Chatbots, Agents, and the Choice of Econometric Software

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

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

July 2026

Abstract

We study how large language models write code for econometric analysis. We measure how performance varies with the statistical software (Stata, R, or Python), the prompt, and the degree of agency, with three levels from a chatbot that only writes code to a more agentic harness that executes its own code. We create a benchmark of applied econometric and statistics tasks. We find that a higher degree of agency afforded to the model through the harness produces higher success rates, at a higher cost. We also analyze this trade-off between gains in correct output from more agency and token costs. The main results use Claude Code. We reproduce the results using OpenAI's Codex and DeepSeek-V4 through OpenCode.

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

1 Introduction

Empirical economists usually pose a causal question and set up an econometric model to answer it. They estimate the parameters of interest and state identification assumptions (Garg and Fetzer, 2025). Code is usually written in R, Stata, or Python to process the data and compute the estimates. Modern LLMs can help a lot with writing code, which is a step that can be quite mechanical and 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 project attempting to replicate several social science papers finds coding errors in about a quarter of the papers, often alongside posted code that reproduces the published numbers exactly (Brodeur et al., 2026).

The practical question is how much of the technical implementation a researcher should delegate to an LLM. The researcher retains the causal 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 code for econometric analysis. We measure how task success and cost vary with the statistical software, the prompt, and the degree of agency. The software dimension compares Stata, R, and Python. 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 agency dimension varies what the model can see and do between receiving the task and submitting 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* runs code in a capped shell-execution loop, reads the output, revises, and decides when to submit. We evaluate every combination on a benchmark of applied econometric tasks. We also measure what each agency condition costs, and we repeat the experiment with models from other providers.

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 tasks in statistics. Each task specifies its dataset, target estimand, reference output, and scoring tolerances. The runs vary by statistical software, prompt, and degree of agency. 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.

A single automated scoring system scores every run, and the main estimates use Claude Sonnet runs across all tasks, software, and agency conditions. We analyze whether examples improve task success, whether higher levels of agency improve task success 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 74.0 percent under the chatbot, 82.9 percent under one-repair, and 95.6 percent under the constrained agent. The gains come primarily from executability and creating the required result file. For Claude Sonnet, the design-averaged few-shot effect is 5.9 percentage points, with a wild-bootstrap p-value of 0.012. Under the zero-shot chatbot, Stata is an estimated 33.3 percentage points below Python, with a wild-bootstrap p-value of 0.005. Across the full benchmark, task success is 88.6 percent in Python, 88.9 percent in R, and 74.9 percent in Stata. The remaining failures involve estimator variants, data conventions, and misinterpretations of the requested calculation that execution alone does not reveal.

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 associated with moving from the chatbot to a higher level of agency. Across the full benchmark, the constrained agent raises mean cost per run from \$0.17 to \$0.25 and costs \$0.26 per successful run, compared with \$0.23 for the chatbot. The incremental cost is \$0.38 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 Claude Code with Codex and DeepSeek. 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 find incomplete computational reproducibility and coding errors in roughly one quarter of studies (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). Recent work also 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 statistical analysis (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).

Execution feedback and tools expand what a model can do 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), and 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). On research code, Kim et al. (2026) mask functions from the codebases of replicated machine-learning papers and find that letting the agent choose its actions and debug after failed runs raises the pass rate of one model more than fourfold relative to a predefined pipeline. The closest design uses the same model across degrees of agency in the repair of existing R code (Shah et al., 2026). Full-agent benchmarks add file inspection, editing, command execution, and model-directed submission (Jimenez et al., 2024; Merrill et al., 2026; Kwa et al., 2025). Li et al. (2026) describe the surrounding engineering layer as an agent harness.

Our design measures how examples, statistical software, and three agency conditions change task success and cost on applied econometric tasks.

3 Design

3.1 Design dimensions

The principal 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. Each cell combines one task, one statistical software environment, one prompt, and one agency condition. The analysis uses five runs in every cell. 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. Recorded expectations for each design cell are reproduced in Appendix H.

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. Figure 1 summarizes the design, the three agency conditions, and the evaluation rule.

Run records retain task success, the number of script executions, and token use. Appendices C and G describe the agency protocol and the model-specific fields (Anthropic, 2026b,c).

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. Generated code runs on the task inputs and is compared 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

