CODE UNDERSTANDING IS A BOTTLENECK FOR CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Repository benchmarks (e.g., SWE-bench) for coding agents often assume that lines of code edited can predict task difficulty, but such datasets' poor control over code and task types makes it hard to know which abilities truly drive agent errors. We present CABRA: a Coding Ability Blueprint for Rigorous Agent evaluation. CABRA builds tasks from scratch as call graph transformations and scales difficulty via a task size parameter on four axes: function traversal, search, runtime resolution, and instruction following. We run eight LLMs and six coding agents on 6,840 CABRA tasks to show: 1) LLM accuracy falls as task size grows, but agents stay near-perfect by offloading work to tools (e.g., `grep`); 2) Larger CABRA tasks elicit more tool calls for reading and analysis, while a separate study on SWE-bench Verified shows these tool call counts predict agents' accuracy better than lines of code edited—suggesting task difficulty for agents can lie in *understanding* code to edit, not just in making edits; 3) Extending CABRA to an intense understanding task where models analyze divergent logic across two classes backs this finding, as agent accuracy finally falls. More broadly, we argue for synthetic evaluations like CABRA to unmask LLM weaknesses trivialized by tools (e.g., needle-in-a-haystack) and abilities beyond just editing (e.g., understanding) that coding agents still find difficult—pairing SWE-bench-style tasks with controlled diagnosis.

1 INTRODUCTION: HERDING CODING AGENT ABILITIES WITH CABRA

Coding agents execute user queries by reasoning about, editing, and running code (Chen et al., 2021). They are often evaluated on benchmarks where researchers collect GitHub repositories, then augment them with problem statements and test cases as issue resolution tasks (Jimenez et al., 2024).

Progress in coding agents requires knowing when agents fail, but repository-augmented benchmarks impede this. Tasks jointly differ in their code and task types, supporting only correlational analyses (Mehtiyev & Assunção, 2026) that can misidentify why agents fail. For example, Deng et al. (2026) attribute agents' low accuracy in SWE-Bench Pro to agents editing more code, but our analysis shows that agents also have low accuracy when they read, analyze, and test their codebases more (§4.3).

To escape such confounders, researchers have long relied on synthetic tasks that isolate model abilities, enabling evaluations of how model performance on these abilities scales based on task size (Merrill et al., 2020). For example, Needle in a Haystack (Kamradt, 2023) offers a controlled evaluation for long-context reasoning over adjustable corpus sizes. While prior work has extended this methodology to reasoning LLMs (Motwani et al., 2026) and RL environments (Stojanovski et al., 2026), it has not yet been studied in coding agents with vastly different behavior, thanks to their ability to call tools.

We present CABRA: a **C**oding **A**bility **B**lueprint for **R**igorous **A**gent evaluation, a framework for generating programming tasks from scratch. Unlike repository-based benchmarks that derive tasks from codebases, CABRA splits task design into two controllable steps: defining task types and defining how those tasks scale in size. We instantiate these two steps by generating programs from directed acyclic call graphs and transforming them into five task types (Figure 1 left, §2.1), then scaling those tasks across four axes synthesized from prior work: function traversal, function search, runtime resolution, and instruction following (Figure 1 right, §2.2).

We generate 6,840 CABRA task instances of increasing sizes (§3), while controlling for program and task type. By varying one scaling axis at a time across many codebases and tasks, we can robustly surface axes that are hardest for models. We run eight LLMs and six coding agents in Copilot's public harness (§3.2) on these tasks to discover how our scaling axes impact task accuracy for LLMs without tools versus agents with them (§3.1), and reveal opportunities for improving agents.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

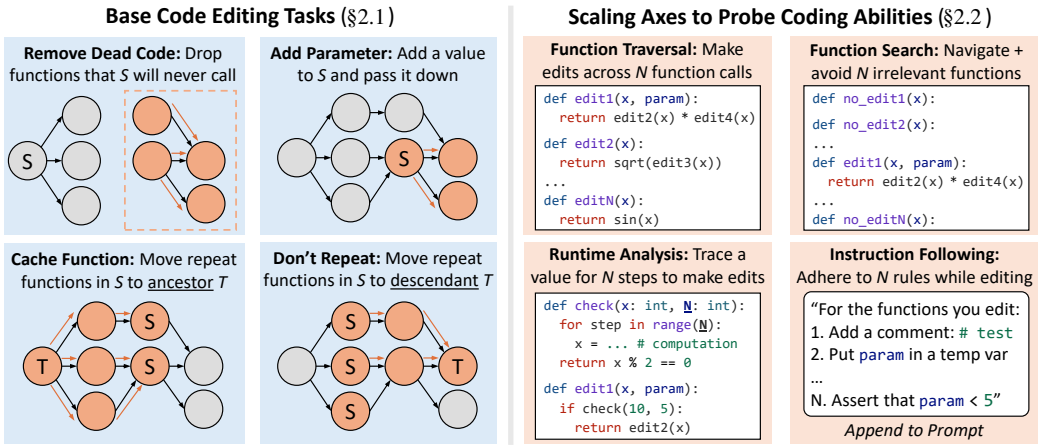


Figure 1: Overview of CABRA, a framework for generating controlled coding tasks in two steps. We first transform directed acyclic call graphs to define base tasks (left). We then scale each task over four axes via task size N to probe abilities ("Add Parameter" example, right).

While LLM accuracy falls on larger CABRA tasks, agents can sustain near-perfect scores by offloading work to tools, like `grep` to search long code (§4.1). To dig deeper, we label agents' tool calls: to read, analyze, search, edit, or test code (§4.2). As task size grows, agents do not always edit more, contrary to prior explanations for low accuracy (Deng et al., 2026). Instead, they often spend more tokens on tools to read and analyze code—*understanding* code to edit. We further verify that this behavior, discovered using CABRA's synthetic tasks, generalizes to more realistic code editing tasks: analyses on SWE-bench Verified traces reveal that the number of tool calls an agent spends to understand code predicts task difficulty better than line edit count (§4.3).

Based on our analyses, we hypothesize that understanding-intensive tasks with relatively few edits can challenge agents. To causally test this hypothesis, we extend CABRA to a new task that targets and controls the amount of code that agents must understand: given two classes, extract all shared logic and preserve any differences (§5.1). When task size grows, agent accuracy falls, tokens spent on understanding tools balloon, and edits stay small—supporting code understanding as a weakness (§5.2). Follow-up analyses suggest agents fail to reason over logical decomposition, over-rely on surface-level naming, and struggle to excise one-line differences.

Our paper has implications for both researchers and developers. For researchers, we argue for more work in controllable tasks like CABRA beyond SWE-bench-style tasks to attribute agent errors. For developers, we highlight code understanding as a key accuracy and efficiency bottleneck, suggesting a direction for future gains. To summarize, our core contributions are:

1. CABRA, a framework that generates coding tasks by sampling task types and scaling their size, affording more controlled analyses than repository-derived benchmarks.
2. Evidence that once-elusive tasks for LLMs are now trivial for agents that execute code, and that code understanding is a scalable ability beyond edit volume to heighten task difficulty.
3. A challenging code merging task to help researchers track agents' code understanding ability.
4. A taxonomy of agent tool calls and classifier to support future coding agent analyses.

2 CABRA: TAKING CONTROL OF CODING AGENT EVALUATIONS

To probe coding agent failures, we present CABRA: a **C**oding **A**bility **B**lueprint for **R**igorous **A**gent evaluation. Rather than augmenting repositories (Jimenez et al., 2024), CABRA creates tasks from scratch via two controlled steps: defining tasks (§2.1) and scaling task size (§2.2).

2.1 TRANSFORMING DIRECTED ACYCLIC GRAPHS AS CODE EDITING TASKS

To control our baseline code editing tasks, we use directed acyclic graphs (DAGs) to represent programs, and then transform them to define code edits. Each DAG $\mathcal{C} = \langle V, E \rangle$ is a function call

Table 1: Four of the coding agent abilities we study, each scaled by a task size parameter n.

Scaling Param n	Ability	Description
# Edit Funcs.	Function Traversal (Wang et al., 2023)	Making n edits over call graphs.
# Non-edit Funcs.	Function Search (Kamradt, 2023)	Avoiding n distractor functions.
# Runtime Steps	Runtime Resolution (Gu et al., 2024)	Resolving values after n steps.
# Instructions	Instruction Following (Pyatkin et al., 2026)	Satisfying n rules while editing.

graph, where $v_i \in V$ is a function and edge $(v_i, v_j) \in E$ means v_i calls v_j. Each function computes a random mathematical function of its input. For v_i with many output vertices, v_i chains the function outputs with random operations. Appendix A.1 describes the sampling process. Figure 1 has example DAGs/programs with floats; we also design string and array versions so our analyses generalize beyond a single code type (Appendix A.2 has examples). We finally add a `__main__` block calling the root vertex in $\mathcal{C}$. This synthetic code generation approach trades downstream realism for simplicity and tight control over tested coding abilities.

We define five diverse tasks based on these DAGs to ensure our later task scaling analyses generalize beyond a single task type. We ground the tasks in Fowler et al. (1999)'s catalogue of refactoring types, also used to derive Gautam et al. (2025)'s tasks in RefactorBench. Each task samples a graph $\mathcal{C}_{\text{init}}$ that models do not edit and a graph $\mathcal{C}_{\text{edit}}$ that models must edit, combined into the input codebase $\mathcal{C}$. We outline our tasks next, visualized in Figure 1 (left).

1. **Remove Dead Code:** We sample $\mathcal{C}_{\text{edit}}$ as a DAG representing functions that $\mathcal{C}$ never calls from `main`. The agent must remove all of the "dead" functions that exist in $\mathcal{C}_{\text{edit}}$.
2. **Add Parameter:** We sample $\mathcal{C}_{\text{edit}}$ as a DAG. Its one root function adds an input parameter (e.g., a multiplier) that must be passed down the call graph and applied at child nodes.
3. **Add Return:** (2)'s inverse: we flip $\mathcal{C}_{\text{edit}}$'s edges to have one sink node. The sink function adds a return value (e.g., a string key) that must be passed all the way up the call graph.
4. **Cache Function:** We sample $\mathcal{C}_{\text{edit}}$ as a tree and call `slow()` at child nodes. Models move `slow()` to the lowest common ancestor and pass its value down to cut redundancy.
5. **Don't Repeat:** The inverse of (4): we flip $\mathcal{C}_{\text{edit}}$'s edges, then at each child node, we call `get(val)` and apply its output. Agents move `get` to the first common descendant and pass its value up, i.e., the DRY principle (Hunt, 1900, **D**on't **R**epeat **Y**ourself).

We sample edges between $\mathcal{C}_{\text{init}}$ and $\mathcal{C}_{\text{edit}}$ with probability $\beta = 0.3$ (except in Remove Dead Code, or $\mathcal{C}_{\text{edit}}$ is no longer dead code). For each task, we develop a problem statement that explains how to solve the task with an example (all problem statements in Appendix A.11).

2.2 Scaling Task Size Across Complexity Axes

We now scale the size of our base tasks on four axes (Figure 1, right). Each axis stress-tests a coding ability by scaling one aspect of the task via a size parameter n. Our axes do not cover all abilities that agents need, but have been studied in prior work and argued as useful for downstream coding tasks. As §5 shows, CABRA can be readily extended to other coding abilities of researchers' interest. The main scaling axes we define are (summary in Table 1):

1. **Function Traversal:** Tasks such as refactoring often require models to propagate edits throughout codebases by tracing call graphs—a commonly-studied graph traversal ability (Wang et al., 2023). This axis scales the number of functions models must edit $|V_{\text{edit}}| = n$.
2. **Function Search:** Models often search through large codebases to decide which sections require edits. This axis scales the number of functions agents do not need to edit $|V_{\text{init}}| = n$, mirroring Needle in a Haystack evaluations for long-context reasoning (Kamradt, 2023).
3. **Runtime Resolution:** Coding tasks may force models to trace increasingly long computations to resolve runtime values (Gu et al., 2024, e.g., CRUXEval). To test this, we wrap an `if check():` around identical code blocks and have models only edit blocks that run. We scale the size of resolving `check()` via a Collatz-based function (Lagarias, 1985) whose `for` loop runs for n steps (Code Block A.4), requiring n steps of reasoning to resolve.

4. **Instruction Following:** Users issue long queries when models should make more edits at once—often assessed via instruction-following benchmarks (Pyatkin et al., 2026). We define task size by appending n distinct instructions to our problem statement, requiring models to also add asserts, comments, temporary values, etc. (full list in Appendix A.2). This axis is unusable with "Remove Dead Code" (§2.1), as deletion is the only valid action.

3 Experimental Setup

We generate $6,840$ tasks with CABRA across three code types, five tasks (four in instruction following), four axes, and six sizes—with 20 tasks each. Varying code and task types lets us more rigorously assess task size scaling per axis. Below, we discuss our tasks, metrics (§3.1), and model design (§3.2).

3.1 Tasks

Task Grid. For each axis in §2.2, we sweep task size parameter $n \in \{5, 10, 25, 50, 100, 200\}$. Function traversal and search fix $|V_{\text{start}}| = 25$ and $|V_{\text{edit}}| = 25$, respectively. We set $|V_{\text{start}}| = |V_{\text{edit}}| = 10$ for runtime resolution/instruction following. We sample 20 $\mathcal{C}$ per configuration, with variation from function calls, operations, `check()` inputs for runtime resolution, and instruction following rules.

Success Metrics. Our tasks are rule-based transformations, so we can form ground-truth programs $\mathcal{C}_{\text{true}}$. We then compare model-edited programs $\mathcal{C}_{\text{pred}}$ with $\mathcal{C}_{\text{true}}$ via black-box tests (Myers et al., 2011): we verify each function in $\mathcal{C}_{\text{pred}}$ returns the same outputs as $\mathcal{C}_{\text{true}}$'s version on 50 random inputs (i.e., $50|V|$ total tests, so 500 minimum). We add rules to verify dead code solutions drop all $\mathcal{C}_{\text{edit}}$ functions, runtime resolution solutions edit valid `if` branches, and instruction following solutions execute all requests. A model gets a score of 1.0 per task iff all checks pass.

3.2 A Flock of Language Models and Agents

We compare how task size affects LLMs and agents. The key difference is that while all models use natural language reasoning, agents have tools. Parameters are default when unspecified.

Language Models. We use GPT (Singh et al., 2025, 5.4 Mini, 5.3 Codex, 5, and 5.5), Grok-4.1 Fast (xAI, 2025), Deepseek-V3.2 (Liu et al., 2025a), GPT-OSS 120B (Agarwal et al., 2025), and Mistral Large 3.1 (MistralAI, 2025). We adapt SWEAgent's prompt (Yang et al., 2024): LLMs view the problem statement, then return a `<thought></thought>` block with natural language reasoning followed by a full rewrite of the code (Prompt A.13). This surpasses generating diff patches (Appendix A.3). We use medium reasoning effort when adjustable.

Agents. We use GitHub Copilot's public harness (Microsoft, 2026, `v1.0.76`) with GPT 5.5 and 5.4 Mini, (OpenAI, 2026), Gemini-3.5 Flash and 3.1 Pro (Google, 2025), Sonnet-4.6, and Opus-4.7 (Anthropic, 2025). We prompt with the problem statement and give read-write access to a folder with the input code. Copilot uses a three-stage ReACT loop (Yao et al., 2023): at each turn, it outputs reasoning r_i, a tool call t_i (`bash`), and adds the tool output o_i. Agents are costly (Table A.1, 25k USD in final runs), so we use just one harness. After our runs, OpenAI disclosed an incident where agents communicated across environments (OpenAI, 2026). This spurred us to look for such behavior in Copilot: 11% of traces accessed shared directories and $< 0.2\%$ queried other agents' session history. However, this minimally impacts results, as fully sandboxed runs have the same accuracy and tool call patterns (Appendix A.8).

4 Agents Spend Effort Understanding Code, not Just Editing

We find that large CABRA tasks challenge LLMs but not agents due to tool use (§4.1). Agent traces show they spend many tokens on tools to understand code in larger tasks—not just edit it (§4.2)—with similar trends in SWE-Bench (§4.3). We thus argue that scaling understanding effort is promising for making hard agentic coding tasks, which we test in §5.

4.1 Comparison of LLM and Coding Agent Accuracy

We display aggregated results on our four complexity axes in Figure 2, with takeaways below:

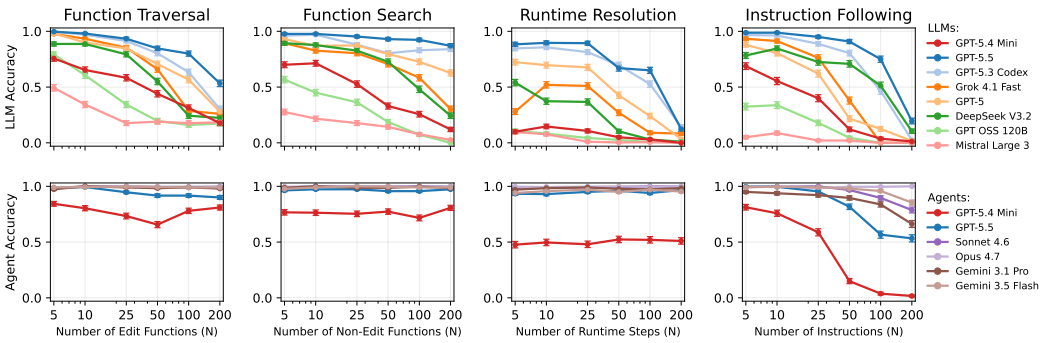


Figure 2: Comparison of LLM and agent accuracy over our four scaling axes, averaged over all tasks with stderr. While all LLMs lower in accuracy as task size n grows, agents often remain near-perfect. Appendix A.5 has similar trends when segmenting by each task.

LLM-Hard $\not\Rightarrow$ Agent-Hard. LLM accuracy drops with n, but agent accuracy is mostly constant and often near perfect. This is not just due to differences in backbone LLMs, as GPT-5.4 Mini and GPT-5.5 degrade in accuracy as LLMs, but score consistently as agents (except instruction following). Appendix A.5 has the same trend with a subset of CABRA tasks with Gemini models. Tasks of certain complexity or size that challenge LLMs may no longer challenge agents, so evaluating agents' limits will either require scaling task sizes to colossal n (Sinha et al., 2026) or new axes for complexity. We pursue the latter path in §5.

Weaker LLMs can Climb via Harnesses. GPT-5.4 Mini, when used in an agent harness, surpasses stronger LLMs without tools (GPT-5.5, Grok) in non-instruction following tasks for $n > 100$. Thus, developers do not always need large LLMs in harnesses; small LLMs that generate and compose tool calls well can generalize to larger tasks (Zhang & Khattab, 2026).

Tools Let Agents Graze Codebases. To explain agents' near-perfect accuracy, we analyze execution traces. Agents often write and run "meta-scripts": `ast` code to move variables in function traversal, `grep` to locate names in function search, or `exec()` to learn outputs in runtime resolution. LLMs often generate more natural language reasoning as task size grows (Figure A.7), but some tasks let agents write scripts with fixed reasoning invariant of n.

The exception is instruction following—rules struggle to map natural language to edits (Zan et al., 2023)—which is also the only axis where agents degrade ($n = 200$, except Opus). A new consideration for synthetic tasks with agents is whether tools bypass intended task size. As we show in §5, there are scalable axes beyond instruction following that are hard to script.

Table 2: Our surfaced tool call taxonomy and the distribution of tool use in our evaluations.

Label	Example	Description
READ (22%)	`view solution.py`	Reading files or lines of code verbatim.
ANALYZE (14%)	`ast.parse(src)`	Analyzing dependencies, call graphs, runtime, etc.
SEARCH (11%)	`grep "def check"`	Pattern matching a parameter, function, or string.
EDIT (36%)	`edit solution.py`	Changing lines of code in the solution program.
TEST (8%)	`pytest test.py`	Validating code via syntax checks or test cases.
OTHER (9%)	`pip install numpy`	Any other tool (summary, file cleanup, etc.)

