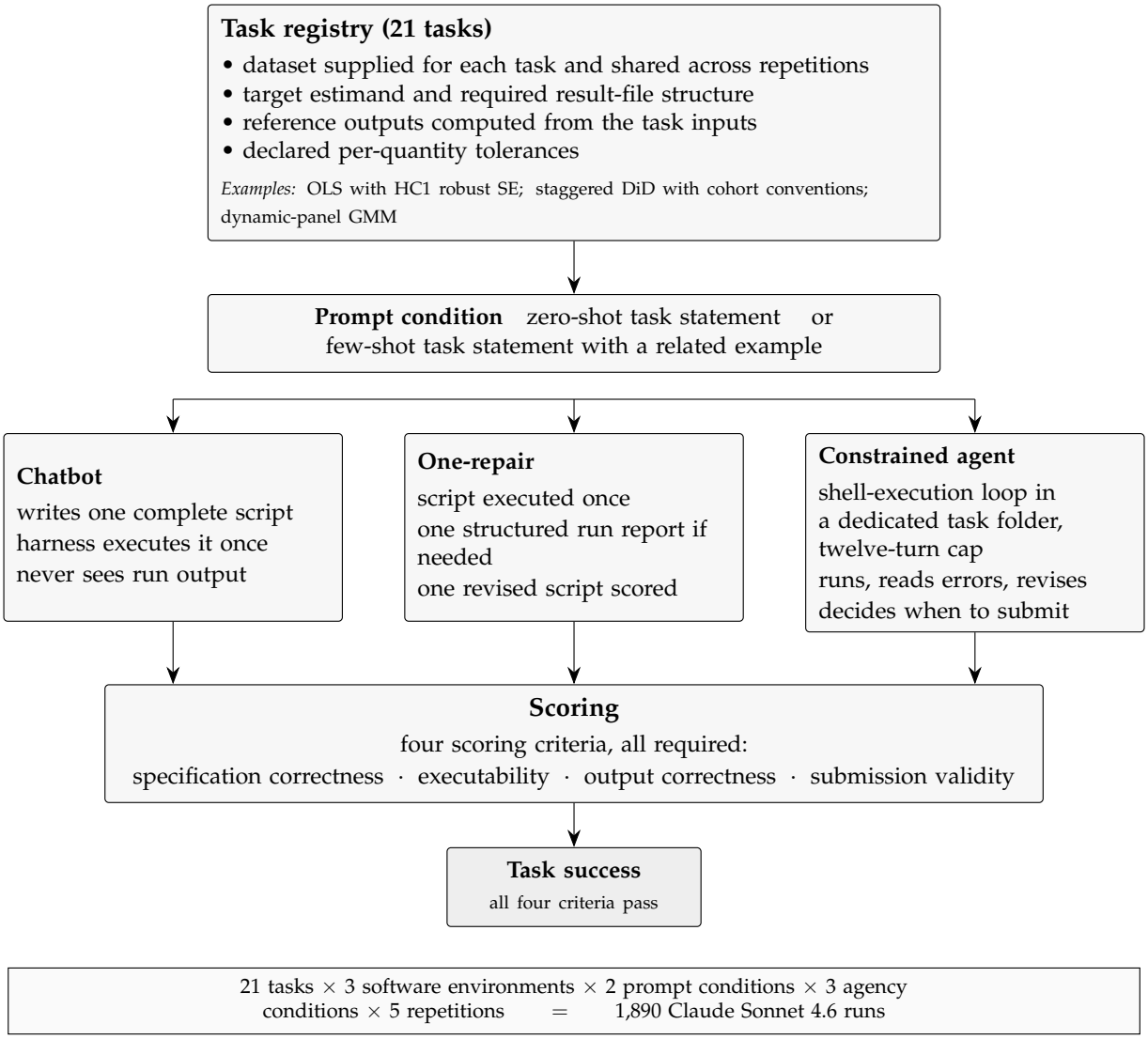

Figure 1: Benchmark design. Each of the 21 tasks specifies a dataset, a target estimand, reference outputs computed from the task inputs, and per-quantity tolerances. The prompt condition changes the task statement, and the agency condition changes what the model can observe and do before submission. The automated scoring system applies four criteria, and a run succeeds when all four pass.

hard-coded reference numbers. The scoring system normalizes serialization dialects before applying the numeric comparison, so the software comparison is not driven by output formatting (Appendix D). Beyond the binary outcome, the diagnostic record tracks each criterion, numerical errors, executions, turns, tokens, and estimated cost (Yang et al., 2023; Merrill et al., 2026; Anthropic, 2026a).[2]

[2]Appendices B and E describe the agency protocol and the model-specific fields (Anthropic, 2026b,c).

3.4 Analysis and inference

Cell rates summarize task success, following executable-code benchmarks (Chen et al., 2021; Zhuo et al., 2025). The regression uses each run as one observation. We estimate a linear probability model with task fixed effects,

$$\text{Success}_{ir} = \tau_i + X'_{ir}\beta + \varepsilon_{ir}, \tag{1}$$

where i indexes tasks and r indexes runs within a task. The vector X_{ir} encodes all combinations in the $2 \times 3 \times 3$ design, with Python, zero-shot, and chatbot as the omitted categories. The model includes all interactions among software, prompt, and agency conditions (Angrist and Pischke, 2009; Wooldridge, 2010). Together, these terms produce a predicted task-success rate for every combination of software, prompt, and agency conditions. The reported estimates compare those predicted rates. The software estimates compare R and Stata with Python among zero-shot chatbot runs. The prompt estimate compares few-shot with zero-shot after averaging across software and agency conditions. The agency estimates compare one-repair and the constrained agent with chatbot after averaging across software and prompts. The interaction estimates show how the few-shot effect changes under one-repair and the constrained agent relative to chatbot. Each average gives the relevant combinations equal weight.

The analysis reports four hypotheses:

- *H1*, the software contrast at the zero-shot chatbot baseline;
- *H2*, the average few-shot effect across software and agency conditions;
- *H3*, the average one-repair and constrained-agent effects against the chatbot, reported separately; and
- *H4*, the interactions between prompt and agency conditions.

Positive interactions indicate complementarity between examples and agency, and negative interactions indicate substitution. H4 tests whether access to execution and revision changes the value of adding an example to the prompt. Standard errors are clustered at the task level; with 21 task clusters the headline p-values use the wild cluster bootstrap with Rademacher weights and 9,999 replications (Cameron et al., 2008; MacKinnon and Webb, 2018).

4 Results

The analysis sample contains 1,890 scored Claude Sonnet 4.6 runs: five repetitions of every cell in the design across the 21 tasks, with 945 in each prompt condition. Each analyzed record has a stored task registry entry, dataset, seed, prompt, model identity, agency condition, run record, reference output, and scorer version.