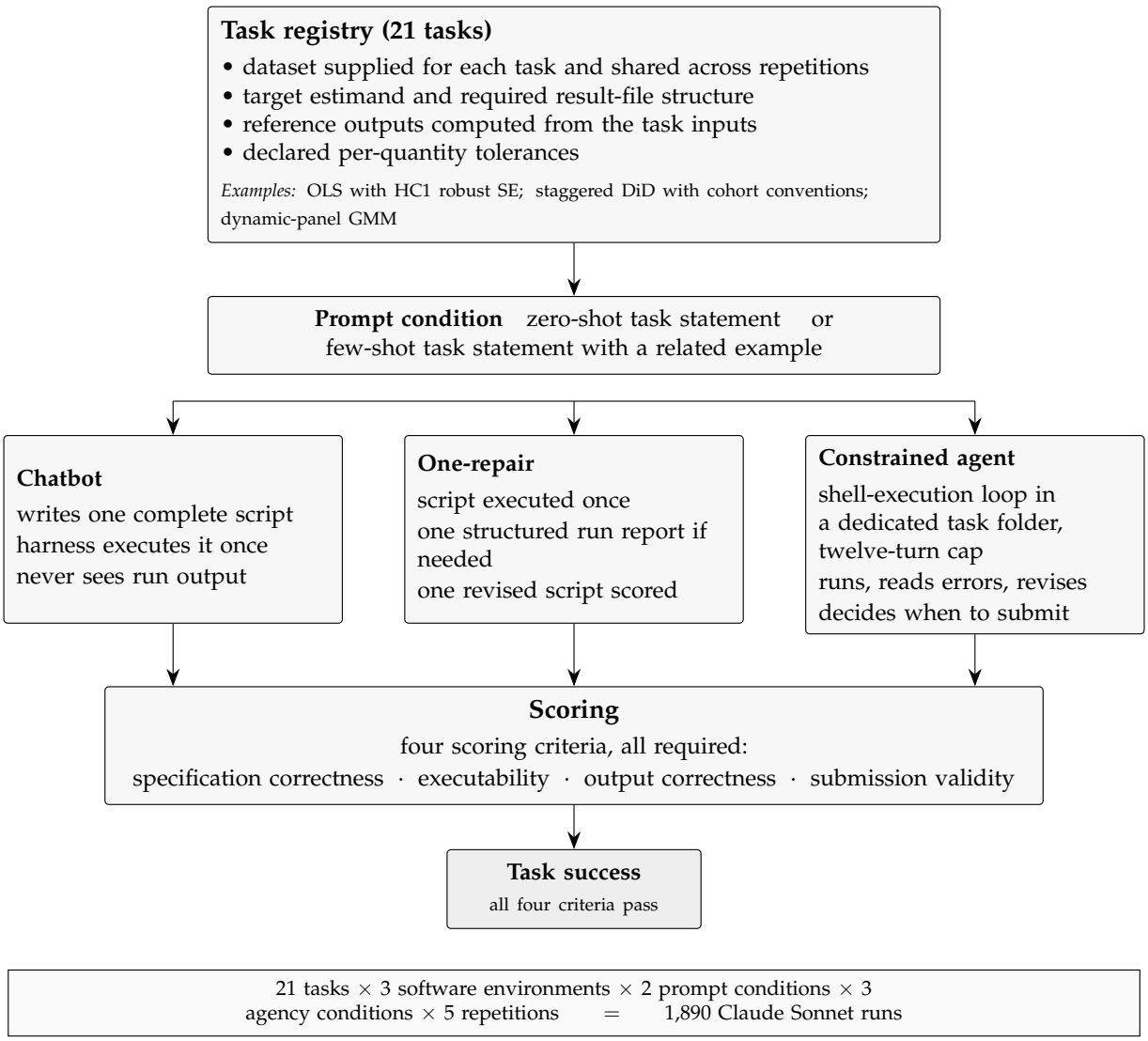

Figure 1: Benchmark design. Each of the 21 tasks specifies a dataset, a target estimand, reference outputs computed from the task inputs, and per-quantity tolerances. A zero-shot or few-shot prompt poses the task under one of three agency conditions. The automated scoring system applies four criteria, and a run succeeds when all four pass.

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. The full registry is in Appendix A, with concrete examples of scored targets and evaluation checks in Table 3.

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 required 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 hard-coded reference numbers. The scoring system normalizes serialization dialects before scoring so the software comparison accounts for output formatting (Appendix E). 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).

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. Cluster-robust standard errors are clustered at the

task level; with 21 analyzed 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). Robustness specifications, power notes, and the extended hypothesis discussion are in Appendix H.

4 Results

4.1 Analysis sample

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

4.2 Effectiveness across agency conditions

Overall task success in the analysis sample is 84.1 percent, 81.2 in the zero-shot condition and 87.1 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 90.4 and 81.1 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 74.0 percent under the chatbot condition to 82.9 percent under one-repair and 95.6 percent under the constrained agent.

Panel B of Table 1 reports the estimated changes in task success defined after equation (1).

On H3, one structured run report adds 8.9 percentage points over the chatbot baseline, and the constrained agent adds 21.6 percentage points; both bootstrap p-values are below one in a thousand. The one-repair estimate applies the one-revision boundary stated in Appendix C, which scores a run as a failure when success requires more than one revision.

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 ($p = 0.869$). The joint test rejects the hypothesis of no software gap at baseline ($p = 0.006$). Software differences are also visible in the raw rates. Across the full

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
Zero-shot	One-repair	90.5	81.3	81.3
Zero-shot	Constrained agent	99.7	94.9	94.9
Few-shot	Chatbot	91.1	81.6	80.6
Few-shot	One-repair	94.6	85.1	84.4
Few-shot	Constrained agent	100.0	96.8	96.2

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

Comparison	Estimated change
H1a: R versus Python	-1.9 (7.9)
H1b: Stata versus Python	-33.3^{***} (10.1)
H2: Few-shot versus zero-shot	5.9^{**} (2.2)
H3a: One-repair versus chatbot	8.9^{***} (2.2)
H3b: Constrained agent versus chatbot	21.6^{***} (4.2)
H4a: Few-shot $\times$ one-repair	-10.2^{**} (4.3)
H4b: Few-shot $\times$ constrained agent	-12.1^{*} (5.9)
Additional: Few-shot versus zero-shot under chatbot	13.3^{***} (4.7)

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.

benchmark, task success is 88.6 percent for Python, 88.9 percent for R, and 74.9 percent for Stata. Table 7 in Appendix I splits task success by software, agency condition, and prompt condition. In the zero-shot condition, Stata rises from 45.7 percent under the chatbot condition to 98.0 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.

4.3 Effects of few-shot examples

Few-shot examples raise overall task success from 81.2 to 87.1 percent. On H2, the design-averaged few-shot effect is 5.9 percentage points ($p = 0.012$). The design average conceals agency heterogeneity. Within the chatbot condition, few-shot examples add 13.3 percentage points ($p = 0.006$). The raw prompt gains are 13.3 percentage points under the chatbot condition, 3.2 under one-repair, and 1.3 under the constrained agent.