4.2 TRACE ANALYSIS: AGENTS OFTEN SPEND TOOL CALLS ON UNDERSTANDING CODE

We analyze tool use further via Bingham (2023)'s protocol: we review a subset of agent traces and derive six tool types (Table 2). We then prompt Qwen3-4B Instruct (Zhang et al., 2025b) to label each tool call in agent traces from §4.1 via its name, inputs, and reasoning tokens—reaching 94% agreement with 20 author-labeled traces per class (details in Appendix A.4).

For each tool type, we find the number of calls and output tokens used, averaged over agents. For tool call t_i, the latter is the sum of the tokens used to create t_i and tokens in reasoning r_i: chain-of-thought between the output o_{i-1} of t_{i-1} and t_i (or after the prompt for $i = 0$). Token use is our proxy for

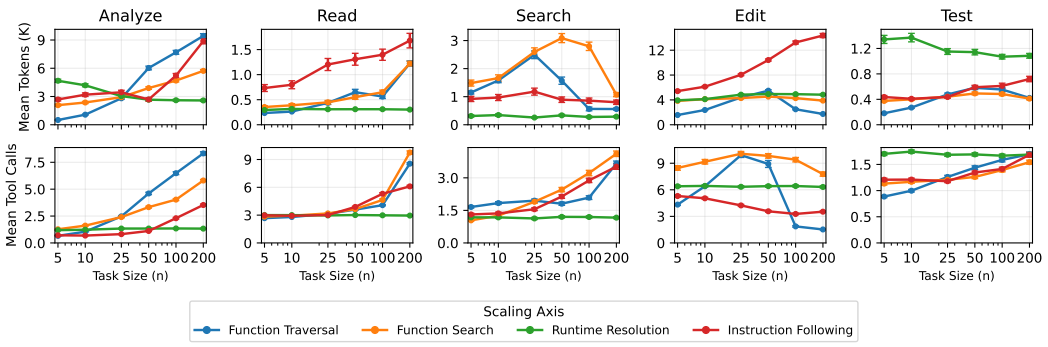


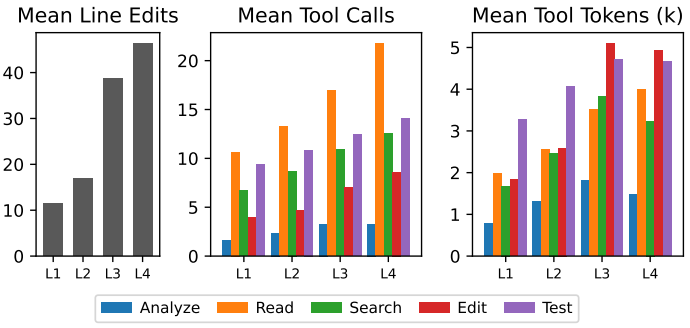
Figure 3: Agents' mean tool call count and output tokens for Table 2's categories (with stderr). Only understanding tools (READ, ANALYZE) scale in calls and tokens with task size in 3/4 axes. EDIT only scales in instruction-following. Per-agent results are similar (Appendix A.6).

effort devoted to each tool type. Some tokens may link to earlier calls, but token attribution is hard, and Chen et al. (2026) study step-wise token use like this.

While prior work uses edits to predict coding task difficulty (Deng et al., 2026; Raghavendra et al., 2026), only instruction following elicits more EDIT tokens with n (Figure 3). Instead, READ and ANALYZE tokens grow for all axes but runtime. We treat these two as **code understanding** actions: acquiring knowledge of a program's construction and reasoning about its behavior to guide later edits (Littman et al., 1987; Maalej et al., 2014). Their position in agent traces backs our interpretation: 48% of READ and 40% of ANALYZE calls occur in the first quarter, and 69% and 56%, respectively, precede the first EDIT—the most of all tools. SEARCH may support understanding, but work treats pattern-matching lookups (`grep`) as distinct actions preceding understanding (Wilde et al., 2001; Bennett et al., 2002). Consistent with this idea, only 12% of SEARCH tools are followed by EDIT, but 19% are followed by READ.

Comparing tool use on our scaling axes with agent accuracy on said axes (Figure 2) shows agents excel on tasks where they UNDERSTAND (READ + ANALYZE) more (search/traversal), but start to fail if they UNDERSTAND *and* EDIT more (instruction following). Appendix A.9 has tasks where EDIT use grows more than UNDERSTAND, validating them as unique constructs. This spurs us to ask if tasks that demand UNDERSTAND tools can thwart agents in §5.

(a) Trace metrics grouped by human-annotated difficulty levels.



(b) Top-eight ranked metrics by agent accuracy correlation.

Metric	r (↑)
UNDERSTAND$_{num}$	-0.200
ALL TOOL$_{tok}$	-0.178
EDIT$_{tok}$	-0.177
READ$_{num}$	-0.171
Lines Edited	-0.159
ANALYZE$_{num}$	-0.158
ALL TOOL$_{num}$	-0.157
EDIT$_{num}$	-0.151

Figure 4: Trace analysis on SWE-Bench Verified (Jimenez et al., 2024) reveals: (a) human-labeled difficulty correlates with edit size and tool use, making failures hard to disentangle; and (b) UNDERSTAND tool call count (READ + ANALYZE) is most strongly linked to agent failures.

4.3 UNDERSTANDING PERSISTS IN SWE-BENCH VERIFIED

CABRA's synthetic tasks offer control but lack ecological validity (De Vries et al., 2020), so we test our hypothesis—code understanding beyond editing explains task difficulty in agents—on more realistic tasks. We collect released `mini-swe-agent` trajectories on SWE-bench Verified (Yang et al., 2024) from Claude-4.5 Opus/Sonnet, Gemini-3 Pro/Flash, GPT-5.2, and GPT-5 Mini. We

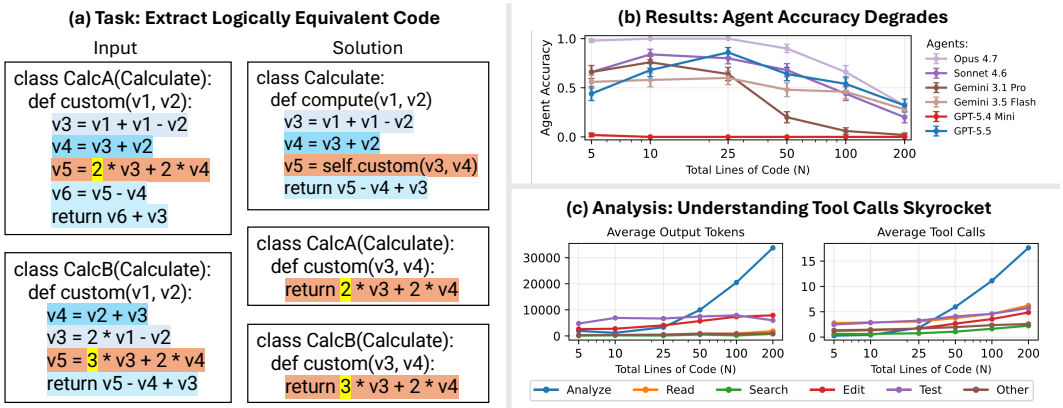


Figure 5: We extend CABRA to an intense understanding task: given two classes with shared logic, move logically equivalent code into a parent class while preserving per-class differences (a). Agent accuracy plummets as lines of code grow (b) and understanding tools balloon (c).

study how the edited lines of code in the agents' final solution (a commonly-used difficulty metric (Deng et al., 2026; Raghavendra et al., 2026)) versus our tool call metrics from §4.2 (by running our classifier on these traces) relate to two measures of task difficulty: SWE-bench's human-annotated difficulty levels and agents' accuracy (linked to their traces).

Edits are not the scapegoat for human-rated difficulty. SWE-bench tasks come with difficulty levels based on the time developers estimate humans would need to finish them (L1 $\leq$15 min; L2 = 15 min – 1 hr; L3 = 1–4 hours; L4 $\geq$4 hours). Most metrics rise with level (Figure 4, left): longer tasks elicit not just more edits, but also reading, analysis, and testing. Since metrics grow jointly, SWE-bench alone cannot discern when errors stem from edit volume or understanding how to edit—motivating controlled evaluations such as CABRA.

Understanding predicts errors better than edit count. We measure the point-biserial correlation r (Tate, 1954) between agent accuracy and each metric. All r are low: it is tough to predict errors, further motivating CABRA-style tasks for more control. The number of understanding tool calls (defined as READ plus ANALYZE calls) is most strongly correlated with accuracy (Figure 4, right), more so than EDIT token use and lines edited. Overall, this further underlines the key role code understanding plays in the evaluation of coding agents.

5 INTENSE UNDERSTANDING TASKS CAN MAKE CODING AGENTS BLEAT

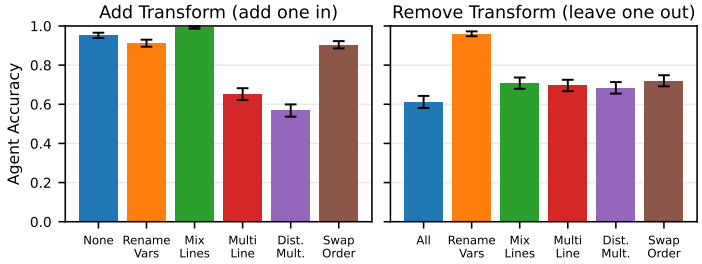
While work often scales *edit size* toward harder coding agent tasks, §4 suggests to control agent accuracy via an analogous *understanding size*: how much understanding occurs before editing. To test this, we apply CABRA's two-stage evaluation approach: we design a synthetic task where models compare lines of code across classes to extract shared logic (§5.1), then show agent accuracy drops over three scaling axes that target understanding effort (§5.2).

5.1 A CODE UNDERSTANDING TASK: EXTRACTING LOGICALLY EQUIVALENT CODE

We extend CABRA to a code understanding task based on program equivalence (Pitts, 1997), which Wei et al. (2025) use to test if models understand code semantics. We define calculator classes as DAGs $\mathcal{C} = \langle V, E \rangle$, where $v_i \in V$ is a line defining a variable and edge $(v_i, v_j) \in E$ means v_j uses v_i in its computation (DAG visualized in Figure A.17). Models start with classes $\mathcal{C}_A, \mathcal{C}_B$ with ℓ lines each that differ in behavior at d lines: models must move the $\ell - d$ equivalent lines into a parent class $\mathcal{C}_{\text{parent}}$ and keep the d differences in child classes (Figure 5, left). As ℓ grows, edits scale modestly (e.g., $O(\ell)$ lines per class), but combinations of lines that can be understood to locate differences skyrockets (e.g., $\Omega(\ell^2)$ lines worst case). We later validate this intuition empirically: agents' tool calls for ANALYZE rise sharply with ℓ (§5.2).

We instantiate tasks by first sampling ℓ line nodes; each line defines a new variable by chaining input parameters, past variables, and non-zero integer coefficients in $[-5, 5]$ via sums and products. We

(a) Mean agent accuracy after adding/removing transforms ($n = 50$).



(b) Mean agent accuracy with varied difference count.

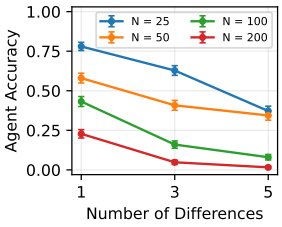


Figure 6: We reveal agent difficulties in extracting equivalent code: aligning identical decompositions (left), mapping renamed variables (middle), and locating differing lines (right).

randomly select d divergent line nodes, done by either: 1) flipping a sign; 2) editing a coefficient; 3) editing a variable; or 4) adding an extra term. To make the code surrounding these differences behaviorally equivalent while syntactically dissimilar, we use five transformations: 1) rename variables; 2) shuffle commutative lines; 3) decompose/merge lines of code; 4) distribute multiplication ($3a = a + a + a$); and 5) swap term order ($a + b = b + a$).

We prompt agents with one example and ask for the most efficient edit, i.e., keep as much code as possible in the parent class (Prompt A.6). Agents receive a score of 1 iff they preserve behavior (black-box testing, §3.1) and each child class has d different lines of code maximum.

5.2 Understanding When Understanding is Challenging

We now scale the size of our code equivalence task (§5.1) to probe agents' code understanding abilities. We first confirm that scaling one axis—(1) the total lines of code per class—raises understanding effort and lowers accuracy. We then study two more scaling axes: (2) the total transformations agents reason over; and (3) the number of differing lines they must locate.

1) Comparing Lines of Code. We sample 50 tasks while scaling $\ell \in \{5, 10, 25, 50, 100, 200\}$, increasing the combinations of lines of code that could be understood and compared. Higher ℓ drops agent accuracy (Figure 5b),[1] and elicits far more UNDERSTAND than EDIT tools (Figure 5c)—supporting that understanding is a weakness in coding agents that can be improved.

2) Aligning Logic. We hypothesize that our understanding task is hard as it requires aligning syntactically distinct but semantically equivalent code (Churchill et al., 2019). Thus, we test the equivalence transform types agents must align as complexity axes for understanding. We ablate transforms at $\ell = 50$: adding each to a no-transform baseline (Figure 6, left) and removing each from an all-transform baseline (Figure 6, middle). Multi-line edits and distributive multiplication cause large drops when added—for Gemini specifically (Appendix A.10)—failing on multi-hop alignment (Zhang et al., 2025a). Removing variable renaming boosts accuracy most, so agents may rely on shallow naming to understand code (Lam et al., 2025).

3) Locating Differences. When agents edit, they generally follow the two-step process of: (i) locating lines that differ; and (ii) executing the refactor. To disentangle where errors occur, we study both metrics in §5.1. Agents preserve code behavior in over 90% of cases, so most failures stem from (i) versus (ii); agents keep too many lines in the child class, failing to find one line differences. We further probe this weakness by varying the number of differing lines $d \in \{1, 3, 5\}$ while holding lines of code ℓ fixed. Accuracy erodes as d grows (Figure 6, right), validating agents struggle increasingly as they need to identify more behavioral differences.

6 Related Work

To ground CABRA, we synthesize work on code evaluations, agent errors, and synthetic tasks.

[1] Prompted LLMs from §3.2 are much less accurate (Appendix A.10).

Evaluating Code Models. Code generation is a long-studied task (Hindle et al., 2012). Early evaluations used NL-to-code translation tasks (Oda et al., 2015; Yin et al., 2018), then scaled difficulty with algorithm tasks based on contests like LeetCode (Chen et al., 2021; Li et al., 2022; Jain et al., 2025). The boom of agents that compose LLMs and tools (Yao et al., 2023) motivated harder tasks like issue resolution (Jimenez et al., 2024; Zan et al., 2026) and full app generation from specifications (Si et al., 2025; Zhao et al., 2025). To scale difficulty, work adds more edits (Deng et al., 2026; Raghavendra et al., 2026), multi-turn prompts (Le et al., 2025; Laban et al., 2026b), and extra requirements (Ding et al., 2026; Yang et al., 2026b). We also show that understanding is a less studied but key axis to control difficulty.

While such tasks simulate real use, they struggle to diagnose model errors, as they differ on many confounded axes. We review these limitations then discuss synthetic tasks as a solution.

Agent Failures. Knowing model failures drives progress. For coding agents, work often studies properties of failed traces on repository benchmarks (Cemri et al., 2026) like execution strategy, tool call type, and length—relating them to agent accuracy to guide methods (Majgaonkar et al., 2025; Liu et al., 2025b; Mehtiyev & Assunção, 2026). However, analyses are correlational—tasks also vary in codebase and task—making it hard to attribute failures, and correlations are often poor (§4.3). We argue CABRA-style tasks that control such confounders are better for this goal, which we use to surface agent weaknesses in code understanding.

Synthetic Tasks. Compared to repository benchmarks, synthetic tasks can better probe model errors (Merrill et al., 2020). Tasks like bAbI (Weston et al., 2016; Tamari et al., 2022) and Needle in a Haystack (Hsieh et al., 2024; Kim et al., 2025) test limits of a model ability (e.g., long-context) by defining tasks (e.g., locate a passkey) and scaling their size (e.g., corpus size); we use this two-stage approach for CABRA. Recent work extends this to reasoning LLMs: LongCoT for long-horizon execution (Motwani et al., 2026) and RELIC for context-free grammar recognition (Petty et al., 2026). In code specifically, related work includes CruxEval for code tracing (Gu et al., 2024) and Bug in the Code Stack for codebase search (Lee et al., 2024), which form our initial CABRA tasks. There are also synthetic RL environments to train coding agents (Yang et al., 2026a; Zhu et al., 2025), but their use of existing repositories or LLM-generated data impede full control over tasks, unlike CABRA.

Different from all of these works, we explore task complexity in coding agents. Our main contribution is exposing that many "LLM-hard" tasks are trivial with tool use—task complexity must evolve for agents—then surfacing and probing code understanding as a model weakness.

7 CONCLUSION: RAISING THE BAA-R FOR CODING AGENT EVALUATIONS

Using CABRA, we show the number of edits alone is a poor proxy for coding agent task difficulty and instead surface code understanding as a key factor. Across our initial task axes, editing effort scales insufficiently with task size. In contrast, we reveal that code understanding more often scales with task size across diverse axes and more reliably predicts SWE-bench Verified errors. We establish this link concretely with our code merging task and surface facets like multi-line decomposition and readability where agents still struggle, motivating threads for future work.

Controlled Evaluation. CABRA-style tasks can both attribute agent error and form axes to probe them. We use tools for diagnosis, but future work can study problem-solving methods (Gao et al., 2024) and error recovery (Huang et al., 2024). The next step is imbuing such weaknesses into SWE-bench-style tasks as more realistic evaluations (Saxon et al., 2024).

Agent-Understandable Code. Agents struggle to compare programs with renamed variables (§5.2), showing some code types are harder to understand. Future work could test if repository indexes (Luo et al., 2026), memory (Zhang et al., 2025c), and readability practices (Buse & Weimer, 2009) help agents. As understanding can encompass up to 70% of SWE work (Minelli et al., 2015), 19% of real coding requests (Baumann et al., 2026), and already demands effort in CABRA and SWE-Bench, progress here is key for maintainability (Le et al., 2025) and limiting context rot (Laban et al., 2026a).

A Theory of Agents. Work on transformers' limits (Bhattamishra et al., 2020) often uses synthetic tasks like CABRA to empirically test theories, but tasks that were once hard for LLMs can now be trivial with tools (§4.1). Discerning "agent-hard" or "-easy" tasks will require formalizing the work used to call tools and work tools can execute (Meyerson & Qiu, 2025).

AI USE STATEMENT

In this work, we used generative AI tools primarily for code implementation and data visualization (e.g., plots). We have not used generative AI for writing from scratch, figure generation, or designing the high-level framing and structure of this paper. We also minimally used generative AI tools for revising writing, brainstorming, reviewing our paper, and literature search. For coding, we have reviewed all AI-assisted work through a combination of manually defined test cases, and visualization tools to verify that our evaluations generate data, elicit model responses, and score responses correctly. Before releasing our code, one author reviewed all generative AI outputs to detect any errors. We take full responsibility for the final content of this work, including any errors from generative AI.

ETHICS STATEMENT

Our paper shows that coding agents can struggle to understand code. Given the importance of understanding in real developer workflows (Minelli et al., 2015; Baumann et al., 2026)—coupled with evidence that programmers struggle to understand their code when working with agents (Shen & Tamkin, 2026)—there are risks for failing to address this problem. We hope that our paper brings more attention to evaluating understanding in coding agents.

REPRODUCIBILITY STATEMENT

Upon acceptance, we will publicly release our code on GitHub and CABRA data on Hugging Face so researchers can extend our work. For the submission, we have attached an anonymized code repository as supplementary material. In Appendix A.3, we provide more details on our model setup to help researchers reproduce our results.