4.1 Effectiveness across agency conditions

Overall task success in the analysis sample is 84.4 percent, 81.2 in the zero-shot condition and 87.7 in the few-shot condition. Panel A of Table 1 decomposes success into the scoring criteria by agency condition within each prompt condition. Agency conditions that return run output increase executability and output correctness. Zero-shot chatbot runs execute successfully in 77.5 percent of analyzed records and satisfy output correctness in 67.3 percent. The one-repair condition raises those rates to 90.5 and 81.3 percent; the constrained agent reaches 99.7 percent executability and 94.9 percent output correctness. The few-shot condition shows the same ordering from a higher chatbot level. Across both prompt conditions, task success rises from 74.4 percent under the chatbot condition to 83.2 percent under one-repair and 95.7 percent under the constrained agent.

Panel B of Table 1 reports the estimated changes in task success from equation (1). On H3, one structured run report adds 8.7 percentage points over the chatbot baseline, and the constrained agent adds 21.3 percentage points; both bootstrap p-values are below one in a thousand. The one-repair estimate applies the one-revision boundary stated in Appendix B: a run that needs more than one revision counts as a failure.

4.2 Prompt engineering and agency

On H2, the average few-shot effect is 6.6 percentage points ($p = 0.009$). This average conceals substantial differences across agency conditions. Within the chatbot condition, few-shot examples add 14.3 percentage points ($p = 0.005$). The raw prompt gains fall from 14.3 percentage points under the chatbot to 3.8 under one-repair and 1.6 under the constrained agent. Figure 2 shows this pattern for Claude Sonnet 4.6 and for GPT-5.4 through Codex.

On H4, both interactions between prompt and agency conditions are negative. The few-shot-by-one-repair interaction is -10.5 percentage points ($p = 0.016$), and the few-shot-by-constrained-agent interaction is -12.7 points ($p = 0.061$). The value of adding an example therefore falls as the Claude Sonnet 4.6 harness provides more agency. In

Table 1: Task success and estimated effects of prompts, agency, and software.

Panel A: Observed success criteria (percent)

Prompt	Agency condition	Executability	Output correctness	Task success
Zero-shot	Chatbot	77.5	67.3	67.3
	One-repair	90.5	81.3	81.3
	Constrained agent	99.7	94.9	94.9
Few-shot	Chatbot	91.1	81.6	81.6
	One-repair	94.6	85.1	85.1
	Constrained agent	100.0	96.8	96.5

Panel B: Estimated changes in task success (percentage points)

Comparison	Estimated change
H3a: One-repair versus chatbot	8.7^{***} (2.2)
H3b: Constrained agent versus chatbot	21.3^{***} (4.2)
H4a: Few-shot × one-repair	-10.5^{**} (4.3)
H4b: Few-shot × constrained agent	-12.7^{*} (6.2)
H2: Few-shot versus zero-shot	6.6^{***} (2.3)
Additional: Few-shot versus zero-shot under chatbot	14.3^{***} (4.9)
H1a: R versus Python	-1.9 (7.9)
H1b: Stata versus Python	-33.3^{***} (10.1)

Notes: Panel A reports percentages; each row contains 315 runs. Timeouts count as failures. Output correctness requires a valid result file and a numeric match to the reference output. Task success requires all four scoring criteria to pass. Panel B reports differences between predicted task-success rates from equation (1). H1 compares software in the zero-shot chatbot condition. H2 averages across software and agency conditions with equal weight. H3 averages across software and prompt conditions with equal weight. H4 measures how the few-shot effect changes under one-repair or the constrained agent. Standard errors in parentheses are clustered by task. *, **, and *** denote significance at the 10, 5, and 1 percent levels using wild-cluster-bootstrap tests with 9,999 replications.

this setting, prompt engineering and agency act as substitutes. As coding agents become more prevalent and capable, increasing the agency provided by the harness may be more valuable than increasingly sophisticated prompt engineering.

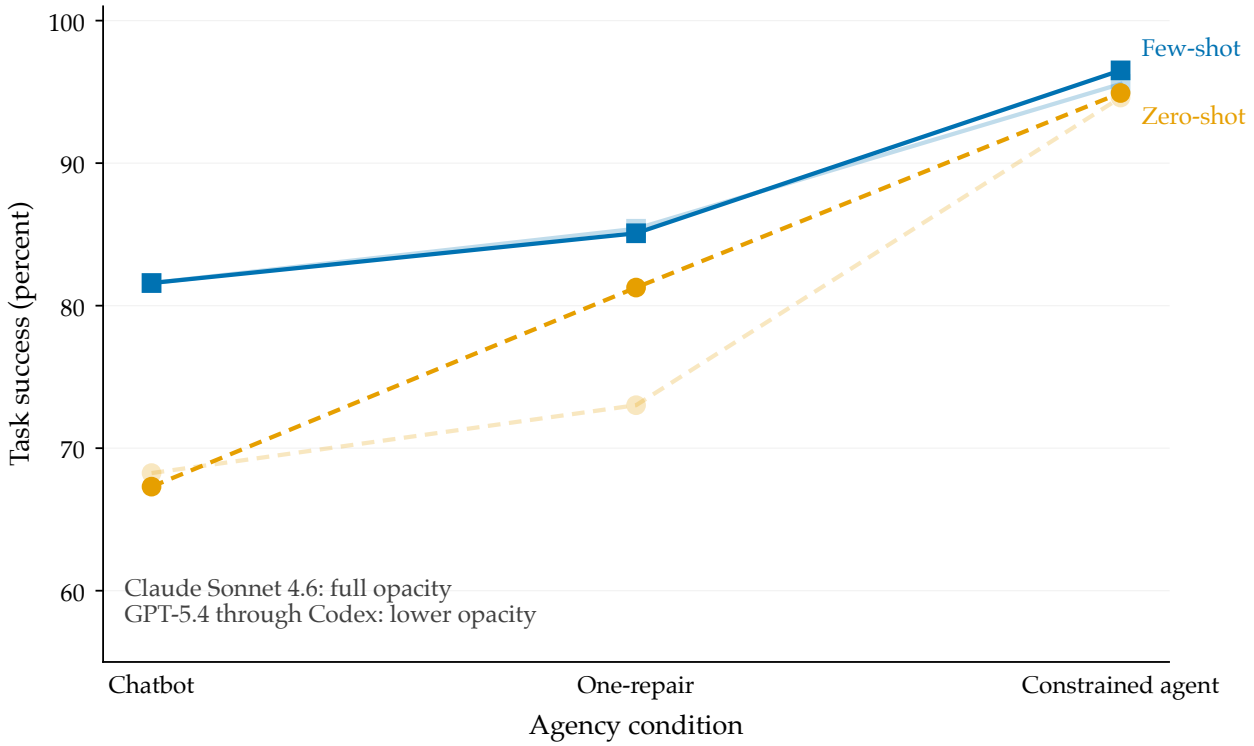

Figure 2: Task success by prompt and agency condition for Claude Sonnet 4.6 and GPT-5.4 through Codex. Claude Sonnet 4.6 appears at full opacity and Codex at lower opacity. Each point averages across tasks and statistical software. Lines connect zero-shot and few-shot results within each model and agency condition.

