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

September 2026

Abstract

We study how large language models write econometric code. On 21 tasks in Stata, R, and Python, we vary the prompt, with or without a worked example, and agency, from a chatbot that only writes the script to an agent that also runs and revises it. For Claude Sonnet 4.6 and GPT-5.4, the agent raises task success from 82 to 97 percent for under ten additional cents per run. The example and the software matter under the chatbot but little under the agent, which repairs scripts that fail to run and leaves those that run but return wrong numbers.

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 typically answer a research question by setting up an econometric model and identifying its parameters of interest (Angrist and Pischke, 2010; Garg and Fetzer, 2024). They then write code in Stata, R, or Python to process the data and compute the estimates. Because much of this coding is mechanical, economists often delegate it to research assistants, although delegation does not remove the need to review, correct, and standardize the resulting code. Coding errors are common and often difficult to detect. Some are semantic bugs, in which the code runs but does not compute what was intended (Tan et al., 2014). Audits find coding errors in about a quarter of published studies, even when the posted code reproduces the published numbers exactly (Brodeur et al., 2026). Large language models (LLMs) can now perform that coding work as well (Cui et al., 2026), and, like research assistants, they can introduce such errors into the estimation code.

In this paper we study how LLMs write econometric code. We measure how task success and cost vary with the degree of agency, the prompt, and the statistical software, for two LLMs. In our design, the researcher chooses the question, the method, and the identification assumptions, while the model receives a task and writes the code that implements it. Writing that code involves five stages: translating the request into a plan, writing the script, running it, inspecting the output, and revising the script when something is wrong. A chatbot performs only the first two, because it answers with a script and never runs it. The last three stages require running the code and reacting to what it returns, and we use *agency* for how much of that work the model can do before a result is submitted.

What the model can do is set by the harness, the program that surrounds the model, runs its code, decides which tools it may use, and passes the output back to it. Our design has three agency conditions, the chatbot, self-repair, and the constrained agent, which give the model progressively more of the last three stages. Under the *chatbot* condition, the model writes one script and submits it without ever seeing it run. Under *self-repair*, the harness runs the script, and if the script fails or produces no result file, it returns the run output to the model, which may then revise the script and submit it again, up to three times. Under the *constrained agent*, the model runs its own code, reads the output, revises as many times as it needs, and decides when it is done. The other two dimensions are the prompt and the software. 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), and the software dimension compares Stata, R, and Python. We run every combination on a benchmark of applied econometric tasks with two models, Claude Sonnet 4.6 in the Claude Code harness and GPT-5.4 in the Codex harness, and for every run we record task

success, cost, and runtime.

The benchmark consists of 21 econometric tasks, which cover linear regression and its diagnostics, classical tests, generalized linear models, difference-in-differences, instrumental variables, regression discontinuity, dynamic panel estimation, survival analysis, and time series. Each task comes with a simulated dataset and with a reference output computed in advance. A run is a single attempt by one of the two models at one task, in a given software environment and under a given prompt and agency condition, and it succeeds when the estimates it reports match the reference output of its task within a tolerance. Because the output of a model varies across otherwise identical runs, we repeat every combination of task, software, prompt, and agency condition twelve times with each model, which gives 4,536 runs per model and 9,072 runs in total. Econometric coding is a useful setting in which to measure the effect of agency, because the task fixes the method and the target output, so that an automated scoring system can check, run after run, whether the submitted code produces the requested result. What we learn about agency may therefore inform other research tasks with verifiable outputs.

The two models and the three agency conditions reflect how AI is used today. Anthropic, OpenAI, and Google account for 88 percent of enterprise language-model spending, and Claude and GPT models lead the coding segment of that market (Tully et al., 2025). Most individual use still happens through a chat window: ChatGPT alone reaches about a tenth of the world's adults (Chatterji et al., 2025), and fewer than one percent of its active individual users have opened the Codex agent (Johnston et al., 2026). Use of coding agents is nonetheless growing quickly (Massenkoff et al., 2026), so the three agency conditions span the interfaces in use, from the chat window to the coding agent.

We estimate the effects of the three dimensions on task success in a regression with task fixed effects, cluster standard errors by task, and base inference on wild-cluster-bootstrap p-values. Task success rises with agency for both models. For Claude Sonnet 4.6, it rises from 83.1 percent under the chatbot to 93.4 percent under self-repair and 95.9 percent under the constrained agent, and GPT-5.4 moves from 81.8 to 97.3 percent. A few-shot example helps the chatbot, but its value falls with agency. For Claude Sonnet 4.6, the example adds 9.0 percentage points under the chatbot but only 1.3 under the constrained agent, and its average effect across conditions, 2.9 points, is not statistically significant. The example therefore adds little once the model can execute and revise its own code, and the same holds for GPT-5.4, where its gain falls from 9.7 to 0.4 points. The software comparison shows the same pattern: differences that are large under the chatbot shrink under the constrained agent. Under the zero-shot chatbot, Stata is an estimated 23.4 percentage points below Python for Claude Sonnet 4.6 and 52.8 points below for GPT-5.4. Under the

constrained agent, zero-shot Stata runs of Claude Sonnet 4.6 reach 98.0 percent, up from 65.1 percent under the chatbot, and the gap all but disappears for GPT-5.4 as well.

Under the chatbot, most failures are loud, since the script does not run or writes no result file, and the few-shot example and agency both remove that kind of failure, which is why the two substitute for each other and why the Stata gap closes under the constrained agent. The failures that remain under the constrained agent are silent, since the script runs and writes a result file whose numbers are wrong, and nothing in the result file reveals the error.

We also report costs and runtime. Agency costs more: for Claude Sonnet 4.6, the extra model turns and tool calls raise mean cost per run by 8.6 cents, from $0.18 to $0.26. Cost per successful run rises by less than mean cost per run, from $0.21 under the chatbot to $0.27 under the constrained agent, because more runs succeed. Dividing the extra spending by the extra successes gives an incremental cost of $0.67 per additional successful run. For the same model, the constrained agent is also slower, with a median runtime 2.3 times that of the chatbot.

2 Related Literature

Empirical research depends on code, and errors in the code can change reported results. Practitioner guides therefore treat code as a production input and set engineering standards for empirical work (Gentzkow and Shapiro, 2014). Audits of research code document widespread reproducibility failures and frequent coding errors (Galiani et al., 2017; Brodeur et al., 2026). AI systems now help researchers review the literature, write code, collect data, and build research datasets (Korinek, 2025), and recent work uses LLM agents to assess replication packages and to execute and re-analyze published replication materials at scale (Hu et al., 2025; Xu and Yang, 2026; Siegel et al., 2024).

Executable-code benchmarks measure how well these systems write code by running the generated programs against tests, and they cover general programming, data science, and multilingual code generation (Chen et al., 2021; Lai et al., 2023; Zhuo et al., 2025; Cassano et al., 2023). Benchmarks of statistical analysis score the analyses with human experts, answer keys, or an LLM judge rather than by running the code (Song et al., 2026; Zhu et al., 2026). In econometrics, Chen et al. (2025) evaluate MetricsAI, an agent that plans, executes code, and revises after errors, on expert-level tasks, where it outperforms general-purpose language models and general-purpose agents. 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; Starace et al., 2025).

A user of these systems chooses the software environment, the prompt, and the harness through which the model acts. Research code in economics concentrates in a few environments: Stata appears in 73 percent of the author-provided replication packages of the American Economic Association journals, MATLAB in 22 percent, R in 4 percent, and Python in 2 percent, and one package can contain more than one of them (Vilhuber et al., 2020). Econometric packages can return different numbers for the same estimator, and each environment has its own programming conventions (McCullough and Vinod, 1999; Cox, 2005).

Few-shot examples in the prompt raise performance on many tasks (Brown et al., 2020), and results can depend enough on the wording of a prompt that evaluations are asked to report more than one prompt (Mizrahi et al., 2024). For programming languages with little training data, in-context examples raise code-generation performance in every configuration that Giagnorio et al. (2025) test in R and Racket, and by more for larger models, although the models in that study write code without running it.

Harnesses expand what a model can do while it codes. ReAct, InterCode, and SWE-agent formalize a model's work as a sequence of model actions and environment observations, with the harness defining the actions available to the model and the feedback it receives (Yao et al., 2023; Yang et al., 2023, 2024). Self-debugging, self-refinement, Reflexion, and self-repair are methods in which the model revises its output after tests, observations, or critiques (Chen et al., 2024; Madaan et al., 2023; Shinn et al., 2023; Olausson et al., 2024). Meng et al. (2026) call the engineering layer that surrounds the model an agent harness. Harness-Bench finds that the harness changes completion, efficiency, and failure behavior enough that capability should be reported for a pair of model and harness rather than for a model alone (Yao et al., 2026).