REFERENCES

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Anthropic. Introducing claude sonnet 4.5. https://www.anthropic.com/news/claude-sonnet-4-5, September 29 2025. Accessed: 2025-11-29.

Joachim Baumann, Vishakh Padmakumar, Xiang Li, John Yang, Diyi Yang, and Sanmi Koyejo. Swe-chat: Coding agent interactions from real users in the wild. *arXiv preprint arXiv:2604.20779*, 2026.

Keith H Bennett, Václav T Rajlich, and Norman Wilde. Software evolution and the staged model of the software lifecycle. In *Advances in Computers*, volume 56, pp. 1–54. Elsevier, 2002.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the Ability and Limitations of Transformers to Recognize Formal Languages. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 7096–7116, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.576. URL https://aclanthology.org/2020.emnlp-main.576/.

Andrea J Bingham. From data management to actionable findings: A five-phase process of qualitative data analysis. *International journal of qualitative methods*, 22:16094069231183620, 2023.

Raymond PL Buse and Westley R Weimer. Learning a metric for code readability. *IEEE Transactions on software engineering*, 36(4):546–558, 2009.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. URL https://openreview.net/forum?id=fAjbYBmonr.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

Sirui Chen, Mengshi Zhao, Lei Xu, Yuying Zhao, Beier Zhu, Hanwang Zhang, Shengjie Zhao, and Chaochao Lu. Depo: Dual-efficiency preference optimization for llm agents. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 30279–30287, 2026.

Wei Cheng, Yongchang Cao, Chen Shen, Binhua Li, Jue Chen, Yongbin Li, and Wei Hu. To diff or not to diff? structure-aware and adaptive output formats for efficient LLM-based code editing. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 29677–29694, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1483. URL https://aclanthology.org/2026.findings-acl.1483/.

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pp. 1027–1040, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314596. URL https://doi.org/10.1145/3314221.3314596.

Harm De Vries, Dzmitry Bahdanau, and Christopher Manning. Towards ecologically valid research on language user interfaces. *arXiv preprint arXiv:2007.14435*, 2020.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa R Kundurthy, Sean M. Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-bench pro: Can AI agents solve long-horizon software engineering tasks? In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=uEVTdoAbnK.

Deming Ding, Shichun Liu, Enhui Yang, Jiahang Lin, Ziying Chen, Shihan Dou, Honglin Guo, Weiyu Cheng, Pengyu Zhao, Chengjun Xiao, Qunhong Zeng, Qi Zhang, Xuanjing Huang, Qidi Xu, and Tao Gui. OctoBench: Benchmarking scaffold-aware instruction following in repository-grounded agentic coding. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 5958–5978, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.269. URL https://aclanthology.org/2026.acl-long.269/.

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*, volume 1. Addison-wesley Reading, 1999.

Chang Gao, Haiyun Jiang, Deng Cai, Shuming Shi, and Wai Lam. Strategyllm: Large language models as strategy generators, executors, optimizers, and evaluators for problem solving. *Advances in Neural Information Processing Systems*, 37:96797–96846, 2024.

Esteban Garces Arias, Meimingwei Li, Christian Heumann, and Matthias Assenmacher. Decoding decoded: Understanding hyperparameter effects in open-ended text generation. In Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert (eds.), *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 9992–10020, Abu Dhabi, UAE, January 2025. Association for Computational Linguistics. URL https://aclanthology.org/2025.coling-main.668/.

Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian. Refactorbench: Evaluating stateful reasoning in language agents through code. In *International Conference on Learning Representations*, volume 2025, pp. 43131–43162, 2025.

Google. A new era of intelligence with gemini 3. https://blog.google/products-and-platforms/products/gemini/gemini-3/, November 2025.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=Ffpg52swvg.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 837–847. IEEE Press, 2012. ISBN 9781467310673.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=kIoBbc76Sy.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=IkmD3fKBPQ.

Andrew Hunt. *The pragmatic programmer*. Pearson Education India, 1900.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pp. 58791–58831, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Gregory Kamradt. Needle in a haystack - pressure testing llms, 2023. URL https://github.com/gkamradt/LLMTest_NeedleInAHaystack.

Yekyung Kim, Jenna Russell, Marzena Karpinska, and Mohit Iyyer. One ruler to measure them all: Benchmarking multilingual long-context language models. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=3vxxB3Ar9r.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. LLMs get lost in multi-turn conversation. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=VKGTGGcwl6.

Philippe Laban, Tobias Schnabel, and Jennifer Neville. Llms corrupt your documents when you delegate. *ArXiv*, abs/2604.15597, 2026b. URL https://api.semanticscholar.org/CorpusID:287607479.

Jeffrey C Lagarias. The 3 x+ 1 problem and its generalizations. *The American Mathematical Monthly*, 92(1):3–23, 1985.

Man Ho Lam, Chaozheng Wang, Jen tse Huang, and Michael Lyu. Codecrash: Exposing LLM fragility to misleading natural language in code reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=CAB0EjD9EK.

Tue Le, Minh VT Thai, Dung Nguyen Manh, Huy Phan Nhat, and Nghi DQ Bui. Swe-evo: Benchmarking coding agents in long-horizon software evolution scenarios. *arXiv preprint arXiv:2512.18470*, 2025.

Hokyung Lee, Sumanyu Sharma, and Bing Hu. Bug in the code stack: Can llms find bugs in large python code stacks. *arXiv preprint arXiv:2406.15325*, 2024.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

David C Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 7(4):341–355, 1987.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025a.

Simiao Liu, Fang Liu, Liehao Li, Xin Tan, Yinghao Zhu, Xiaoli Lian, and Li Zhang. An empirical study on failures in automated issue solving. *ArXiv*, abs/2509.13941, 2025b. URL https://api.semanticscholar.org/CorpusID:281333210.

Jane Luo, Chengyu Yin, Xin Zhang, Qingtao Li, Steven Liu, Yiming Huang, Jie Wu, Hao Liu, Yangyu Huang, Yu Kang, Fangkai Yang, Ying Xin, and Scarlett Li. Closing the loop: Universal repository representation with RPG-encoder. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=am8FJ31GWi.

Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):1–37, 2014.

Oorja Majgaonkar, Zhiwei Fei, Xiang Li, Federica Sarro, and He Ye. Understanding code agent behaviour: An empirical study of success and failure trajectories. *ArXiv*, abs/2511.00197, 2025. URL https://api.semanticscholar.org/CorpusID:282739603.

Tural Mehtiyev and Wesley Assunção. Beyond resolution rates: Behavioral drivers of coding agent success and failure. *arXiv preprint arXiv:2604.02547*, 2026.

William Merrill, Gail Weiss, Yoav Goldberg, Roy Schwartz, Noah A. Smith, and Eran Yahav. A formal hierarchy of RNN architectures. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (eds.), *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 443–459, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.43. URL https://aclanthology.org/2020.acl-main.43/.

Elliot Meyerson and Xin Qiu. Position: Scaling LLM agents requires asymptotic analysis with LLM primitives. In *Forty-second International Conference on Machine Learning Position Paper Track*, 2025. URL https://openreview.net/forum?id=LEYmr1TsBW.

Microsoft. vscode-copilot-chat, 2026. URL https://github.com/microsoft/vscode-copilot-chat. GitHub repository.

Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer-an investigation of how developers spend their time. In *2015 IEEE 23rd international conference on program comprehension*, pp. 25–35. IEEE, 2015.

MistralAI. Introducing mistral 3. https://mistral.ai/news/mistral-3/, 2025.

Sumeet Ramesh Motwani, Daniel Nichols, Charles London, Peggy Li, Fabio Pizzati, Acer Blake, Hasan Hammoud, Tavish McDonald, Akshat Naik, Alesia Ivanova, et al. Longcot: Benchmarking long-horizon chain-of-thought reasoning. *arXiv preprint arXiv:2604.14140*, 2026.

Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pp. 574–584, Lincoln, Nebraska, USA, November 2015. IEEE Computer Society. ISBN 978-1-5090-0025-8. doi: 10.1109/ASE.2015.36. URL https://doi.org/10.1109/ASE.2015.36.

OpenAI. Openai and hugging face partner to address security incident during model evaluation. https://openai.com/index/hugging-face-model-evaluation-security-incident/, July 2026. Accessed: 2026-09-09.

OpenAI. Gpt-5.5 system card. https://openai.com/index/gpt-5-5-system-card/, 2026.

Jackson Petty, Michael Y Hu, Wentao Wang, Shauli Ravfogel, William Merrill, and Tal Linzen. Relic: Evaluating complex reasoning via the recognition of languages in-context. *Transactions of the Association for Computational Linguistics*, 14:1410–1431, 2026.

Andrew M Pitts. Operationally-based theories of program equivalence. *Semantics and Logics of Computation*, 14:241, 1997.

Valentina Pyatkin, Saumya Malik, Victoria Graf, Hamish Ivison, Shengyi Huang, Pradeep Dasigi, Nathan Lambert, and Hannaneh Hajishirzi. Generalizing verifiable instruction following. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. URL https://openreview.net/forum?id=yfYgwjj5F8.

Mohit Raghavendra, Soham Dan, Miguel Romero Calvo, Yannis Yiming He, Johannes Baptist Mols, Gautam Anand, Cole McCollum, Edgar Arakelyan, Vijay Bharadwaj, Andrew Park, Jeff Da, MohammadHossein Rezaei, Bing Liu, Brad Kenstler, and Yunzhong He. SWE atlas: Benchmarking coding agents beyond issue resolution. In *Deep Learning for Code: Towards Human-Centered Coding Agents*, 2026. URL https://openreview.net/forum?id=HN6TySnI0i.

Michael Saxon, Ari Holtzman, Peter West, William Yang Wang, and Naomi Saphra. Benchmarks as microscopes: A call for model metrology. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=bttKwCZDkm.

Judy Hanwen Shen and Alex Tamkin. How ai impacts skill formation, 2026. URL https://arxiv.org/abs/2601.20245.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2Code: Benchmarking multimodal code generation for automated front-end engineering. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 3956–3974, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.199. URL https://aclanthology.org/2025.naacl-long.199/.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Akshit Sinha, Arvindh Arun, Shashwat Goel, Steffen Staab, and Jonas Geiping. The illusion of diminishing returns: Measuring long horizon execution in LLMs. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=3lm8lWYxiq.

Zafir Stojanovski, Oliver Stanley, Joe Sharratt, Richard Jones, Abdulhakeem Adefioye, Jean Kaddour, and Andreas Köpf. Reasoning gym: Reasoning environments for reinforcement learning with verifiable rewards. *Advances in Neural Information Processing Systems*, 38, 2026.

Ronen Tamari, Kyle Richardson, Noam Kahlon, Aviad Sar-shalom, Nelson F. Liu, Reut Tsarfaty, and Dafna Shahaf. Dyna-bAbI: unlocking bAbI's potential with dynamic synthetic benchmarking. In Vivi Nastase, Ellie Pavlick, Mohammad Taher Pilehvar, Jose Camacho-Collados, and Alessandro Raganato (eds.), *Proceedings of the 11th Joint Conference on Lexical and Computational Semantics*, pp. 101–122, Seattle, Washington, July 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.starsem-1.9. URL https://aclanthology.org/2022.starsem-1.9/.

Robert F. Tate. Correlation between a discrete and a continuous variable: Point-biserial correlation. *The Annals of Mathematical Statistics*, 25(3):603–607, 1954. doi: 10.1214/aoms/1177728730.

Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. Can language models solve graph problems in natural language? *Advances in Neural Information Processing Systems*, 36:30840–30861, 2023.

Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yuan Liu, Thiago SFX Teixeira, Diyi Yang, Ke Wang, et al. Equibench: Benchmarking large language models' reasoning about program semantics via equivalence checking. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 33856–33869, 2025.

Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M. Rush, Bart Van Merriënboer, Armand Joulin, and Tomas Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016. URL https://arxiv.org/abs/1502.05698. Poster.

Norman Wilde, Michelle Buckellew, Henry Page, and Václav Rajlich. A case study of feature location in unstructured legacy fortran code. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, pp. 68–76. IEEE, 2001.

xAI. Grok 4.1 model card. https://data.x.ai/2025-11-17-grok-4-1-model-card.pdf, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026a. URL https://openreview.net/forum?id=63iVrXc8cC.

John Yang, Kilian Lieret, Jeffrey Jian Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Programbench: Can language models rebuild programs from scratch? In *Deep Learning for Code: Towards Human-Centered Coding Agents*, 2026b. URL https://openreview.net/forum?id=2AJ8PBgp4l.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pp. 476–486. ACM, 2018. doi: https://doi.org/10.1145/3196398.3196408.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. Large language models meet NL2Code: A survey. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7443–7464, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.411. URL https://aclanthology.org/2023.acl-long.411/.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, et al. Multi-swe-bench: A multilingual benchmark for issue resolving. *Advances in Neural Information Processing Systems*, 38, 2026.

Alex Zhang and Omar Khattab. Language model harnesses are compositional generalizers. July 2026. URL https://alexzhang13.github.io/blog/2026/harness/.

Taolin Zhang, Dongyang Li, Qizhou Chen, Chengyu Wang, and Xiaofeng He. BELLE: A bi-level multi-agent reasoning framework for multi-hop question answering. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4184–4202, Vienna, Austria, July 2025a. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.211. URL https://aclanthology.org/2025.acl-long.211/.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, et al. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025b.

Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model-based agents. *ACM Transactions on Information Systems*, 43(6):1–47, 2025c.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=MMwaQEVsAg.

Kan Zhu, Mathew Jacob, Chenxi Ma, Yi Pan, Stephanie Wang, Arvind Krishnamurthy, and Baris Kasikci. Tracelab: Characterizing coding agent workloads for llm serving. *arXiv preprint arXiv:2606.30560*, 2026.

Yiqi Zhu, Apurva Gandhi, and Graham Neubig. Training versatile coding agents in synthetic environments, 2025. URL https://arxiv.org/abs/2512.12216.

A Appendix

A.1 DAG Task Sampling Procedure

We now describe our sampling procedures for the DAG-based tasks in §2. Given N vertices, we sample a height $h \sim \mathrm{Uniform}(0.45, 0.55)$ and set the topological level count to $\mathtt{round}(N^h)$. We set vertices across levels via Poisson draws, where mean λ is the number of vertices left to assign divided by the number of levels left. For each non-root vertex v_j, we sample an incoming edge (v_i, v_j) from an earlier level with an offset $\Delta \sim \mathrm{Geom}(0.9)$, biasing edges to adjacent levels but allowing skips (i.e., $i + \Delta = j$). Figure A.1 shows example DAGs and Algorithm 1 summarizes the sampling procedure.

A.2 Example Programs

We present examples of math, string, and array programs for the Add Parameter task (§2.1) in Code Block A.1, A.2, and A.3, respectively. Code Block A.4 shows the Collatz-inspired `check()` function (Lagarias, 1985) we use for runtime resolution, and Code Block A.5 shows an example of two `Calculator` classes in the code merging task (§5); the new line separates the synonymous from divergent code, but this formatting is stripped during evaluation.

For the instruction following task, we use the following bank of rules:

- Add a number: `new_var = var + rand`.
- Subtract a number: `new_var = var - rand`.
- Multiply by a number: `new_var = var * rand`.
- Divide by a number: `new_var = var / rand`.
- Add a character prefix: `new_var = rand + var`.
- Add a character suffix: `new_var = var + rand`.
- Replace a character: `new_var = var.replace(rand, rand2)`.
- Surround a string with characters: `new_var = rand + var + rand2`.
- Assert that a value is unequal to a sampled value: `assert var != rand`.
- Assert an upper bound: `assert var < rand`.
- Assert a lower bound: `assert var > rand`.
- Print a value with a prefix: `print(prefix, var)`.
- Print a value with a suffix: `print(var, suffix)`.
- Add an exact comment of the form `# label: var`.

Sampling values for `rand`, `prefix`, and `label` allows us to scale the total number of unique instructions.

A.3 Model Implementation Details

We run all LLMs via litellm[2] using the following models:

- `gpt-5.3-codex_2026-02-24`
- `gpt-5_2025-08-07`
- `gpt-5.4-mini_2026-03-17`
- `gpt-5.5_2026-04-24`
- `grok-4-1-fast-reasoning_1`
- `DeepSeek-V3.2_1`

[2] https://github.com/BerriAI/litellm

- `gpt-oss-120b_1`
- `Mistral-Large-3_1`

All models use the Completions API, except Codex (Responses API). We enable reasoning for OpenAI and Grok models—which is `medium` by default, except for Codex and GPT-5.5—where we set it manually. We use default parameters otherwise to simulate out-of-the-box behavior (Garces Arias et al., 2025). We prompt models until they generate a parseable output codebase—ensuring failures stem from weaknesses in coding abilities rather than output formatting—which typically takes no more than ten tries.

We evaluate whether LLMs should perform full codebase generation or diff patch generation in CABRA by empirically testing both. We run a subset of LLMs in §3.2 on a task that combines function traversal and search—scaling $\mathcal{C}_{\text{init}}$ and $\mathcal{C}_{\text{edit}}$ with $n \in \{5, 10, 25, 50, 100, 200\}$. Full codebase generation yields higher accuracy than patch generation overall, and for all models except GPT OSS 120B (Figure A.2). Thus, we use it as the configuration in all experiments. This replicates results of Cheng et al. (2026), who find structured diff formats can lead to unstable LLM responses.

For the agent harness, we script Copilot `v1.0.76` (run in July 2026). We script the model using the following command:

```
copilot \
  --model m \
  -C c \
  -p p \
  --mode autopilot \
  --silent \
  --allow-all-tools \
  --session-id=i \
  --no-ask-user \
  --no-remote
```

where c is the path to where we write the CABRA files, p is the problem statement, and i is a randomly-generated UUID so we can access the session later. The other commands remove output streaming, ensure Copilot does not ask clarification questions, give access to all tools, and keep execution local. The models m are one of the following endpoints:

- `claude-sonnet-4.6`
- `claude-opus-4.7`
- `gpt-5.5`
- `gpt-5.4-mini`
- `gemini-3.1-pro-preview`
- `gemini-3.5-flash`

We retry until Copilot returns a parseable output codebase, which typically takes no more than three tries at 10 minutes per try (with the exception of the challenging logical equivalence task from §5.1, where runtimes between 10 and 30 minutes are common).

A.4 CLASSIFIER IMPLEMENTATION

We classify tool calls using `Qwen/Qwen3-4B-Instruct-2507` (Yang et al., 2025) with 0 temperature, a context length of 10000 tokens, and `sglang` for efficiency. Along with the six labels in Table 2, we also classify tools as `summary` (generates a summary of output changes) and `plan` (creates a TODO list); we find these are rare, so we merge them with `other` for analyses. In rare cases, Qwen output an invalid label; these instances were routed to `gpt-4.1_2025-04-14` as a slower but more powerful fallback. To validate the classifier, we sample 20 examples for each of the Qwen-predicted labels—yielding 160 in total. One author then blindly labels each tool call. We repeat this process for the initial CABRA tasks (§4.2), SWE-Bench tasks (§4.3), and merge codebases task (§5.2)—which achieved agreement of 94%, 89%, and 94%, respectively.

A.5 Further CABRA Accuracy Across Task Types and Models

We decompose results from Figure 2 across two dimensions as a robustness check: 1) the five task types; and 2) the three program types (math, string, array; all described in §2.1). We present the task breakdowns for (1) across LLMs and agents in Figure A.3 and Figure A.4, respectively, and program breakdowns for (2) across LLMs and agents in Figure A.5 and Figure A.6, respectively. Trends persist even in these decompositions: higher-complexity tasks tend to challenge LLMs, but agent accuracy remains nearly perfect.

Finally, we also run Gemini-3.5 Flash and Gemini-3.1 Pro as reasoning LLMs without tools to mirror their use in the Copilot Agent harness in §4.1. Due to cost constraints, we only run these models on 15 programs (five math/string/array); in total these experiments cost $952 USD. When controlling for model (GPT and Gemini), the trend remains consistent across most axes: LLM accuracy degrades more with larger N compared to agents (Figure A.8). This further backs our claim that tool use is what allows agents to excel in CABRA tasks, not differences in underlying model abilities.