4.3 Agency and statistical software

On H1, the estimated baseline software gap is a Stata gap. At the zero-shot chatbot baseline, the Stata contrast is -33.3 percentage points ($p = 0.005$) and the R contrast is -1.9 points ($p = 0.808$). The joint test rejects the hypothesis of no software gap at baseline ($p = 0.006$). Across the full benchmark, task success is 88.7 percent for Python, 88.9 percent for R, and 75.7 percent for Stata. In the zero-shot condition, Stata rises from 45.7 percent under the chatbot condition to 98.1 percent under the constrained agent, the largest agency gain across the three software environments. Under the constrained agent, Stata has the highest task-success rate in both prompt conditions. The software comparison therefore illustrates the broader agency result. Differences that are large under the chatbot become small when the harness can execute and revise code.

4.4 Task-level variation

Figure 3 shows task success across the three agency conditions for each task, with GPT-5.4 through Codex as a lighter overlay. Task-level rates locate the sources of agency gains and

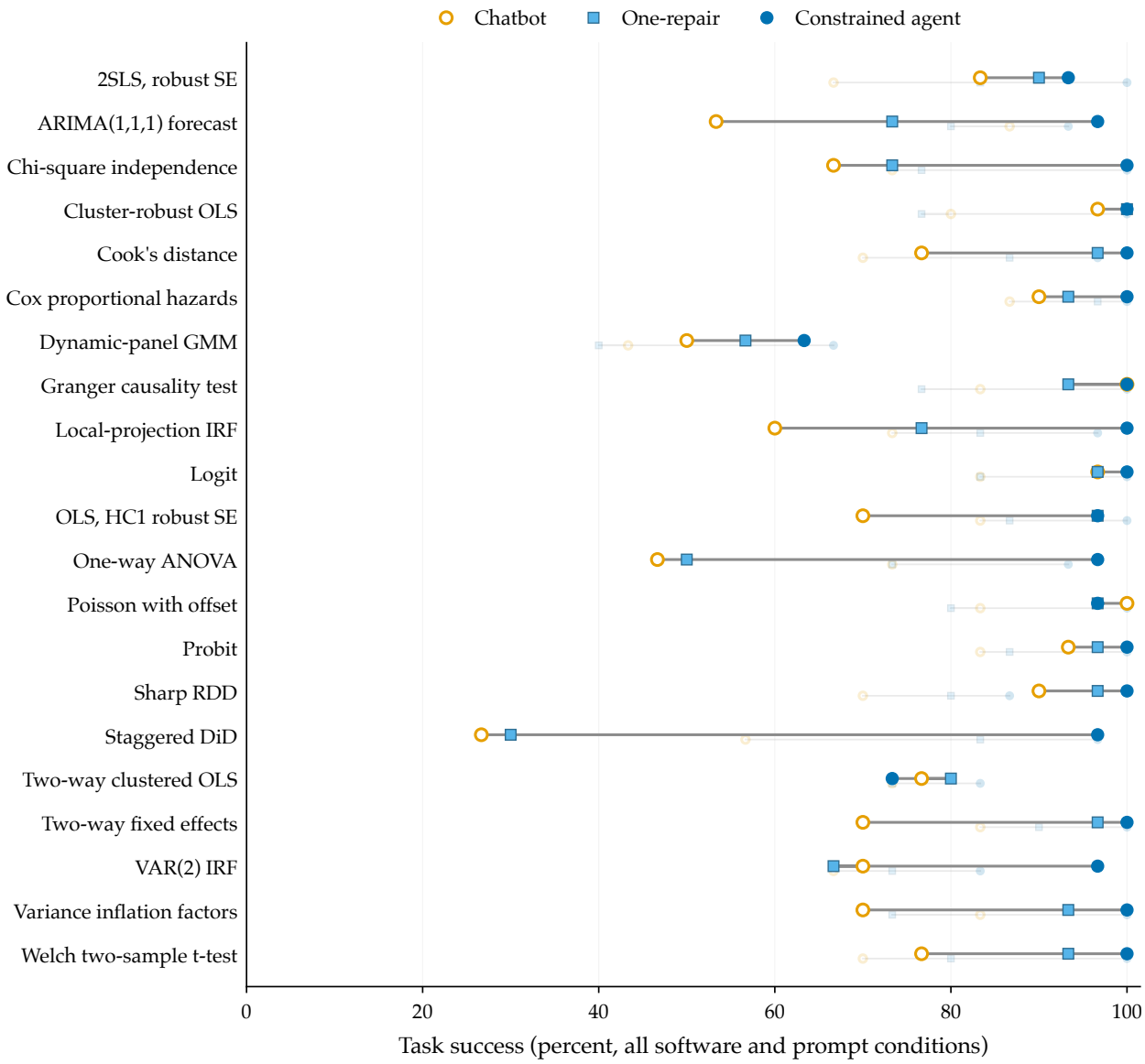

Figure 3: Task success by task, model, and agency condition. Tasks are listed alphabetically. Each path averages across software and prompts within a model. Claude Sonnet 4.6 appears at full opacity. GPT-5.4 through Codex appears as an adjacent, more transparent comparison. Open circles show chatbot success, squares show one-repair success, and filled circles show constrained-agent success.

residual failures. Cluster-robust OLS, Poisson with an offset, logit, VAR Granger causality, and probit all succeed in 95 percent or more of analyzed records. The two tasks with the lowest success rates are staggered-adoption DiD at 51.1 percent and dynamic-panel GMM at 56.7 percent. Both have failure modes that a successful execution can leave undetected.

Dynamic-panel GMM succeeds in 90.0 percent of Stata runs, 66.7 percent of R runs, and 13.3 percent of Python runs. The observed Python failures often implement a related

estimator, including Anderson–Hsiao-style instrumental variables, instead of the requested one-step Arellano–Bond estimator (Anderson and Hsiao, 1981; Arellano and Bond, 1991). These scripts can execute while estimating a different dynamic-panel specification. Few-shot examples raise success on this task from 44.4 to 68.9 percent.

Another risk is silent failure: among runs that execute and write a valid result file, 10.5 percent still fail task success under the chatbot condition. The corresponding rates are 7.3 percent under one-repair and 4.1 percent under the constrained agent. Run output solves nearly all runtime and result-file failures and most numeric ones. Residual failures involve estimator variants, data conventions, and misinterpretations of the requested calculation. Specification review remains necessary because these scripts can execute successfully.

4.5 Scoring checks

The scoring system treats R's `jsonlite` empty-object convention and Stata's leading-dot decimals as valid serialization forms. This normalization keeps numerically correct runs from failing because of software-specific JSON formatting.[3] The same scoring rules apply to all rates in this section.