Our agency comparison has its closest precedents in studies that compare alternative harness designs for a given language model, so that the harness, rather than the model, accounts for any difference in outcomes. Kim et al. (2026) mask functions from the codebases of replicated machine-learning papers and find that the pass rate of one model rises more than fourfold when the agent chooses its own actions and debugs after failed runs, relative to a predefined pipeline. On CORE-Bench, where an agent reproduces the results of published papers from their code and data, Siegel et al. (2024) find that a task-specific agent raises the accuracy of GPT-4o on the hardest tasks from 6.7 to 21.5 percent relative to a general-purpose agent, and they report the API cost of each agent. On PaperBench, where agents replicate machine-learning papers from scratch, Starace

et al. (2025) find that a scaffold that removes the model's option to end its own run and prompts it step by step raises the replication score of two OpenAI models and lowers that of Claude 3.5 Sonnet. Shah et al. (2026) repair deliberately broken R scripts from published studies with a prompt-based and with an agent-based workflow, and reproduction success is 69 to 96 percent under the agent-based workflow against 31 to 79 percent under the prompt-based one, depending on the complexity of the failure and on the context given in the prompt. In each of these studies the design of the harness changes performance by a wide margin, and in the PaperBench comparison the direction of the change depends on the model. Weinberger and Hozez (2026) study the interaction between the prompt and the harness that we also measure. They run frozen prompt variants of 24 coding tasks under two agent harnesses for six open-weight reasoning models and find that the effect of a prompt does not transfer from one harness to the other, although their tasks are small, with high success ceilings, so that the interaction appears in reasoning tokens and in cost per success rather than in task success.

Unlike the repair and reproduction settings above, every run in our design generates new econometric code from a task statement, in Stata, R, or Python, with its cost and runtime recorded. The tasks are scored by numerical agreement with a reference output rather than by a judge. The software comparison asks the same question as the prompt comparison, because the environment, like the prompt, changes how often a script runs and writes its result file at the first attempt.

3 Design

The design has three dimensions: the prompt (zero-shot or few-shot), the degree of agency (chatbot, self-repair, or constrained agent), and the statistical software (Stata, R, or Python). Each of the two models, Claude Sonnet 4.6 in the Claude Code harness and GPT-5.4 in the Codex harness, attempts 21 econometric tasks under every combination of prompt, agency condition, and software. Each combination of task, software, prompt, and agency condition forms one cell of the design. Because the output of a model varies across otherwise identical runs, each model runs every cell twelve times. The analysis sample therefore holds 4,536 runs per model and 9,072 runs in total. Figure 1 summarizes the design.

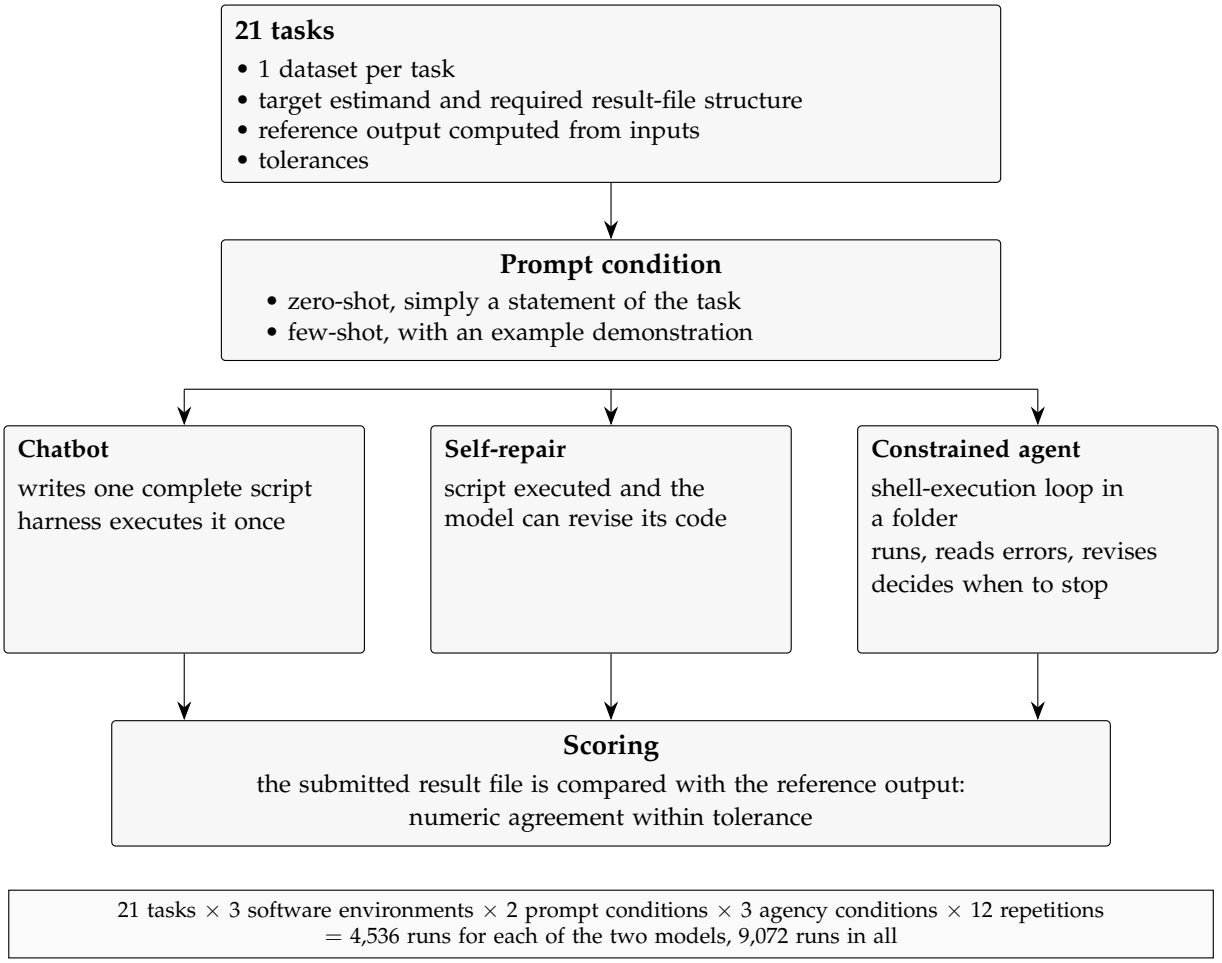

Figure 1: Design of the benchmark. Each task specifies a dataset, a target estimand, a required result-file structure, a reference output, and tolerances. The prompt condition changes the task statement, the agency condition changes what the model can observe and do before it submits a result, and every run succeeds when its result file agrees with the reference output within tolerance. Each model runs every cell twelve times.

3.1 Degree of agency

The agency conditions differ in what the model can see and do between receiving the task and submitting a result. In every condition the harness sets the context the model receives, the tools it may call, the way its code is executed, the rule that ends the run, and the record the run leaves behind (Meng et al., 2026). Under the *chatbot* condition, the harness obtains one complete script in a single model call, executes it once, and returns nothing to the model, as in a chat window where the researcher pastes a request and runs the script that comes back. Under *self-repair*, the harness executes the script in the same way, but when the script fails or writes no result file, the harness returns a fixed message with the

captured run output, and the model may submit up to three revised scripts. Under the *constrained agent*, the model executes shell commands in a per-run working folder that holds the input data, and within a session time limit it writes, runs, and revises its script as often as it chooses and declares when it is finished. The condition is constrained in that the model is limited to writing and running scripts for the task.[1] In every condition, script executions and model calls also run under time limits, and a run that reaches a limit without a scored result file counts as a failure.

3.2 Tasks and scoring

Each task specifies a dataset, a target estimand, a required result-file structure, a reference output computed in advance, method checks, and numerical tolerances. Every dataset is simulated, and the same dataset and the same reference output serve every software environment, prompt condition, agency condition, and repetition of a task, so the design dimensions change nothing about the data or the target. The 21 tasks cover linear regression and its diagnostics, classical tests, generalized linear models, difference-in-differences, instrumental variables, regression discontinuity, dynamic panel estimation, survival analysis, and time series.[2]

The task statement is the same in both prompt conditions. It names the software, the dataset and its variables, the quantity to estimate and the sample, the required estimator and inference convention, and the required structure and location of the result file. The zero-shot prompt contains only that statement. The few-shot prompt places one worked example before it: a short statement of a related task from the same method family, complete code in the target software on small inline data, and the result-file conventions that the scored task requires. The example is never the scored task itself and does not mention its dataset, variables, or reference values.

The primary outcome is task success, which is binary. A run succeeds when the result file it submitted reports estimates that match the stored reference output within the task's tolerances and when the script computed them from the task's input data, so a hard-coded answer or a read of the stored reference output fails. The scoring system judges the final result file the run saved, whatever number of permitted revisions produced it. Before the comparison, it applies a formatting fix to that file that changes no reported digit, leaves a truncated or empty file as a failure, and normalizes the conventions with which the three environments write numbers and empty fields, so output formatting does not drive the

[1] Appendix A.3 records, for each harness, the tools it authorized and the limits it applied.
[2] Appendix A summarizes the task definitions.

software comparison (Appendix A.4).

Beyond the binary outcome, the scoring system stores five checks for every run: whether the script executed, whether it wrote a well-formed result file, whether the reported numbers agree with the reference output, whether the script shows evidence of the required method, and whether the submission is valid. Task success requires the numeric-agreement and submission-validity checks, and the other three are recorded as diagnostics. We call the verdict that requires all five checks the strict five-check diagnostic, and Section 4 and Appendix B use it to describe how runs fail.

3.3 Analysis and inference

We summarize task success as cell rates, the share of successful runs in each cell. We estimate the effects of the design dimensions with a regression model in which each run is one observation and each task has its own intercept,

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

where Success_{ir} equals one if run r of task i succeeds and zero otherwise, τ_i is the task fixed effect, and X_{ir} holds indicators for every combination of software, prompt, and agency condition in the $2 \times 3 \times 3$ design, with Python, zero-shot, and chatbot as the omitted categories. Because X_{ir} includes all interactions among the three dimensions (Angrist and Pischke, 2009) and every cell holds twelve runs for each model, the regression reproduces the observed success rate of each of the 18 combinations of software, prompt, and agency condition. Every estimate we report is a difference between two such rates or between averages of them, so the regression organizes the cell comparisons and supplies their standard errors rather than imposing a functional form on them.

