Frontier Coding Agents Use Metaprogramming to Adapt to Unfamiliar Programming Languages

Aman Sharma* **Sushrut Thorat†** **Paras Chopra†**
Lossfunk
{aman.sharma,sushrut.thorat,paras}@lossfunk.com

Abstract

LLM-based coding agents are usually evaluated in familiar software settings: mainstream languages, common libraries, and public repositories. These benchmarks remain important, but they can hide how agents behave when the language itself is unfamiliar. We evaluate six contemporary coding agents on four esoteric programming languages using a sequential setup with file editing, local execution, and hidden-test grading. Our protocol exposes capability differences between these agents that mainstream coding and agentic benchmarks such as SWE-Bench Verified and Terminal-Bench 2.0 compress into much narrower bands. We observe that the strongest agents, Claude Opus 4.6 and GPT-5.4 xhigh, often avoid writing the target language directly. On Brainfuck and Befunge-98, they write Python programs that generate target-language code and debug those generators locally. Forbidding this metaprogramming strategy causes large performance drops. Text guidance distilled from this strategy does not materially improve weaker agents. In contrast, Opus-derived Python reference code for building generators, including task-specific examples, sharply improves Sonnet 4.6 and GPT-5.4 mini on the same problems, while Haiku 4.5 remains low. More interpreter calls improve stronger agents but leave weaker agents near their original performance; observed token-use efficiency also differs, suggesting that resources amplify useful strategies rather than create them. Together, these results show that **strong coding agents adapt to unfamiliar languages by using tools, feedback, and workspace state to build a working model of the target language**. Metaprogramming is the clearest case, but the broader gap is constructing and debugging a strategy that works under the target language's rules.

1 Introduction

Coding is one of the central applications of large language model (LLM) agents. Most prominent benchmarks for coding agents evaluate them in familiar software ecosystems: mainstream languages, common libraries, and public open-source repositories. SWE-Bench Verified [Jimenez et al., 2024] is a canonical example, testing agents on real GitHub issues from widely used Python projects. These benchmarks measure important progress on realistic software-engineering tasks, but they also test settings where frontier models have extensive prior exposure to the relevant syntax, APIs, libraries, coding patterns, and repository structure. A complementary question is how the same agents behave when the programming language itself is unfamiliar: when the agent must figure out how to write, run, debug, and revise code in a language whose syntax and execution rules are not already familiar. This setting has received comparatively less attention in agentic coding evaluation, despite its practical relevance. Production systems often require models to work with internal domain-specific

*Primary author. Correspondence: aman.sharma@lossfunk.com.
†Equal advising.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

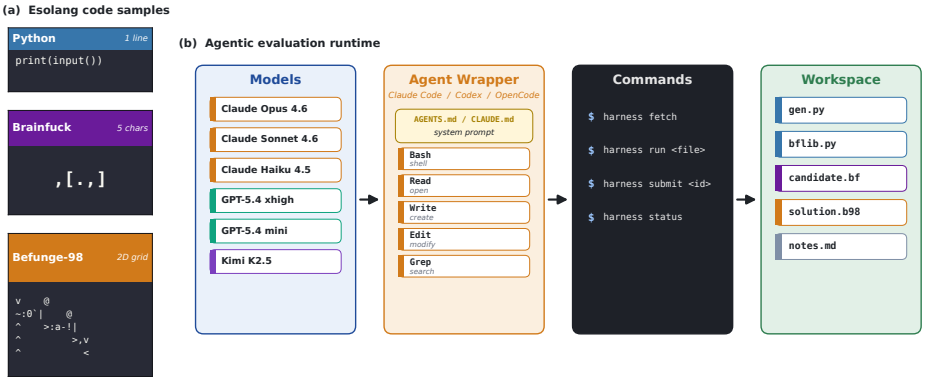

Figure 1: **Task substrate and agentic runtime.** (a) The same simple input-and-print task in Python, Brainfuck, and Befunge-98 shows how different esolang code looks from ordinary code. (b) Each model runs in a coding harness (Claude Code, Codex, or OpenCode) with file editing, shell access, benchmark commands, and a persistent workspace for local execution and hidden-test submission.

languages, proprietary configuration files, generated APIs, and local tool conventions that are absent from public corpora or differ from standard programming environments. In such settings, success depends less on recalling familiar code patterns and more on building a working understanding of the target interface during the session.

To study how contemporary LLM-based coding agents behave when the programming language itself is unfamiliar, we use languages from EsoLang-Bench [Sharma and Chopra, 2026]. These esoteric languages are not realistic production targets; they are controlled proxies for unfamiliar executable interfaces. For example, Brainfuck is a minimal pointer-machine language, and Befunge-98 introduces two-dimensional control flow over a stack-based grid (Figure 1a). This makes them useful for testing whether agents can learn an unfamiliar language well enough during a session to write, run, debug, and improve working programs.

We therefore build an agentic evaluation pipeline around EsoLang-Bench and use it to compare a capability ladder of six contemporary LLM-based coding agents under a common tool-use protocol: Claude Opus 4.6, Sonnet 4.6, and Haiku 4.5; GPT-5.4 xhigh and GPT-5.4 mini (xhigh and medium reasoning effort, respectively); and Kimi K2.5 (Figure 1). Each agent works in a persistent workspace where it can edit files, run code locally, and submit final answers to hidden tests. The evaluation therefore tests an interactive problem-solving process, not a single code completion. We analyze final pass rates together with agent logs and targeted interventions, allowing us to ask which agents succeed and how they adapt.

Central observations:

1. **Unfamiliar-language evaluation separates agents that look similar on mainstream coding benchmarks.** Under our EsoLang-Bench protocol, where the target language must be worked out within the session, these agents are separated over a much wider range than on mainstream coding benchmarks such as SWE-Bench Verified and Terminal-Bench 2.0, exposing capability differences that those benchmarks compress into narrower bands (Table 2).
2. **The strongest agents use metaprogramming.** They write Python generators that emit target-esolang programs, reuse helpers across problems, and test locally before submission. This emerges without language-specific prompting. Forbidding metaprogramming on Brainfuck and Befunge-98 drops performance by tens of percentage points.
3. **Strategy transfer works through executable scaffolds, but not with distilled written strategies.** A textual description of the strategy does not close the gap, but providing the implementation of that strategy as a reference library of working Python generators substantially improves Sonnet 4.6 and GPT-5.4 mini. Haiku 4.5 remains low, showing that some agents still cannot compose the provided machinery into working solutions.