The headline rates change little when the declared tolerances are rescaled. Halving every numeric tolerance lowers overall task success from 81.2 to 80.5 percent in the zero-shot condition and from 87.7 to 87.3 percent in the few-shot condition. Doubling every tolerance leaves the zero-shot rate unchanged and moves the few-shot rate to 87.8 percent. Runs that fail on execution, result-file, specification, or integrity grounds never flip under rescaling, so the tolerance choice affects only the numeric-match margin.

5 The Cost of Agency

For each run, the ledger records the model, agency condition, prompt, statistical software, token use, estimated cost, and task success. The estimated-dollar field is a client-side estimate used for comparing cells.[4] Dollar figures for the main sample use both prompt conditions unless a prompt condition is named.

5.1 Cost by agency condition

One-repair and constrained-agent runs can use additional model turns, script executions, and tool calls. The cost comparison therefore reports expenditure per run and per success-

[3] Appendix D lists the normalization and verification rules.
[4] Appendix E describes field availability and reporting rules.

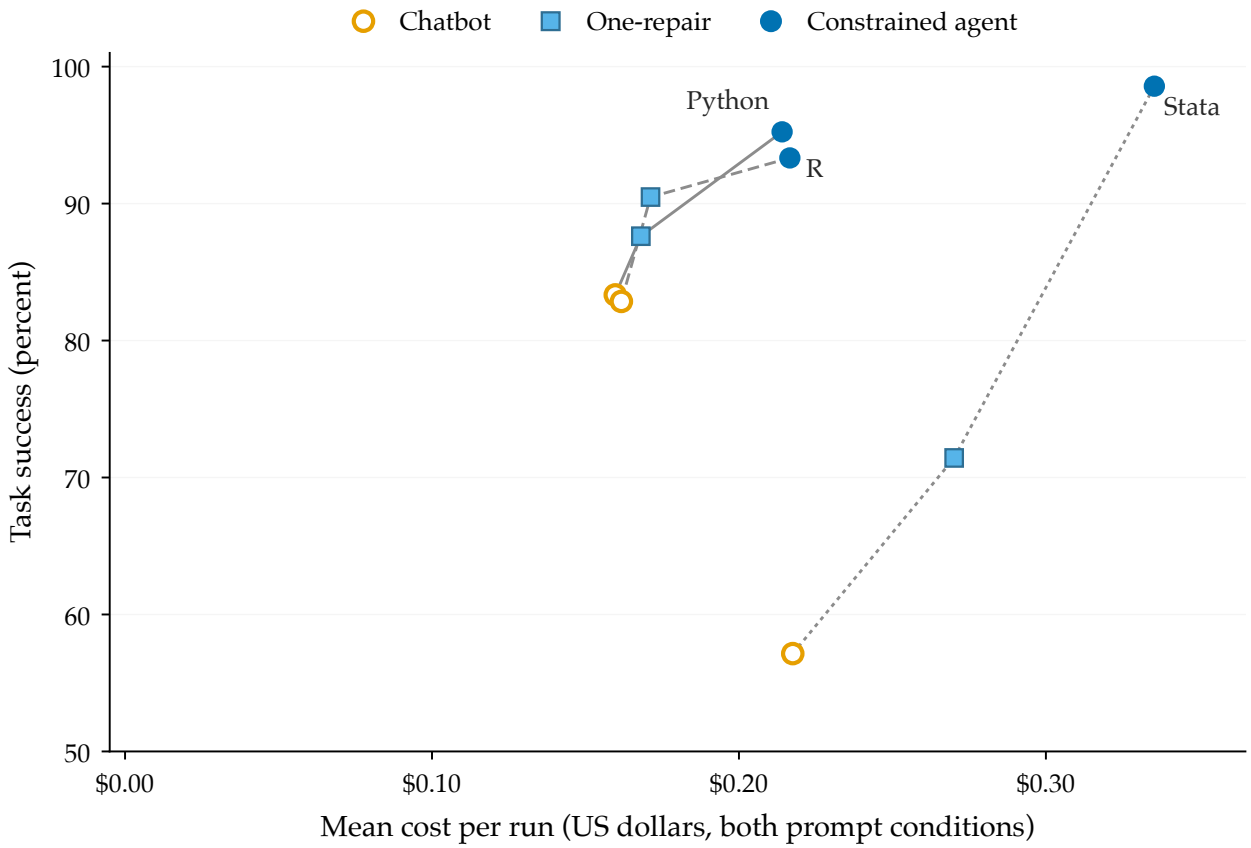

Figure 4: Claude Sonnet 4.6 cost and task success by statistical software and agency condition. Points use recorded mean cost per run. Open circles show chatbot runs, squares show one-repair runs, and filled circles show constrained-agent runs. Lines connect the three agency conditions within each software environment.

ful run alongside task success. Cost per successful run equals mean cost per run divided by the task-success rate. Figure 4 plots task success against mean cost per run for the nine combinations of software and agency conditions across both prompt conditions.

The median run costs $0.19 under the chatbot, $0.20 under one-repair, and $0.24 under the constrained agent. The one-repair condition costs $0.23 per successful run against $0.22 for the chatbot while raising task success from 74.4 to 83.2 percent. Its median cost per run matches the chatbot's within each prompt condition. Within each prompt condition the constrained agent adds about five cents to the chatbot's median per-run cost, $0.18 against $0.12 under zero-shot prompts and $0.25 against $0.20 under few-shot prompts. The constrained agent costs the most per successful run, $0.27 against $0.22 for the chatbot.

Typical costs are highest in Stata. Median per-run cost is $0.19 for both Python and R against $0.26 for Stata. Cost per successful run is $0.20 for Python, $0.21 for R, and $0.34 for Stata. The pattern mirrors the success results.

5.2 Incremental cost of agency

The incremental cost statistic divides the change in mean cost per run by the change in task success. Moving from the chatbot to the constrained agent raises task success by 21.3 percentage points and mean cost per run by about eight cents, an incremental cost of $0.36 per additional successful run. The incremental cost differs by prompt condition. Under zero-shot prompts, the constrained agent adds 27.6 percentage points of task success at $0.27 per additional successful run. Under few-shot prompts, the same change adds 14.9 percentage points at $0.55 per additional successful run.

6 Other Models

The main estimates use Claude Sonnet 4.6. We apply the same design and analysis to GPT-5.4 through the Codex command line interface and DeepSeek V4 Flash. Each additional arm contains 1,890 runs, with five repetitions in every cell. All three model arms use the same tasks, prompts, statistical software environments, reference outputs, scoring rule, and agency conditions. In the one-repair condition, every model receives at most one revision after a failed script or a missing required result file. In the constrained-agent condition, Codex can take somewhat more actions than Claude Sonnet 4.6 or DeepSeek. Table 2 reports task success, the estimated effects, and costs for the three arms.