We estimate equation (1) separately for each model, and once on the two models pooled, with a fixed effect for every pair of model and task and with cell effects common to the two models. Because each model contributes twelve runs to every cell, each pooled estimate is the simple average of the two per-model estimates. Table 1 reports the per-model estimates in columns 1 and 2 and the pooled estimates in column 3. A second pooled specification with model-specific cell effects tests whether an estimate differs between the two models, with cluster-robust standard errors and a t distribution with 20 degrees of freedom.

The software contrasts in Table 1 compare R and Stata with Python among zero-shot chatbot runs, the baseline condition of the design. The prompt contrast compares few-shot with zero-shot after averaging over the nine combinations of software and agency condition. The agency contrasts compare self-repair and the constrained agent with

the chatbot after averaging over the six combinations of software and prompt. The interaction contrast measures how the few-shot effect changes under the constrained agent relative to the chatbot, averaged over the three software environments, and a positive interaction indicates complementarity between the example and agency, whereas a negative interaction indicates substitution. Table 1 also prints two descriptive contrasts for Stata against Python, one averaged over the six prompt and agency conditions and one for the zero-shot constrained agent.

Runs of the same task share its dataset and its difficulty, so we cluster standard errors at the task level. Five datasets serve thirteen of the 21 tasks, and task-level clustering does not absorb the dependence that those shared datasets create across tasks (Appendix A). With only 21 clusters, conventional cluster-robust inference can over-reject, so the headline p-values come from the wild cluster bootstrap with Rademacher weights, the null imposed, and 9,999 replications (Cameron et al., 2008; MacKinnon and Webb, 2018). Table 1 reports the cluster-robust standard errors in parentheses and marks significance levels from the bootstrap p-values.

4 Results

Task success over the whole sample is 90.4 percent, 90.8 percent for Claude Sonnet 4.6 and 90.0 percent for GPT-5.4. The two models reach similar levels and respond to the three dimensions of the design (agency, prompt, and software) in the same direction.

4.1 Agency effects

Task success rises with agency for both models. For Claude Sonnet 4.6, it is 83.1 percent under the chatbot, 93.4 percent under self-repair, and 95.9 percent under the constrained agent. For GPT-5.4, it is 81.8, 90.9, and 97.3 percent. Table 1 reports these differences as the agency contrasts of equation (1). Relative to the chatbot, self-repair raises task success by 10.3 percentage points for Claude Sonnet 4.6 and by 9.1 points for GPT-5.4, and the constrained agent raises it by 12.8 and 15.5 points. All four estimates are significant at the 1 percent level, as are the pooled estimates, 9.7 points for self-repair and 14.2 points for the constrained agent.

Self-repair alone achieves four fifths of the constrained agent's improvement for Claude Sonnet 4.6 and about three fifths for GPT-5.4. It does so although it only lets the model revise a script that has failed, and at most three times. Most self-repair runs revise at most once (Appendix A.3). Letting the model run and revise its own code as often as it

Table 1: Effects of software, prompting, and agency on task success.

	(1) Claude Sonnet 4.6	(2) GPT-5.4 (Codex)	(3) Total
Software (zero-shot chatbot, percentage points)			
R versus Python	−6.3	−3.2**	−4.8
	(6.5)	(1.6)	(3.4)
Stata versus Python	−23.4***	−52.8***	−38.1***
	(8.1)	(5.8)	(6.2)
Stata versus Python, design average	−8.2*	−19.8***	−14.0***
	(4.4)	(4.2)	(4.1)
Stata versus Python, constrained agent	3.2	−2.0	0.6
	(4.6)	(4.3)	(4.2)
Prompting (percentage points)			
Few-shot versus zero-shot	2.9	6.5***	4.7***
	(2.4)	(1.0)	(1.6)
Agency (percentage points)			
Self-repair versus chatbot	10.3***	9.1***	9.7***
	(2.0)	(1.4)	(1.2)
Constrained agent versus chatbot	12.8***	15.5***	14.2***
	(3.1)	(1.4)	(1.8)
Interaction (percentage points)			
Few-shot × constrained agent	−7.7*	−9.3***	−8.5***
	(3.8)	(2.2)	(2.2)
Observations	4,536	4,536	9,072
Task fixed effects	Yes	Yes	Yes

Notes: Estimates are differences between predicted task-success rates from equation (1), a regression model with task fixed effects, fit on one model in columns 1 and 2 and on the two models pooled with model-by-task fixed effects in column 3. Cluster-robust (CR1) standard errors on the 21 task clusters are in parentheses. *, **, and *** mark the 10, 5, and 1 percent levels from wild-cluster-bootstrap tests (Rademacher weights, 9,999 replications, null imposed). The smallest reportable p is 0.0001. The interaction is the change in the few-shot effect under the constrained agent, so a negative value means the prompt and the agent substitute for each other.

chooses raises task success further, from 93.4 to 95.9 percent for Claude Sonnet 4.6 and from 90.9 to 97.3 percent for GPT-5.4. Most of the gain from agency therefore requires no more than one revision of a script that has failed, and the open-ended interaction of the constrained agent adds the remainder. The primary outcome scores each run once. Appendix C.2 reports pass@k, the probability that at least one of k runs of a cell succeeds, and with twelve chatbot runs that probability is at least as high as the success rate of a single constrained-agent run. The user, however, pays for twelve runs and must identify the correct script among them, so that the running and checking that agency delegates to the model stay with the user.

4.2 Prompt and software effects

The few-shot example raises task success under the chatbot and adds little under the constrained agent. For Claude Sonnet 4.6, averaging over the three software environments, the example raises task success by 9.0 percentage points under the chatbot, changes it by -1.6 points under self-repair, and raises it by 1.3 points under the constrained agent. For GPT-5.4, it raises task success by 9.7 points under the chatbot, by 9.5 points under self-repair, and by 0.4 points under the constrained agent. In Panel A of Figure 2, which pools the two models, the few-shot line lies about nine points above the zero-shot line under the chatbot and almost coincides with it under the constrained agent.

The prompt contrast of Table 1, the average effect of the example over the nine combinations of software and agency condition, is 2.9 percentage points for Claude Sonnet 4.6 and 6.5 points for GPT-5.4, and only the estimate for GPT-5.4 is statistically significant. The interaction contrast, the change in the effect of the example from the chatbot to the constrained agent, is -7.7 points for Claude Sonnet 4.6, -9.3 points for GPT-5.4, and -8.5 points when the two models are pooled, significant at the 1 percent level for GPT-5.4 and for the pooled sample and at the 10 percent level for Claude Sonnet 4.6. A negative interaction means that the example and the constrained agent are substitutes, because the example adds less to task success once the model can run and revise its code. When every cell effect is allowed to differ by model, the two interactions differ by 1.6 points, and the difference is not statistically significant ($p = 0.719$), so the substitution has the same sign and, within sampling error, the same size for the two models. The substitution suggests that the example works by preventing failures that the model would otherwise repair on its own once it can run its code.

The choice of software shows the same dependence on agency. Under the zero-shot chatbot, the baseline condition of the design, Stata is 23.4 percentage points below Python for Claude Sonnet 4.6 and 52.8 points below for GPT-5.4, and R is 6.3 and 3.2 points below. As agency increases, the Stata gap narrows for both models. For Claude Sonnet 4.6, zero-shot Stata runs rise from 65.1 percent under the chatbot to 98.0 percent under the constrained agent. Under zero-shot self-repair, Stata is 2.8 points below Python for Claude Sonnet 4.6, so self-repair alone closes most of the gap for this model. For GPT-5.4, Stata is still 27.8 points below Python under zero-shot self-repair, about half of the baseline gap. Under the zero-shot constrained agent, the gap is closed for both models, with Stata 3.2 points above Python for Claude Sonnet 4.6 and 2.0 points below for GPT-5.4, and neither difference is statistically significant. Panel B of Figure 2 shows the same pattern pooled over the two models. Because self-repair only revises scripts that have failed to run or to write their result file, the disadvantage of Stata under the chatbot appears to lie in getting

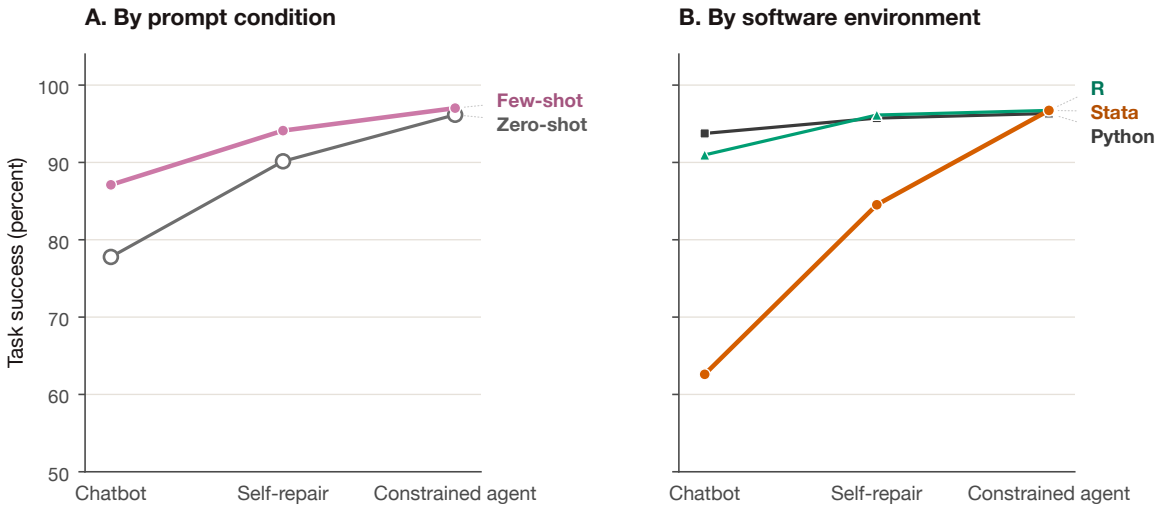

Figure 2: Task success by agency condition, pooled over the two models. Panel A draws the two prompt conditions and Panel B the three software environments. Labels at the end of the lines name the series.

scripts to run rather than in the estimates they report, and Section 4.3 bears this out for both models. Taken together, the prompt and the software, the two choices on which task success depends under the chatbot, matter far less once the model can run and revise its code.