On H4, both interactions between prompt and agency conditions are negative. The few-shot-by-one-repair interaction is -10.2 percentage points ($p = 0.019$), and the few-shot-by-constrained-agent interaction is -12.1 points ($p = 0.060$).

4.4 Task-level variation

Figure 2 groups the task-level results by econometric family.

Task-level rates locate the sources of agency gains and 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 55.6 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 10.0 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 66.7 percent.

Staggered DiD is a data-discovery failure. Success is 26.7 percent under the chatbot condition, 30.0 percent under one-repair, and 96.7 percent under the constrained agent. The input dataset codes never-treated firms as cohort -1, a convention the model must discover from the data.

Table 8 in Appendix I records the selection rules used in the design and the examples they identify.

Silent failures quantify the residual risk. Among runs that execute and write a valid result file, 11.1 percent still fail task success under the chatbot condition. The corresponding rates are 7.6 percent under one-repair and 4.3 percent under the constrained agent. These failures require inspection of the specification and results because execution alone does not reveal them.

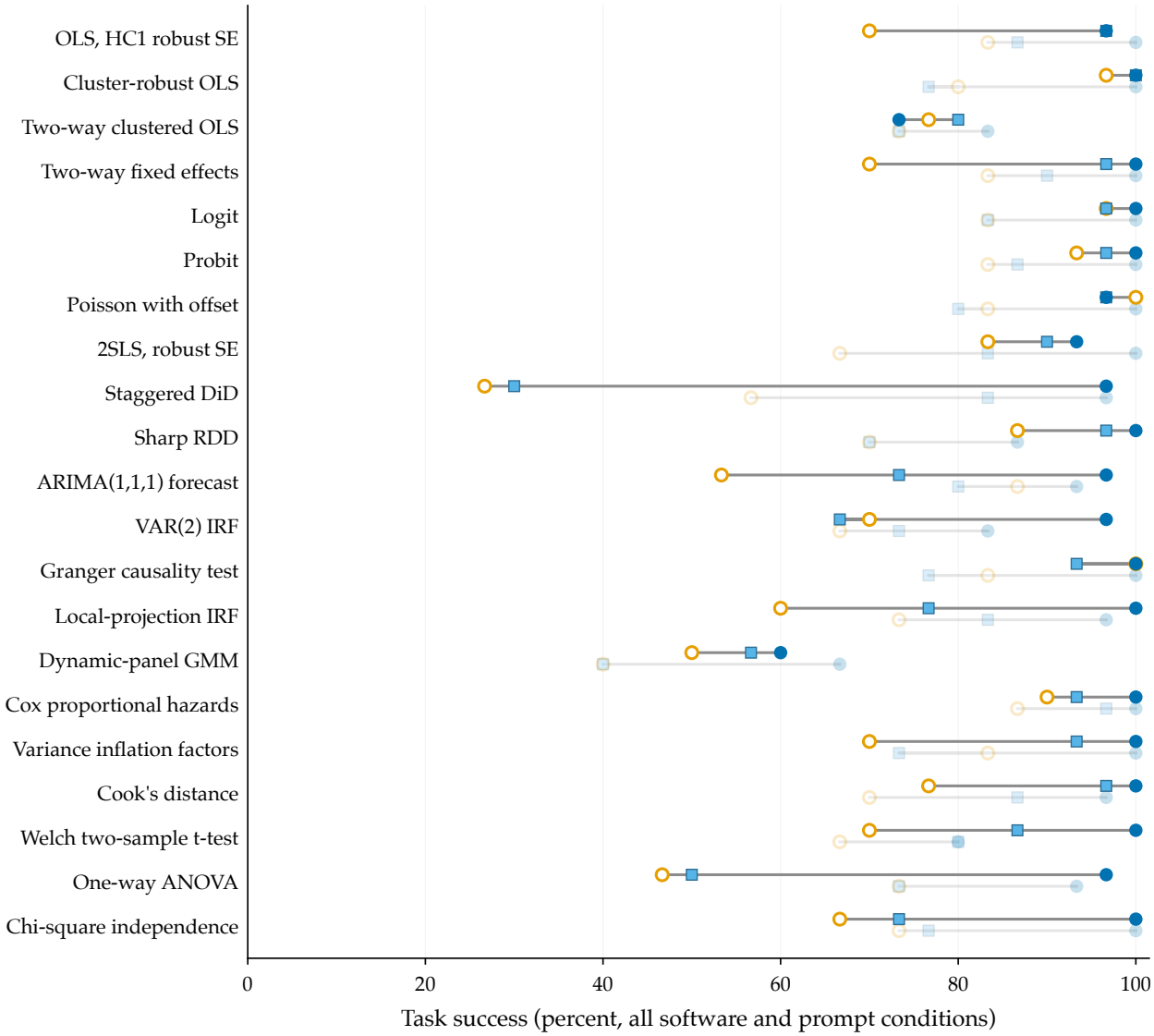

Figure 2: Task success by task and agency condition for Claude Sonnet and GPT-5.4 through Codex. Claude Sonnet appears at full opacity and Codex at lower opacity. Each path uses all software and prompt conditions. Open circles show chatbot success, squares show one-repair success, and filled circles show constrained-agent success. Rows group regression, instrumental variables, difference-in-differences and regression discontinuity, time series and dynamic panels, and other tasks.

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. Appendix E lists the normalization and verification rules. 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 87.1 to 86.7 percent in the few-shot condition. Doubling every tolerance leaves the zero-shot rate unchanged and moves the few-shot rate to 87.2 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.

Agency conditions that return run output solve nearly all runtime and result-file failures and most numeric failures. Residual failures involve estimator variants, data conventions, and misinterpretations of the requested calculation. Specification review remains necessary because these scripts can execute successfully.

5 The Cost of Agency

This section compares the expenditure and task success associated with each agency condition. 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. Dollar figures for the main sample use both prompt conditions unless a prompt condition is named. Appendix G describes field availability and reporting rules.