Within each model arm, execution-enabled agency raises task success relative to the chatbot. The constrained-agent contrast is 21.3 percentage points for Claude Sonnet 4.6, 20.2 points for GPT-5.4, and 18.7 points for DeepSeek. Few-shot examples also raise average task success in every arm. The estimated gain is 6.6 points for Claude Sonnet 4.6, 8.9 for GPT-5.4, and 14.9 for DeepSeek. The Stata contrast at the zero-shot chatbot baseline is negative in all three arms and largest in magnitude for GPT-5.4 and DeepSeek. DeepSeek is also the only arm where R outperforms Python at that baseline, by 21.9 points. The prompt-by-agency interaction differs across models. The few-shot gain falls under the constrained agent for Claude Sonnet 4.6 and GPT-5.4, while it rises for DeepSeek. The substitution result therefore describes the main Claude Sonnet 4.6 analysis and the constrained-agent Codex comparison, not every model arm.

All three cost measures value recorded token use at the relevant API price. Claude Sonnet 4.6 records a client-side API-cost estimate, and DeepSeek records the OpenRouter charge. Codex records tokens but no billed dollar amount because the runs used a ChatGPT subscription. We value its tokens at the GPT-5.4 Standard base token rates, so its entries are API-equivalent amounts rather than billed charges. Panel C compares cost per successful

Table 2: Task success, estimated effects, and costs across models.

Panel A: Task success (percent)

Prompt	Claude Sonnet 4.6	GPT-5.4	DeepSeek V4 Flash
Zero-shot	81.2	78.6	57.7
Few-shot	87.7	87.5	72.6

Panel B: Estimated changes in task success (percentage points)

Comparison	Claude Sonnet 4.6	GPT-5.4	DeepSeek V4 Flash
H1a: R versus Python	-1.9	-3.8	21.9^{**}
	(7.9)	(3.0)	(8.4)
H1b: Stata versus Python	-33.3^{***}	-77.1^{***}	-66.7^{***}
	(10.1)	(4.7)	(7.9)
H2: Few-shot versus zero-shot	6.6^{***}	8.9^{***}	14.9^{***}
	(2.3)	(2.2)	(2.1)
H3a: One-repair versus chatbot	8.7^{***}	4.3^{**}	8.1^{***}
	(2.2)	(2.0)	(2.4)
H3b: Constrained agent versus chatbot	21.3^{***}	20.2^{***}	18.7^{***}
	(4.2)	(1.7)	(3.2)
H4a: Few-shot $\times$ one-repair	-10.5^{**}	-1.0	8.6^{**}
	(4.3)	(3.4)	(2.9)
H4b: Few-shot $\times$ constrained agent	-12.7^{*}	-12.4^{***}	15.2^{***}
	(6.2)	(3.7)	(3.6)

Panel C: Cost per successful constrained-agent run (US dollars)

Prompt	Claude Sonnet 4.6	GPT-5.4	DeepSeek V4 Flash
Zero-shot	\$0.22	\$0.13	\$0.0051
Few-shot	\$0.31	\$0.08	\$0.0025

Notes: Panel A reports observed task-success rates. Panel B reports percentage-point differences between predicted task-success rates from separate estimates of equation (1) for each model. H1a and H1b compare R and Stata with Python in the zero-shot chatbot condition. H2 averages across software and agency conditions with equal weight. H3 averages across software and prompt conditions with equal weight. H4 measures how the few-shot effect changes under one-repair or the constrained agent. Standard errors in parentheses are clustered by task. *, **, and *** denote significance at the 10, 5, and 1 percent levels using wild-cluster-bootstrap tests with 9,999 replications. Panel C reports API-equivalent dollar costs calculated from recorded tokens and the relevant API price.

constrained-agent run: \$0.22 and \$0.31 for Claude Sonnet 4.6 under zero-shot and few-shot prompts, \$0.13 and \$0.08 for GPT-5.4, and less than one cent for DeepSeek.

7 Conclusion

This paper builds a controlled benchmark for LLM econometric coding and scores 1,890 Claude Sonnet 4.6 runs on 21 tasks across prompts, statistical software, and degrees of agency. Moving from the chatbot to the constrained agent raises task success from 74.4 to

95.7 percent, removing about five of every six failures at an incremental cost of $0.36 per additional successful run. A single structured repair captures about 40 percent of that gain with a one-cent increase in cost per successful run. Some runs execute successfully while implementing a different specification. We do not evaluate that aspect of LLM performance. Testing the LLM's ability to decide the correct specification given an economic problem would require further research.

The results apply to the tested models, English prompts, statistical software environments, and agency conditions. Each benchmark task defines its dataset, statistical method, target quantity, and required output. Open-ended empirical research falls outside this scope.

The transition toward greater agency is changing how economists interact with AI. For the models and tasks we study, the agency provided by the harness matters more than sophisticated prompting.

References

Afonso, S., S. Galiani, R. H. Gálvez, and R. A. Sosa (2026, May). Deep research on a loop: Using AI agents to construct economic datasets. Working Paper 35188, National Bureau of Economic Research.

Anderson, T. W. and C. Hsiao (1981). Estimation of dynamic models with error components. *Journal of the American Statistical Association 76*(375), 598–606.

Angrist, J. D. and J.-S. Pischke (2009). *Mostly Harmless Econometrics: An Empiricist's Companion*. Princeton University Press.

Anthropic (2026a). Demystifying evals for AI agents. Anthropic Engineering, https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents. Accessed July 1, 2026.

Anthropic (2026b). How the agent loop works. Claude Code Docs. Accessed May 14, 2026.

Anthropic (2026c). Track cost and usage. Claude Code Docs. Accessed May 14, 2026.

Arellano, M. and S. Bond (1991). Some tests of specification for panel data: Monte Carlo evidence and an application to employment equations. *The Review of Economic Studies 58*(2), 277–297.

Brodeur, A., D. Mikola, N. Cook, L. Fiala, T. Brailey, R. Briggs, A. de Gendre, Y. Dupraz, J. Gabani, R. Gauriot, et al. (2026). Reproducibility and robustness of economics and political science research. *Nature 652*(8108), 151–156.

Brown, T. B., B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei (2020). Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, Volume 33, pp. 1877–1901.

Callaway, B. and P. H. C. Sant'Anna (2021). Difference-in-differences with multiple time periods. *Journal of Econometrics 225*(2), 200–230.

Cameron, A. C., J. B. Gelbach, and D. L. Miller (2008). Bootstrap-based improvements for inference with clustered errors. *Review of Economics and Statistics 90*(3), 414–427.

Cassano, F., J. Gouwar, F. Lucchetti, C. Schlesinger, A. Freeman, C. J. Anderson, M. Q. Feldman, M. Greenberg, A. Jangda, and A. Guha (2024). Knowledge transfer from high-resource to low-resource programming languages for code LLMs. *Proceedings of the ACM on Programming Languages 8*(OOPSLA2), 677–708.

Cassano, F., J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda (2023). MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering 49*(7), 3675–3691.

Chen, M., J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

Chen, Q., T. Han, J. Li, Y. Luo, Z. Wang, Y. Wu, X. Zhang, and T. Zhou (2025). Can AI Master Econometrics? Evidence from Econometrics AI Agent on Expert-Level Tasks. arXiv preprint arXiv:2506.00856.