4.3 Failures

We classify every run that misses the strict five-check diagnostic by the first of the five checks of Section 3.2 that it fails (Appendix B). We call a failure loud when the script does not run or writes no result file, so that the user sees an error or a missing file. We call a failure silent when the script writes a result file whose numbers or method evidence fail the corresponding check, so that nothing in the file reveals the error. For Claude Sonnet 4.6, loud failures fall from 10.6 per 100 runs under the chatbot to 0.1 under the constrained agent, whereas silent failures fall only from 7.1 to 4.1 per 100 runs. For GPT-5.4, loud failures fall from 17.7 per 100 runs under the chatbot to 0.1 under the constrained agent, and silent failures stay between 4.8 and 5.9 per 100 runs in the three agency conditions. Agency therefore removes almost all loud failures and less than half of the silent ones. Of the failures that remain under the constrained agent, 98 percent are silent for Claude Sonnet 4.6 and 99 percent for GPT-5.4. The silent failures that remain involve estimator variants, data conventions, and misinterpretations of the requested calculation.

This accounting helps explain the two patterns of Section 4.2, the falling value of the

example and the closing of the software gap. Under the chatbot, the few-shot example mainly reduces loud failures, and the number of numeric failures does not fall (Table 4). The example therefore raises task success by making more scripts run and write their result file, not by making the numbers more accurate. The constrained agent removes the same loud failures on its own, so once the model can run and revise its code the example has little left to fix, which is why the two are substitutes. Unlike the example, the constrained agent also removes some silent failures, although less than half of them. The software environments also differ mainly in loud failures. Under the chatbot, 20.8 percent of the Stata scripts of Claude Sonnet 4.6 fail to run or to write their result file, against 5.2 percent of its Python scripts and 5.8 percent of its R scripts. For GPT-5.4, the shares are 49.8 percent for Stata against 1.0 and 2.4 percent for Python and R. The shares that fail the numeric check differ far less across the three environments (Table 5). Under the constrained agent, those loud failures all but disappear in the three environments for both models, and with them most of the Stata gap.

Figure 3 plots task success by task and agency condition, pooled over the two models. Under the constrained agent, most tasks reach or approach full success, and the remaining failures concentrate in a few tasks. Two-way clustered OLS is the only task in which the constrained agent does not improve on the chatbot, and for Claude Sonnet 4.6 its success rate falls from 84.7 percent under the chatbot to 66.7 percent under the constrained agent. Its failures under the constrained agent are silent, because the scripts run and write a result file whose numbers fall outside the tolerance. For Claude Sonnet 4.6, the two tasks with the lowest success rates across all conditions are Arellano–Bond GMM, at 68.1 percent, and one-way ANOVA, at 74.1 percent, and they fail for different reasons. One-way ANOVA fails mainly under the chatbot and self-repair, and its failures nearly disappear once the model runs its own code. Arellano–Bond GMM has a low success rate in every condition, and it reaches only 69.4 percent under the constrained agent for Claude Sonnet 4.6 and 75.0 percent for GPT-5.4.

For Claude Sonnet 4.6, Arellano–Bond GMM succeeds in 95.8 percent of the Stata runs but only 37.5 percent of the Python runs, where the failing scripts often implement a related estimator, such as Anderson–Hsiao-style instrumental variables, instead of the requested one-step Arellano–Bond estimator (Anderson and Hsiao, 1981; Arellano and Bond, 1991). Such a script runs without error and writes a result file, so the run output gives the model no sign that the estimator is wrong, and a researcher still needs to review the specification that the model chose.

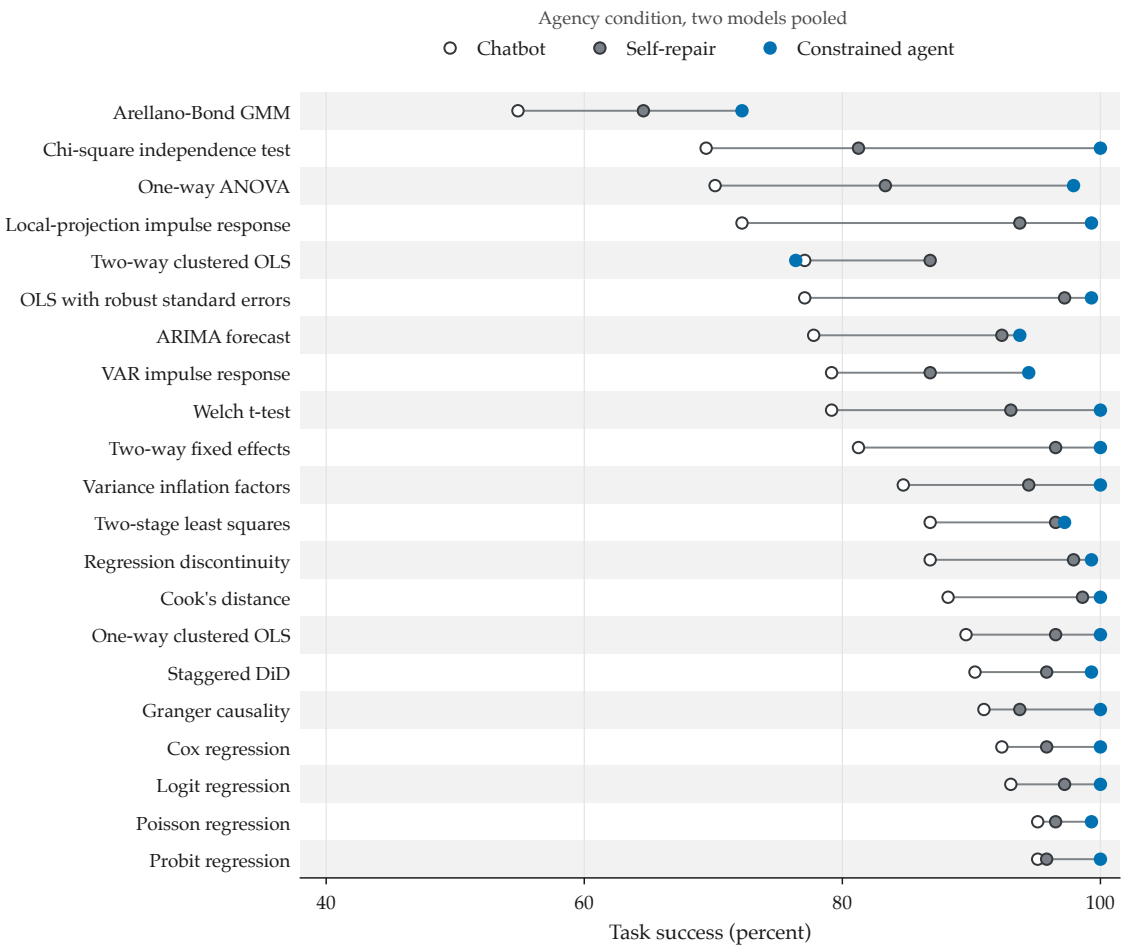

Figure 3: Task success by task and agency condition, pooled over the two models, with the tasks ordered by success under the chatbot. White markers are the chatbot, gray markers self-repair, and blue markers the constrained agent, each averaging 144 runs. The horizontal axis begins at 40 percent.

4.4 Scoring checks

Over the whole sample, the strict five-check diagnostic is 2.1 percentage points below the primary outcome, and halving or doubling every tolerance moves the Claude Sonnet 4.6 rates by one point or less. Appendix A.4 details both checks and the planted-failure tests that document the one kind of error that survives the scoring rule. A run that implements the wrong estimator but reports numbers inside the tolerance is scored a success under the primary outcome, and under the strict diagnostic as well when the required method appears only in the script's comments, so the reported rates are upper bounds with respect to this kind of error.

5 Cost and Runtime

The cost of a run is the price of the tokens consumed by its model calls, at the list prices of the model's API. For Claude Sonnet 4.6, the harness records the cost of each model call, and for GPT-5.4 we price the recorded tokens at the list prices that OpenAI publishes, which Appendix C reports together with the fields of the cost ledger. The cost of a self-repair run sums every attempt, and the cost of a constrained-agent run sums every model turn of the session, so the reported cost covers the whole run (Appendix A.3). Cost per run is the mean or the median cost over the runs of a condition that carry a cost, and cost per successful run divides the total cost of those runs by the number of successes among them.[3] Unless stated otherwise, the dollar figures below refer to Claude Sonnet 4.6 and pool the two prompt conditions.

Self-repair and constrained-agent runs can use more tokens than chatbot runs. A self-repair run adds a model call each time its script fails or writes no result file, up to three times, and a constrained-agent run adds a model turn each time the model runs its script, reads the output, and revises the code.