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 successful run alongside task success. Cost per successful run equals mean cost per run divided by the task-success rate. Figure 3 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. Within each prompt condition the constrained agent adds about five cents to the chatbot's median per-run cost, $0.17 against $0.12 under zero-shot prompts and $0.25 against $0.20 under few-shot prompts. The constrained agent has the highest cost per successful run, at $0.26 against $0.23 for the chatbot. The one-repair condition has the same $0.23 cost per successful run as the chatbot while raising task success from 74.0 to 82.9 percent. Its median cost per run matches the chatbot's within each prompt condition. The constrained agent produces higher task success and higher expenditure per successful run.

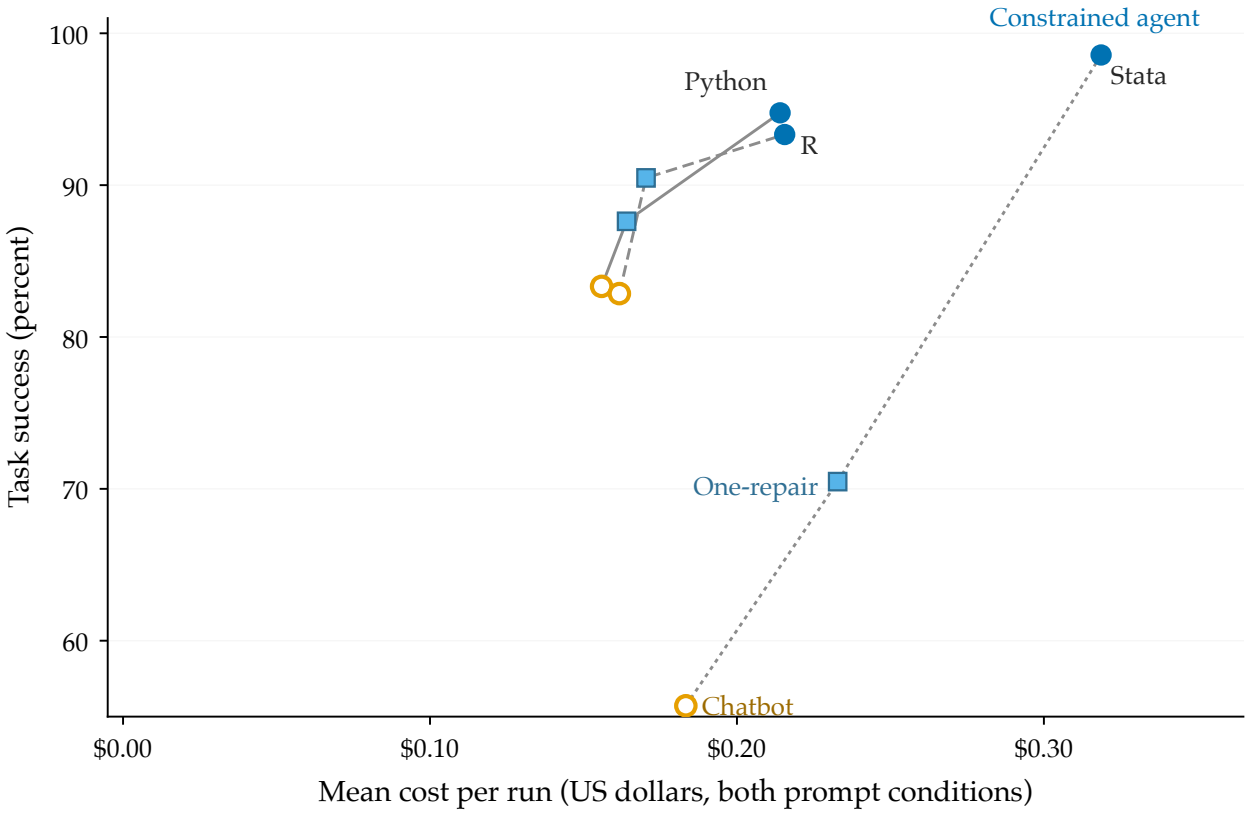

Figure 3: Claude Sonnet cost and task success by statistical software and agency condition. Points use recorded mean cost per run. Lines connect chatbot, one-repair, and constrained-agent cells within each software environment.

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 21.6 percentage points and mean cost per run by about eight cents. This gives an incremental cost of $0.38 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 $0.30 per additional successful run. Under few-shot prompts, the same change adds 15.6 percentage points at $0.53 per additional successful run.

Typical costs are highest in Stata. Median per-run cost is $0.19 for both Python and R against $0.25 for Stata. Cost per successful run is $0.20 for Python, $0.21 for R, and $0.33 for Stata. The pattern mirrors the success results.

6 Other Models

The main estimates use Claude Sonnet. We apply the same design and analysis to GPT-5.4 through the Codex command line interface and DeepSeek V4 Flash through OpenRouter. 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 categories. 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 or DeepSeek.

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

Panel A: Task success (percent)

Prompt	Claude Sonnet	GPT-5.4	DeepSeek V4 Flash
Zero-shot	81.2	78.1	57.7
Few-shot	87.1	86.9	72.1

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

Comparison	Claude Sonnet	GPT-5.4	DeepSeek V4 Flash
H1b: Stata versus Python	-33.3^{***}	-76.2^{***}	-66.7^{***}
	(10.1)	(5.3)	(7.9)
H2: Few-shot versus zero-shot	5.9^{**}	8.8^{***}	14.4^{***}
	(2.2)	(2.3)	(2.2)
H3a: One-repair versus chatbot	8.9^{***}	4.1^{*}	8.3^{***}
	(2.2)	(2.0)	(2.4)
H3b: Constrained agent versus chatbot	21.6^{***}	19.5^{***}	18.7^{***}
	(4.2)	(1.7)	(3.2)

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

Prompt	Claude Sonnet	GPT-5.4	DeepSeek V4 Flash
Zero-shot	\$0.22	\$0.13	\$0.0051
Few-shot	\$0.30	\$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. H1b compares 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. 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.