Chen, X., M. Lin, N. Schärli, and D. Zhou (2024). Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*.

Cox, N. J. (2005). Suggestions on Stata programming style. *Stata Journal 5*(4), 560–566.

Cui, K., M. Demirer, S. Jaffe, L. Musolff, S. Peng, and T. Salz (2026). The effects of generative AI on high-skilled work: Evidence from three field experiments with software developers. *Management Science*. Forthcoming.

Eltokhy, K. (2026). LLM stata benchmark. Online benchmark. Accessed August 3, 2026.

Galiani, S., P. Gertler, and M. Romero (2017, July). Incentives for replication in economics. NBER Working Paper 23576, National Bureau of Economic Research.

Garg, P. and T. Fetzer (2025). Causal claims in economics. CEPR Discussion Paper 19701, Centre for Economic Policy Research. arXiv:2501.06873.

Gentzkow, M., B. Kelly, and M. Taddy (2019). Text as data. *Journal of Economic Literature 57*(3), 535–574.

Gentzkow, M. and J. M. Shapiro (2014, March). Code and data for the social sciences: A practitioner's guide. Mimeo, University of Chicago.

Gertler, P. J., S. Galiani, and M. Romero (2018). How to make replication the norm. *Nature 554*(7693), 417–419.

Hu, C., L. Zhang, Y. Lim, A. Wadhwani, A. Peters, and D. Kang (2025). REPRO-Bench: Can agentic AI systems assess the reproducibility of social science research? In *Findings of the Association for Computational Linguistics: ACL 2025*, Vienna, Austria, pp. 23616–23626. Association for Computational Linguistics.

Jimenez, C. E., J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan (2024). SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*.

Kim, G. J., A. Wilf, L.-P. Morency, and D. Fried (2026). From reproduction to replication: Evaluating research agents with progressive code masking. In *International Conference on Learning Representations (ICLR)*. arXiv:2506.19724.

Kohler, B., D. Zollikofer, J. Einsiedler, A. Hoyle, and E. Ash (2026). Read the paper, write the code: Agentic reproduction of social-science results. *arXiv preprint arXiv:2604.21965*.

Korinek, A. (2025, September). AI agents for economic research. NBER Working Paper 34202, National Bureau of Economic Research.

Kwa, T., B. West, J. Becker, A. Deng, K. Garcia, M. Hasin, S. Jawhar, M. Kinniment, N. Rush, S. Von Arx, et al. (2025). Measuring AI ability to complete long software tasks. arXiv preprint arXiv:2503.14499.

Lai, Y., C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, S. W.-t. Yih, D. Fried, S. Wang, and T. Yu (2023). DS-1000: A natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*.

Li, J., X. Xiao, Y. Zhang, C. Liu, L. Zhao, X. Liao, Y. Ji, J. Wang, J. Gu, Y. Ge, W. Xu, X. Fang, X. Xu, T. Zhao, Y. Kim, T. Wang, J. Hamm, S. Krishnaswamy, J. Huan, and C. K. Reddy (2026). Agent harness engineering: A survey. OpenReview preprint.

MacKinnon, J. G. and M. D. Webb (2018). The wild bootstrap for few (treated) clusters. *Econometrics Journal 21*(2), 114–135.

Madaan, A., N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, S. Gupta, B. P. Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, and P. Clark (2023). Self-Refine: Iterative refinement with self-feedback. arXiv preprint arXiv:2303.17651.

McCullough, B. D. and H. D. Vinod (1999). The numerical reliability of econometric software. *Journal of Economic Literature 37*(2), 633–665.

Merrill, M. A., A. G. Shaw, N. Carlini, B. Li, H. Raj, I. Bercovich, L. Shi, J. Y. Shin, T. Walshe, E. K. Buchanan, et al. (2026). Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. arXiv preprint arXiv:2601.11868.

Olausson, T. X., J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama (2024). Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations (ICLR)*.

Orlanski, G., K. Xiao, X. Garcia, J. Hui, J. Howland, J. Malmaud, J. Austin, R. Singh, and M. Catasta (2023). Measuring the impact of programming language distribution. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*.

Shah, S. M. H., F. Hopfgartner, and A. Bleier (2026). Automating computational reproducibility in social science: Comparing prompt-based and agent-based approaches. In

Companion Proceedings of the ACM Web Conference 2026 (WWW Companion '26), New York, NY, USA. ACM.

Shinn, N., F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao (2023). Reflexion: Language agents with verbal reinforcement learning. arXiv preprint arXiv:2303.11366.

Song, X., L. Lee, K. Xie, X. Liu, X. Deng, and Y. Hong (2026). StatLLM: A dataset for evaluating the performance of large language models in statistical analysis. *Scientific Data 13*, 369.

Song, X., K. Xie, L. Lee, R. Chen, J. M. Clark, H. He, H. He, J. Min, X. Zhang, S. Zheng, Z. Zhang, X. Deng, and Y. Hong (2025). Performance evaluation of large language models in statistical programming. arXiv preprint arXiv:2502.13117.

Tan, L., C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai (2014). Bug characteristics in open source software. *Empirical Software Engineering 19*(6), 1665–1705.

Vilhuber, L., J. Turrito, and K. Welch (2020). Report by the AEA data editor. *AEA Papers and Proceedings 110*, 764–775.

Wooldridge, J. M. (2010). *Econometric Analysis of Cross Section and Panel Data* (2 ed.). MIT Press.

Xu, Y. and L. Y. Yang (2026). Scaling reproducibility: An AI-assisted workflow for large-scale replication and reanalysis. arXiv preprint arXiv:2602.16733.

Yang, J., C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press (2024). SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Yang, J., A. Prabhakar, K. Narasimhan, and S. Yao (2023). InterCode: Standardizing and benchmarking interactive coding with execution feedback. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*.

Yao, S., J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao (2023). ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Zhuo, T. Y., M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, S. Brunner, C. Gong, T. Hoang, A. R. Zebaze, X. Hong, W.-D. Li, J. Kaddour, M. Xu, Z. Zhang, P. Yadav, N. Jain, A. Gu, Z. Cheng, J. Liu, Q. Liu, Z. Wang, D. Lo, B. Hui,

N. Muennighoff, D. Fried, X. Du, H. de Vries, and L. Von Werra (2025). BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations (ICLR)*.

Appendices

A Benchmark Tasks

The analysis uses 21 econometric and statistical tasks from the task registry. Each task supplies one dataset, one task statement, a required result-file structure, a shared reference output, numerical tolerances, and method checks. The same task dataset and reference output apply across software, prompt conditions, agency conditions, and repetitions. Table 3 lists the required output for every analyzed task.

Table 3: Tasks and scored outputs.