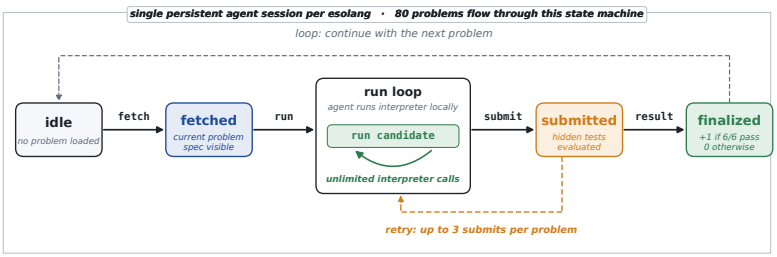

Figure 2: **Per-problem state machine under the primary protocol.** Each model–language run is a fixed forward session over 80 problems. For each problem, the agent fetches the specification, edits and executes candidate programs locally, and makes up to three hidden submissions. Hidden submissions return only aggregate hidden-test feedback; finalized problems are not revisited.

4. **Extra inference-time resources help only when agents can use them.** More interpreter calls improve stronger agents but leave weaker agents near their original performance. Observed token-use efficiency also differs; the token analysis does not manipulate output budgets.

2 Experimental setup

Task substrate. We evaluate on EsoLang-Bench [Sharma and Chopra, 2026], using the original 80-problem sequences for Brainfuck, Befunge-98, Whitespace, and Shakespeare. EsoLang-Bench includes a fifth language, Unlambda, which we exclude because its interpreter made local execution substantially slower in our agentic setting; including it would make wall-clock runtime depend heavily on interpreter latency rather than adaptation behavior. The problems themselves are short standard programming tasks (echoing input, sorting integers, GCD/LCM, and similar list- and number-manipulation tasks across the four difficulty tiers; the full per-tier task list is in Appendix A.2). The original EsoLang-Bench evaluation reports near-ceiling performance on these same problem statements when models answer in Python or JavaScript, so here the difficulty is primarily expressing, implementing, and debugging solutions in an unfamiliar target language.

Agentic protocol. We use the same four-language task substrate as EsoLang-Bench, but evaluate each model as an interactive coding agent rather than a one-pass generator. Each model$\times$language run is one sequential session over all 80 problems for that language. Problems are fetched in fixed forward order. For each problem, the agent receives the statement, edits files in a separate workspace, runs candidates locally, and may make up to three hidden submissions. A problem is finalized when one submission passes all six hidden tests or when the three submissions are exhausted; finalized problems are not revisited.

Local interpreter calls expose ordinary execution feedback such as stdout, stderr, and runtime errors. Hidden submissions return only the aggregate number of private tests passed, not the private inputs, expected outputs, or per-test diagnostics. Figure 2 summarizes the state machine. The primary protocol uses 80 problems per language, six hidden tests per problem, up to three hidden submissions, unlimited local interpreter calls, a 32k-token output budget per assistant turn, and separate workspaces; all parameters are summarized in Appendix Table 4.

Models and harnesses. We evaluate deployed coding agents rather than bare models. Claude Opus 4.6, Claude Sonnet 4.6, and Claude Haiku 4.5 run under Claude Code [Anthropic, 2026]; GPT-5.4 xhigh and GPT-5.4 mini run under Codex [OpenAI, 2026a,b]; and Kimi K2.5 runs under OpenCode [Moonshot AI, 2026]. This model$\times$harness pairing is part of what we evaluate, because tool mediation, file editing, shell access, and workspace management are part of deployed coding-agent systems. Per-agent API endpoints, model identifiers, sampling settings, and harness invocations are documented in Appendix A.

Every agent receives the same benchmark-facing operations and the same per-language system prompt (a simple task prompt for benchmarking, with no problem-specific guidance, no solved examples, and no hidden-test material); the system prompt and the per-condition deviations from

this primary prompt are in Appendix A.12. As a cross-harness check, we re-ran Opus 4.6 and GPT-5.4 xhigh under OpenCode on Brainfuck and Befunge-98; we observed similar performance and the qualitative ordering is unchanged (Appendix B.4).

Logging and behavioral measurements. For each run, we log problem fetches, shell commands, local interpreter calls, hidden submissions, file edits, generated files, command outputs, and final workspace state. We use *metaprogramming* to mean that the agent writes a program in a familiar host language, such as Python, JavaScript, or Rust, whose output is source code in the target esolang. This differs from direct authoring, where the agent edits the target esolang source itself. A helper file is reusable if it persists in the workspace and is called, imported, copied, or modified across multiple problems in the same session. These labels describe behavior only; scoring depends solely on hidden-test success.

Scoring and reporting. A problem is counted as solved if and only if one of the agent's submissions passes all six private hidden tests for that problem; a submission that passes only a subset of the hidden tests counts as a failure for that submission, with no partial credit, and we do not aggregate hidden-test passes across the up-to-three submissions allowed per problem. Primary scores are solved problems out of 80 for each model$\times$language run.

For each model$\times$language cell, we run three independent sessions under the same task order and protocol. We report the first session's solved count as the headline value in Table 1 and Table 2, with nominal Wilson 95% binomial confidence intervals [Wilson, 1927] computed over its 80 per-problem outcomes ($k/80$ per-language and $k/320$ pooled). These intervals are descriptive: sequential workspace reuse violates independence, so nominal coverage is not guaranteed. The remaining two sessions serve as session-to-session sanity checks; per-session counts for all three sessions are tabulated in Appendix B.7. Full reporting and confidence-interval details are in Appendices A.6, A.7, and B.6.

Diagnostic protocol variants. All headline results use the primary protocol above. We use protocol variants only for targeted diagnostic experiments. In Section 3.3, we restrict metaprogramming by requiring agents to author Brainfuck and Befunge-98 directly, without using a host-language generator. In Section 3.4, we test strategy transfer by giving weaker agents either text guidance distilled from Claude Opus 4.6's traces or a reference library of generator code; its reusable primitives, examples, and overlap audit are documented in Appendix J.5. In Section 3.5, we vary local interpreter-call budgets and analyze observed output-token use while holding the task substrate fixed.