For Claude Sonnet 4.6, mean cost per run rises from \$0.18 under the chatbot to \$0.26 under the constrained agent, 49 percent more, and cost per successful run from \$0.21 to \$0.27, 29 percent more, because the conditions that cost more also produce more successes. The median rises less than the mean, by 37 percent, because a minority of expensive constrained-agent runs raises the mean but not the median. Self-repair accounts for about half of the increase in cost and, as Section 4.1 shows, for four fifths of the gain in task success. Claude Sonnet 4.6 runs cost 5.7 times as much as GPT-5.4 runs under the chatbot and 2.7 times as much under the constrained agent, because the mean cost of a GPT-5.4 run triples with agency, from \$0.03 to \$0.10, while that of a Claude Sonnet 4.6 run rises by half. Figure 4 plots task success against mean cost per run by software environment and agency condition, with one panel for each model. In every software environment and for both models, the constrained agent costs more per run than the chatbot and succeeds more often, and both differences are largest in Stata.

The few-shot example raises the mean cost of a Claude Sonnet 4.6 run by 16 percent and lowers that of a GPT-5.4 run by 37 percent. The opposite signs are consistent with two ways in which the few-shot example changes the cost of a run. It lengthens every prompt, which raises the cost of Claude Sonnet 4.6 runs in every condition. It also prevents loud failures, and GPT-5.4, whose scripts fail loudly far more often under zero-shot prompts

[3] Nine of the 4,536 Claude Sonnet 4.6 runs carry no cost, because a model call timed out or a harness error ended the run before any model call was recorded. They stay in the analysis sample, and every cost measure uses the other 99.8 percent of the runs. Every GPT-5.4 run carries a cost.

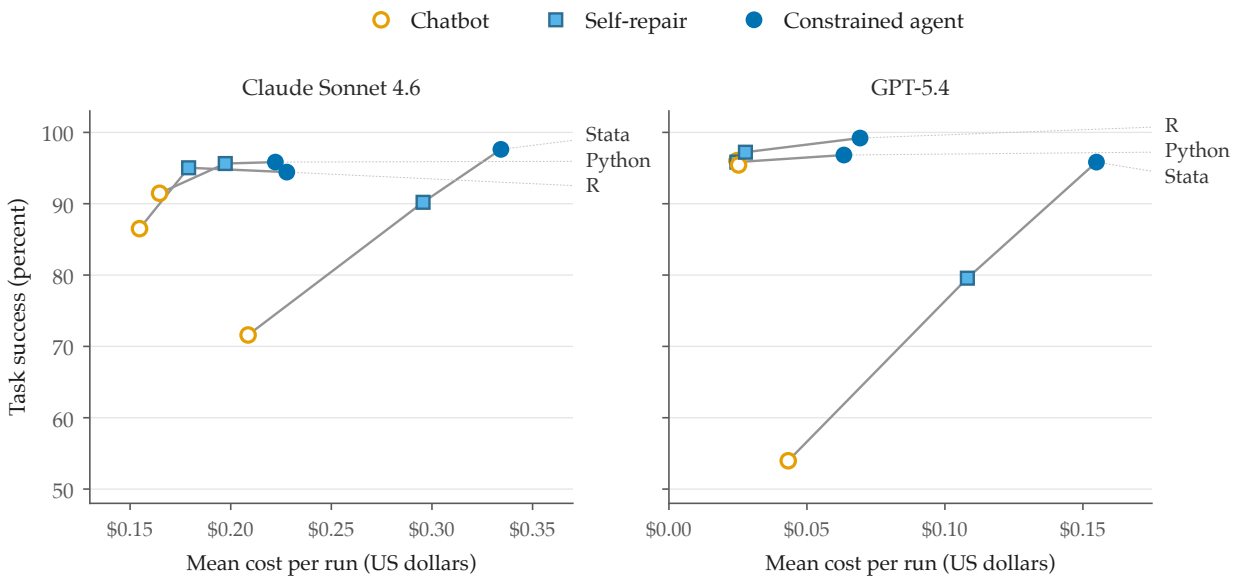

Figure 4: Task success and mean cost per run by software environment and agency condition, one panel per model. Each line joins the chatbot (open circle), self-repair (square), and constrained agent (filled circle) of one software environment, named in the right-hand margin. Each point pools the two prompt conditions, covers up to 504 runs, and uses only the runs that carry a cost. The panels share the vertical axis, and each has its own cost axis.

(Section 4.3), is spared the revisions and turns that those failures require. Appendix C reports the medians by prompt and agency condition, which show that the cost of agency is smaller under few-shot prompts.

Stata is the most expensive software environment for both models. For Claude Sonnet 4.6, a Stata run costs 44 percent more than a Python run on average, and a successful Stata run 57 percent more, $0.32 against $0.21, since fewer Stata runs succeed (Table 1). For GPT-5.4, a Stata run costs 2.7 times as much as a Python run and a successful one 3.4 times as much. The Stata difference in cost per run already exists under the chatbot and widens as agency increases, which is consistent with the constrained agent spending turns on the loud failures that Stata scripts produce more often.

The incremental cost of agency is the additional spending per additional success. We compute it as the change in mean cost per run among the runs that carry a cost divided by the change in task success among all runs, both measured from the chatbot to the constrained agent. For Claude Sonnet 4.6, the constrained agent raises task success by 12.8 percentage points and mean cost per run by 8.6 cents, from $0.18 to $0.26, so each additional successful run costs $0.67. The two steps differ in what an additional success costs. The move from the chatbot to self-repair adds 10.3 points of task success for 4.8

cents, or $0.46 per additional successful run, whereas the move from self-repair to the constrained agent adds 2.5 points for 3.8 cents, or $1.50. For GPT-5.4, the two steps cost $0.25 and $0.66 per additional successful run. The incremental cost is lower under zero-shot prompts, where the constrained agent adds 16.7 percentage points of task success at $0.63 per additional successful run, than under few-shot prompts, where it adds 9.0 points at $0.74. The difference arises because the example has already removed most loud failures under the chatbot, so the constrained agent adds fewer successes. These amounts are small relative to the cost of research time. At research-assistant wages of $15 to $50 an hour, the $0.67 cost of an additional successful run pays for between one and three minutes of an assistant's time.

We measure the runtime of a run as the wall-clock time of its calls to the model provider, which includes the script executions that the constrained agent runs inside its session but excludes the harness's own executions in the other two conditions and harness overhead in every condition. For Claude Sonnet 4.6, the median runtime is 10.9 seconds under the chatbot, 11.8 seconds under self-repair, and 24.8 seconds under the constrained agent, and for GPT-5.4 it is 20.2, 20.6, and 35.1 seconds (Figure 7 in Appendix C.3). The median constrained-agent run therefore takes 2.3 times as long as the median chatbot run for Claude Sonnet 4.6 and 1.7 times as long for GPT-5.4. Self-repair adds only 8 and 2 percent to the median runtime, since most self-repair runs revise at most once (Section 4.1).

6 Conclusion

This paper measures how often, and at what cost, the two large language models that lead the coding market write correct econometric code when the researcher has already chosen the question, the method, and the identification assumptions. The answer depends more on whether the model can run and revise its own code than on whether the prompt includes a few-shot example or on the software in which the code is written. Under the chatbot, most failures are loud, since the script does not run or writes no result file. The few-shot example reduces failures of this kind, and Stata produces them more often than R and Python. Letting the model run and revise its code removes most of the failures that the chatbot produces, and much of that gain requires no more than one revision of a script that has failed. Because the failures it removes are the loud ones, little is left for the example to fix, and the software gap largely closes. The additional cost of agency is small relative to the cost of research time, so the price of the runs is unlikely to limit the use of these tools for econometric work.

A script that runs and returns numbers is not for that reason correct: the failures that

remain once the model runs its code are silent, since nothing in the output reveals that the numbers are wrong. Agency therefore changes the kind of error that the researcher must catch as much as the number of errors, because it removes those that are cheapest to detect and leaves those that only someone who knows the method can recognize. As coding agents spread (Massenkoff et al., 2026), an increasing share of the errors that researchers encounter is likely to be silent.

The results apply to the two models, the English prompts, the three software environments, and the three agency conditions that we tested. Models and harnesses change quickly, and the benchmark can be run again on each new model or harness. Whether an LLM can choose the correct specification for an economic problem is a different question, which our benchmark does not address. With either interface, deciding whether the specification and the numbers are right remains the researcher's work.

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.

Angrist, J. D. and J.-S. Pischke (2010). The credibility revolution in empirical economics: How better research design is taking the con out of econometrics. *Journal of Economic Perspectives 24*(2), 3–30.

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, 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.

Chatterji, A., T. Cunningham, D. J. Deming, Z. Hitzig, C. Ong, C. Y. Shan, and K. Wadman (2025, September). How people use ChatGPT. NBER Working Paper 34255, National Bureau of Economic Research.

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 (2024). 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. Mimeo, University of Chicago.

Giagnorio, A., A. Martin-Lopez, and G. Bavota (2025). Enhancing code generation for low-resource languages: No silver bullet. In *IEEE/ACM International Conference on Program Comprehension (ICPC)*. arXiv:2501.19085.

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)*.

Johnston, D., D. Holtz, A. M. Richmond, C. Ong, P. Tambe, and A. Chatterji (2026). The shift to agentic AI: Evidence from Codex. arXiv preprint arXiv:2606.26959.

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.

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

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)*.

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.