A.6 Further Tool Call Analysis

We now provide more analyses on agent tool calls. First, we report the names of tool calls agents use in Table A.3. The three most common tool types are consistently `edit`, `bash`, and `view`, which are also popular tool calls in Codex and Copilot (Zhu et al., 2026).

We also decompose the analysis from Figure 3 across the six models in our agent harness, showing the output tokens each model spends on different tool call types. Our analysis is mostly consistent with our three main trends: 1) function traversal and function search often elicit more ANALYZE tokens not EDIT; 2) runtime resolution elicits near-constant tokens of all tool call types; and 3) instruction following elicits more EDIT tokens—but often times more ANALYZE tokens as well.

A.7 SWE-Bench Analysis with Per-Model Breakdown

We now provide model-specific analyses for the SWE-Bench tool call breakdowns and correlations in §4.3. We use the following model endpoints in the format `harness/model`:

- `mini-v2.0.0/claude-4-5-opus-high`
- `mini-v2.0.0/claude-4-5-sonnet-high`
- `mini-v2.0.0/gpt-5-2-high`
- `mini-v2.0.0/gpt-5-mini`
- `mini-v2.0.0/gemini-3-flash-high`
- `mini-v1.15.0/gemini-3-pro-preview-20251118`

All models use the SWE-Agent Mini harness `mini-v2.0.0` (Yang et al., 2024), except Gemini-3 Pro, as results on this harness are corrupted (the model fails every task).

We present a per-model breakdown of Figure 4 (left) in Figure A.10, showing that our trend persists across models: they do not only spend more tokens on EDIT operations. To account for model-specific differences in our correlational analysis (Figure 4, right), we fit a separate logistic regression for each metric m:

$$\text{logit}[\Pr(\text{score}_i = 1 \mid \text{model}_i, m_i)] = \beta_0 + C(\text{model}_i) + \beta_m m_i, \tag{1}$$

where $score \in 0, 1$ denotes whether the agent completed the task, and $C(model)$ controls for agent baseline success. For each metric, we compare the augmented regression in Equation 1 with a baseline regression using just $C(model)$. We measure improvement in model fit via the likelihood-ratio (LR) statistic:

$$2\left(\ell_{\text{augmented}} - \ell_{\text{baseline}}\right), \tag{2}$$

where $\ell_{\text{augmented}}$ and ℓ_{baseline} are the maximized log-likelihoods of the two models. Larger LR indicates m improves fit beyond model identity alone. Edit volume is not the most predictive signal for coding agent accuracy (Table A.4)—mirroring the trend in Figure 4 (right). Since SEARCH is also arguably a tool call for understanding, we also run regressions to predict agent accuracy from tool call tokens and count when combining READ, SEARCH, and ANALYZE. These metrics are still more predictive of agent accuracy than edit volume.

A.8 STUDYING THE IMPACT OF AGENT COLLABORATION ON OUR EVALUATIONS

When reviewing our agent traces, we found cases where our agents attempted to get information from other parallel runs (recalling the OpenAI security incident reported with Hugging Face (OpenAI, 2026)). In particular, we found that some agents used the Copilot CLI session store database to search for similar sessions (although this was very rare; only 72 / ∼43k traces). Others explicitly searched the local filesystem for other agents' `solution.py`. Attempts to read these files were permission-denied; however, all agents shared access to `/tmp`, which some used for intermediate tasks. One example of an agent looking for other solutions (this tool call was permission-denied):

```
{
  "command": "find /input_dir/ -name \"solution.py\" -type f -readable 2>/dev/null | head -n
     20",
  "description": "Find readable solution.py files"
}
```

To ensure this does not impact our results, we first examine agent accuracy (Figure A.11) and tool use (Figure A.12) after filtering traces that could have potentially accessed other runs (i.e., accessed the `/tmp` folder with other agent solutions or used the `session_store_sql` tool). This filters just 11% of our traces (mostly innocuous uses of `/tmp`), and trends are consistent with our main results.

Next, we use a Docker image that exposes only the input codebase to better sandbox the agent (preventing any filesystem-based cross-trace leak). We set up the Docker image to be minimal but provide tools the agent might find useful:

```
FROM node:22-bookworm-slim

RUN apt-get update \
    && apt-get install --yes --no-install-recommends \
        bash \
        build-essential \
        ca-certificates \
        coreutils \
        curl \
        diffutils \
        file \
        findutils \
        gawk \
        grep \
        jq \
        less \
        patch \
        procps \
        python3 \
        python3-pip \
        python3-venv \
        ripgrep \
        sed \
        tree \
        wget \
    && npm install --global "@github/copilot@1.0.81" \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/* \
    && useradd --create-home --uid 10001 agent

USER 10001:10001
```

```
WORKDIR /workspace

RUN copilot --version
```

Using this file, we script Copilot `v1.0.81` (run in September 2026) to operate inside the Docker image while also disabling the `session_store_sql` tool to prevent unecessary searches:

```
docker create \
  --name <container_name> \
  --label cabra.agent=copilot \
  --network bridge \
  --init \
  --mount type=bind,source=<workdir>,target=/workspace \
  --workdir /workspace \
  --user 10001:10001 \
  --cap-drop ALL \
  --security-opt no-new-privileges \
  --memory 32g \
  --env HOME=/home/agent \
  --env COPILOT_GITHUB_TOKEN \
  <docker_image> \
  copilot \
  --model <model_name> \
  -C /workspace \
  -p <prompt> \
  --mode autopilot \
  --silent \
  --allow-all-tools \
  --excluded-tools=session_store_sql \
  --deny-tool=session_store_sql \
  --session-id=<session_id> \
  --no-ask-user \
  --no-remote \
  --no-remote-export
```

Using this approach, we run $n = 20$ examples using the math version of our codebases across all tasks, with Opus-4.7, GPT-5.5, GPT-5.4 Mini, and Gemini-3.5 Flash (cost breakdown in Table A.2). We present accuracy in Figure A.13 and tool use metrics in Figure A.14 which are also consistent with our main results: agent accuracy mostly degrades with task size for our understanding-intensive merge codebases task, and usage of READ and ANALYZE tools scale beyond just EDIT with task size—with ANALYZE growing especially in the merge codebases task.

Overall, as (1) filtering traces with potential cross-agent communication; and (2) re-running agents with problematic tools disabled in contained Docker images both did not impact our conclusions, we are confident that the main trends we present in this paper are unaffected by cross-agent communication.

A.9 EDITING TASKS WITHOUT MUCH UNDERSTANDING

Our evaluations with CABRA in §4 reveal two task types where scaling complexity changes behavior in agent tool calls: 1) UNDERSTAND tool calls increase while EDIT tool calls remain stable (i.e., function search and function traversal); and 2) Both UNDERSTAND and EDIT tool calls increase (i.e., instruction following). To confirm that understanding and editing are separate constructs, we now design a task that scales editing requirements with complexity without large changes in understanding requirements.

We study two different tasks:

1. A comment copying task which uses the CABRA tasks in §2 similar to instruction-following, but the model is asked to add a comment with n words (Prompt A.11)

2. A variable renaming task where a model receives a simple calculation with 200 operations in a block of code (Code A.6) and must rename n variables as defined in the prompt (Prompt A.12).

Evaluations on this task reveal that as n increases: 1) LLM and agent accuracy remains consistent for comment copying, but LLM accuracy drops in variable renaming with n (Figure A.15); and 2) EDIT tool calls scale in both tasks without large changes in UNDERSTAND tool calls, especially compared to our first instruction-following task in CABRA (Figure A.16). The number of EDIT tool calls does not scale for any task, but one can arbitrarily make a require more EDIT tool calls by forcing models to make smaller edits at a time.

Overall, this further backs our argument that understanding and editing are distinct coding agent abilities that can be uniquely targeted and measured in evaluations.

A.10 ADDITIONAL RESULTS ON EXTRACTING EQUIVALENT CODE

We first show both LLM and coding agent accuracy on the merge codebases task (§5). Both model types degrade in accuracy with large n, but many LLMs fail to ever clear 0.5 accuracy—confirming the task's difficulty.

We also present the follow-up analyses in Figure 6 segmented across our six agents. The add-one-in ablation accuracy results are model-specific, but 3/6 models struggle with ones related to multi-line differences (Figure A.22, left). In the leave-one-out ablations, results are consistent: all models improve most when variables are not renamed (Figure A.22, right), which could suggest over-reliance on surface-level naming when inferring code behavior. Since Opus-4.7 saturates the task at $n = 50$, we repeat this experiment for $n = 200$ with Opus alone in Figure A.21, which shows similar trends: the model struggles with multi-line differences and variable renaming.

Finally, the trend for locating increasing number of different lines is consistent across models: all agents struggle more when they must identify more one-line differences (Figure A.19). This also suggests that difficulty arises from navigating the search space of all possible solutions to find the most efficient one (i.e., least lines of code in child classes). To validate this, we run the same experiment as Figure 5b but instruct models that there is only a single one-line difference; as expected, agent accuracy is much higher (Figure A.20).

A.11 PROMPTS

We provide prompts with our problem statements for Remove Dead Code (Prompt A.1), Add Parameter (Prompt A.2), Add Return Value (Prompt A.3), Cache Function (Prompt A.4), Don't Repeat Yourself (Prompt A.5), and Merge Codebases (Prompt A.6) tasks. We also show how we minimally tailor these prompts for the string (Prompt A.7) and array (Prompt A.8) versions of our code. Prompt A.9 has the prompt for tasks involving scaling runtime complexity, and Prompts A.10 and A.11 for instruction following tasks with and without scaling understanding tool calls, respectively.

For §3.2, Prompt A.13 and A.14 are used for evaluating LLMs when they generate the full codebase or diff patches, respectively. Prompts A.15 and A.16 are the classification prompts of model tool calls on our synthetic CABRA tasks and SWE-Bench, respectively.

Algorithm 1 Sampling Procedure for Directed Acyclic Call Graphs (§2.1)

Require: Number of vertices N
Ensure: A directed acyclic graph $G = (V, E)$

Step 1: Sample the number of topological levels

Sample $h \sim \mathrm{Uniform}(0.45, 0.55)$
$L \leftarrow \max\{1, \mathrm{round}(N^h)\}$

Step 2: Distribute vertices across levels

$V_{\text{left}} \leftarrow N$
$V \leftarrow \varnothing$, $E \leftarrow \varnothing$
for $\ell \leftarrow 1$ to L **do**
  $L_{\text{left}} \leftarrow L - \ell + 1$
  $\lambda_\ell \leftarrow V_{\text{left}} / L_{\text{left}}$ ▷ Poisson mean based on nodes left
  if $\ell = L$ **then**
    $n_\ell \leftarrow V_{\text{left}}$ ▷ Assign all remaining vertices
  else
    $\widetilde{n}_\ell \sim \mathrm{Poisson}(\lambda_\ell)$
    $n_\ell \leftarrow \min\{\max\{1, \widetilde{n}_\ell\}, V_{\text{left}} - (L_{\text{left}} - 1)\}$ ▷ At least one vertex per level
  end if
  Create level V_ℓ containing n_ℓ new vertices
  $V \leftarrow V \cup V_\ell$
  $V_{\text{left}} \leftarrow V_{\text{left}} - n_\ell$
end for

Step 3: Sample one incoming edge per non-root vertex

for $\ell \leftarrow 2$ to L **do**
  for all $v \in V_\ell$ **do**
    Sample $\widetilde{\Delta} \sim \mathrm{Geom}(0.9)$ ▷ Prefer nearby levels
    $\Delta \leftarrow \min\{\widetilde{\Delta}, \ell - 1\}$ ▷ Clip to an existing earlier level
    Sample $u \sim \mathrm{Uniform}(V_{\ell - \Delta})$ ▷ Choose a parent within that level
    $E \leftarrow E \cup \{(u, v)\}$
  end for
end for
return $G = (V, E)$

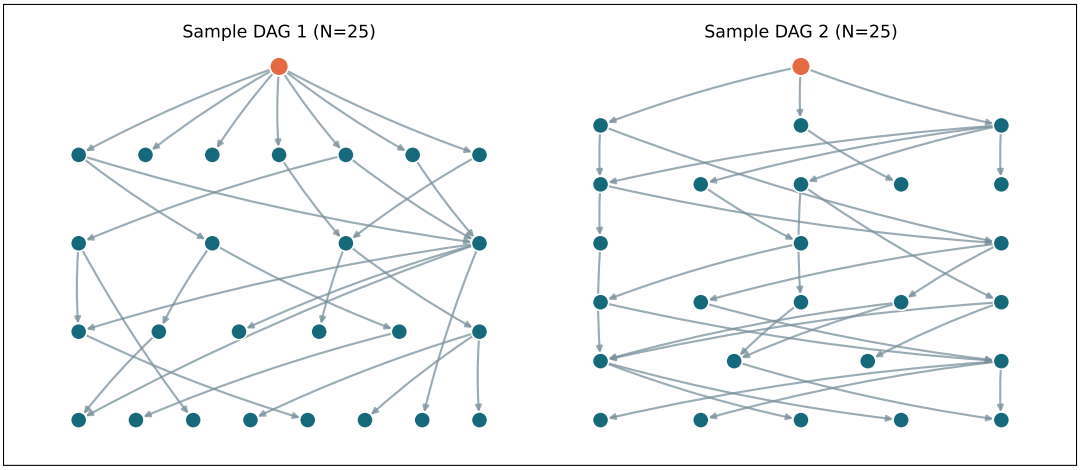


Figure A.1: Examples of $|V| = 25$ DAGs sampled from the procedure described in §2.1.

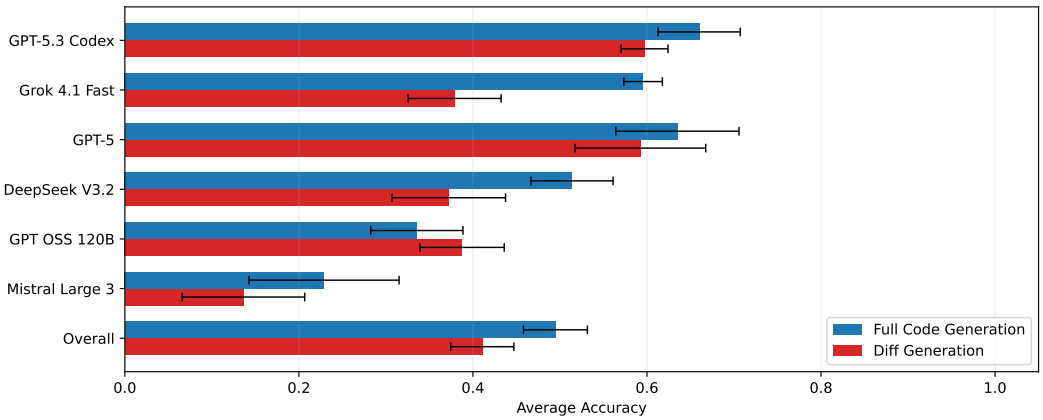


Figure A.2: Comparison of LLM task completion when editing code via full codebase regeneration versus structured diff patches when both $\mathcal{C}_{\text{init}}$ and $\mathcal{C}_{\text{edit}}$ sweep $n \in \{5, 10, 25, 50, 100, 200\}$. The former yields higher accuracy overall, so we use this format for remaining experiments.

Table A.1: Input and output tokens (in millions) used by our final evaluation runs by model and task family. We calculate costs in USD from LLM API providers. Cached tokens are included in the "In" column but accounted for separately in the USD cost. Total cost: $28,262.

Model	Traversal		Instruction		Search		Runtime		Merge		Overall		
	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	Cost (USD)
LLMs													
DeepSeek-V3.2	7.0	17.4	5.9	22.1	7.1	18.8	5.1	22.2	1.0	3.6	26.2	84.1	$41
Mistral-Large-3	6.9	7.6	6.0	14.6	6.9	8.0	5.1	5.3	1.0	1.7	26.0	37.2	$69
gpt-5.3-codex	6.6	9.3	5.8	11.8	6.7	9.2	5.0	12.4	1.0	1.7	25.0	44.4	$665
gpt-5.4-mini	6.6	19.0	5.8	24.1	6.7	16.4	5.0	18.9	1.0	2.6	25.0	80.9	$383
gpt-5.54	6.6	8.5	5.8	13.6	6.7	8.6	5.0	11.3	1.0	1.8	25.0	43.9	$1,440
gpt-5	6.6	15.1	5.8	19.8	6.7	15.8	5.0	21.7	1.0	1.8	25.0	74.2	$773
gpt-oss-120b	6.7	11.9	5.9	14.0	6.8	14.1	5.1	13.4	1.0	1.2	25.5	54.6	$10
grok-4-1-fast-reasoning	6.6	2.8	5.7	4.3	6.6	4.4	5.0	2.9	1.0	0.4	25.0	14.9	$12
Agents													
claude-opus-4.7	1874.4	39.1	1229.6	48.9	2527.1	43.6	1452.0	38.9	468.1	15.6	7551.1	186.0	$10,407
claude-sonnet-4.6	1309.3	39.1	1078.5	57.7	2055.5	38.5	1108.4	35.2	308.2	20.5	5859.9	191.1	$5,822
gemini-3.1-pro-preview	711.8	5.4	710.8	12.8	628.5	4.6	450.7	5.8	598.4	5.6	3100.2	34.2	$1,950
gemini-3.5-flash	1701.9	9.3	1675.6	19.7	1552.7	9.3	1141.2	9.0	1033.8	6.1	7105.1	53.4	$2,775
gpt-5.4-mini	849.2	23.2	666.3	27.7	838.6	20.4	469.9	17.0	53.1	2.5	2877.0	90.7	$873
gpt-5.5	359.9	8.4	304.7	13.3	376.7	8.7	297.4	7.3	83.1	2.8	1421.7	40.5	$3,042

Table A.2: Sandboxed runs cost breakdown using public pricing data (as of September 2026). "In" includes cached input, but those tokens are accounted for separately in cost estimates. Total cost: $3,624.