These variants are used to explain the performance gaps observed under the primary protocol, not to define the main score. Unless explicitly stated otherwise, model prompts, task order, hidden tests, submission limits, and scoring rules remain unchanged from the primary protocol.

Table 1: **Main EsoLang-Bench results under the primary protocol.** EsoLang cells report Session 1 percentage solved (the headline session; two further sessions per cell are reported in Appendix B.7) with Wilson 95% binomial confidence intervals as subscripts. The **Mean** column averages the four esolang scores.

Agent	WS	Sh	B98	BF	**Mean**
Kimi K2.5	$31.3_{\pm 10.8}$	$2.5_{+6.2}$	$6.3_{+7.6}$	$5.0_{+7.2}$	$\mathbf{11.3_{\pm 3.9}}$
Haiku 4.5	$81.3_{\pm 9.9}$	$7.5_{+7.9}$	$5.0_{+7.2}$	$5.0_{+7.2}$	$\mathbf{24.7_{\pm 5.0}}$
Sonnet 4.6	$100.0_{-4.6}$	$70.0_{\pm 10.8}$	$80.0_{\pm 10.0}$	$15.0_{\pm 9.4}$	$\mathbf{66.3_{\pm 5.3}}$
Opus 4.6	$100.0_{-4.6}$	$87.5_{\pm 9.0}$	$80.0_{\pm 10.0}$	$80.0_{\pm 10.0}$	$\mathbf{86.9_{\pm 4.1}}$
GPT-5.4 mini	$88.8_{\pm 8.8}$	$21.3_{\pm 10.2}$	$13.8_{\pm 9.2}$	$6.3_{+7.6}$	$\mathbf{32.5_{\pm 5.3}}$
GPT-5.4 xhigh	$100.0_{-4.6}$	$100.0_{-4.6}$	$100.0_{-4.6}$	$98.8_{-5.5}$	$\mathbf{99.7_{-1.4}}$

BF=Brainfuck, B98=Befunge-98, WS=Whitespace, Sh=Shakespeare. Subscripts denote Wilson 95% intervals: $-x$ for ceiling cells, $+x$ for near-floor cells, and $\pm x$ for interior cells. Raw counts are in Appendix B.2, per-session counts in Appendix B.7, and full asymmetric intervals in Appendix B.6.

3 Results

3.1 Unfamiliar-language evaluation sharply separates contemporary coding agents

We first ask how contemporary agentic models fare under unfamiliar-language evaluation. Table 1 reports per-language EsoLang-Bench scores for the six evaluated agents under the primary protocol, with Wilson 95% binomial confidence intervals (Section 2, Appendix B.6). We observe a large performance spread between the agents. The separation is not driven by a single uniformly hard language. Per-language ranges are 5.0–98.8 on Brainfuck, 5.0–100 on Befunge-98, 31.3–100 on Whitespace, and 2.5–100 on Shakespeare. Whitespace is near-ceiling for several agents, whereas Brainfuck and Befunge-98 expose large separations. What these two languages share is that they are low-ecosystem targets whose syntax and idioms are far from mainstream software work; the spread is therefore informative about within-session adaptation to an unfamiliar executable interface.

Comparing the same six agents across mainstream coding benchmarks reveals a much smaller spread. Table 2 reports the spread and standard deviation across three mainstream coding benchmarks (SWE-Bench Verified, Terminal-Bench 2.0, and LiveCodeBench v6) together with the EsoLang-Bench mean; EsoLang-Bench produces both the widest spread and the largest SD of the four. Figure 6 in the appendix visualizes the SWE-V versus EsoLang-Bench cell as a scatter plot. This comparison is descriptive: benchmark tasks, harnesses, and budgets differ, so score dispersion alone does not isolate the effect of language familiarity.

Table 2: **Mainstream coding scores cluster while unfamiliar-language scores separate.** Six contemporary coding agents on three mainstream coding benchmarks and the EsoLang-Bench four-language mean. Spread is best minus worst across the six agents; SD is the sample standard deviation (n=6, $n-1$ denominator). The EsoLang SD (36.0) is roughly $12\times$ SWE-Bench Verified's (2.9) and $2\times$ LiveCodeBench v6's (17.2). † vendor-reported; * Vals.ai. Per-agent sourcing in Appendix Table 6; scatter view in Figure 6.

Benchmark	Opus 4.6	Sonnet 4.6	Haiku 4.5	GPT-5.4 xhigh	GPT-5.4 mini	Kimi K2.5	Spread (pp)	SD
SWE-Bench Verified†	79.6	79.2	73.3	78.2	73.0	76.8	**6.6**	**2.9**
Terminal-Bench 2.0†	65.4	59.1	41.8	75.1	60.0	50.8	**33.3**	**11.5**
LiveCodeBench v6*	84.7	82.1	41.2	84.1	81.5	83.9	**43.5**	**17.2**
EsoLang-Bench mean	86.9	66.3	24.7	99.7	32.5	11.3	**88.4**	**36.0**

3.2 Strong agents discover metaprogramming strategies

The performance spread in Section 3.1 raises a behavioral question: what do high-performing agents do differently during the session? Inspecting the logged trajectories shows a consistent pattern on the low-level languages, especially Brainfuck and Befunge-98. The strongest agents often avoid direct target-language authoring. Instead, they write generators in a familiar host language that emit source code in the target esolang, then run the generated programs against the local interpreter before submission. We call this metaprogramming. This subsection describes the behavior we observe in the logs; the next subsection tests whether removing it changes performance.

This strategy is not requested by the primary prompt. The system prompts are fixed across primary runs and reproduced in Appendix A.12; the harness accepts either direct target-language files or generated target-language files. The strategy therefore emerges from the agent's interaction with the task substrate, rather than from language-specific prompting.

A representative within-session switch occurs in Brainfuck E04. Opus 4.6 first submits a hand-written Brainfuck program of 1884 bytes, which fails the hidden tests. After the failure, it writes a Python generator; the generated Brainfuck source is 24500 bytes and passes all six hidden tests. This illustrates why the strategy helps: the host program can name and reuse structure that is implicit and fragile in raw Brainfuck, such as cell allocation, pointer position, sign flags, decimal-digit layout, BCD arithmetic, and conditional macros. Sample programs and trajectory excerpts are provided in Appendices F and C.5.