Massenkoff, M., E. Lyubich, S. Sacher, Z. Hitzig, S. Zhang, R. Heller, and P. McCrory (2026, June). Anthropic Economic Index report: Cadences. Anthropic, https://www.anthropic.com/research/economic-index-june-2026-report. Accessed August 24, 2026.

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

Meng, Q., Y. Wang, L. Chen, W. Wu, Y. Li, W. Jiang, Q. Wang, C. Lu, Y. Gao, Y. Wu, and Y. Hu (2026). Agent harness for large language model agents: A survey. Preprints.org.

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.

Mizrahi, M., G. Kaplan, D. Malkin, R. Dror, D. Shahaf, and G. Stanovsky (2024). State of what art? A call for multi-prompt LLM evaluation. *Transactions of the Association for Computational Linguistics 12*, 933–949.

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)*.

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.

Siegel, Z. S., S. Kapoor, N. Nadgir, B. Stroebl, and A. Narayanan (2024). CORE-Bench: Fostering the credibility of published research through a computational reproducibility agent benchmark. arXiv preprint arXiv:2409.11363. Version 2, June 2026.

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.

Starace, G., O. Jaffe, D. Sherburn, J. Aung, J. S. Chan, L. Maksin, R. Dias, E. Mays, B. Kinsella, W. Thompson, J. Heidecke, A. Glaese, and T. Patwardhan (2025). PaperBench: Evaluating AI's ability to replicate AI research. arXiv preprint arXiv:2504.01848.

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.

Tully, T., J. Redfern, D. Das, and D. Xiao (2025, December). 2025: The state of generative AI in the enterprise. Menlo Ventures, https://menlovc.com/perspective/2025-the-state-of-generative-ai-in-the-enterprise/. Accessed August 24, 2026.

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

Weinberger, S. and A. Hozez (2026). Prompt-induced waste in coding agents: Reasoning, effort, harness design, and end-to-end cost. arXiv preprint arXiv:2608.01347. Version 2, August 2026.

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)*.

Yao, Y., X. Tan, C.-H. Liu, Y. Li, Z. Wang, W. Yu, Z. Tan, Y. Tian, G. Zhao, L. Sun, X. Zhang, and T. Yang (2026). Harness-Bench: Measuring harness effects across models in realistic agent workflows. arXiv preprint arXiv:2605.27922.

Zhu, Y., Y. Ding, P. Lai, L. Wang, B. Jing, and G. Chen (2026). StatABench: Dataset and framework for evaluating statistical analysis capabilities of LLMs. arXiv preprint arXiv:2606.22977.

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 Tasks and Scoring

A.1 Tasks and data

The benchmark uses 21 tasks and 13 simulated datasets. Each task specifies the estimand, input data, result-file structure, reference output, numerical tolerances, and method checks. The dataset and reference output are the same across software, prompt and agency conditions, and repetitions. The wild-cluster-bootstrap task is excluded because its statement changed after collection began and it has no few-shot or GPT-5.4 runs. Table 2 lists the analyzed tasks and their scored outputs by type of method.

Five datasets serve several tasks. OLS with robust standard errors, logit, and probit share one dataset. One-way clustered OLS, Poisson, and two-stage least squares share another. Two-way clustered OLS, two-way fixed effects, and staggered DiD use a third. Cook's distance and variance inflation factors share a fourth, and VAR impulse responses and Granger causality share a fifth. The other eight tasks use separate datasets. Shared inputs can create dependence across tasks that task-level clustering does not absorb.

Table 2: Tasks and scored outputs.

Task	Required result
Binary regression	
Probit regression	Intercept and coefficients on `x1` and `x2`, with asymptotic standard errors
Logit regression	Intercept and coefficients on `x1` and `x2`, with asymptotic standard errors
OLS	
OLS with robust standard errors	Intercept and coefficients on `x1` and `x2`, with HC1 standard errors
One-way clustered OLS	Intercept and coefficients on `x1` and `x2`, with standard errors clustered by `cluster`
Two-way clustered OLS	Intercept and coefficient on `x`, with standard errors clustered by firm and year
Difference-in-differences	
Staggered DiD	Equal-weight average of supported cohort-time treatment effects. The standard error is not scored

(continued)

Task	Required result
Two-way fixed effects	Coefficient on `x`, with firm-clustered standard error
Instrumental variables	
Two-stage least squares	Intercept and coefficient on the endogenous regressor, with HC1 standard errors
Regression discontinuity	
Regression discontinuity	Difference between local-linear intercepts at the cutoff, using bandwidth 0.5 and triangular weights
Time series	
ARIMA forecast	One-step-ahead forecast and 95 percent prediction-interval bounds
Granger causality	Aligned-OLS F statistic, numerator and denominator degrees of freedom, and p-value
VAR impulse response	Horizon-four orthogonal response of `y1` to a one-standard-deviation shock in `y2`, with `y1` ordered first in the Cholesky factorization
Local-projection impulse response	Horizon-four response coefficient, with Newey–West standard error using lag four
Poisson regression	
Poisson regression	Intercept and coefficients on `x1` and `x2`, with model-based standard errors, using an exposure offset
Other	
Cook's distance	Zero-based observation identifiers and Cook's distance for the five most influential observations
Variance inflation factors	VIF for each of five regressors
Arellano–Bond GMM	One-step difference-GMM coefficients on lagged outcome and `x`. Standard errors are not scored
Welch t-test	Test statistic, degrees of freedom, and p-value
One-way ANOVA	Omnibus F statistic, numerator and denominator degrees of freedom, and p-value
Chi-square independence test	Pearson χ^2 statistic, degrees of freedom, and p-value
Cox regression	Log hazard ratios for treatment and `x`, with model-based standard errors

Notes: Italic rows name the type of method, and tasks appear in the analysis reading order. Figure 3 orders tasks by chatbot success. The task prompts define the model specification, sample, row names, and output schema. Appendix A.2 states the reference-output conventions and Appendix A.4 states the scoring rules.

A.2 Reference outputs

Each reference output contains the quantities listed in Table 2. The estimate rows identify the term, its estimate, standard error, and effective sample size through the fields `term`, `estimate`, `se`, and `n_obs`. Standard errors are null when the task does not score them. The scorer does not compare sample sizes for staggered DiD, VAR impulse responses, ARIMA forecasts, Cox regression, or Arellano–Bond GMM, where reporting conventions differ. For example, the Cox data contain 2,000 subjects and 1,470 events, and the dynamic panel contains 800 observations from 100 firms. The estimates remain scored in these tasks.

For example, the Arellano–Bond task uses one-step difference GMM (Arellano and Bond, 1991). Its reference coefficients are $\rho = 0.5583$ and $\beta_x = 0.5517$. The stored values come from R's `plm::pgmm`, with `effect = "individual"`, `model = "onestep"`, and `transformation = "d"`. A separate one-step Stata check, `xtabond y x, lags(1)`, agrees within the scoring tolerance.

The staggered-DiD task assigns equal weight to the supported cohort-time 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:

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

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

The reference is computed from these long differences using basic means and arithmetic in each software environment. Package estimates must use the same weights. For example, `aggte(type="simple")` weights cohort-time effects by cohort size and generally returns a different quantity. On the benchmark data, generated with seed 20260515, the stored reference estimate is approximately 1.589. The population value is $20/12 \approx 1.667$, given cohort effects $(\tau_4, \tau_6, \tau_8) = (1, 2, 3)$. The task scores the sample estimate and leaves the standard error null.

A.3 Execution protocol

The chatbot submits one script and receives no execution feedback. Self-repair starts with the same prompt and can submit up to three revisions after an execution error or a

result-file problem. The harness returns the preceding script's output and the file error to the model. Harness-run scripts in these two conditions have a 180-second execution limit. The final saved result determines success, and the cost includes all attempts. The share of self-repair runs with more than one revision is 4.2 percent for Claude Sonnet 4.6 (Sonnet) and 16.3 percent for GPT-5.4. The largest observed revision counts are 3 and 3, respectively.

The constrained agent starts in a working directory containing the task data. It can execute code, inspect results, and revise its script within the session. Claude Code authorizes the shell and permits read-only file tools under its default permission mode. Its shell can access the file system and network beyond the working directory. Codex uses a read-only sandbox for chatbot and self-repair calls and a workspace-write sandbox for the constrained agent. The agent retains its native file-patch and web-search tools. Tool permissions therefore differ between the two harnesses. The submission-validity check tests for access to stored reference outputs and hard-coded reference values.

Sonnet's chatbot and self-repair model calls have a default limit of 180 seconds, separate from the script-execution limit. The August 24 replacement wave overrides the model-call limit. The analyzed sample also contains calls stopped by the default limit. Sonnet's constrained-agent session has a 600-second limit. Codex permits 900 seconds per chatbot or self-repair call and 1,500 seconds per agent session. A model can finish before its limit. Missing-cost runs remain in the success denominator, as described in Section 5. The sample manifest identifies the selected runs and their result-file hashes.

Available run-record timestamps span July 4 to August 25, 2026, for Sonnet and July 14 to August 25 for GPT-5.4. These timestamps cover all GPT-5.4 runs and 63.6 percent of Sonnet runs. The remaining Sonnet runs come from legacy records without a run-level timestamp in this source.

Collection documentation reports StataNow 19.5, R 4.6.0, and Python 3.14.6. Python packages are NumPy 2.4.2, pandas 3.0.0, SciPy 1.18.0, statsmodels 0.14.6, and linearmodels 6.1. Recorded 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 contain executable paths, but a package snapshot is not available for every run. The saved records do not establish a complete inventory of community-contributed Stata packages for each run. The records retain model responses, execution results, and available usage fields. Agent traces and turn counts have different coverage across harnesses.