Model	Traversal		Instruction		Search		Runtime		Merge		Overall		
	In	Out	In	Out	In	Out	In	Out	In	Out	In	Out	Cost (USD)
Agents													
claude-opus-4.7	318.9	4.6	129.2	5.8	415.8	4.8	342.1	4.7	82.8	2.5	1288.9	22.4	$1,532
gemini-3.5-flash	630.3	3.0	517.3	6.3	569.6	3.1	446.0	2.7	318.2	2.6	2481.5	17.7	$896
gpt-5.4-mini	335.6	7.9	264.4	9.8	332.6	6.8	198.9	5.7	27.2	0.8	1158.6	31.0	$299
gpt-5.5	142.6	2.9	119.1	4.7	153.5	2.9	125.3	2.5	44.3	1.2	584.7	14.2	$924

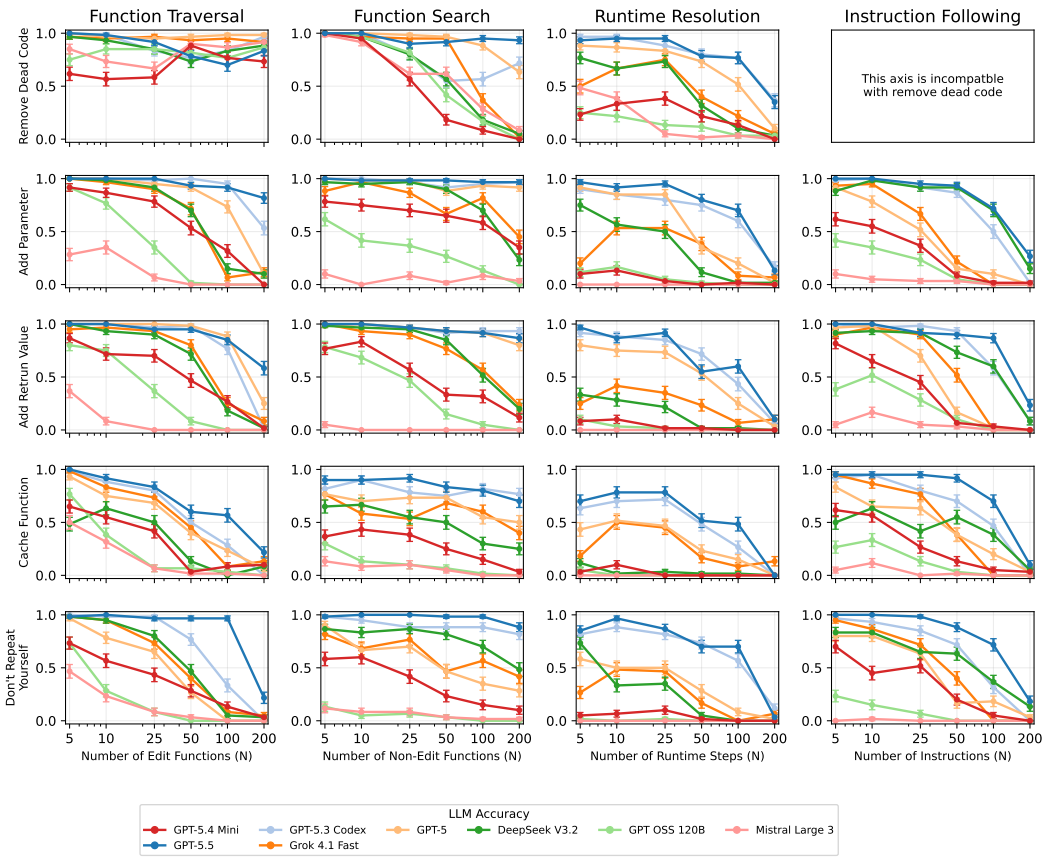


Figure A.3: Evaluation of CABRA tasks in Figure 2 with **LLMs**, segmented by the task types in §2.1. The trend persists: higher complexity tasks tend to challenge LLMs.

Table A.3: The ten most frequent tool call names agents call across each complexity axis in §2.2. We report the raw number (in thousands) and the proportion of all tools. The model also called `web_search` in 48 out of 700k+ cases, which suggests cheating (Yang et al., 2026b, e.g., the model Googling our prompt verbatim), but this is rare enough to not impact results.

Function Traversal	Function Search	Instruction Following	Runtime Resolution
`bash` (75.7k; 37.7%)	`edit` (82.6k; 35.1%)	`bash` (43.8k; 33.8%)	`edit` (58.7k; 35.9%)
`edit` (45.9k; 22.9%)	`bash` (62.9k; 26.7%)	`view` (24.5k; 18.9%)	`bash` (44.4k; 27.2%)
`view` (31.7k; 15.8%)	`view` (36.9k; 15.6%)	`edit` (21.9k; 16.9%)	`view` (22.5k; 13.8%)
`complete` (10.8k; 5.4%)	`rg` (10.9k; 4.6%)	`rg` (8.8k; 6.8%)	`complete` (10.8k; 6.6%)
`create` (8.9k; 4.4%)	`complete` (10.8k; 4.6%)	`complete` (8.6k; 6.7%)	`apply_patch` (7.2k; 4.4%)
`rg` (8.4k; 4.2%)	`apply_patch` (9.7k; 4.1%)	`glob` (6.6k; 5.1%)	`glob` (6.5k; 4.0%)
`glob` (7.3k; 3.6%)	`create` (8.0k; 3.4%)	`apply_patch` (5.7k; 4.4%)	`rg` (5.1k; 3.1%)
`apply_patch` (5.9k; 2.9%)	`glob` (7.3k; 3.1%)	`create` (3.6k; 2.7%)	`sql` (3.6k; 2.2%)
`grep` (4.2k; 2.1%)	`grep` (4.6k; 2.0%)	`sql` (3.4k; 2.6%)	`grep` (2.8k; 1.7%)
`sql` (2.0k; 1.0%)	`sql` (1.6k; 0.7%)	`grep` (1.6k; 1.2%)	`create` (1.9k; 1.2%)

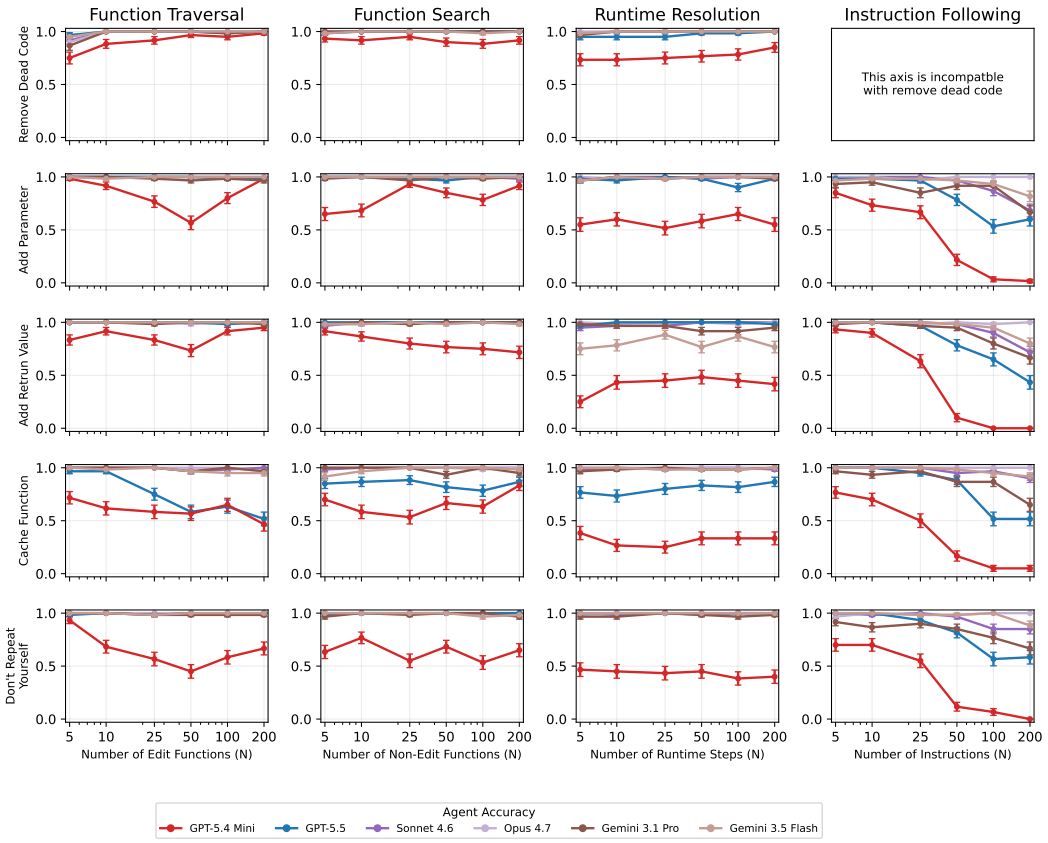


Figure A.4: Evaluation of CABRA tasks in Figure 2 with **agents**, segmented by the task types in §2.1. The trend persists: the strongest agents often have near-perfect scores on larger tasks.

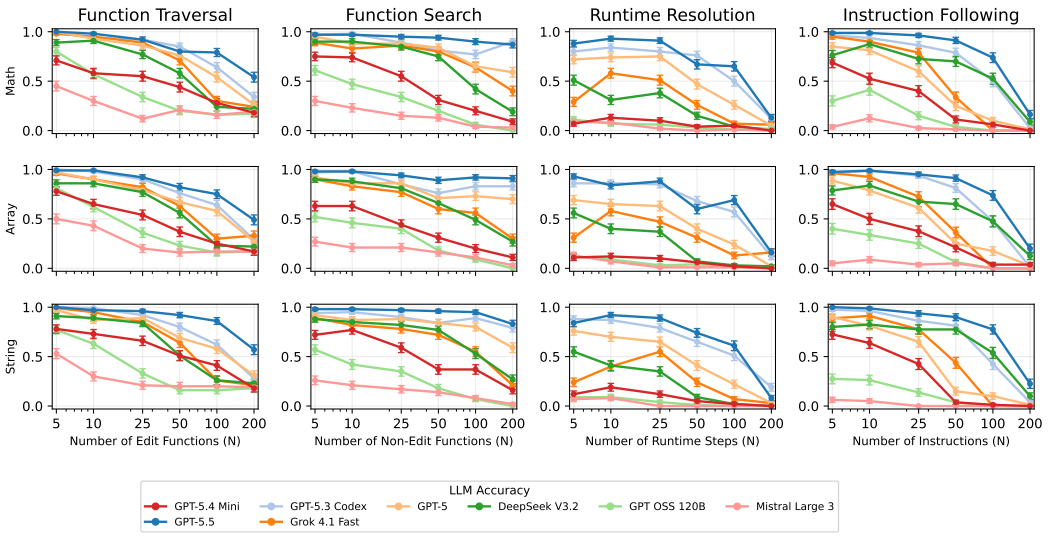


Figure A.5: Evaluation of CABRA tasks in Figure 2 with **LLMs**, segmented by the three program types (math, string, array) in §2.1. The trend persists: higher complexity tasks tend to challenge LLMs.

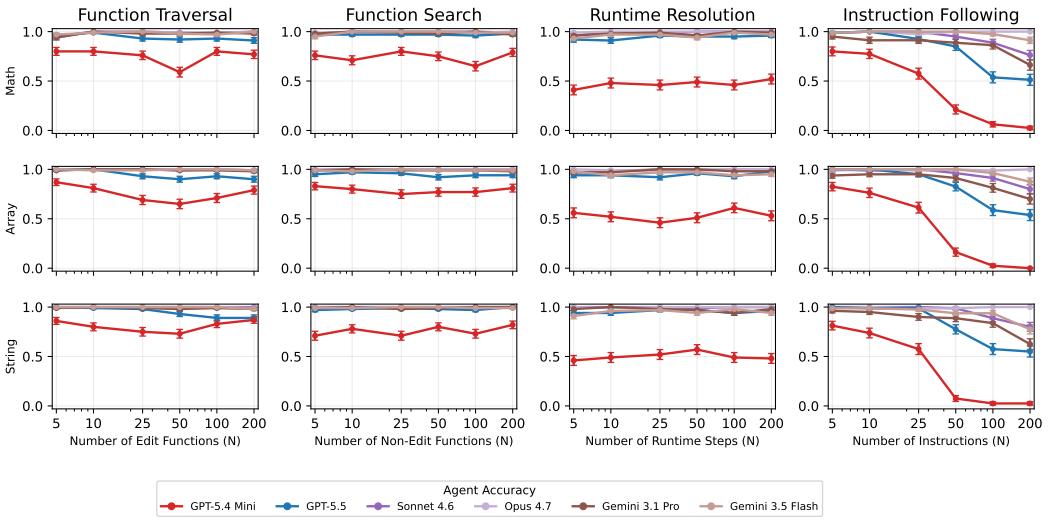


Figure A.6: Evaluation of CABRA tasks in Figure 2 with **agents**, segmented by the task types in §2.1. The trend persists: the strongest agents often have near-perfect scores on larger tasks.

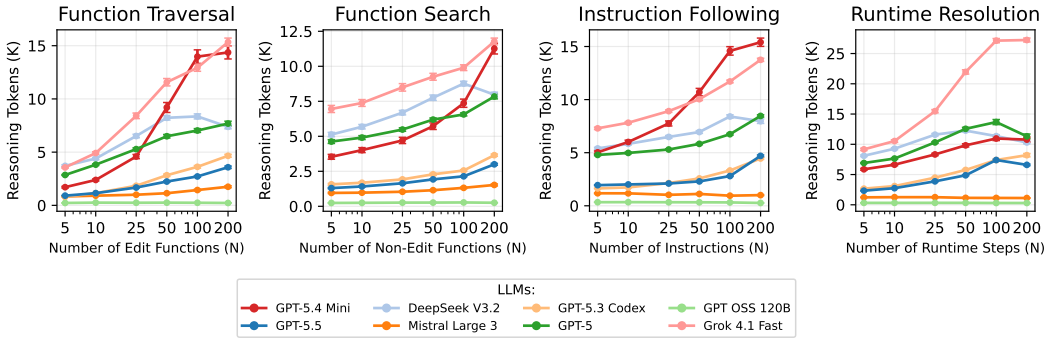


Figure A.7: Number of reasoning tokens LLMs use for our four axes of CABRA in §2.2: the sum of tokens in `<thought></thought>` blocks via tiktoken plus the number of reasoning tokens reported by the model provider (in thousands). Reasoning token count typically increases with N.

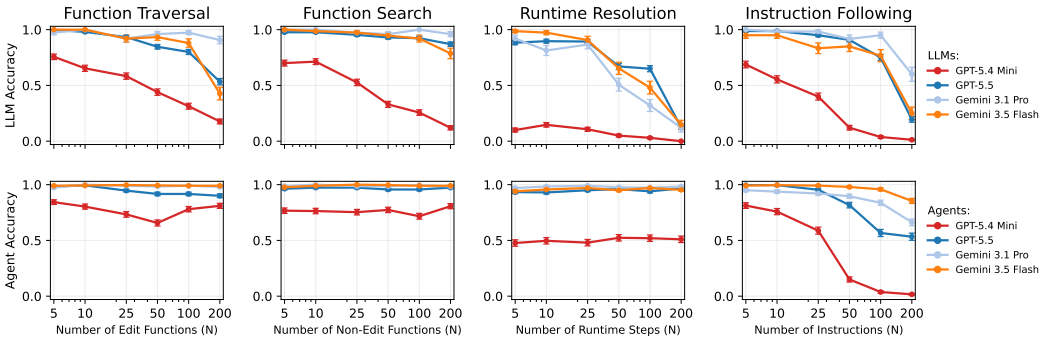


Figure A.8: Evaluation of reasoning LLM and agent accuracy when controlling for baseline models: GPT-5.5, GPT-5.4 Mini, Gemini-3.1 Pro, and Gemini-3.5 Flash. The trend is consistent with §4.1: LLM accuracy degrades more when N scales versus agent accuracy.

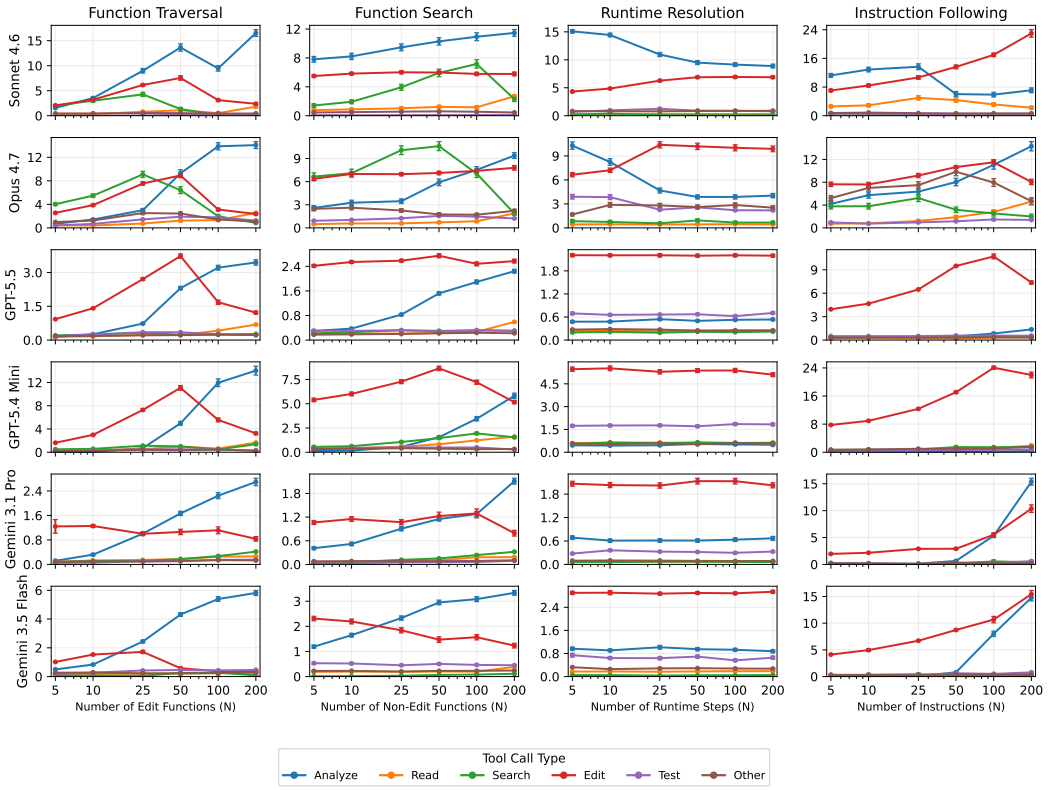


Figure A.9: Analysis of model-specific **output tokens (in thousands) used for tool calls**. Three trends emerge: most models spend more tokens on ANALYZE in function traversal and search, no tool call type scales in runtime resolution, and instruction-following always requires more EDIT tokens (and often more ANALYZE tokens).

Table A.4: Logistic regression to predict agent's task completion using each tool call metric in §4.3, controlling for agent. Edit volume is not the most predictive signal for task accuracy.

Metric	Coefficient	Odds Ratio	p-value	LR Statistic (↓)
All Tokens	-0.788	0.455	< 0.001	224.31
UNDERSTAND+SEARCH$_{\text{tok}}$	-0.659	0.517	< 0.001	189.52
UNDERSTAND$_{\text{num}}$	-0.589	0.555	< 0.001	183.63
All Tools	-0.629	0.533	< 0.001	171.46
UNDERSTAND$_{\text{tok}}$	-0.564	0.569	< 0.001	156.47
UNDERSTAND+SEARCH$_{\text{num}}$	-0.556	0.574	< 0.001	154.33
READ$_{\text{num}}$	-0.492	0.611	< 0.001	133.21
EDIT$_{\text{tok}}$	-0.468	0.626	< 0.001	115.01
ANALYZE$_{\text{num}}$	-0.430	0.650	< 0.001	104.47
SEARCH$_{\text{tok}}$	-0.449	0.638	< 0.001	95.57
READ$_{\text{tok}}$	-0.437	0.646	< 0.001	94.45
ANALYZE$_{\text{tok}}$	-0.403	0.669	< 0.001	94.14
Lines Edited	-0.457	0.633	< 0.001	85.10
TEST$_{\text{tok}}$	-0.492	0.611	< 0.001	78.68
EDIT$_{\text{num}}$	-0.361	0.697	< 0.001	77.11
TEST$_{\text{num}}$	-0.497	0.608	< 0.001	67.62
SEARCH$_{\text{num}}$	-0.364	0.695	< 0.001	65.54
OTHER$_{\text{tok}}$	-0.170	0.843	< 0.001	15.88
Files Edited	-0.154	0.857	< 0.001	14.89
OTHER$_{\text{num}}$	-0.106	0.899	0.016	5.67

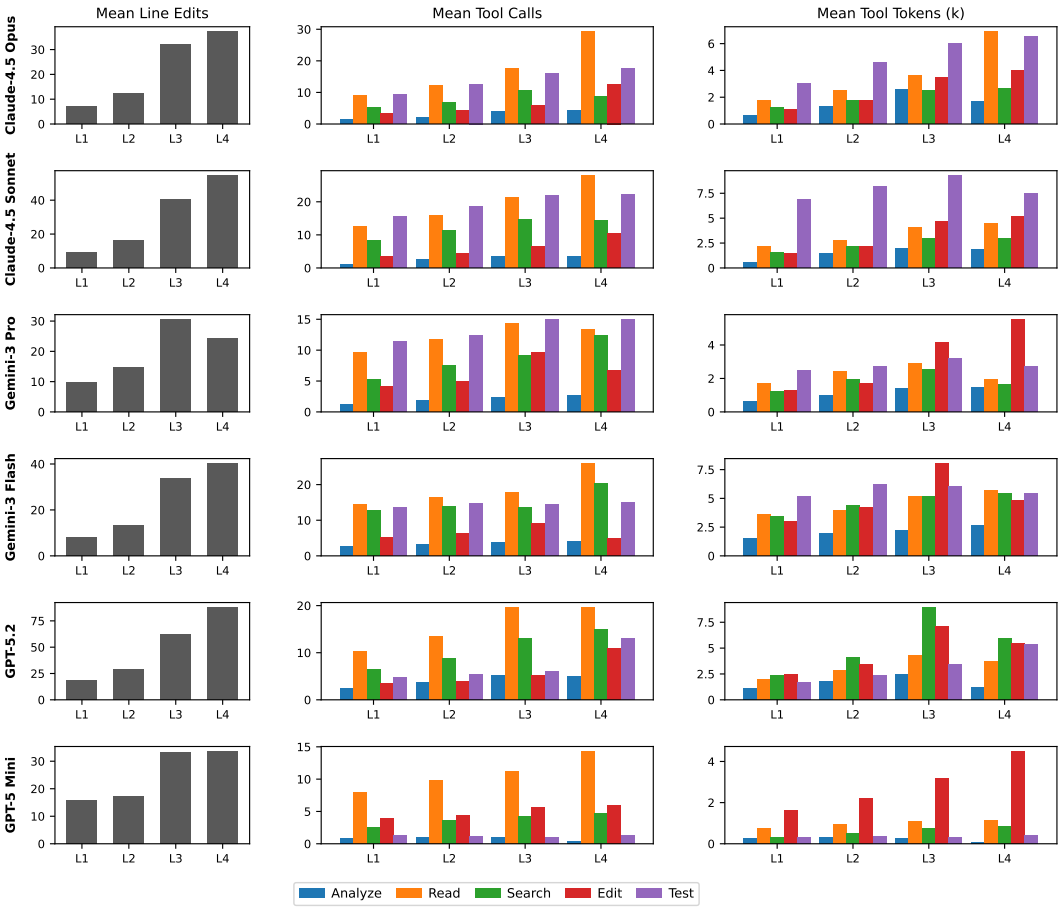


Figure A.10: Number of edits, tool calls, and tokens spent on tools calls in Figure 4 (left) segmented by agent. As human-annotated difficulty levels rise ($L1 \rightarrow ... \rightarrow L4$), agents do not only edit more code—they also read, analyze, or test more.