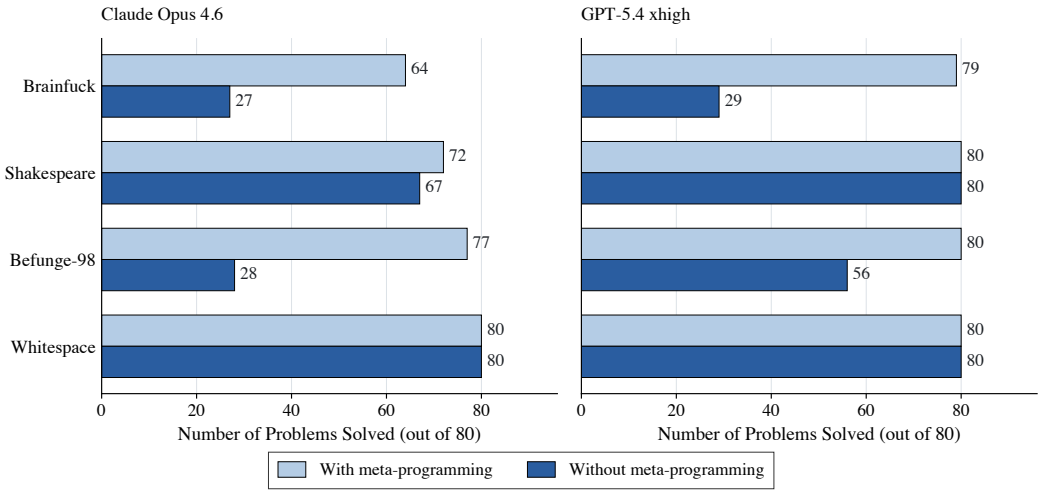

Figure 3: **Forcing direct authoring sharply reduces performance on Brainfuck and Befunge-98.** Solved problems out of 80 for Opus 4.6 and GPT-5.4 xhigh with metaprogramming allowed versus forced direct authoring. The largest drops occur on Brainfuck and Befunge-98. These ablation sessions are distinct from the headline sessions in Table 1.

3.3 Metaprogramming is causally important on Brainfuck and Befunge-98

To test whether metaprogramming merely correlates with success or supports it, we run a no-metaprogramming variant for the two strongest agents. In this variant, agents must author the target esolang directly and may not use a host-language program to generate target source. All other aspects of the protocol are held fixed.

Figure 3 shows that removing metaprogramming causes large drops on Brainfuck and Befunge-98 for both Opus 4.6 and GPT-5.4 xhigh. These are the two languages where direct authoring requires the agent to maintain long, low-level program state across edits. In direct Brainfuck authoring, for example, the agent must track cell offsets, pointer position, flags, and numeric encodings implicitly while editing raw symbols. Small local changes can invalidate this bookkeeping, and local smoke tests often miss hidden edge cases. A host-language generator externalizes this state into named variables and reusable functions, so the same cell allocation, arithmetic, and branching patterns can be emitted consistently across problems.

By contrast, removing metaprogramming has a smaller effect on Shakespeare and no effect at ceiling on Whitespace in these sessions. This comparison does not isolate why the effect varies across languages. The result supports a narrower claim: generator use is important for high performance on Brainfuck and Befunge-98 under the tested protocol. Code excerpts and trajectory examples are provided in Appendices C.5 and C.6.

The benefit is host-language generation, not Python specifically. On Brainfuck, swapping the generator host language preserves most of the gain: Opus 4.6 solves 64/80 with Python, 63/80 with JavaScript, and 55/80 with Rust, while GPT-5.4 xhigh solves 79/80, 77/80, and 79/80, respectively. Direct authoring remains low for both agents (27/80 and 29/80). Thus the critical ingredient is access to a familiar general-purpose host language for constructing target programs, not Python itself; the corresponding generator code in each host language is shown in Appendix C.7.

3.4 Strategy transfer works through executable scaffolds, but not with distilled written strategies

We next ask whether lower-performing agents fail because they lack the high-level idea or because they cannot construct the machinery needed to execute it. We use the strongest agents' traces (primarily Claude Opus 4.6, plus a single generic Brainfuck builder pattern from a successful GPT-5.4 xhigh session) to create two forms of strategy transfer for three lower-performing agents. In the

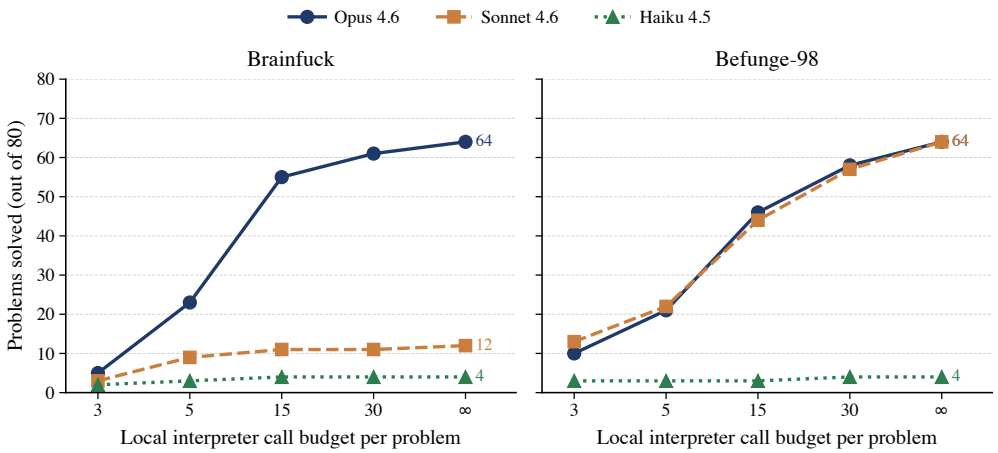

Figure 4: **More interpreter calls help only agents that can use feedback.** Problems solved out of 80 on Brainfuck and Befunge-98 under local-interpreter-call budgets of 3, 5, 15, 30, and unlimited. Opus improves with budget; Haiku remains near the floor; Sonnet improves on Befunge-98 but not Brainfuck.

text condition, we add a system-prompt preamble summarizing the strategy: use a generator, build reusable primitives, verify locally, and regenerate components rather than hand-patching target code. In the *library* condition, we additionally provide host-language reference code and notes distilled from those traces. The retained library contains reusable primitives (cell allocation, BCD arithmetic, decimal printing, and a Befunge-98 simulator) and task-specific generator examples. Thus +Lib tests transfer through executable reference code, without isolating generic primitives from examples. Library contents and a byte-identity audit are in Appendices G.2 and J.5.