A.4 Scoring rules

A run succeeds when the final result file matches the reference within tolerance and passes the submission-validity check. Before comparison, a model-blind formatting correction repairs recoverable result files while preserving every reported digit. Empty or truncated documents remain unrepairable. The primary outcome can therefore accept a recoverable file from a run that fails the execution or original result-file check. Appendix B distinguishes this outcome from the diagnostic classification of failures.

Table 3: Requirements for task success.

Requirement	Condition for passing
Numeric agreement	The final result file, after any formatting correction, matches the reference within the task's tolerances.
Submission validity	Input-file hashes agree, and the available script passes the checks for reference access and copied reference numbers.

Both requirements must pass. Appendix B describes the separate five-check diagnostic. A coefficient matches within 1 percent relative tolerance or 0.005 absolute tolerance. A standard error matches within 5 percent relative tolerance or 0.005 absolute tolerance. These rules apply across the three software environments. The absolute floor can admit differences larger than the relative tolerance when a reference value is close to zero.

Rows match by the exact term name. The numeric comparison ignores extra terms and uses the last row when a term appears more than once. Numeric strings are accepted when they convert to finite numbers. Except for the tasks listed in Appendix A.2, reported sample sizes must equal the reference after rounding to integers. Cook's-distance observation identifiers must match exactly as integers.

The comparison allows several serialization and reporting conventions. Empty arrays and empty objects are equivalent in R's `diagnostics` and `method_metadata` fields. When strict JSON parsing fails, Stata decimals without a leading zero and bare-dot missing values are converted to valid JSON. The Welch statistic is compared in magnitude because its sign depends on group ordering. For the Granger test, the denominator degrees of freedom may equal the reference or twice the reference. The test statistic and reported probability are compared separately with their reference values.

Method checks search the available script, logs, and metadata for text patterns associated with the requested estimator and forbidden alternatives. They accept equivalent software idioms, including `felm`/`lfe` and `fixest`, and analytic weights for kernel weighting. A comment can satisfy a pattern without the estimator being executed. The method

check is therefore a diagnostic and does not enter the primary outcome. Tests of the scorer include hard-coded outputs, invalid files, incorrect variance estimators, and valid submissions. A wrong estimator can pass both outcomes if its numbers fall within tolerance and its comments satisfy the method check. Success rates can overstate correct implementation through this route.

Submission validity compares the input file with the task data and searches the saved script for reference-access patterns and reference numbers. When the final script is missing, only the input-file comparison remains available. The literal check searches selected full-precision and six-decimal representations and can miss more coarsely rounded copies of reference values.

The scorer version is `strict_posthoc_2026-08-03`, and the primary-outcome rule is `final_artifact_2026-08-13`. Both identifiers appear in the generated analysis files. The strict diagnostic is 2.1 percentage points below the primary outcome over the full sample and 0.5 points below for Sonnet. No run passes the other four checks and fails submission validity.

We also rescale the numerical tolerances and recompute the strict five-check diagnostic for Claude Sonnet 4.6 at each scale. Halving every tolerance lowers the diagnostic from 88.7 to 87.7 percent in the zero-shot condition and from 91.9 to 91.5 percent in the few-shot condition. Doubling every tolerance changes the zero-shot rate by less than 0.1 points and raises the few-shot rate from 91.9 to 92.3 percent. Runs that fail the execution, result-file, or method check never change status when the tolerances are rescaled, so the choice of tolerance affects only the numeric comparison.

B Failure Types

Table 4 classifies runs by their first failed diagnostic check, in the order execution, result file, numeric agreement, method evidence, and submission validity. The last column contains runs that pass all five. A loud failure fails the recorded execution or result-file check. A silent failure passes those checks but fails numeric agreement or method evidence. This classification differs from primary task success, which evaluates the repaired result file and submission validity. A recoverable file can pass the primary outcome even when execution or the original file check fails. The first-failure classification assigns such a run to the earlier check. The execution check can also pass when the original file satisfies validation despite a nonzero process exit status.

Execution and result-file failures fall by two orders of magnitude from chatbot to constrained agent for both models. Numeric failures fall by about two fifths for Sonnet and one third for GPT-5.4. These counts describe the first failed check, so they omit numeric defects in runs classified as execution or file failures. No run fails submission validity

Table 4: Where runs fail: first check failed, by model, prompt condition, and agency condition.

		Loud failure		Silent failure			
Prompt	Agency condition	Execution check	Result-file check	Numeric check	Method check	Submission validity check	All five checks pass
Claude Sonnet 4.6							
Zero-shot	Chatbot	108	13	49	0	0	586
	Self-repair	18	0	31	1	0	706
	Constrained agent	1	0	35	0	0	720
Few-shot	Chatbot	38	1	56	2	0	659
	Self-repair	5	0	51	3	0	697
	Constrained agent	0	0	26	1	0	729
GPT-5.4 (Codex)							
Zero-shot	Chatbot	173	36	20	10	0	517
	Self-repair	102	0	33	12	0	609
	Constrained agent	0	1	21	20	0	714
Few-shot	Chatbot	48	11	42	14	0	641
	Self-repair	9	0	25	19	0	703
	Constrained agent	0	0	19	13	0	724

Notes: Counts of runs. Each row holds 756 runs, which is 21 tasks $\times$ 3 software environments $\times$ 12 repetitions. Every run enters exactly one column, namely the first check it fails in the order execution, result file, numeric agreement, method evidence, submission validity, or the last column when it misses none. Row totals are 756. Loud failures fail the execution or result-file check. Silent failures pass those checks and fail a later check. No run in the analysis sample fails submission validity first. This table decomposes the five-check anatomy, which the analysis keeps as a diagnostic. Task success in the main text is the primary outcome, numeric agreement on the repaired file and submission validity. A recoverable file can pass this outcome after an earlier diagnostic check fails, so the last column sits at or below the reported task-success rate.

Table 5: Where runs fail: first check failed, by model, agency condition, and software environment, prompt conditions pooled.

		Loud failure		Silent failure			
Agency condition	Software	Execution check	Result-file check	Numeric check	Method check	Submission validity check	All five checks pass
Claude Sonnet 4.6							
Chatbot	Python	26	0	17	0	0	461
	R	29	0	39	0	0	436
	Stata	91	14	49	2	0	348
Self-repair	Python	6	0	16	2	0	480
	R	0	0	25	0	0	479
	Stata	17	0	41	2	0	444
Constrained agent	Python	0	0	21	1	0	482
	R	1	0	28	0	0	475
	Stata	0	0	12	0	0	492
GPT-5.4 (Codex)							
Chatbot	Python	5	0	15	6	0	478
	R	11	1	11	11	0	470
	Stata	205	46	36	7	0	210
Self-repair	Python	0	0	21	7	0	476
	R	0	0	14	12	0	478
	Stata	111	0	23	12	0	358
Constrained agent	Python	0	0	16	8	0	480
	R	0	0	4	5	0	495
	Stata	0	1	20	20	0	463

Notes: Counts of runs. Each row holds 504 runs, which is 21 tasks $\times$ 2 prompt conditions $\times$ 12 repetitions. Every run enters exactly one column, namely the first check it fails in the order execution, result file, numeric agreement, method evidence, submission validity, or the last column when it misses none. Row totals are 504. Loud failures fail the execution or result-file check. Silent failures pass those checks and fail a later check. This table decomposes the five-check anatomy, which the analysis keeps as a diagnostic. Task success in the main text requires numeric agreement on the repaired file and submission validity. A recoverable file can pass this outcome after an earlier diagnostic check fails, so the last column sits at or below the reported task-success rate.

first. Table 5 shows that Stata's lower chatbot success coincides with more execution and result-file failures. These failures become rare under the constrained agent.

For Sonnet, loud failures fall from 10.6 to 0.1 percent of all runs, and silent failures from 7.1 to 4.1 percent. For GPT-5.4, silent failures range from 4.8 to 5.9 percent across agency conditions. Under the constrained agent, silent failures account for 98 percent of diagnostic failures for Sonnet and 99 percent for GPT-5.4. Among Sonnet runs that pass execution and return a valid file, the silent-failure rate falls from 7.9 percent under chatbot to 4.1 percent under the constrained agent. This conditional rate has a smaller denominator than the rate among all runs.