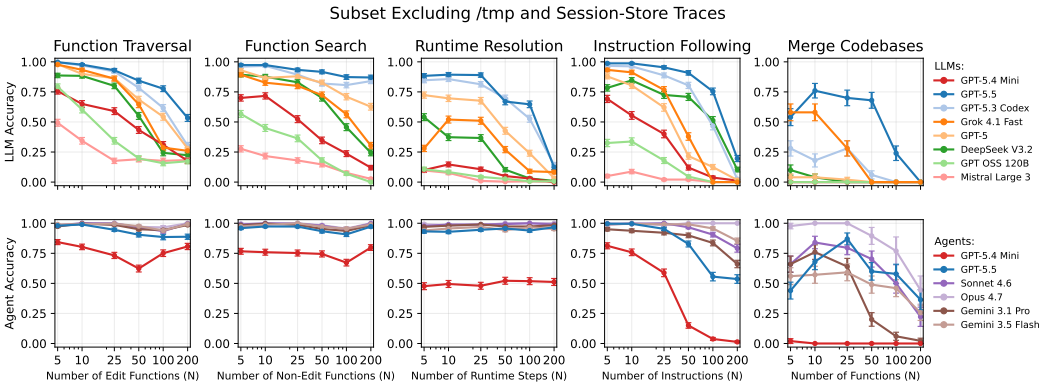


Figure A.11: Agent accuracy on CABRA tasks after we filter out suspicious runs where agents may attempt to communicate with other agents. The trend mirrors Figure 2 and Figure 5b: agent accuracy typically only scales with task size for the merge codebases task that demands more understanding.

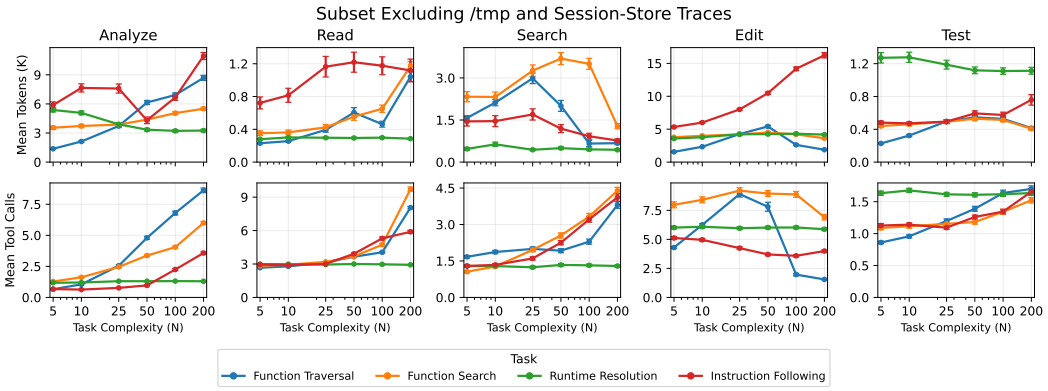


Figure A.12: Comparison of tool use on CABRA tasks after we filter out suspicious runs where agents may attempt to communicate with other agents. The trend mirrors Figure 3: tool use for ANALYZE and READ more consistently scale than EDIT for our initial tasks in §2, and tool calls for ANALYZE balloon for the merge codebases task in §5.

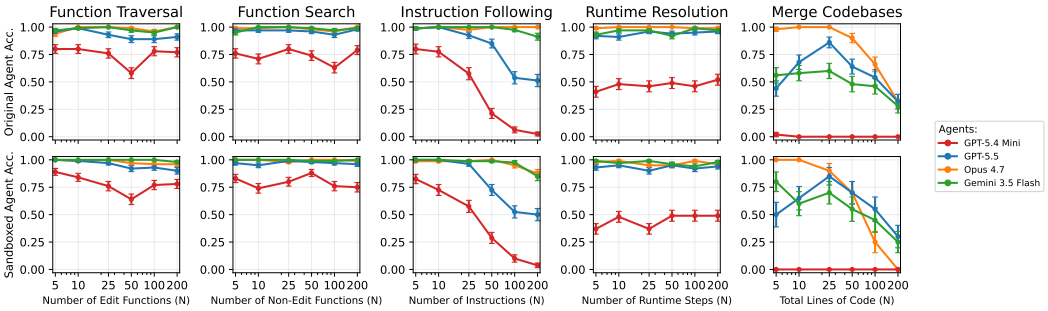


Figure A.13: Comparison of agent accuracy on initial CABRA tasks in a fully sandboxed environment versus our reported non-sandboxed runs. Trends remain consistent: agent accuracy often does not degrade as task size increases—except for our merge codebases task which targets understanding.

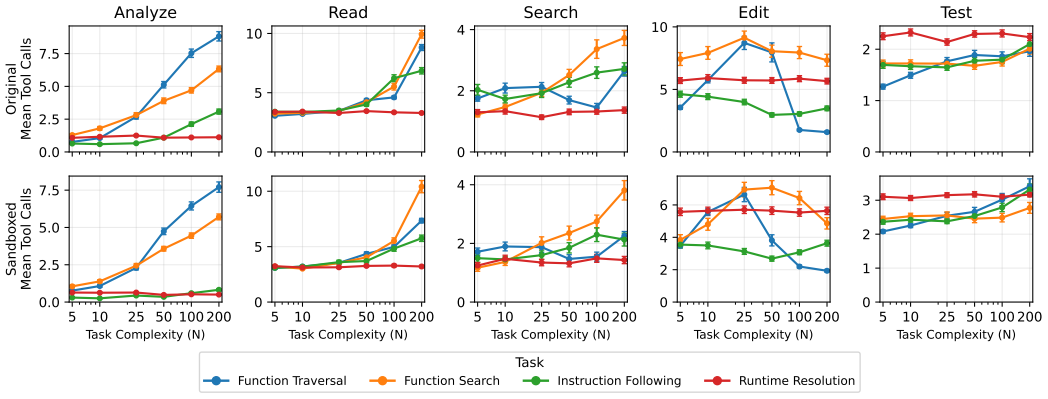


Figure A.14: Comparison of tool use on initial CABRA tasks in a fully sandboxed environment versus our reported non-sandboxed runs. Trends remain consistent: tool use for EDIT do not always scale, but READ and ANALYZE tools do more consistently.

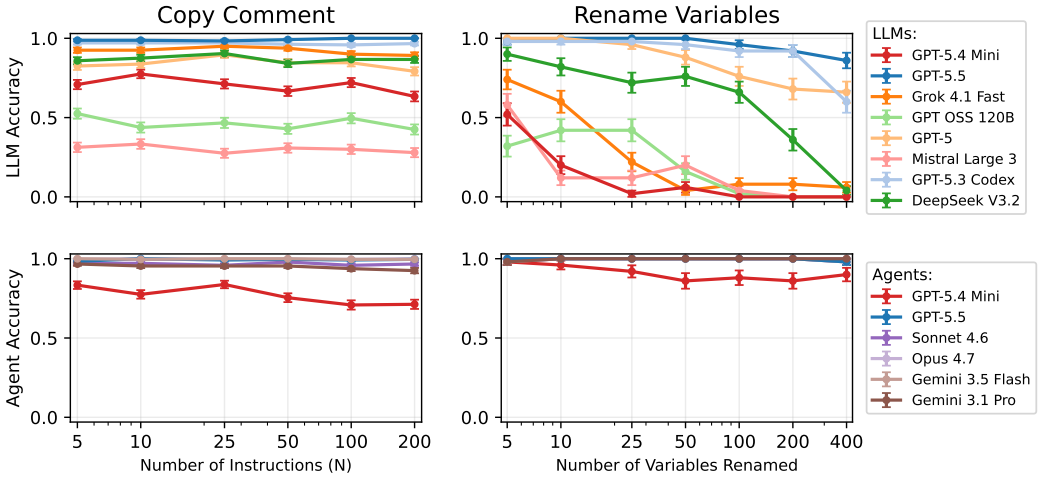


Figure A.15: Evaluation of our synthetic tasks that: 1) demand adding comments of increased words n; and 2) renaming n variables in a block of code. As n increases, LLM and agent accuracy remains stable for (1), but in (2), LLM accuracy decreases while agent accuracy remains stable.

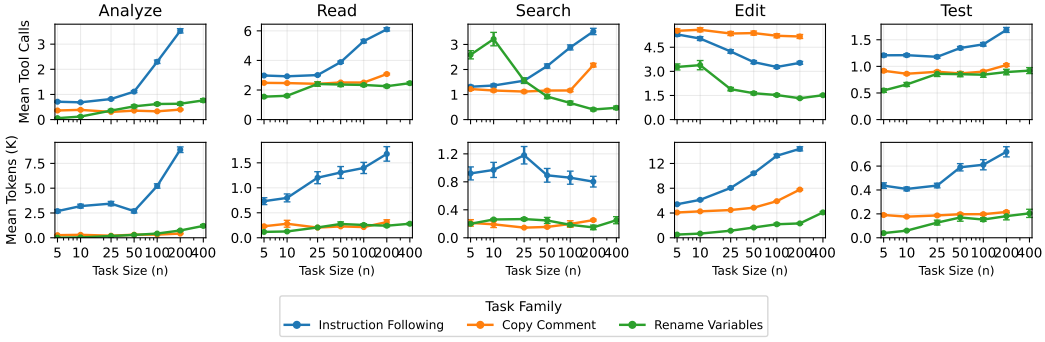


Figure A.16: Tool call categorization of the instruction-following task in Figure 3 versus tasks where models: 1) copy comments with n words; and 2) rename n variables in a block of code. As n increases, output tokens for EDIT scales for (1) and (2), but ANALYZE and READ scale modestly compared to instruction-following. Thus, there exist tasks that can increase editing requirements without largely increasing understanding requirements.

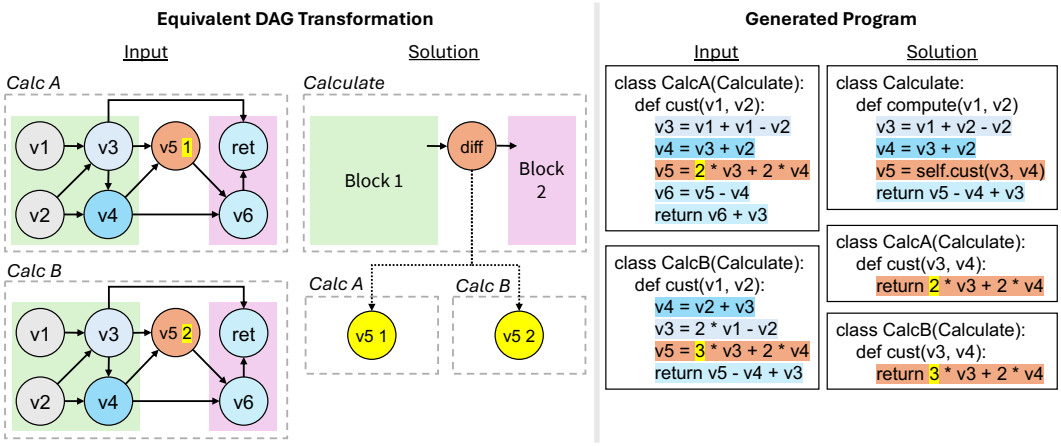


Figure A.17: Our merge codebase task in §5.1 can also be represented as a DAG transformation, mirroring the approach used in the original CABRA tasks (Figure 1, left). Each vertex v represents a defined variable, with edge $v_i \to v_j$ indicating that variable v_j uses v_i in its computation (with randomly sampled operations). We then sample vertices $\mathcal{V}_{\text{diff}}$ to inject differences. To construct the ground-truth codebase $\mathcal{C}_{\text{true}}$, we move the subgraphs between each difference node into a parent class, and keep each difference node in the original classes.

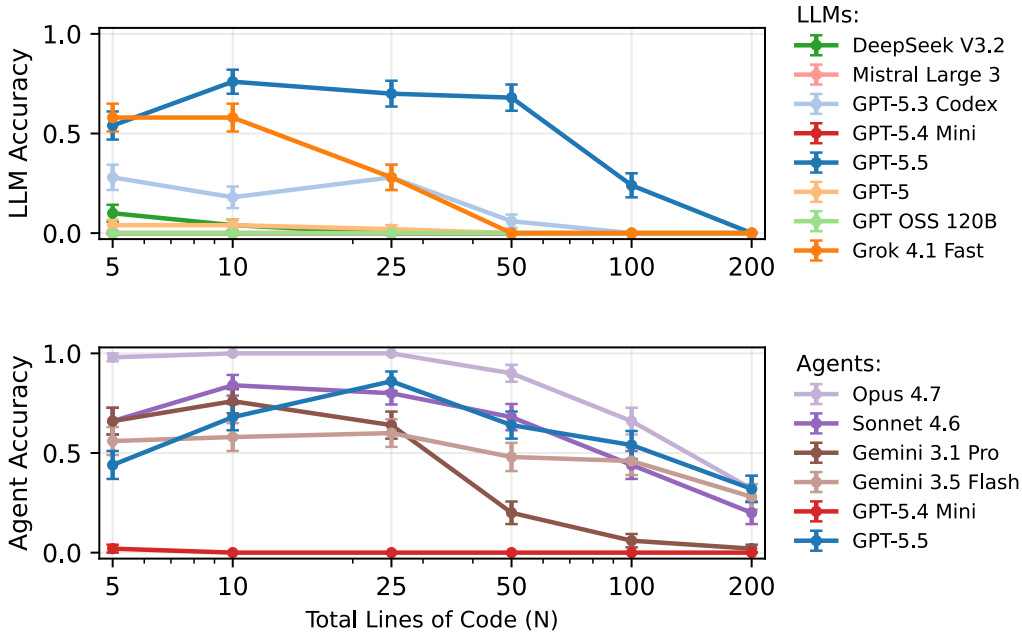


Figure A.18: Comparison of LLM and Agent accuracy in the merge codebases task (§5). Both LLMs and agents struggle, with most LLMs failing to ever clear 0.5 accuracy.

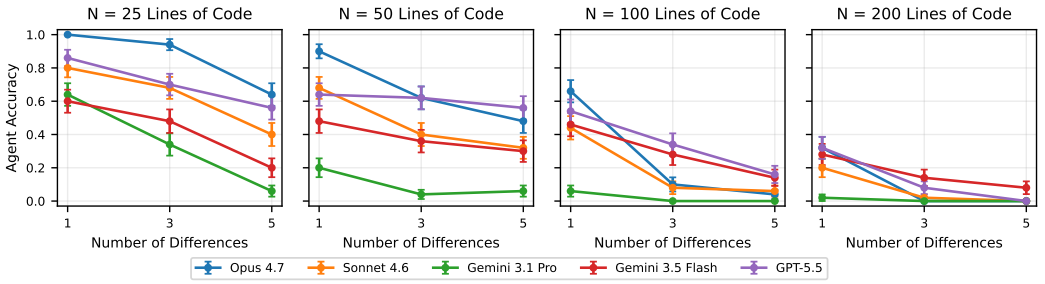


Figure A.19: Evaluation of increased number of differences in Figure 6 (right) segmented by agent. The trend is consistent: agents struggle to identify increased number of differences.

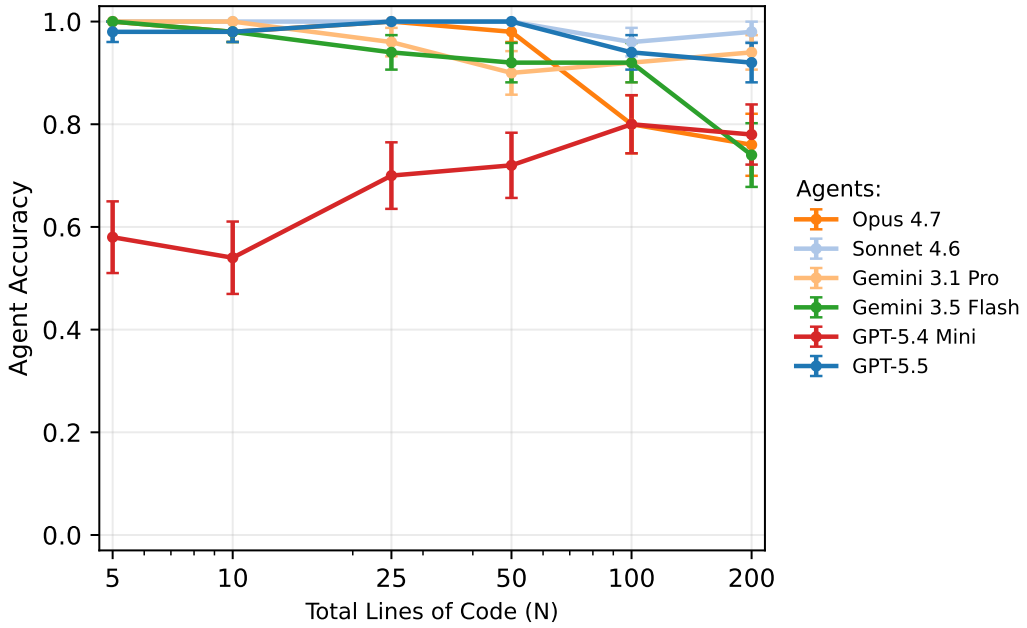


Figure A.20: Evaluation of the extract logic task in §5 but instructing agents that the optimal solution is exactly a one-line difference. Agent accuracy is much higher compared to Figure 5b, showing difficulty lies in navigating the search space to find the optimal solution with minimal edits.