Table 3 reports solved problems out of 80 for Brainfuck and Befunge-98. Written advice alone produces little improvement. The reference library, in contrast, substantially improves Sonnet 4.6 and GPT-5.4 mini, while Haiku 4.5 remains near the floor. This pattern is consistent with difficulty implementing the strategy, but +Lib supplies both executable machinery and information about usable operations. It does not separately identify affordance extraction, implementation, or iterative debugging as the binding constraint. Appendix J.5 reports a sensitivity check for exact overlap with retained examples; the original results in Table 3 are unchanged.

Table 3: **Strategy transfer works through executable scaffolds, but not with distilled written strategies.** Results are problems solved out of 80; Base is the primary protocol. *+Text* adds written strategy guidance distilled from Opus 4.6's trajectory; *+Lib* adds executable reference code and notes from strong-agent traces. Retained libraries include reusable primitives and task-specific examples; Appendix J.5 gives the overlap audit and its limits.

	Brainfuck				Befunge-98			
Agent	Base	+Text	+Lib	Opus	Base	+Text	+Lib	Opus
Haiku 4.5	4	3	7	64	4	5	4	64
Sonnet 4.6	12	12	64	64	64	66	78	64
GPT-5.4 mini	5	8	53	64	11	15	64	64

3.5 Additional inference-time resources help only when agents can use them

Interpreter-call budget. We cap local interpreter calls per problem at 3, 5, 15, 30, or unlimited, holding the task substrate, hidden submissions, and scoring rule fixed. Figure 4 shows the result on Brainfuck and Befunge-98. Additional interpreter access helps agents that already convert local feedback into progress: Opus 4.6 improves on both languages, and Sonnet 4.6 improves on Befunge-98. Agents that are near the floor at the smallest budget remain near the floor even when given many

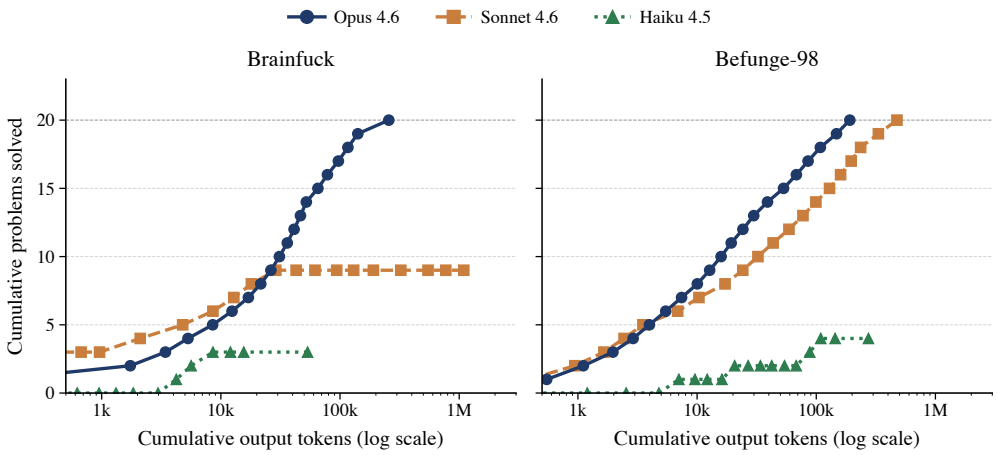

Figure 5: **Observed token efficiency differs across agents.** Cumulative solves versus cumulative API output tokens on the first 20 Brainfuck and Befunge-98 problems for Claude agents. Opus reaches 20/20 on both languages with fewer tokens than Sonnet; Haiku saturates early.

more local runs. Thus, tool access is not a uniform substitute for strategy construction; it amplifies agents that can use the feedback.

Output-token use. We also ask whether the gap is explained simply by stronger models spending more output tokens. For the first 20 Brainfuck and Befunge-98 problems, we log cumulative API output tokens for the three Claude agents, including extended-thinking tokens for Opus and Sonnet. Figure 5 plots cumulative solves against cumulative output tokens. Opus 4.6 solves more problems with fewer tokens than Sonnet 4.6 on Brainfuck and reaches 20/20 on Befunge-98 with roughly half Sonnet's token use. The difference is therefore not just that Opus spends more; it finds a reusable strategy earlier, after which additional problems become cheaper to solve. This observational comparison does not estimate the causal effect of increasing output-token budgets.

Additional diagnostics. Appendix J collects the APL extension, shuffled-order comparisons, cost and latency examples, weak-agent behavioral summaries, and library-overlap check discussed during review. These follow-up sessions supplement the primary results; their scores do not replace the headline sessions. The appendix distinguishes reported summaries from counts reconstructed from retained records.

4 Limitations and validity checks

Closed models and training exposure. The strongest agents we evaluate are closed-source, so we cannot inspect their training data, post-training environments, or exact exposure to esolang examples. We therefore do not claim formal distributional novelty. Our claim is operational: these are low-ecosystem programming targets relative to mainstream languages, and they induce large differences in how deployed coding agents adapt during a session. Appendix D reports public-code prevalence and n-gram overlap analyses showing orders-of-magnitude less public-code presence for these esolangs than for mainstream languages. Hidden tests, separate workspaces, and fixed forward sessions reduce shallow memorization but do not prove zero exposure.

Artificial but controlled proxies for testing adaptation. The esolangs are artificial, but that is what makes them useful here. They are public, fully specified, runnable, automatically graded languages with unusual syntax, execution rules, and debugging surfaces. This gives us a controlled proxy for a practical pressure that is otherwise hard to study publicly: whether an agent can build, test, and revise a working interface model when familiar language and library priors are weak. We therefore treat success on these tasks as evidence about adaptation to low-ecosystem executable interfaces, not as evidence that esolangs themselves are important production targets.

Harnesses and mechanism scope. We evaluate deployed coding agents rather than bare language models, because tool use, file editing, shell access, and workspace management are part of the systems users actually run. The tradeoff is that Claude Code, Codex, and OpenCode are not identical internally. Our protocol fixes the benchmark-facing interface: every agent receives the same problem sequence, local interpreters, hidden-test rule, and scoring criterion. Selected OpenCode re-runs preserve the qualitative ordering (Appendix B.4), making the separation unlikely to be wrapper-only. The metaprogramming claim is specific to Brainfuck and Befunge-98 where direct authoring is fragile, not Whitespace or Shakespeare.