No.	Task	Required result
1	OLS with HC1 standard errors	Intercept and coefficients on `x1` and `x2`, with HC1 standard errors
2	ARIMA(1,1,1) forecast	One-step-ahead forecast and 95 percent prediction-interval bounds
3	OLS with one-way clustered standard errors	Intercept and coefficients on `x1` and `x2`, with standard errors clustered by `cluster`
4	OLS with two-way clustered standard errors	Intercept and coefficient on `x`, with standard errors clustered by firm and year
5	Variance inflation factors	VIF for each of five regressors
6	Cook's distance	Observation identifiers and Cook's distance for the five most influential observations
7	Welch two-sample test	Test statistic, degrees of freedom, and p-value
8	One-way ANOVA	Omnibus F statistic, numerator and denominator degrees of freedom, and p-value
9	Chi-square test of independence	Pearson χ^2 statistic, degrees of freedom, and p-value
10	Binary logit	Intercept and coefficients on `x1` and `x2`, with asymptotic standard errors
11	Binary probit	Intercept and coefficients on `x1` and `x2`, with asymptotic standard errors
12	Poisson regression with an exposure offset	Intercept and coefficients on `x1` and `x2`, with model-based standard errors
13	Cox proportional-hazards model	Log hazard ratios for treatment and `x`, with model-based standard errors
14	Two-way fixed effects	Coefficient on `x`, with firm-clustered standard error
15	Just-identified 2SLS	Intercept and coefficient on the endogenous regressor, with HC1 standard errors

(continued)

No.	Task	Required result
16	Arellano–Bond dynamic-panel GMM	One-step difference-GMM coefficients on lagged outcome and `x`; standard errors are not scored
17	Staggered difference-in-differences	Equal-weight average of supported cohort-time treatment effects; the standard error is not scored
18	Sharp regression discontinuity	Difference between local-linear intercepts at the cutoff, using bandwidth 0.5 and triangular weights
19	Local-projection impulse response	Horizon-four response coefficient, with Newey–West standard error using lag four
20	VAR(2) impulse response	Horizon-four orthogonal response of `y1` to a one-standard-deviation shock in `y2`
21	Granger causality test	Aligned-OLS F statistic, numerator and denominator degrees of freedom, and p-value

Notes: The task prompts define the model specification, sample, row names, and output schema. Appendix C describes selected reference definitions. Appendix D describes the scoring rules.

B Claude Sonnet 4.6 Agency Protocol

This appendix describes the agency conditions in the main Claude Sonnet 4.6 analysis. The protocol defines what the model can observe and do before submission.

Agency conditions The protocol covers the three agency conditions in the analysis sample. In the chatbot condition, the model has no tools. It writes one complete script; the harness executes it once after submission for the scoring record, with a 180-second timeout, and returns no run report to the model. In the one-repair condition, the model likewise has no computer-control tools. If execution fails or the required result file is missing, the harness returns a fixed message with the captured run output. The harness accepts one complete revised script, with the same 180-second execution timeout on both attempts. In the constrained-agent condition, shell execution is the only pre-authorized tool; the model works in a dedicated task folder holding the input data, and the outer session runs under a 600-second timeout. The prompt instructs it to write one script in the target language, run it through the shell tool, read errors and output, iterate until the result file is written, and declare completion. Requests for any other tool, including file read, write, or edit tools and web, API, or MCP tools, are denied and recorded; only the shell tool can execute.

Stopping rules Each agency condition has a fixed stopping rule for auditability and cost recording. These limits are part of the agency-condition definition. The chatbot condition ends after one submitted script. The collection harness could retain as many as three revisions in a one-repair record. The analysis applies a one-revision boundary. A record that requires more than one revision counts as a failure, and only the initial submission and first revision enter the cost ledger. The constrained-agent loop is capped at twelve model turns; the model may stop earlier by declaring completion. Total tool calls and turn counts are recorded per run.

Missing runs An attempted run that produces no scored script within its execution limit counts as a failure in the analysis. Provider interruptions, such as session or usage limits, produce no model attempt and are excluded. The next eligible response fills the corresponding analysis cell.

Configuration and audit fields The harness configures each Claude CLI invocation programmatically. For auditability, each run stores the model ID, prompt condition, available tools, permission mode, execution limits, setting sources, session identifier, stop reason, usage fields, cache-token fields, and estimated cost. The isolated configuration disables session continuation and records any deviation from the declared tool surface. The declared tools and permission mode define the agency condition. Requests for undeclared tools are denied and recorded, and `bypassPermissions` stays disabled.

Execution environment Saved Stata logs identify StataNow 19.5. The recorded worker environment uses R 4.6.0 and Python 3.14.6. The core Python packages are NumPy 2.4.2, pandas 3.0.0, SciPy 1.18.0, statsmodels 0.14.6, and linearmodels 6.1. The core R packages include AER 1.2.16, ivreg 0.6.7, fixest 0.14.1, plm 2.6.7, vars 1.6.1, forecast 9.0.2, survival 3.8.6, sandwich 3.1.1, and lmtest 0.9-40. Run records store the executable paths, although the zero-shot records do not contain a complete package-version snapshot for every run.

Execution result The execution component accepts a complete script or a command that runs the script and returns a structured result:

> statistical software; exit status; timeout indicator; stdout; stderr or Stata log excerpt; expected output file indicator; output file preview when present.

In all conditions, statistical scripts are routed to the selected Stata, R, or Python executable. The Claude Sonnet 4.6 conditions receive no hidden package search, hidden

retrieval, uncontrolled shell access, or undeclared MCP tools. Section 6 describes the broader Codex action surface.

C Reference Outputs and Selected Definitions

Each task declares one or more scored quantities. The automated scoring system compares every declared result row with the shared reference output. These comparisons enter output correctness, and the main analysis outcome is binary task success.

Each reference output is a task-keyed JSON record shared across the three software environments. Each estimate row contains `term`, `estimate`, `se`, and `n_obs`. The value of `se` can be null when the task does not score a standard error. Every task reports `n_obs`, although the scoring system does not compare it for tasks with convention-dependent sample counts. Statistics such as a p-value appear as named estimate rows when the task requires them.

The task definition and scoring rules jointly record the target quantities, inference convention, numerical tolerances, accepted implementation evidence, row keys, and comparison rules.

Reference definitions The dynamic-panel task uses the canonical one-step difference-GMM estimator (Arellano and Bond, 1991). In the benchmark dataset, the reference values are $\rho = 0.5583$ and $\beta_x = 0.5517$. Stata `xtabond` and R `plm::pgmm` agree to ten digits when they use the declared instrument set.

Sample-size reporting follows a declared convention: `n_obs` records the effective estimation sample. The input-file row count is stored separately. For five tasks the effective sample is genuinely convention-dependent: staggered DiD, VAR impulse response, ARIMA forecast, Cox proportional hazards, and dynamic-panel GMM. For example, the Cox model has 2,000 subjects against 1,470 events, and the dynamic panel has 800 observations against 100 firms. The scoring system does not compare `n_obs` for these tasks; the estimates remain scored.