Arellano–Bond GMM remains difficult under the constrained agent, with success rates of 69.4 percent for Sonnet and 75.0 percent for GPT-5.4. For Sonnet, success across agency

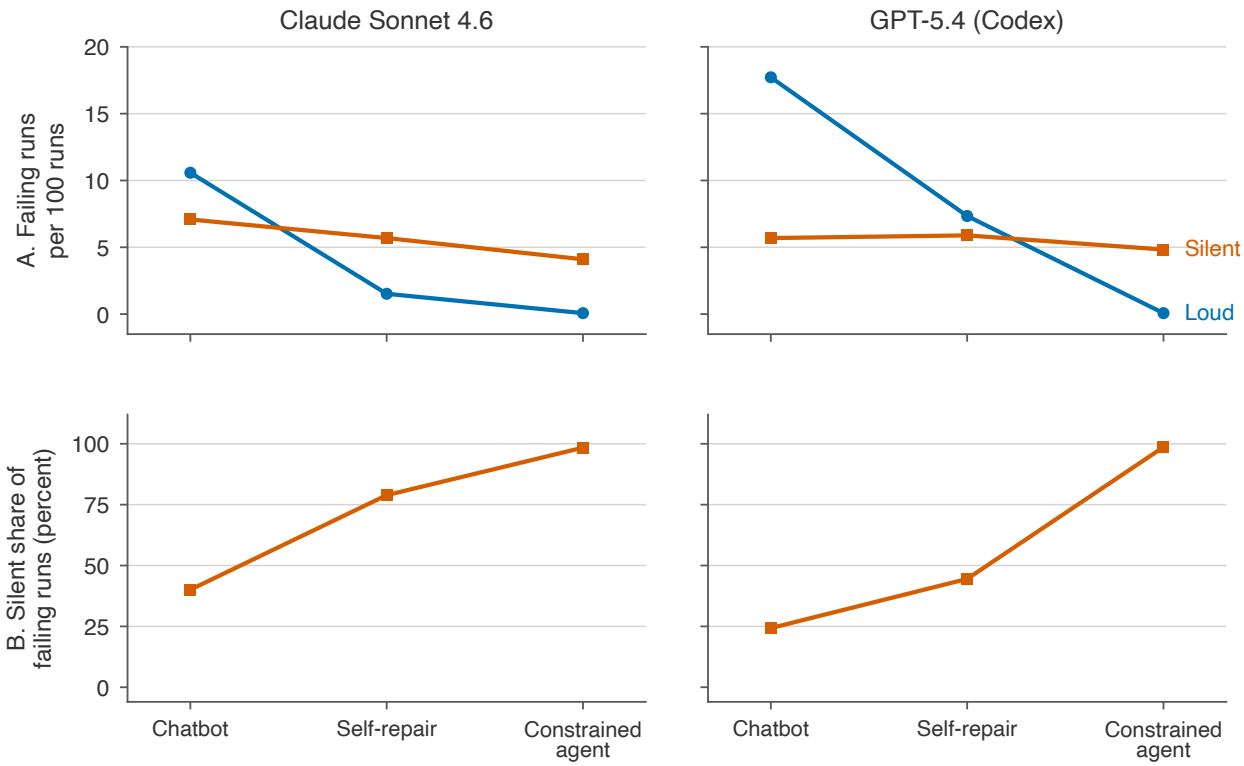

Figure 5: Failure types by model and agency condition. Panel A reports execution and result-file failures (loud) and numeric and method failures (silent) as percentages of all runs. Panel B reports silent failures as a percentage of diagnostic failures. Each point pools both prompt conditions and covers 1,512 runs.

conditions is 95.8 percent in Stata, 70.8 percent in R, and 37.5 percent in Python. The few-shot example raises its task success from 49.1 to 87.0 percent.

C Costs

C.1 Cost measurement

Cost per run includes every model call and permitted revision. Sonnet's records report input, output, cache-creation, and cache-read tokens and the cost recorded by the harness. The cost comparisons reflect recorded cache use and API pricing, not an allocation of subscription fees across runs. For GPT-5.4, we price input, cached-input, and output tokens at the API list prices read on July 30, 2026. These prices are $2.50, $0.25, and $15.00 per million tokens, respectively. The average zero-shot constrained-agent GPT-5.4 run uses 128,945 input tokens, including 109,007 cached tokens, and 2,897 output tokens. Its imputed cost is $0.12. Section 5 defines the priced sample and cost per successful run.

For Sonnet, few-shot prompts raise median cost from $0.12 to $0.20 under chatbot, from $0.18 to $0.20 under self-repair, and from $0.23 to $0.25 under the constrained agent. Self-repair adds about six cents to the zero-shot chatbot median and leaves the few-shot median unchanged at the reported precision. The constrained agent adds about eleven and five cents, respectively. These comparisons describe differences between cost distributions. They do not isolate the cost of revisions from differences in the initial calls or cache use.

Cost per successful constrained-agent run is $0.26 under zero-shot and $0.28 under few-shot for Sonnet. The corresponding GPT-5.4 costs are $0.12 and $0.07. Across agency and prompt conditions, Sonnet's median cost is $0.19 in Python, $0.19 in R, and $0.26 in Stata. Cost per successful run is $0.21, $0.20, and $0.32, respectively.

C.2 Repeated runs

Pass@k is the probability that at least one of k runs from a design cell succeeds. We use the unbiased estimator for draws without replacement from the cell's twelve repetitions (Chen et al., 2021). Each curve in Figure 6 averages over the tasks, software environments, and prompt conditions within one agency condition.

Under chatbot, pass@12 reaches 96.8 percent for Sonnet and 99.2 percent for GPT-5.4. Under the constrained agent, pass@1 is 95.9 and 97.3 percent, respectively, and pass@12 reaches 100.0 and 100.0 percent. Repeated chatbot draws increase the chance of obtaining a correct result, but the user must identify that result among the submissions. For a cell with complete cost records, the total recorded cost is twelve times its mean cost per run, before the user's review time. The pass@k calculation does not estimate a stopping rule or the cost of identifying a correct result.

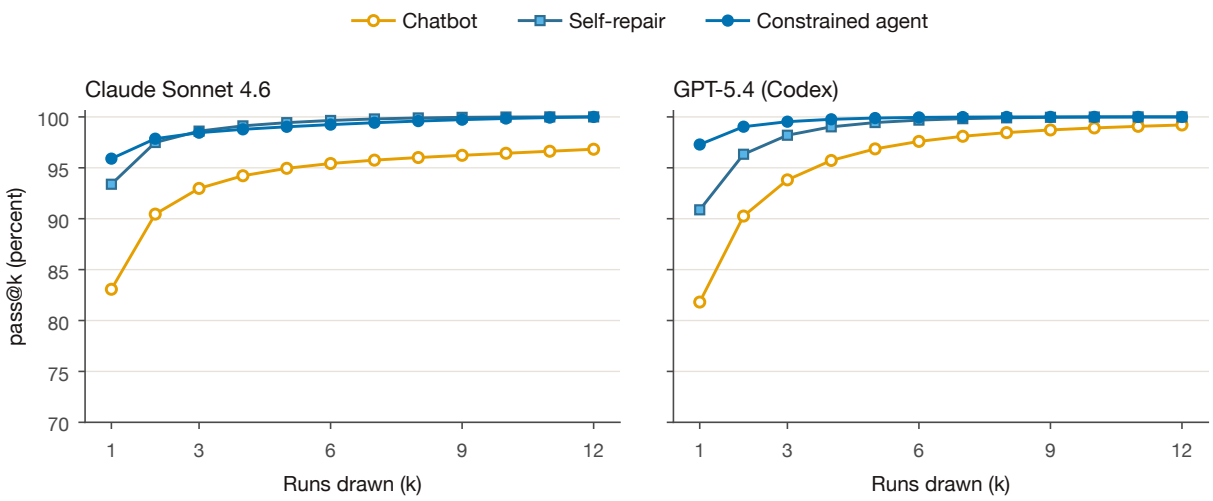

Figure 6: pass@k by agency condition, one panel per model. The horizontal axis is k, the number of runs drawn without replacement from the twelve repetitions of a design cell. Each curve averages the unbiased per-cell estimator over the cells of one agency condition, with both prompt conditions pooled. The vertical axis begins at 70 percent, and the panels share both axes.

C.3 Runtime

Runtime sums the wall-clock duration of model calls across a run's attempts. The agent-session clock includes the script executions within that session. The measure excludes the harness's own script executions under chatbot and self-repair and excludes external harness overhead. Figure 7 reports medians by model and agency condition. Across the analyzed runs, provider calls total 101.8 hours, including 49.2 for Sonnet and 52.7 for GPT-5.4. Retries, provider queueing, and local machine load affect these durations. They measure elapsed time under the collection conditions.

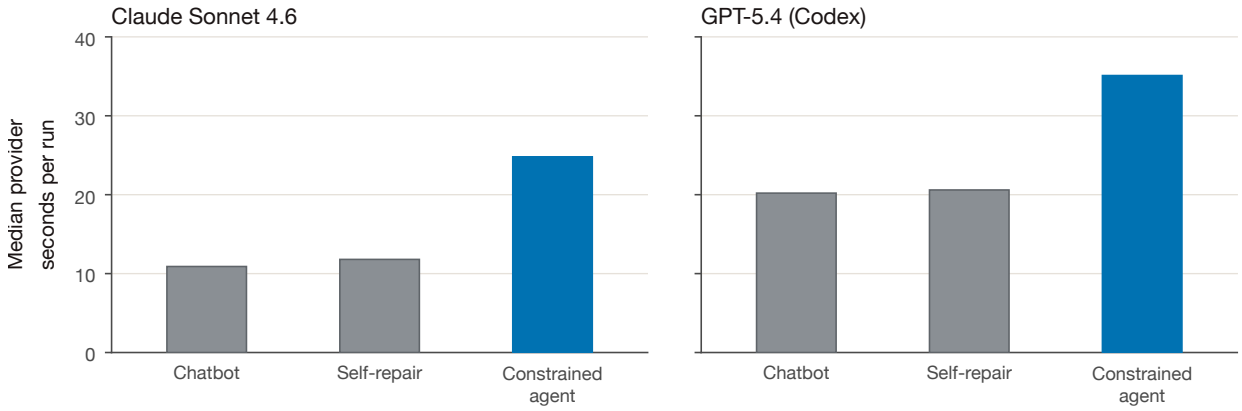

Figure 7: Median runtime by model and agency condition. Each bar pools software and prompt conditions over 1,512 runs. Runtime sums model-call durations, including script execution within agent sessions and all self-repair attempts. It excludes the harness's own script executions and external overhead. Both panels use the same scale.