Mechanism scope. The ablations establish the importance of generator use in the tested protocols. They do not isolate which language property determines its benefit, or separate affordance knowledge from implementation and debugging in +Lib. State naming remains a hypothesis; Appendix J.4 discusses it alongside the observed failure modes.

5 Related work

Code and agentic coding benchmarks. Execution-based grading is central to modern code evaluation, from HumanEval [Chen et al., 2021], MBPP [Austin et al., 2021], APPS [Hendrycks et al., 2021], DS-1000 [Lai et al., 2023], and MultiPL-E [Cassano et al., 2023] to LiveCodeBench [Jain et al., 2025], BigCodeBench [Zhuo et al., 2025], OJBench [Wang et al., 2025], SciCode [Tian et al., 2024], MHPP [Dai et al., 2024], and SWE-bench [Jimenez et al., 2024]. Agentic benchmarks extend this to multi-step work in repositories, terminals, desktops, browsers, and research environments [Yang et al., 2024, Xia et al., 2024, Zhang et al., 2024b, Merrill et al., 2026, Xie et al., 2024, Drouin et al., 2024, Koh et al., 2024, He et al., 2024, Starace et al., 2025, Chan et al., 2025]. These settings are realistic but mix many factors: repository navigation, dependency management, long-horizon planning, environment quirks, and familiarity with public software ecosystems. We inherit executable grading, but change the controlled variable: using EsoLang-Bench [Sharma and Chopra, 2026], we keep the tasks simple and vary the familiarity of the executable interface itself. ARC-AGI-3 [ARC Prize Foundation, 2026] is close in spirit, but stresses rule inference in unfamiliar games; we stress unfamiliar programming interfaces with explicit task specifications.

Tools, feedback, and language transfer. Prior work shows that tools, feedback, and intermediate computation can improve LLM performance [Yao et al., 2023b, Schick et al., 2023, Shinn et al., 2023, Madaan et al., 2023, Yao et al., 2023a, Gao et al., 2023, Chen et al., 2023, 2024, Nye et al., 2021], and benchmarks such as TAU-bench [Yao et al., 2025] and BFCL [Patil et al., 2025] evaluate interface following and tool use. Our question is which agents can turn local execution and workspace persistence into reliable program construction when the target language is unfamiliar. The cross-host experiment also connects to multilingual code generation and translation [Cassano et al., 2023, Paul et al., 2024, Twist et al., 2026, Roziere et al., 2020, Ahmad et al., 2021, Wang et al., 2021, Fried et al., 2023, Nijkamp et al., 2022, Li et al., 2023, Roziere et al., 2023]. Our strongest agents often do not simply generate the target language directly; they write generators in a familiar host language and treat the target language as generated output.

Metaprogramming, skills, and affordances. Zhang et al. [2024a] study metaprogramming representations for visual dataflow audio programming; Voyager builds reusable executable skills [Wang et al., 2023], and CodeAct uses executable code as agent actions [Wang et al., 2024]. Our contribution is measuring which deployed agents spontaneously discover generator construction, and how its use and transfer affect performance under a common benchmark-facing protocol. Affordance-based partial world models [Khetarpal et al., 2026] provide a related perspective on the executable operations encoded by a reference library.

Benchmark validity and long-tail coding. Benchmark-design and contamination work [Ribeiro et al., 2020, Nie et al., 2020, Ye et al., 2021, Bowman and Dahl, 2021, Oren et al., 2024, Deng et al., 2024, Xu et al., 2024] warns that high scores can mask weaknesses and that training overlap is hard to rule out for proprietary models. We therefore avoid formal OOD claims and report public-code frequency and n-gram overlap analyses. The motivation also connects to long-tail production coding, where agents face internal DSLs, generated APIs, proprietary configuration formats, and platform-specific frameworks that are sparse in public training data. Industry evaluations

such as Convex Evals [Hunt, 2025] and Fullstack-Bench [Jayakar, 2025] document failures on such platform-specific invariants. Our esolang setting makes an analogous pressure public, runnable, inspectable, and automatically graded.

Software ecosystems and the Matthew Effect. Gu et al. [2026] document performance asymmetries between mainstream and niche languages and frameworks in AI programming assistance. Our study examines how such disadvantages vary across deployed agents on a shared set of tasks, and the strategies agents use to compensate. Host-language generation offers one observed route: agents construct and debug machinery in a familiar ecosystem while emitting code for the less familiar target. The cross-host results show that this route is not specific to Python. The APL extension and its scope are reported in Appendix J.1; possible implications for host-language dependence are discussed in Appendix J.6, without treating ecosystem evolution as an experimentally measured outcome.

6 Conclusion

Unfamiliar programming languages make a normally hidden agent capability visible. When the target language is low-ecosystem, success requires more than writing code in a familiar software environment. The agent has to work out how the language behaves, test candidate programs, revise failures, and decide when a solution is ready. Because the problems are short standard tasks and every agent receives the same benchmark-facing interface, the separation reflects how well agents turn tools, feedback, and workspace state into a working strategy for the target language.

For the strongest agents, the clearest strategy is metaprogramming: host-language generators, reusable primitives, and local verification loops. The no-metaprogramming ablation shows this machinery is causally important on Brainfuck and Befunge-98, where direct authoring is long and fragile. Strategy transfer sharpens the point: written advice does little, while working helper code lifts mid-tier agents and still fails on the weakest. The key capability is not knowing that a strategy should help, but building and debugging machinery that works under the target language's rules.

Agentic generalization in this setting means reorganizing an unfamiliar problem into a form the agent can solve. The strongest agents do not only retrieve familiar patterns; they create intermediate code, tests, and reusable structure that make the target language usable. Real deployments often involve internal DSLs, generated APIs, proprietary configuration formats, and local tool conventions that are sparse in public code. Making this capability reliable in smaller or open-source agents should become a target for training, distillation, and model analysis.

Code and data. The evaluation code is available at `https://github.com/Lossfunk/esolang-metaprogramming`; the task set is at `https://huggingface.co/datasets/Lossfunk/Esolang-Bench`. The accompanying supplementary archive includes the retained-record audit and documents its coverage and limitations.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, 2021. doi: 10.18653/v1/2021.naacl-main.211. URL https://aclanthology.org/2021.naacl-main.211/.