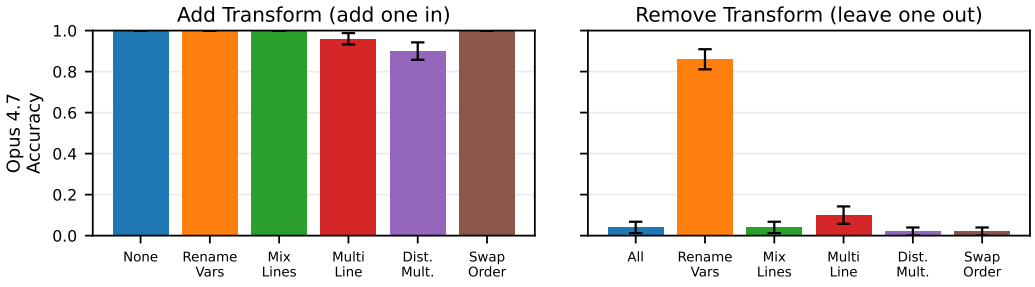


Figure A.21: Evaluation of transformation type ablations in Figure 6 (right) for $n = 200$ with Opus 4.7—our strongest agent. The trend follows Figure A.22: Opus struggles more with the multi-line and distributive transformations and sees the largest accuracy improvements when variables are not renamed.

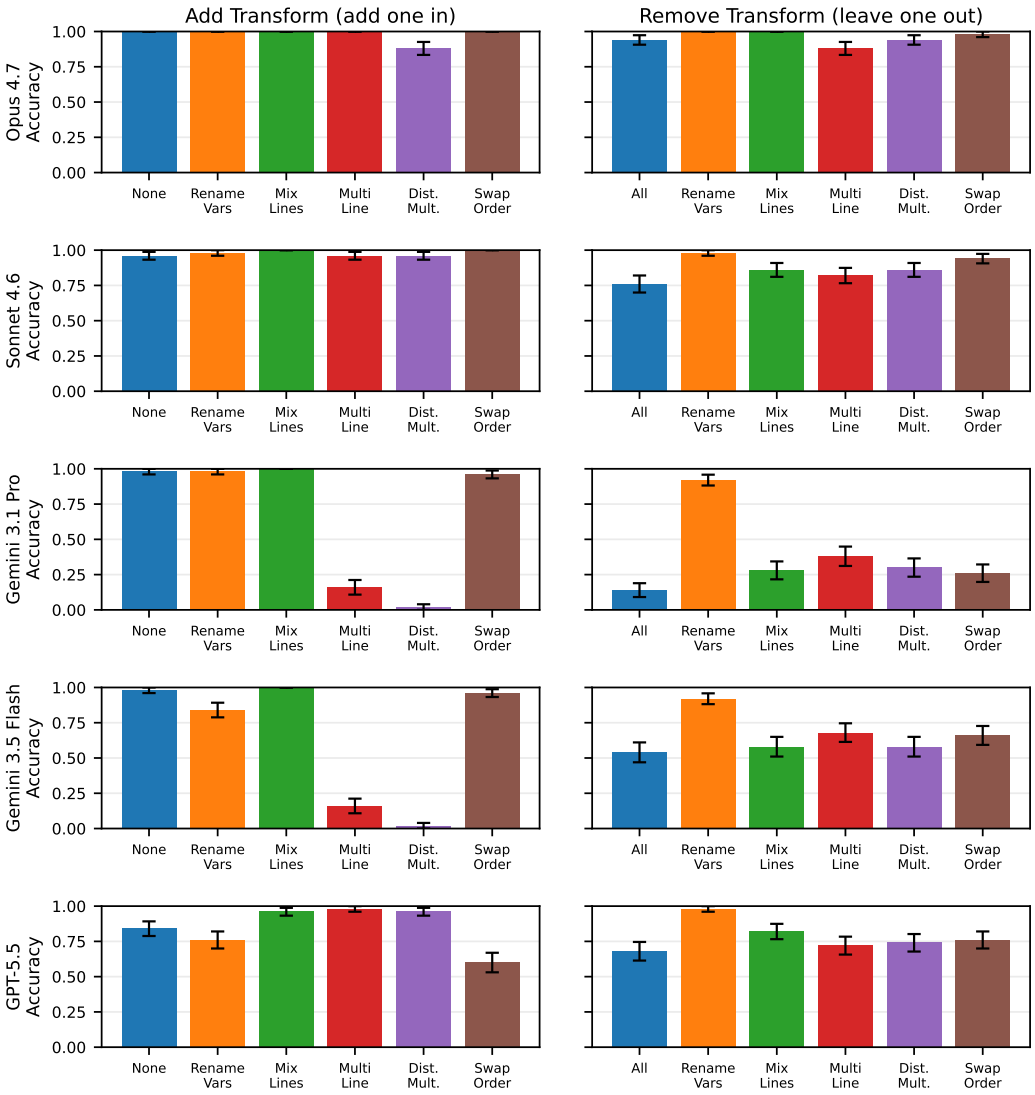


Figure A.22: Evaluation of transformation type ablations in Figure 6 (right) segmented by agent ($n = 50$). Gemini and Opus 4.7 struggle more with the multi-line and distributive transformations, while GPT-5 struggles with the swap order transformation. All models see the largest accuracy improvements when comments are not renamed, further supporting that models may over-rely on surface-level naming over behavior when understanding code.

Prompt A.1: Remove Dead Code Problem Statement

```
Some functions are unused when the solution.py script is run. Remove
    any function that is not reachable from main() through the call
    graph.

For example, if you were given the following code block:
<example>
import sys

def a(val: float) -> float:
    b_val = b(val)
    return b_val

def b(val: float) -> float:
    return val

def main(val: float):
    return b(val)

if __name__ == '__main__':
    main(float(sys.argv[1]))
</example>

You should remove the function `a()`, since it is not reached through
    `main()`, and leave `b()` as is:
<output>
import sys

def b(val: float) -> float:
    return val

def main(val: float):
    return b(val)

if __name__ == '__main__':
    main(float(sys.argv[1]))
</output>

You CANNOT add new functions, imports, or variables outside of the
    provided functions. You MUST preserve all function names that you
    do not remove (do not rename them). Do not change the "__main__"
    block or main() function.
```

Prompt A.2: Add Parameter Problem Statement

```
For any function in solution.py with the parameter mult, recursively
    propagate mult down the call graph by passing it to every function
     call used to compute the return expression. This will require
    adding mult to the parameter signatures of any called functions
    with a default value of `mult: float = default()`, and then
    recursively applying the same transformation to those functions.
    Continue propagating mult until reaching terminal functions whose
    return expressions directly compute a value `X` and do not call
    other functions in the file (apart from library functions). Only
    those terminal functions should apply the multiplier by returning
    mult * X; intermediate functions should NOT multiply their results
     by mult.
```

```
For example, if you were given the following code block:
<example>
import sys
from mylibrary import default

def a(val: float) -> float:
    b_val = b(val)
    return b_val

def b(val: float, mult: float = default()) -> float:
    c_val = c(val)
    return 2.0 * c_val

def c(val: float) -> float:
    return val + 0.5

def main(val: float):
    return a(val)

if __name__ == '__main__':
    main(float(sys.argv[1]))
</example>

You should modify the code such that a() is unchanged, and b() passes
    the `mult` parameter to c(). Then, you should modify the call
    signature of c() to accept `mult` with a default value of default
    (). Since c() is a terminal function, it will apply the multiplier
     to its original computation, returning `mult * (val + 0.5)`.
<output>
import sys
from mylibrary import default

def a(val: float) -> float:
    b_val = b(val)
    return b_val

def b(val: float, mult: float = default()) -> float:
    c_val = c(val, mult=mult)
    return 2.0 * c_val

def c(val: float, mult: float = default()) -> float:
    return mult * (val + 0.5)

def main(val: float):
    return a(val)

if __name__ == '__main__':
    main(float(sys.argv[1]))
</output>

You must follow this procedure exactly; adding function parameters or
    returning values that diverge from what we specify will cause your
     solution to be marked incorrect automatically.

Do NOT modify functions that are upstream of the function that already
     has the mult parameter. In the example above, a() is unchanged.

default() and mylibrary are undefined and will be defined later; do
    not try to define them. You CANNOT add new functions, imports, or
    variables outside of the provided functions. You MUST preserve all
     function names (do not rename them). You may define variables
    within functions. Do not change the "__main__" block or main()
```

```
    function. The code may change; do not assume that any default
    values will persist.
```

Prompt A.3: Add Return Value Problem Statement

```
In solution.py, external() previously returned a float, but mylibrary
    now defines it as returning tuple[float, str]. Update the code so
    the float preserves the original computation, while the str is
    propagated upward through the call chain. Any function that
    directly or indirectly uses the result of external() should return
     tuple[float, str]: the original computed value as the first
    element, and the propagated string as the second.

For example, if you were given the following code block:
<example>
import sys
from mylibrary import external

def a(val: float) -> float:
    b_val = b(val)
    c_val = c(val)
    return b_val + c_val

def b(val: float) -> float:
    return external(val)

def c(val: float) -> float:
    return val

def main(val: float):
    return a(val)

if __name__ == '__main__':
    main(float(sys.argv[1]))
</example>

You should modify the code such that b() now returns a tuple of type
    tuple[float, str] with the new values from external(). The string
    should propagate up the call graph recursively, modifying a() next
    , to preserve the original float computation. c() will remain
    unchanged.
<output>
import sys
from mylibrary import external

def a(val: float) -> tuple[float, str]:
    b_val, key = b(val)
    c_val = c(val)
    return b_val + c_val, key

def b(val: float) -> tuple[float, str]:
    return external(val)

def c(val: float) -> float:
    return val

def main(val: float):
    return a(val)

if __name__ == '__main__':
```

```
    main(float(sys.argv[1]))
</output>

You must follow this procedure exactly; adding function parameters or
    returning values that diverge from what we specify will cause your
     solution to be marked incorrect automatically.

If a function receives multiple propagated strings, it may return any
    one of them, since they are guaranteed to be identical. external()
     and mylibrary are undefined and will be defined later; do not try
     to define them. You CANNOT add new functions, imports, or
    variables outside of the provided functions. You MUST preserve all
     function names (do not rename them). You may define variables
    within functions. Do not change the "__main__" block or main()
    function.
```

Prompt A.4: Cache Function Problem Statement

```
In solution.py, some functions call slow(), which is expensive and
    deterministic, so it should be run as few times as possible.
    Refactor the code so that each repeated call to slow() is
    performed exactly once at the lowest shared caller (i.e., lowest
    common ancestor) of the functions that need it. Store the result
    in a temporary variable called `temp`, then pass that value down
    as an additional argument to the functions that use it (e.g., a(
    val: float) -> a(val: float, temp: float)).

For example, if you were given the following code block:
<example>
import sys
from mylibrary import slow

def a(val: float) -> float:
    b_val = b(val)
    return b_val

def b(val: float) -> float:
    c_val = c(val)
    d_val = d(val)
    return c_val + d_val

def c(val: float) -> float:
    return -4 * slow()

def d(val: float) -> float:
    e_val = e(val)
    return 2 * slow() + e_val

def e(val: float) -> float:
    return val

def main(val: float):
    return a(val)

if __name__ == '__main__':
    main(float(sys.argv[1]))
</example>

You should modify the code such that the calls to slow() in c() and d
    () are moved to b(), as b() is the lowest common ancestor of c()
```

```
    and d(). Then, b() stores the intermediate value of slow() and
    passes it down to the other functions:
<output>
import sys
from mylibrary import slow

def a(val: float) -> float:
    b_val = b(val)
    return b_val

def b(val: float) -> float:
    temp: float = slow()
    c_val = c(val, temp)
    d_val = d(val, temp)
    return c_val + d_val

def c(val: float, temp: float) -> float:
    return -4 * temp

def d(val: float, temp: float) -> float:
    e_val = e(val)
    return 2 * temp + e_val

def e(val: float) -> float:
    return val

def main(val: float):
    return a(val)

if __name__ == '__main__':
    main(float(sys.argv[1]))
</output>
```

Prompt A.5: Don't Repeat Yourself Problem Statement

```
In solution.py, multiple functions independently call get_multiplier(
    val), which returns a multiplier for a given value. To reduce this
     duplication, refactor the code so the multiplier is defined once
    at the earliest shared downstream function reached by following
    calls from each of those functions, then return that multiplier
    back up to the functions that need it. Do not move the multiplier
    computation into a caller/ancestor of the duplicated functions.

For example, if you were given the following code block:
<example>
import sys
from mylibrary import get_multiplier

def a(val: float) -> float:
    b_val = b(val)
    c_val = c(val)
    return b_val + c_val

def b(val: float) -> float:
    d_val = d(val)
    e_val = e(val)
    return get_multiplier(val) * (d_val + e_val)

def c(val: float) -> float:
    d_val = d(val)
```

```
    return get_multiplier(val) * d_val

def d(val: float) -> float:
    e_val = e(val)
    return e_val

def e(val: float) -> float:
    return val

def main(val: float):
    return a(val)

if __name__ == '__main__':
    main(float(sys.argv[1]))
</example>

First, you should identify that b() and c() both call get_multiplier(
    val). Then, trace their calls downward toward their callees. The
    first shared downstream function is d(), so move get_multiplier(
    val) into d(), return it along with d()'s original result, and
    reuse that returned multiplier in b() and c().
<output>
import sys
from mylibrary import get_multiplier

def a(val: float) -> float:
    b_val = b(val)
    c_val = c(val)
    return b_val + c_val

def b(val: float) -> float:
    d_val, mult = d(val)
    e_val = e(val)
    return mult * (d_val + e_val)

def c(val: float) -> float:
    d_val, mult = d(val)
    return mult * d_val

def d(val: float) -> tuple[float, float]:
    e_val = e(val)
    return e_val, get_multiplier(val)

def e(val: float) -> float:
    return val

def main(val: float):
    return a(val)

if __name__ == '__main__':
    main(float(sys.argv[1]))
</output>
```

Prompt A.6: Merge Codebases Problem Statement

```
CalculateA and CalculateB are subclasses of the shared Calculate class
    . Their compute() methods contain redundant logic. Refactor the
    code using the template method pattern: move as much identical
    compute() logic as possible into Calculate.compute(), extract
    subclass-specific behavior into helper methods with the name
```

```
    custom_compute_{idx}, and keep only the differences in CalculateA
    and CalculateB.

For example, if you were given the following code block:
<example>
class Calculate:
    pass

class CalculateA(Calculate):
    def compute(self, a: int) -> int:
        b = a * 2
        c = b + 3
        return c * 10

class CalculateB(Calculate):
    def compute(self, c: int) -> int:
        a = 2 * c
        b = a - 1
        return b * 10
</example>

You should modify the code such that Calculate.compute() contains the
    shared control flow and shared operations. The child classes
    should only implement the custom part that differs between them:
<output>
class Calculate:
    def compute(self, val_0: int) -> int:
        val_1 = val_0 * 2
        val_2 = self.custom_compute_1(val_1)
        return val_2 * 10

class CalculateA(Calculate):
    def custom_compute_1(self, val: int) -> int:
        return val + 3

class CalculateB(Calculate):
    def custom_compute_1(self, val: int) -> int:
        return val - 1
</output>

You may need to repeat this process multiple times with multiple
    custom_comput_{idx}() functions if there are multiple identical
    regions and multiple smaller differences inside compute(). The
    final Calculate.compute() method should contain all shared logic
    and call child class-specific helper methods only where CalculateA
     and CalculateB differ.

You must preserve the behavior of CalculateA.compute() and CalculateB.
    compute(). You MUST keep CalculateA and CalculateB as subclasses
    of Calculate. You CANNOT add new imports or unrelated classes. You
     may add helper methods to Calculate, CalculateA, and CalculateB
    as needed. Do not rename Calculate, CalculateA, CalculateB, or any
     public methods used outside these classes. Keep as few lines of
    code as possible in the child class functions custom_compute_{idx
    }; only subclass-specific differences should remain there.
```

Prompt A.7: Add Parameter Problem Statement — String Variation

```
For any function in solution.py with the parameter prefix, recursively
     propagate prefix down the call graph by passing it to every
```

```
    function call used to compute the return expression. This will
    require adding prefix to the parameter signatures of any called
    functions with a default value of `prefix: str = default()`, and
    then recursively applying the same transformation to those
    functions. Continue propagating prefix until reaching terminal
    functions whose return expressions directly compute a value `X`
    and do not call other functions in the file (apart from library
    functions). Only those terminal functions should add the prefix by
     returning prefix + X; intermediate functions should NOT append
    the prefix to their results.

For example, if you were given the following code block:
<example>
import sys
import strops
from mylibrary import default

def a(val: str) -> str:
    b_val = b(val)
    return b_val

def b(val: str, prefix: str = default()) -> str:
    c_val = c(val)
    return strops.reverse(c_val)

def c(val: str) -> str:
    return strops.interleave(val, "abc")

def main(val: str):
    return a(val)

if __name__ == '__main__':
    main(str(sys.argv[1]))
</example>

You should modify the code such that a() is unchanged, and b() passes
    the `prefix` parameter to c(). Then, you should modify the call
    signature of c() to accept `prefix` with a default value of
    default(). Since c() is a terminal function, it will apply the
    prefix to its original computation, returning `prefix + strops.
    interleave(val, "abc")`.
<output>
import sys
import strops
from mylibrary import default

def a(val: str) -> str:
    b_val = b(val)
    return b_val

def b(val: str, prefix: str = default()) -> str:
    c_val = c(val, prefix=prefix)
    return strops.reverse(c_val)

def c(val: str, prefix: str = default()) -> str:
    return prefix + strops.interleave(val, "abc")

def main(val: str):
    return a(val)

if __name__ == '__main__':
    main(str(sys.argv[1]))
</output>
```

```
You must follow this procedure exactly; adding function parameters or
    returning values that diverge from what we specify will cause your
     solution to be marked incorrect automatically.

Do NOT modify functions that are upstream of the function that already
     has the prefix parameter. In the example above, a() is unchanged.

strops, default(), and mylibrary are undefined and will be defined
    later; do not try to define them. You CANNOT add new functions,
    imports, or variables outside of the provided functions. You MUST
    preserve all function names (do not rename them). You may define
    variables within functions. Do not change the "__main__" block or
    main() function. The code may change; do not assume that any
    default values will persist.
```

Prompt A.8: Add Parameter Problem Statement — Array Variation

```
For any function in solution.py with the parameter mult, recursively
    propagate mult down the call graph by passing it to every function
     call used to compute the return expression. This will require
    adding mult to the parameter signatures of any called functions
    with a default value of `mult: float = default()`, and then
    recursively applying the same transformation to those functions.
    Continue propagating mult until reaching terminal functions whose
    return expressions directly compute a value `X` and do not call
    other functions in the file (apart from library functions). Only
    those terminal functions should apply the multiplier by returning
    mult * X; intermediate functions should NOT multiply their results
     by mult.

For example, if you were given the following code block:
<example>
import sys
import numpy as np
from mylibrary import default

def a(val: np.ndarray) -> np.ndarray:
    b_val = b(val)
    return b_val

def b(val: np.ndarray, mult: float = default()) -> np.ndarray:
    c_val = c(val)
    return 2.0 * c_val

def c(val: np.ndarray) -> np.ndarray:
    return val + 0.5

def main(val: np.ndarray):
    return a(val)

if __name__ == '__main__':
    main(np.array(eval(sys.argv[1])))
</example>

You should modify the code such that a() is unchanged, and b() passes
    the `mult` parameter to c(). Then, you should modify the call
    signature of c() to accept `mult` with a default value of default
    (). Since c() is a terminal function, it will apply the multiplier
     to its original computation, returning `mult * (val + 0.5)`.
```

```
<output>
import sys
import numpy as np
from mylibrary import default

def a(val: np.ndarray) -> np.ndarray:
    b_val = b(val)
    return b_val

def b(val: np.ndarray, mult: float = default()) -> np.ndarray:
    c_val = c(val, mult=mult)
    return 2.0 * c_val

def c(val: np.ndarray, mult: float = default()) -> np.ndarray:
    return mult * (val + 0.5)

def main(val: np.ndarray):
    return a(val)

if __name__ == '__main__':
    main(np.array(eval(sys.argv[1])))
</output>

You must follow this procedure exactly; adding function parameters or
    returning values that diverge from what we specify will cause your
     solution to be marked incorrect automatically.

Do NOT modify functions that are upstream of the function that already
     has the mult parameter. In the example above, a() is unchanged.

default() and mylibrary are undefined and will be defined later; do
    not try to define them. You CANNOT add new functions, imports, or
    variables outside of the provided functions. You MUST preserve all
     function names (do not rename them). You may define variables
    within functions. Do not change the "__main__" block or main()
    function. The code may change; do not assume that any default
    values will persist.
```