Within each model arm, execution-enabled agency raises task success relative to the chatbot. The constrained-agent contrast is 21.6 percentage points for Claude Sonnet, 19.5

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 5.9 points for Claude Sonnet, 8.8 for GPT-5.4, and 14.4 for DeepSeek. Stata has the lowest zero-shot chatbot rate in all three arms.

All three cost measures value recorded token use at the relevant API price. Claude Sonnet 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 the Codex tokens at the GPT-5.4 Standard base token rates. Its table entries are API-equivalent amounts, and the ChatGPT subscription did not bill these amounts. The final two rows of Table 2 compare cost per successful constrained-agent run under each prompt condition.

7 Conclusion and Limitations

This paper builds a controlled benchmark for LLM econometric coding and scores 1,890 Claude Sonnet runs on 21 tasks across prompt structures, statistical software, and degrees of agency. Each task uses its own dataset, reference output, and scoring rule in every condition. Across both prompt conditions, task success is 74.0 percent under the chatbot, 82.9 percent under one-repair, and 95.6 percent under the constrained agent. The constrained agent costs $0.26 per successful run, compared with $0.23 for the chatbot, and the incremental cost is $0.38 per additional successful run. Some remaining failures execute successfully while implementing a different specification. Domain review therefore remains necessary.

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 the benchmark's scope.

References

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*.

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. and J. M. Shapiro (2014, March). Code and data for the social sciences: A practitioner's guide. Manuscript, 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*.

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 Task List and DGP Schema

The registry defines the 21 scored tasks. Each retained task has one benchmark dataset used across software, prompt conditions, agency conditions, and repetitions. The dataset may be simulated from a DGP or adapted from a documented benchmark source. The reference output is always generated before model runs begin.

Task schema Each registry entry records the following required fields.

Task ID and family. Stable identifier, task family, and difficulty tier (easy, medium, hard).

Dataset. Path, source, generation seed when simulated, unit of observation, and sample size.

DGP or data source. Seeded DGP, real dataset, or adapted benchmark dataset; for DGP tasks, the baseline parameter values and sensitivity grid.

Econometric design. Target estimator, sample restriction, covariates, fixed effects, and inference convention.

Primary estimand. One primary row used for task success and the main analysis contrasts.

Reference scripts. Stata, R, and Python scripts that produce the software-specific reference output, including the reference package for each software environment.

Required result file. Required column names, row names, and file format of the result file written by the run.

Scoring tolerances. Per-quantity numerical tolerances for estimates, standard errors, test statistics, and any declared p-value or impulse-response quantity.

Method checks. Deterministic command, function, package, or code-structure evidence for the intended method.

Asymmetry flag. Whether implementation burden or package support differs materially across software.

Variation rationale. Which design-factor contrast the task informs, and whether it is expected to be near-floor, near-ceiling, or in the informative middle of the success rate.

For simulated tasks, the DGP note records the sampling process, parameter values, target population estimand, finite-sample reference estimand, and any reason the reference output may differ across software. For adapted tasks, the note records the source, any modifications, and why the task remains a common applied statistical or econometric task. Tasks that remain asymmetric across software after reference script harmonization carry the asymmetry flag in the registry.

Task list Table 3 illustrates the unit of scoring. A generated script must write the declared result file and match the reference output for that task and software environment.

Table 3: Concrete examples of task sources and reference-output targets.

Example	Scored output	Scoring checks
OLS with HC1 SE	Slope coefficient and HC1 robust SE	Correct regressor, sample size, HC1 variance convention, no hidden reference read
Welch two-sample test	Difference in group means, Welch t, degrees of freedom, p-value	Direction of the contrast, unequal-variance formula, group labels
VIF and Cook's distance	VIF values or top-five influence rows	Same model matrix, same observation identifiers, correct diagnostic definition
Logit, probit, and Poisson offset	Coefficients, marginal effect when declared, offset coefficient convention, standard errors	Link function, offset handling, reference category, sample
Cox proportional hazards	Log hazard ratio and model-based SE	Risk set construction, censoring treatment, covariate specification
ARIMA, VAR IRF, and Granger test	Forecast, impulse-response coefficient, or Granger F-test and p-value	Lag length, ordering, horizon, deterministic terms
Staggered DiD and RDD	ATT or local treatment effect at the declared target	Cohort-time target, control group, bandwidth or support rule, inference convention

Notes: The examples show the unit of scoring for the full 21-task registry listed below.

The list below contains the $N = 21$ tasks in the benchmark. Task IDs, datasets, and reference packages are recorded in the registry. The list is designed to give the analysis in Section 3.4 the variation it relies on. Three design rules govern selection:

1. **Family coverage.** The eight families listed in Section 3.3 and illustrated in Table 3 are all represented, so that the software main effects and software-by-agency interactions are identified from variation across estimators.
2. **Difficulty spread.** The list spans easy, medium, and hard tasks, with most tasks in the informative middle and final difficulty tiers calibrated by locked run batches. The spread is what gives the contrasts in Section 3.4 room to detect positive gains and ceiling effects.
3. **Software symmetry.** Most tasks are implementable in a comparable way on Stata, R, and Python (the symmetric subset). A small number are deliberately asymmetric, with one software environment providing the reference workflow; these are flagged and reported separately.

Table 4: The $N = 21$ task list with difficulty, symmetry, and reference target.