Anthropic. Claude Opus 4.6, Claude Sonnet 4.6, and Claude Haiku 4.5: Model overview. Anthropic model overview, system cards, and release announcements, 2026. URL https://platform.claude.com/docs/en/about-claude/models/overview. Opus 4.6 system card February 2026; Sonnet 4.6 system card February 17, 2026. Accessed 2026-05-04.

ARC Prize Foundation. ARC-AGI-3: A new challenge for frontier agentic intelligence, 2026. URL https://arxiv.org/abs/2603.24621.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021. URL https://arxiv.org/abs/2108.07732.

Samuel R. Bowman and George Dahl. What will it take to fix benchmarking in natural language understanding? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4843–4855, 2021. doi: 10.18653/v1/2021.naacl-main.385. URL https://aclanthology.org/2021.naacl-main.385/.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023. doi: 10.1109/TSE.2023.3267446. URL https://doi.org/10.1109/TSE.2023.3267446.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/7e3767db483c942b883eb4f8cfb74e31-Abstract-Conference.html.

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. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, 2023. URL https://arxiv.org/abs/2211.12588. arXiv:2211.12588.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/2460396f2d0d421885997dd1612ac56b-Abstract-Conference.html.

Andy Clark and David Chalmers. The extended mind. *Analysis*, 58(1):7–19, 1998. doi: 10.1093/analys/58.1.7. URL https://doi.org/10.1093/analys/58.1.7.

Jianbo Dai, Jianqiao Lu, Yunlong Feng, Guangtao Zeng, Rongju Ruan, Ming Cheng, Dong Huang, Haochen Tan, and Zhijiang Guo. MHPP: Exploring the capabilities and limitations of language models beyond basic code generation. *arXiv preprint arXiv:2405.11430*, 2024. URL https://arxiv.org/abs/2405.11430.

Chunyuan Deng, Yilun Zhao, Xiangru Tang, Mark Gerstein, and Arman Cohan. Investigating data contamination in modern benchmarks for large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 8706–8719, 2024. doi: 10.18653/v1/2024.naacl-long.482. URL https://aclanthology.org/2024.naacl-long.482/.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. WorkArena: How capable are web agents at solving common knowledge work tasks? In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 11642–11662, 2024. URL https://proceedings.mlr.press/v235/drouin24a.html.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Evaluating large language models in class-level code generation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024. doi: 10.1145/3597503.3639219. URL https://doi.org/10.1145/3597503.3639219.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A generative model for code infilling and synthesis. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2204.05999. arXiv:2204.05999.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799, 2023. URL https://proceedings.mlr.press/v202/gao23f.html.

Fei Gu, Zi Liang, Jiahao Ma, and Hongzong Li. The matthew effect of AI programming assistants: A hidden bias in software evolution. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/6fde96479648d71e4fd9724374bf76eb-Abstract-Conference.html.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. WebVoyager: Building an end-to-end web agent with large multimodal models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6864–6890, 2024. doi: 10.18653/v1/2024.acl-long.371. URL https://aclanthology.org/2024.acl-long.371/.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*, 2021. URL https://arxiv.org/abs/2105.09938.

Jordan Hunt. Convex Evals: Behind the scenes of AI coding with Convex. Convex Stack Blog, 2025. URL https://stack.convex.dev/convex-evals. Accessed 2026-04-22.

Edwin Hutchins. *Cognition in the Wild*. MIT Press, Cambridge, MA, 1995. URL https://mitpress.mit.edu/9780262581462/cognition-in-the-wild/.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida I. 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*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/94074dd5a072d28ff75a76dabed43767-Abstract-Conference.html.

Sujay Jayakar. Introducing Fullstack-Bench. Convex Stack Blog, 2025. URL https://stack.convex.dev/introducing-fullstack-bench. Accessed 2026-04-22.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html.

Khimya Khetarpal, Gheorghe Comanici, Jonathan Richens, Jeremy Shar, Fei Xia, Laurent Orseau, Aleksandra Faust, and Doina Precup. Affordances enable partial world modeling with LLMs. *arXiv preprint arXiv:2602.10390*, 2026. URL https://arxiv.org/abs/2602.10390.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 881–905, 2024. doi: 10.18653/v1/2024.acl-long.50. URL https://aclanthology.org/2024.acl-long.50/.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida Wang, and Tao Yu. DS-1000: A natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345, 2023. URL https://proceedings.mlr.press/v202/lai23b.html.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, Joao Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, et al. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023. URL https://arxiv.org/abs/2305.06161.

Lossfunk. Lossfunk/Esolang-Bench: Hugging face dataset. Hugging Face Datasets, 2026. URL https://huggingface.co/datasets/Lossfunk/Esolang-Bench. CC BY 4.0. Accessed 2026-05-07.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026. URL https://arxiv.org/abs/2601.11868.

Moonshot AI. Kimi K2.5: Open visual agentic model for real work. Moonshot AI model documentation, 2026. URL https://www.kimi.com/ai-models/kimi-k2-5. Released January 27, 2026. Accessed 2026-05-04.

Sumeet Ramesh Motwani, Daniel Nichols, Charles London, Peggy Li, Fabio Pizzati, Acer Blake, Hasan Hammoud, Tavish McDonald, Akshat Naik, Alesia Ivanova, Vignesh Baskaran, Ivan Laptev, Ruben Glatt, Tal Ben-Nun, Philip Torr, Natasha Jaques, Ameya Prabhu, Brian Bartoldson, Bhavya Kailkhura, and Christian Schroeder de Witt. LongCoT: Benchmarking long-horizon chain-of-thought reasoning. *arXiv preprint arXiv:2604.14140*, 2026. URL https://arxiv.org/abs/2604.14140.

Yixin Nie, Adina Williams, Emily Dinan, Mohit Bansal, Jason Weston, and Douwe Kiela. Adversarial NLI: A new benchmark for natural language understanding. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4885–4901, 2020. doi: 10.18653/v1/2020.acl-main.441. URL https://aclanthology.org/2020.acl-main.441/.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022. URL https://arxiv.org/abs/2203.13474.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021. URL https://arxiv.org/abs/2112.00114.

OpenAI. GPT-5.4 thinking system card. OpenAI system card, March 2026, 2026a. URL https://openai.com/index/gpt-5-4-thinking-system-card/. Released March 5, 2026. Accessed 2026-05-04.