Prompt A.9: Add Parameter Task — Scaling Runtime (See Code Block A.4)

```
For any function in solution.py with the parameter mult, recursively
    propagate mult down the call graph by passing it to every function
     call used to compute the return expression. This will require
    adding mult to the parameter signatures of any called functions
    with a default value of `mult: float = default()`, and then
    recursively applying the same transformation to those functions.
    Continue propagating mult until reaching terminal functions whose
    return expressions directly compute a value `X` and do not call
    other functions in the file (apart from library functions). Only
    those terminal functions should apply the multiplier by returning
    mult * X; intermediate functions should NOT multiply their results
     by mult. Some code may be guarded by conditional expressions;
    only the if/else branch that actually executes influences the
    output. Apply this transformation only inside branches that
    actually run; if another branch contains the same calls or
    computation, preserve it unchanged.

For example, if you were given the following code block:
<example>
import sys
```

```
from mylibrary import default

def check():
    return True

def a(val: float) -> float:
    b_val = b(val)
    return b_val

def b(val: float, mult: float = default()) -> float:
    if check():
        c_val = c(val)
        return 2.0 * c_val
    else:
        c_val = c(val)
        return 2.0 * c_val

def c(val: float) -> float:
    return val + 0.5

def main(val: float):
    return a(val)

if __name__ == '__main__':
    main(float(sys.argv[1]))
</example>

You should modify the code such that a() is unchanged, and b() passes
    the `mult` parameter to c() only in the branch that actually
    executes. Then, you should modify the call signature of c() to
    accept `mult` with a default value of default(). Since c() is a
    terminal function on the running branch, it will apply the
    multiplier to its original computation, returning `mult * (val +
    0.5)`. Do not pass `mult` in the branch that does not run, even
    though it contains the same computation.
<output>
import sys
from mylibrary import default

def check():
    return True

def a(val: float) -> float:
    b_val = b(val)
    return b_val

def b(val: float, mult: float = default()) -> float:
    if check():
        c_val = c(val, mult=mult)
        return 2.0 * c_val
    else:
        c_val = c(val)
        return 2.0 * c_val

def c(val: float, mult: float = default()) -> float:
    return mult * (val + 0.5)

def main(val: float):
    return a(val)

if __name__ == '__main__':
    main(float(sys.argv[1]))
</output>
```

```
...
```

Prompt A.10: Add Parameter Task — Scaling Instruction Following

```
...

In addition to the propagation above, each generated parameter carries
     a set of CONSTRAINTS that are applied in every eligible function.
     The constraints for all generated parameters are listed together
    in the "requirements" section below and may be shuffled rather
    than grouped by parameter. Apply these constraints inside every
    function that must take the named parameter as part of the
    propagation. For each such function, apply every requirement for
    the parameter named in that requirement. Do not apply a
    requirement inside main(), inside functions that do not take the
    named parameter.

When several requirements match the same function, apply them in this
    fixed order within the function body: 1. comments 2. assert
    statements 3. print statements 4. temporary-variable assignments

Insert these constraint lines after the function has received the
    named parameter and before the named parameter is passed to
    another function or used in a terminal return computation. Comment
     requirements should be written as real # comments on their own
    lines, exactly as specified, including the named generated value.
    Assert, print, and temporary-variable assignment requirements
    should be written as separate Python statements on their own lines.


Temporary-variable assignment requirements compute a value from the
    named parameter and store it in the exact temporary variable named
     by the requirement. They do not update the parameter itself. For
    example, if a requirement says to set mult1_temp to mult1 + 0.2,
    write mult1_temp = mult1 + 0.2; continue passing and using mult1,
    not mult1_temp. If there are multiple matching requirements of the
     same kind, write each one as a separate line.

Here are the 10 requirements for the mult parameters (mult1, mult2,
    mult3, mult4, mult5, mult6, mult7, mult8, mult9, and mult10).
    These requirements apply to every eligible function.

1. For mult8: In every function, assert() that mult8 is not equal to
    30.
2. For mult7: In every function, assert() that mult7 differs from 30.
3. For mult6: In every function, print mult6 prefixed with the string
    'Log: ' in an fstring.
4. For mult4: In every function, assert() that mult4 is not equal to
    -30.
5. For mult8: In every function, add the comment '# TODO: mult8' in
    the function.
6. For mult7: In every function, compute mult7/0.4 into snapshot_mult7
     on a separate line.
7. For mult5: In every function, make sure mult5 doesn't equal -30 via
     an assert() statement.
8. For mult2: In every function, add the exact comment '# verify:
    mult2' to the function.
9. For mult2: In every function, compute mult2-0.7 into mult2_local on
     a separate line.
```

```
10. For mult4: In every function, assert() that mult4 is not equal to
    -10.
```

Prompt A.11: Add Parameter Task — Comment Copying

```
...

For every function you edit, add this comment on a new line verbatim:
    "# buoyancy victory majesty fibre audible lancer lovemaking fathom
     readability proletarian bathroom jogging copernicus fight
    nutrition thatcher subset search means nowadays cochin prologue
    entrails reissue cyberspace"
```

Prompt A.12: Variable Replacement Task

```
solution.py has a single function called main(val: float) that
    performs a math operation using intermediate variables. Rename the
     variables as follows:
- Change the name of variable 'gg' to 'bt'
- Change the name of variable 'cn' to 'gi'
- Change the name of variable 'fv' to 'g'
- Change the name of variable 'd' to 'ac'
- Change the name of variable 'dh' to 'fi'

You MUST apply every requested rename consistently to assignments and
    uses. You MUST NOT rename any other variables or change the code
    in any other way.
```

Prompt A.13: LLM Generation — Full Response

```
<instructions>
You are a helpful assistant that solves programming tasks.

You will be given an initial code repository and a problem statement.
    Your goal is to modify the repository to solve the problem.

Your response must contain exactly ONE <solution> code block with your
     solution to the problem.
Include a <thought> section before your <solution> where you explain
    your reasoning process.
Format your response as shown in <format_example>.

<format_example>
<thought>
Your reasoning and analysis here. Explain why you came up with this
    solution
</thought>

<solution>
Your solution to the problem. Return the entire modified repository in
     full in the format below:

# === solution.py (any file name) ===
...[insert code]...
```

````
This would correspond to writing all of `solution.py`.

</solution>
</format_example>

Failure to follow these rules will cause your response to be rejected.
</instructions>

Here is the initial code repository you must modify:
<code repository>
{code_repository}
</code repository>

Here is the problem statement you must satisfy:
<problem statement>
{problem_statement}
</problem statement>
````

Prompt A.14: LLM Generation — Diff Patch

````
...

<format_example>

<thought>
Your reasoning and analysis here. Explain why you came up with this
    solution
</thought>

<solution>
Your solution to the problem. For each file that needs to be changed,
    write out the changes similar to a unified diff like `diff -U0`
    would produce. Start each hunk of changes with a `@@ ... @@` line.
     Don't include line numbers like `diff -U0` does.

Return your modifications in the diff format shown below:

```diff
--- app.py
+++ app.py
@@ ... @@
-class MathWeb:
+import sympy
+
+class MathWeb:
@@ ... @@
-def is_prime(x):
-    if x < 2:
-        return False
-    for i in range(2, int(math.sqrt(x)) + 1):
-        if x % i == 0:
-            return False
-    return True
@@ ... @@
-@app.route('/prime/<int:n>')
-def nth_prime(n):
-    count = 0
-    num = 1
-    while count < n:
````
````

```
-        num += 1
-        if is_prime(num):
-            count += 1
-    return str(num)
+@app.route('/prime/<int:n>')
+def nth_prime(n):
+    count = 0
+    num = 1
+    while count < n:
+        num += 1
+        if sympy.isprime(num):
+            count += 1
+    return str(num)
```

This would correspond to replacing the function call to `is_prime` with a call to sympy in a file called app.py
</solution>

Overall, remember to format your solution as:
<thought>
...
</thought>
<solution>
...
</solution>

</format_example>

Failure to follow these rules will cause your response to be rejected.
</instructions>

...

Prompt A.15: Classifying Tool Calls on CABRA

<task>
You are an expert at classifying the reason that a coding agent generated a script or tool call for completing a programming task.

You will be given information about an agent's tool call, including the content of the tool call and a summary of what the tool call does. Your job is to classify the tool call into one of the following categories:
1. Read: Call view or a similar tool to list all of the files/folders in the codebase, review the entire codebase, look at an entire file, or view a certain section of the file (i.e., span of lines). The point is we are mostly reading the code exactly without doing analysis on it
2. Search: Use a search tool such as `grep`, `find`, `glob`, or `rg` to look for a certain parameter, function, or snippet in the code. The point is we are searching for something specifically that exists exactly in the code.
3. Understand: Use a script (e.g., Abstract Syntax Tree) to comprehend the current codebase, including dependencies, call graph analysis, and runtime analysis (such as checking the parity returned by the `check()` output function). The point is we are doing analysis on top of the code beyond just reading or searching. This also includes creating new files/scripts to do this analysis.

```
4. Edit: Make edits to the code in solution.py, including any initial
    changes or any revisions (e.g., based on failed tests). This also
    includes the generation of meta-scripts that can perform code
    edits after the script is run
5. Test: Test the current code, including syntax validation, test case
     generation, mock executions, or smoke tests. This also includes
    making changes or creating files that are not solution.py (e.g.,
    create test.py file)
6. Summary: The model calls no tool and instead just generates a
    summary of the changes made, as the task is complete. The
    tool_name will be "task_complete"
7. Plan: Create a plan (usually with SQL) or TODOs with steps on what
    to do
8. Other: Any other change that is not described above (e.g., web
    search, removing files, package installations)
</task>

If the model creates a new file, you need to determine what is the
    intent of this script and classify it accordingly (typically:
    understand, edit, or test).

Here is the input tool call and message information that you must
    classify:
<tool call information>
{tool_call}
</tool call information>

<format>
Return only a single word with your classification, which must be one
    of the following labels:
- read
- search
- understand
- edit
- test
- summary
- plan
- other

Do not generate ANYTHING else
</format>
```

Prompt A.16: Classifying Tool Calls on SWE-Bench Verified

```
<task>
You are an expert at classifying the reason that a coding agent
    generated a script or tool call for completing a programming task.

You will be given information about an agent's tool call, including
    the content of the tool call and a summary of what the tool call
    does. Your job is to classify the tool call into one of the
    following categories:
1. Read: Call view or a similar tool to list all of the files/folders
    in the codebase, review the entire codebase, look at an entire
    file, or view a certain section of the file (i.e., span of lines).
     The point is we are mostly reading the code exactly without doing
     analysis on it
2. Search: Use a search tool such as `grep`, `find`, `glob`, or `rg`
    to look for a certain parameter, function, or snippet in the code.
```

```
     The point is we are searching for something specifically that
    exists exactly in the code.
3. Understand: Use a script (e.g., Abstract Syntax Tree) to comprehend
     the current codebase, including dependencies, call graph analysis
    , and runtime analysis (e.g., checking the output of a function).
    The point is we are doing analysis on top of the code beyond just
    reading or searching. This also includes creating new files/
    scripts to do this analysis.
4. Edit: Make edits to the code, including any initial changes or any
    revisions (e.g., based on failed tests). This also includes the
    generation of meta-scripts that can perform code edits after the
    script is run.
5. Test: Test the current code or reproduce the current error,
    including syntax validation, test case generation/running, mock
    executions, or smoke tests. This also includes making changes or
    creating files used for testing.
6. Summary: The model calls no tool and instead just generates a
    summary of the changes made, as the task is complete. The
    tool_name will similar to "task_complete"
7. Plan: Create a plan with explicit TODOs on what to do
8. Other: Any other change that is not described above (e.g., web
    search, removing files, package installations)
</task>

If the model creates a new file, you need to determine what is the
    intent of this script and classify it accordingly (typically:
    understand, edit, or test).

Here is the input tool call and message information that you must
    classify:
<tool call information>
{tool_call}
</tool call information>

<format>
Return only a single word with your classification, which must be one
    of the following labels:
- read
- search
- understand
- edit
- test
- summary
- plan
- other

Do not generate ANYTHING else
</format>
```

Code Block A.1: Example Math Program

```python
from mylibrary import default
import math
import operator
import sys

def n(val: float) -> float:
    return abs(val)

def d(val: float) -> float:
```

```
    g_val = g(val)
    r_val = r(val)
    return operator.add(g_val, r_val)

def b(val: float) -> float:
    return abs(val)

def q(val: float) -> float:
    b_val = b(val)
    f_val = f(val)
    a_val = a(val)
    j_val = j(val)
    return operator.sub(min(math.hypot(b_val, f_val), a_val), j_val)

def s(val: float, mult: float=default()) -> float:
    q_val = q(val)
    return round(q_val)

def g(val: float) -> float:
    l_val = l(val)
    i_val = i(val)
    return max(l_val, i_val)

def i(val: float) -> float:
    return math.degrees(val)

def a(val: float) -> float:
    return operator.neg(val)

def j(val: float) -> float:
    return math.cos(val)

def f(val: float) -> float:
    return abs(val)

def l(val: float) -> float:
    return math.tanh(val)

def r(val: float) -> float:
    n_val = n(val)
    i_val = i(val)
    s_val = s(val)
    return math.hypot(math.hypot(n_val, i_val), s_val)

def main(val: float):
    return d(val)

if __name__ == '__main__':
    main(float(sys.argv[1]))
```

Code Block A.2: Example String Program

```
from mylibrary import default
import strops
import sys

def n(val: str) -> str:
    return strops.rot13(val)

def d(val: str) -> str:
```

```
    g_val = g(val)
    r_val = r(val)
    return strops.interleave(g_val, r_val)

def b(val: str) -> str:
    return strops.rot13(val)

def q(val: str) -> str:
    b_val = b(val)
    f_val = f(val)
    a_val = a(val)
    j_val = j(val)
    return strops.swap_halves(strops.zip_chars(strops.xor_chars(b_val,
     f_val), a_val), j_val)

def s(val: str, prefix: str=default()) -> str:
    q_val = q(val)
    return strops.mirror(q_val)

def g(val: str) -> str:
    l_val = l(val)
    i_val = i(val)
    return strops.alternate_chars(l_val, i_val)

def i(val: str) -> str:
    return strops.char_rotate(val)

def a(val: str) -> str:
    return strops.char_rotate_by_3(val)

def j(val: str) -> str:
    return strops.reverse(val)

def f(val: str) -> str:
    return strops.rot13(val)

def l(val: str) -> str:
    return strops.swap_case(val)

def r(val: str) -> str:
    n_val = n(val)
    i_val = i(val)
    s_val = s(val)
    return strops.xor_chars(strops.xor_chars(n_val, i_val), s_val)

def main(val: str):
    return d(val)

if __name__ == '__main__':
    main(str(sys.argv[1]))
```

Code Block A.3: Example Array Program

```
from mylibrary import default
import numpy as np
import sys

def n(val: np.ndarray) -> np.ndarray:
    return np.abs(val)
```

```
def d(val: np.ndarray) -> np.ndarray:
    g_val = g(val)
    r_val = r(val)
    return np.add(g_val, r_val)

def b(val: np.ndarray) -> np.ndarray:
    return np.abs(val)

def q(val: np.ndarray) -> np.ndarray:
    b_val = b(val)
    f_val = f(val)
    a_val = a(val)
    j_val = j(val)
    return np.subtract(np.multiply(np.minimum(b_val, f_val), a_val),
    j_val)

def s(val: np.ndarray, mult: float=default()) -> np.ndarray:
    q_val = q(val)
    return np.sign(q_val)

def g(val: np.ndarray) -> np.ndarray:
    l_val = l(val)
    i_val = i(val)
    return np.maximum(l_val, i_val)

def i(val: np.ndarray) -> np.ndarray:
    return np.cumprod(val)

def a(val: np.ndarray) -> np.ndarray:
    return np.flip(val)

def j(val: np.ndarray) -> np.ndarray:
    return np.sin(val)

def f(val: np.ndarray) -> np.ndarray:
    return np.abs(val)

def l(val: np.ndarray) -> np.ndarray:
    return np.cos(val)

def r(val: np.ndarray) -> np.ndarray:
    n_val = n(val)
    i_val = i(val)
    s_val = s(val)
    return np.minimum(np.minimum(n_val, i_val), s_val)

def main(val: np.ndarray):
    return d(val)

if __name__ == '__main__':
    main(np.array(eval(sys.argv[1])))
```

Code Block A.4: Collatz-Inspired Runtime Function

```
def check(x: int, a: int, b: int, target: int, num_steps: int) -> bool
    :
    for step in range(num_steps):
        if x % 2 == 0:
            x = x // 2
        else:
```

```
            x = a * x + b

    return x % 2 == target
```

Code Block A.5: Example Calculator Classes

```
from abc import ABC, abstractmethod


class Calculate(ABC):
    """Abstract base: turn the input parameters into a single result."
    ""

    @abstractmethod
    def compute(self, *args):
        ...


class CalculateA(Calculate):

    def compute(self, al, ac, ae, aj, l, ak, ag, m, ad, ab):

        af = 4 * ac - 2 * (aj * l)
        s = -5 * (ag * af) - aj - aj
        v = 2 * s - ab

        ah = -4 * ac + v

        y = -4 * (aj * s) - 2 * ac
        q = y - 2 * af
        x = 5 * ah + 4 * ac

        return x + q

class CalculateB(Calculate):

    def compute(self, l, aa, ab, t, o, ah, w, y, v, r):

        aj = 4 * aa - 2 * (o * t)
        s = -2 * t - 5 * (w * aj)
        al = -r + 2 * s

        ai = -4 * (aa * v) + al

        af = 4 * aa + 5 * ai
        m = -4 * (t * s) - 2 * aa
        z = -2 * aj + m

        return af + z


if __name__ == "__main__":
    calc_a = CalculateA()
    calc_b = CalculateB()

    args = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

    result_a = calc_a.compute(*args)
    result_b = calc_b.compute(*args)
```

```
    print(f"Result A: {result_a}")
    print(f"Result B: {result_b}")
```

Code Block A.6: Rename Variable Program

```
def main(val: float):
    ak = val
    n = ak + 4
    gr = n + ak
    ds = gr + ak
    bk = ds - ak
    f = bk - n
    bn = f + 4
    ae = bn + 8
    bb = ae + 7
    dz = bb - bn
    bj = dz + ae
    cd = bj + 2
    cm = cd + f
    gk = cm - ds
    al = gk - gr
    ft = al + gk
    dg = ft + gk
    at = dg + ak
    p = at + 4
    gq = p + ft
    c = gq + 1
    ee = c - cm
    ff = ee - 8
    aq = ff - at
    dr = aq + dz
    fx = dr + 9
    fw = fx + 1
    gx = fw + 9
    dy = gx + dg
    fk = dy + 2
    d = fk - 4
    cc = d + dy
    ed = cc + 1
    bw = ed - c
    cz = bw + c
    fc = cz - bb
    cb = fc + bk
    fu = cb + bn
    fh = fu - 9
    j = fh + ff
    m = j - 6
    am = m - cz
    aw = am + 9
    au = aw - 9
    ch = au + 2
    cx = ch + bn
    bz = cx + fh
    return bz
```