No.	Task (family)	Tier	Sym.	Reference target and notes
1	OLS, single regressor, HC1 robust SE (core regression)	E	Y	Coefficient and HC1 SE; simulated, $n = 1000$
2	ARIMA(1,1,1) one-step-ahead forecast with 95% prediction interval (time series)	M	Y	Point forecast at $h = 1$ and 95% PI bounds; simulated ARIMA series, $n = 250$
3	OLS with one-way clustered SE (core regression)	M	Y	Coefficient and cluster-robust SE; simulated panel, 50×20
4	OLS with two-way clustered SE, firm $\times$ year (core regression)	M	Y	Coefficient and two-way clustered SE; same simulated panel as task 3
5	VIF for a five-variable linear model (diagnostics)	E	Y	VIF vector; translated dataset
6	Cook's distance, list the five most influential observations (diagnostics)	M	Y	Top-five observation IDs and Cook's D values; translated dataset
7	Two-sample t-test, unequal variance (tests)	E	Y	Test statistic, degrees of freedom, p-value; simulated
8	One-way ANOVA with Tukey HSD post-hoc (tests)	M	Y	F, p, pairwise comparison table; simulated
9	Chi-square test of independence (tests)	E	Y	χ^2, degrees of freedom, p-value; translated contingency dataset
10	Logit with marginal effects at the mean (GLM)	M	Y	Coefficients and average marginal effects; simulated
11	Probit with one continuous and one binary regressor (GLM)	M	Y	Coefficient and asymptotic SE; simulated
12	Poisson regression with an offset term (GLM)	M	Y	Coefficient and SE; simulated count data
13	Cox proportional hazards with binary treatment and one continuous covariate (survival)	M	Y	Log hazard ratios for treatment and covariate with model-based SE; simulated right-censored panel, $n = 2{,}000$, Weibull baseline, $\approx 30\%$ censoring
14	Two-way fixed effects with clustered SE (panels)	M	Y*	Coefficient and cluster-robust SE; Y* records additional cross-software verification for the high-dimensional fixed-effects workflow
15	2SLS with first-stage F-statistic (endogeneity)	M	Y	Coefficient, SE, and first-stage F; simulated IV design
16	Dynamic panel GMM (Arellano–Bond) (panels, dynamic)	H	N	One-step difference-GMM coefficient and SE; reference verified identically in Stata and R; see Appendix D

(continued)

No.	Task (family)	Tier	Sym.	Reference target and notes
17	Staggered-adoption difference-in-differences under heterogeneous cohort effects, uniformly weighted ATT(g,t) (treatment)	M	Y	Uniformly weighted Callaway–Sant'Anna $\text{ATT}_{\text{uniform}}$ over supported post-treatment cells with never-treated controls; manual scripts in Python, R, and Stata agree at ≈ 1.589 on the locked dataset (seed 20260515); DGP population truth $20/12 \approx 1.667$; scored quantity is the point estimate; estimand and verification protocol in Appendix D
18	Sharp regression discontinuity at a known cutoff with bandwidth selection (treatment)	H	N*	Local-linear LATE and bandwidth; N* records a software difference in package support
19	Local projection IRF at horizon four, Newey–West SE (time series)	M	Y	IRF coefficient and SE at $h = 4$; simulated AR(1) shock process
20	VAR(2) orthogonal impulse response of y_1 to a one-SD shock in y_2 at horizon four (time series)	M	Y	IRF coefficient at $h = 4$ under Cholesky identification with y_1 ordered first; simulated bivariate VAR(2), $n = 200$
21	Granger causality F-test: does y_2 Granger-cause y_1 with two lags (time series)	M	Y	F-statistic, degrees of freedom, p-value from a two-lag OLS lag-augmented regression; same simulated VAR dataset as task 20

Notes: Tier is E = easy, M = medium, H = hard, defined by expected baseline (chatbot, Python) task-success rate. Sym. is Y if the task is implementable in a comparable way on Stata, R, and Python at the level of standard packages; N if one software environment provides the reference workflow; Y* and N* indicate borderline cases that are reported in both the full and the symmetric-subset analyses. Reference packages recorded in the registry: task 2, `statsmodels.tsa.arima.ARIMA` (Python), `forecast::Arima` (R), `arima` (Stata); task 13, `survival::coxph` (R), `lifelines.CoxPHFitter` or `statsmodels.PHReg` (Python), `stcox` (Stata); task 14, `reghdfe` (Stata), `fixest` (R), `pyfixest` (Python); task 16, `xtabond` (Stata), `plm::pgmm` (R); task 18, `rdrobust` (R, Stata), `rdrobust-py` (Python); task 20, `statsmodels.tsa.api.VAR` (Python), `vars::VAR` with `vars::irf` (R), `var` and `irf` (Stata); task 21, `lmtest::grangertest` (R), `statsmodels.tsa.stattools.grangercausalitytests` (Python), `vargranger` (Stata). Manual verification is recorded in the task registry for tasks 2 (long-run check), 20, and 21. Software and package pinning is described in Appendix F.

The list distributes tasks across the analysis-relevant axes as follows. Expected difficulty includes four easy, fifteen medium, and two hard tasks. Software symmetry includes eighteen symmetric tasks, one asymmetric task, and two borderline cases. Family coverage includes three core-regression-with-inference tasks, two diagnostics tasks, three classical-tests tasks, three GLM tasks, one survival task, three panel-and-endogeneity tasks, two treatment-effect tasks, and four time-series tasks. This composition provides variation

in baseline success rates, estimator families, and software symmetry for the analysis in equation (1) and the robustness checks described in Section 3.4.

B Prompt Template

Each prompt is rendered from the same template, with software-specific syntax instructions and path names inserted by the experimental implementation. The prompt has five blocks:

1. target software: Stata, R, or Python;
2. dataset path and variable dictionary;
3. econometric task and sample restriction;
4. required estimator, inference convention, and output quantities;
5. required result-file structure and output file path.