Staggered-adoption DiD estimand and verification protocol The reference estimand is the uniformly weighted average of cell-level treatment effects:

$$\text{ATT}_{\text{uniform}} = \frac{1}{|S|} \sum_{(g,t)\in S} \text{ATT}(g,t), \qquad S = \{(g,t) : g \in \{4,6,8\},\ g \le t \le 9\},\ |S| = 12,$$

where each cell-level effect is the Callaway–Sant'Anna long difference (Callaway and Sant'Anna, 2021) with never-treated firms as the control group and the pre-treatment year $g-1$ as the base:

$$\text{ATT}(g,t) \;=\; \big[\,\overline{y}_{g,t} - \overline{y}_{g,g-1}\,\big] - \big[\,\overline{y}_{-1,t} - \overline{y}_{-1,g-1}\,\big],$$

with $\overline{y}_{c,t}$ the within-(cohort, year) mean of y_{did}.

The registry defines the estimand as the mathematical object above. A package call is documented as canonical for a software environment after it reproduces the manual long-difference value within tolerance. The manual-first verification protocol is:

1. Implement the long-difference average above in Python, R, and Stata using only basic mean, merge, and arithmetic operations.
2. Run each implementation on the locked benchmark dataset and confirm the three sample values agree within floating-point tolerance.
3. Register that cross-software sample value as the reference target.
4. Test candidate package calls (for example, `did::att_gt` with `aggte()`, `csdid`, `pyfixest.did2s`, or manual loops) against the manual reference; flag any aggregator whose weighting differs. For example, `aggte(type="simple")` uses cohort-time group-size weights, which differ from uniform weights across the supported cells.

On the locked staggered-DiD dataset (seed 20260515), the three manual reference scripts return $\text{ATT}_{\text{uniform}} \approx 1.589$. The DGP population truth derived from the heterogeneous cohort effects $(\tau_4, \tau_6, \tau_8) = (1, 2, 3)$ is $20/12 \approx 1.667$, and the sample-to-population gap is consistent with the variance structure of the panel (firm-level random effects, linear time trend, idiosyncratic noise). The scored output stores the point estimate and sets its standard error to null.

D Success Criteria and Planted Failure Tests

Success criteria Task success is a binary outcome. A run must satisfy all four criteria in Table 4. If any criterion fails, the primary outcome is failure, while the remaining diagnostic fields are still stored when they can be computed.

Table 4: Task-success criteria and evidence.

Criterion	Passing rule	Evidence used
Specification correctness	Script implements the intended estimator, sample restriction, controls or fixed effects, target quantity, and inference procedure	Generated script, execution log, task registry, sample diagnostics, task-specific method checks
Executability	Script runs within the time and execution caps without a runtime failure	Exit status, timeout flag, stdout, stderr, execution count
Output correctness	Run writes the required result file with the declared rows, columns, and types; declared target quantities match the shared reference output within declared tolerances	Submitted result file, result-file validator, row keys, column names, parse errors, shared reference output, tolerance table, row-matching rule
Submission validity	Run uses the declared input data and performs the requested computation; hard-coded answers, hidden-output reads, changes to the task inputs, and bypasses fail	Generated script, result file, output timing, hidden-path checks, reference-number checks, planted failure tests

The deterministic scorer evaluates execution, required output fields, numerical agreement, specification evidence, and submission validity.

Example scoring entry The task definition and scoring rules jointly encode the information illustrated below. The example below is the OLS task with one-way clustered standard errors.

Task. Estimate `y_panel` on `x1` and `x2` with an intercept.

Sample. Use the observations in the declared task dataset and report the estimation sample size.

Estimator. Linear regression with an intercept and both declared regressors.

Inference. One-way cluster-robust standard errors clustered by `cluster`.

Required result file. `result.json` with top-level fields `task_id`, `platform`, `estimates`, `method_metadata`, and `diagnostics`. The estimates contain rows for `const`, `x1`, and `x2`, with fields `term`, `estimate`, `se`, and `n_obs`.

Output-correctness rule. Required structure, with each coefficient and standard error within its declared tolerance relative to the shared reference output.

Serialization and convention normalizations The scoring system normalizes output dialects and reporting conventions that are orthogonal to econometric correctness before applying the numeric comparison. It records these rules and applies them uniformly to all runs.

- R `jsonlite` serialization: an empty `diagnostics` or `method_metadata` object can be written as an empty array; empty arrays and empty objects are treated as equivalent in these fields.
- Stata numeric serialization: decimals without a leading zero and bare-dot missing values are repaired into valid JSON only if a strict parse fails.
- Sign conventions: the Welch t statistic is compared in magnitude because its sign encodes an arbitrary group ordering; degrees of freedom and p-value are compared directly.
- Reporting conventions: for the Granger test both the aligned-OLS and the system-Wald denominator degrees-of-freedom conventions are accepted, since the F statistic and p-value are consistent across them.
- Method evidence: recognized software idioms for the same estimator are accepted (for example, `felm/lfe` alongside `fixest` for high-dimensional fixed effects, and analytic-weight syntax for kernel weighting), so specification correctness does not penalize a legitimate implementation route.
- Execution status: a run that produced a parseable result file counts as executable and as a model response even when the wrapper process exits with a nonzero status.

Sample-size conventions and reference definitions are documented in Appendix C.

Planted-failure fixtures A runnable seven-fixture suite tests the scorer against the following planted-failure taxonomy:

- wrong estimator: executable code estimates a different model;
- wrong sample: code silently changes the sample restriction;
- wrong inference: the coefficient is correct; the variance estimator or clustering rule is wrong;
- wrong fixed effects: code omits or changes fixed effects;
- wrong output: printed estimates are correct; the required result file is malformed or incomplete;
- hard-coded output: code writes the reference number without implementing the estimator;
- package-default failure: output differs because defaults are not harmonized.

E Cost Ledger

Schema The cost ledger is built from the same run records used for scoring. Each row records the model, agency condition, prompt, statistical software, task, repetition, task

success, script executions, token use, and estimated cost when available. Claude Sonnet 4.6 records include input, output, cache-creation, and cache-read tokens plus estimated cost. GPT-5.4 records include input, output, cached-input, and reasoning-output tokens. DeepSeek records include the same token fields and the billed OpenRouter charge. Turn counts and stopping reasons are available for Claude Sonnet 4.6 and for execution-enabled DeepSeek runs. The paper reports API-equivalent dollar amounts for all three model arms. Claude Sonnet 4.6 uses the client-side estimate produced from its recorded token categories. Codex records token use without a billed dollar amount, so its value applies the GPT-5.4 Standard base token prices; the ChatGPT subscription did not bill these imputed amounts. DeepSeek uses the billed API charge stored with the run.

Reporting For each model, statistical software, prompt, and agency condition, the ledger reports the available token, execution, and cost fields. Runs without recorded cost telemetry are excluded from cost averages. For the one-repair condition, cost per run sums only the first two attempts, the initial script and one revision, matching the one-revision boundary in Appendix B. Cost per successful run equals mean cost per run divided by the task-success rate.