The literal prompt files are stored in the repository and omitted from this appendix.

C Agency-Condition Protocol

The protocol defines what each agency condition exposes to the model. The benchmark uses the same logical protocol in the Claude Code command line interface and in the Claude Agent SDK interface, with any unsupported control recorded in the run metadata.

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 script is executed after submission for the scoring record with a 180-second timeout, and no run report is returned 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 plus the captured run output and 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 one-repair condition ends after the single revision round. A one-repair run without a valid result after its revision counts as a failure. Only the initial submission and its 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 completeness record counts both categories for every cell.

Configuration and audit fields The SDK implementation is configured 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 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 pinned Stata, R, or Python executable according to the software condition. No condition receives hidden package search, hidden retrieval, uncontrolled shell access, or undeclared MCP tools.

D Reference Outputs and Primary Estimand Protocol

The task registry declares one primary estimand for each of the 21 scored tasks. The primary estimand is the reference quantity used by reviewers when assigning the task-success label and in the main analysis contrasts. Additional reportable quantities can be required for output correctness. The primary estimand remains the quantity used in the main analysis contrasts.

Each reference output is a JSON record produced by the software-specific reference script. The required fields per estimate are `term`, `estimate`, and `se`, unless a task explicitly declares that the standard error is inapplicable. A `p_value` field is required only for declared inference targets, and an `n_obs` field is required when sample size is part of the estimand. Additional fields for specific tasks can record quantities such as horizon or treatment cohort.

For every task, the reference output note records the software-specific command or function, package version, inference convention, and numerical tolerance. When the primary target is an impulse response, treatment effect estimate, or other nonstandard quantity, the task definition records the exact row key and comparison rule. The exact table by task is stored with the task registry.

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, dynamic-panel GMM; for example, 2,000 subjects against 1,470 events in the Cox model, or 800 observations against 100 firms in the dynamic panel), and `n_obs` is not scored 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:

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

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:

$$\mathrm{ATT}(g,t) \;=\; \left[\,\overline{y}_{g,t} - \overline{y}_{g,g-1}\,\right] - \left[\,\overline{y}_{-1,t} - \overline{y}_{-1,g-1}\,\right],$$

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 / 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 standard error proxy recorded alongside the point estimate is the cross-cell standard deviation divided by $\sqrt{|S|}$; the scored quantity is the point estimate.

The same manual-first protocol applies to the ARIMA forecast, VAR impulse response, Granger causality, and simulated Cox proportional-hazards tasks described in Table 4.

E Success Criteria and Planted Failure Tests

Success criteria Task success is a binary outcome. A run must satisfy all four criteria in Table 5. If any criterion fails, the primary outcome is failure, while the remaining diagnostic fields are still stored when they can be computed.

Table 5: 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, package versions, sample diagnostics, task-specific method checks

(continued)

Criterion	Passing rule	Evidence used
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 software-specific reference output within declared tolerances	Submitted result file, result-file validator, row keys, column names, parse errors, software-specific 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	Script, file-access trace when available, output timing, hidden-path checks, planted failure tests

The deterministic scorer evaluates execution, required output fields, numerical agreement, specification evidence, and submission validity. A runnable seven-fixture suite tests legitimate controls, copied values, truncated hard-coding, and incorrect inference methods. The suite also records known limits of the current scorer.

Example scoring entry The registry contains a record of this form for each task-software pair. The example below is an OLS task with one-way clustered standard errors.

Task. Estimate the effect of `treat` on `y` with controls `x1` and `x2`.

Sample. All observations in the task dataset after dropping rows with missing values in the declared variables; report the final sample size.

Estimator. Linear regression with the declared controls; no transformation of `y` or `treat`.

Inference. One-way cluster-robust standard error clustered by `firm_id`.

Required result file. CSV with one row keyed by `treat` and columns `parameter`, `estimate`, `std_error`, and `nobs`.

Output-correctness rule. Required structure, with the coefficient and standard error within their declared tolerances relative to the software-specific 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. The rules are recorded in the task registry and applied 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 D.

Planted-failure fixtures A planted-failure taxonomy guides scoring-system tests:

- 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 standardized CSV 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.

F Packages and Software Notes

The package note for each task records the reference command or function in each software environment and any harmonization needed for defaults.

The following differences are recorded before scoring:

- robust and clustered standard-error conventions;
- finite-sample corrections for OLS, IV, panel, and GMM estimators;
- likelihood parameterizations for limited dependent variable models;

- tie handling, bandwidth choice, or bootstrap rule when applicable;
- package support gaps that require custom reference code in one software environment.

Tasks that are asymmetric across software remain in the benchmark because software ecosystem burden is part of the research question. Each asymmetric task is flagged before data collection, with the reason stated as absent canonical command, longer reference script, nonmatching package default, or a software-specific output convention.

G 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 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 OpenRouter charge. Turn counts and stopping reasons are available for Claude Sonnet and for execution-enabled DeepSeek runs. The estimated-dollar field is a client-side estimate used for planning and cell comparison. All three model arms report API-equivalent dollar amounts from recorded token use. 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 the imputed amounts.

Reporting For each model, statistical software, prompt, and agency condition, the ledger reports the available token, execution, and cost fields. 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 C. Cost per successful run equals mean cost per run divided by the task-success rate.

H Extended Design Notes

H.1 Recorded expectations by cell

Table 6 reproduces the recorded expectations for each combination of software, prompt, and agency conditions.

Table 6: Experimental cells and design interpretation.

Software	Zero-shot	Few-shot
Panel A. Chatbot condition: prompt and software only		
Stata	Standardized Stata syntax may be enough on its own; failures here would likely come from semantic mistakes that running code cannot catch.	Examples might still help if they disambiguate option choice or the required result-file structure.
R	Failures may surface package, formula, and object errors that the model cannot self-correct without seeing execution output.	Examples might guide package choice and creation of the required result file, or induce template copying if the example anchors too strongly.
Python	Broad training exposure could compete with fragmented econometrics APIs; we may see uneven performance across estimators.	Examples might offset API fragmentation, or pull the model toward the demonstrated API even when a better one exists.
Panel B. One-repair condition: one structured run report		
Stata	Terse Stata logs can be too sparse to diagnose semantic errors; the gain over Panel A is the test.	Examples plus logs might correct command, option, and syntax choices.
R	R errors may support package and formula correction when the model recognizes ecosystem cues.	The run report might target the intended estimator, or drift toward nearby package fixes that compile but mis-estimate.
Python	Detailed tracebacks may improve reliability where fragmented APIs cause silent or partial failures.	Examples and tracebacks could be complements (different information) or substitutes (redundant information).
Panel C. Constrained-agent condition: capped command loop		
Stata	An execution loop might raise the value of standardized commands by letting the model run the script, read the log, and rerun.	Examples could keep constrained-agent Stata runs closer to the required result-file structure; without them the agent might spend turns exploring.
R	A live loop could help the model navigate package alternatives and stop at a correct output, or expand search without converging.	Examples may reduce search over packages and output formats.
Python	Iterating against tracebacks could fix API errors, or burn turns exploring alternatives without converging.	Examples might reduce costly exploration and semantic drift, or anchor the model on the example's specific pattern.

Notes: Each cell uses the same task set, dataset, reference output, execution limit, and evaluation protocol. Panel text reproduces the recorded design expectations.

H.2 Extended hypothesis discussion

Hypothesis H1 (software contrast at the baseline). The software contrast at the baseline summarizes the software main effect before any prompting or agency intervention is applied. It asks whether Stata's standardized command syntax, R's package-driven ecosystem, or Python's broader general-coding exposure translates into different starting points for econometric work, and is the joint test that the two software indicators have zero coefficients at the zero-shot, chatbot baseline. The design summarizes a bundle of software-level differences (training-data exposure, syntax conventions, package maturity, default verbosity, error-message format) and cannot identify which channel drives the effect.

Hypothesis H2 (prompt contrast). The prompt contrast asks whether adding an econometric example improves success on average across agency conditions and software. This is the few-shot premise of Brown et al. (2020) applied to econometric coding: an in-context example of a related task could carry over to a fresh scored task, or could anchor the model on the example template. The restriction is that the design-averaged few-shot effect equals zero.

Hypothesis H3 (agency contrast). The agency contrast asks whether the one-repair and constrained-agent conditions raise success against the chatbot baseline. The one-repair arm tests whether a single structured run report lifts performance when the model remains in a chat-style protocol. The constrained-agent arm tests whether a live execution loop adds more when the model can run its script, read errors, and revise within the capped loop. We report two separate design-averaged contrasts, one for each agency condition.

Hypothesis H4 (interaction between prompt and agency conditions). A positive interaction means examples and a more agentic condition exceed the sum of their parts and the two design features are complements; a negative one means they substitute. The design-averaged difference-in-differences identifies how the few-shot effect changes when the model sees a run report or acts in a live loop.

H.3 Power, precision, and robustness

The primary estimates use the intention-to-treat sample. Model-side timeouts enter as failures, while provider-side interruptions are excluded because no model response was available. The responded-only sensitivity removes the timeout rows. Task success is 81.2 percent with zero-shot prompts and 87.1 percent with few-shot prompts in the primary sample. The corresponding responded-only rates are 85.0 and 89.8 percent. The contrast is largest for zero-shot Stata, whose rate is 69.8 percent in the primary sample and 80.9 percent among records with a model response.

The Claude Sonnet inference clusters observations by task and uses a wild cluster bootstrap with 9,999 Rademacher draws. The GPT-5.4 and DeepSeek results use the same within-model regression and inference as the Claude Sonnet analysis. The three model arms use the same agency categories. Codex can take somewhat more actions in its agent condition.

I Supplementary Results Tables

Table 7 reports the full software-by-agency split of task success in each prompt condition, discussed in Section 4.2.

Table 7: Task-success rate by statistical software, agency condition, and prompt condition.

Panel A: Zero-shot condition			
Software	Chatbot	One-repair	Constrained agent
Python	79.0	87.5	95.2
R	77.1	90.4	91.4
Stata	45.7	65.7	98.1

Panel B: Few-shot condition			
Software	Chatbot	One-repair	Constrained agent
Python	87.6	87.6	94.3
R	88.6	90.5	95.2
Stata	65.7	75.2	99.0

Notes: Entries are percentages over analyzed records in each software-by-agency cell of the five-repetition analysis sample. Intention-to-treat failure rows enter as failures. Each cell contains 105 records, with five repetitions for each of the 21 tasks.

Table 8: Examples used to interpret the aggregate results.

Example role	Selection rule	Selected case
Calibration	Highest analysis-sample success among simple reference-output tasks	Cluster-robust OLS (98.9 percent); Poisson with offset and logit (97.8 percent each); VAR Granger causality and probit just below
Agency-sensitive task	Largest one-repair or constrained-agent gain over the chatbot condition	Staggered DiD (+69.9 points chatbot to constrained agent); one-way ANOVA (+50.0 points)
Semantic failure	High execution success with low specification or numeric success	Dynamic-panel GMM (wrong estimator variant); one-way ANOVA in R (linear trend in place of ANOVA)
Software contrast	Largest software gap within the same task	Dynamic-panel GMM: Stata 90.0, R 66.7, Python 10.0 percent

Notes: Rates are percentages over analyzed records in the five-repetition analysis sample across both prompt conditions.