OpenAI. Introducing GPT-5.4 mini and nano. OpenAI release announcement, 2026b. URL https://openai.com/index/introducing-gpt-5-4-mini-and-nano/. Accessed 2026-05-04.

Yonatan Oren, Nicole Meister, Niladri Chatterji, Faisal Ladhak, and Tatsunori B. Hashimoto. Proving test set contamination in black-box language models. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/46e624c244cff669223d488defd4e835-Abstract-Conference.html.

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 48371–48392, 2025. URL https://proceedings.mlr.press/v267/patil25a.html.

Indraneil Paul, Goran Glavaš, and Iryna Gurevych. IRCoder: Intermediate representations make language models robust multilingual code generators. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15023–15041, 2024. doi: 10.18653/v1/2024.acl-long.802. URL https://aclanthology.org/2024.acl-long.802/.

Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4902–4912, 2020. doi: 10.18653/v1/2020.acl-main.442. URL https://aclanthology.org/2020.acl-main.442/.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*, volume 33, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jeremy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Defossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023. URL https://arxiv.org/abs/2308.12950.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.

Aman Sharma and Paras Chopra. EsoLang-Bench: Evaluating genuine reasoning in large language models via esoteric programming languages. *arXiv preprint arXiv:2603.09678*, 2026. URL https://arxiv.org/abs/2603.09678.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. PaperBench: Evaluating AI's ability to replicate AI research. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 56843–56873, 2025. URL https://proceedings.mlr.press/v267/starace25a.html.

Minyang Tian, Luyu Gao, Shizhuo Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, Shengyan Liu, Di Luo, Yutao Ma, Hao Tong, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Yanyu Xiong, Shengzhu Yin, Minhui Zhu, Kilian Lieret, Yanxin Lu, Genglin Liu, Yufeng Du, Tianhua Tao, Ofir Press, Jamie Callan, Eliu Huerta, and Hao Peng. SciCode: A research coding benchmark curated by scientists. *arXiv preprint arXiv:2407.13168*, 2024. URL https://arxiv.org/abs/2407.13168.

Lukas Twist, Mark Harman, Don Syme, Joost Noppen, Helen Yannakoudakis, Detlef Nauck, and Jie M. Zhang. A study of llms' preferences for libraries and programming languages. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 331–351, 2026. doi: 10.18653/v1/2026.findings-acl.15. URL https://aclanthology.org/2026.findings-acl.15/.

Vals AI. LiveCodeBench v6: Vals.ai public leaderboard. Vals.ai third-party model evaluation leaderboard, 2026a. URL https://www.vals.ai/benchmarks/lcb. Accessed 2026-05-06.

Vals AI. SWE-Bench Verified: Vals.ai public leaderboard (bash-tool-only harness). Vals.ai third-party model evaluation leaderboard, 2026b. URL https://www.vals.ai/benchmarks/swebench. Used for GPT-5.4 mini and GPT-5.4 xhigh SWE-Bench Verified scores because OpenAI does not publish vendor SWE-V numbers for the GPT-5.4 family. Vals.ai bash-tool-only harness scores: 73.0 for GPT-5.4 mini, 78.2 for GPT-5.4 xhigh. Accessed 2026-05-06.

Vals AI. Terminal-Bench 2.0: Vals.ai public leaderboard. Vals.ai third-party model evaluation leaderboard, 2026c. URL https://www.vals.ai/benchmarks/terminal-bench-2. Accessed 2026-05-06.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023. URL https://arxiv.org/abs/2305.16291.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 50208–50232, 2024. URL https://proceedings.mlr.press/v235/wang24h.html.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.org/2021.emnlp-main.685/.

Zhexu Wang, Yiping Liu, Yejie Wang, Wenyang He, Bofei Gao, Muxi Diao, Yanxu Chen, Kelin Fu, Flood Sung, Zhilin Yang, Tianyu Liu, and Weiran Xu. OJBench: A competition level code benchmark for large language models. *arXiv preprint arXiv:2506.16395*, 2025. URL https://arxiv.org/abs/2506.16395.

Edwin B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927. doi: 10.1080/01621459.1927.10502953. URL https://doi.org/10.1080/01621459.1927.10502953.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024. URL https://arxiv.org/abs/2407.01489.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5d413e48f84dc61244b6be550f1cd8f5-Abstract-Datasets_and_Benchmarks_Track.html. Datasets and Benchmarks Track.

Cheng Xu, Shuhao Guan, Derek Greene, and M. Tahar Kechadi. Benchmark data contamination of large language models: A survey. *arXiv preprint arXiv:2406.04244*, 2024. URL https://arxiv.org/abs/2406.04244.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*, volume 36, 2023a. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/271db9922b8d1f4dd7aaef84ed5ac703-Abstract-Conference.html.

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*, 2023b. URL https://arxiv.org/abs/2210.03629.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R. Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/1b126cc38b8638e07bef37e7b2bb72bf-Abstract-Conference.html.

Qinyuan Ye, Bill Yuchen Lin, and Xiang Ren. CrossFit: A few-shot learning challenge for cross-task generalization in NLP. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7163–7189, 2021. doi: 10.18653/v1/2021.emnlp-main.572. URL https://aclanthology.org/2021.emnlp-main.572/.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. CoderEval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024. URL https://arxiv.org/abs/2302.00288. arXiv:2302.00288.

William Zhang, Maria Leon, Ryan Xu, Adrian Cardenas, Amelia Wissink, Hanna Martin, Maya Srikanth, Kaya Dorogi, Christian Valadez, Pedro Perez, Citlalli Grijalva, Corey Zhang, and Mark Santolucito. Benchmarking LLM code generation for audio programming with visual dataflow languages. *arXiv preprint arXiv:2409.00856*, 2024a. URL https://arxiv.org/abs/2409.00856.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024b. doi: 10.1145/3650212.3680384. URL https://doi.org/10.1145/3650212.3680384.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm

de Vries, and Leandro von Werra. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a6a90bcc2aa470c3871b2d39a67d26e8-Abstract-Conference.html.

A Experimental details

A.1 Problem order and task structure

Each esolang benchmark contains 80 tasks in a fixed fetch order (20 easy, 20 medium, 20 hard, 20 extra-hard). Under the primary protocol used for every main-text result, problems are fetched and finalized in that fixed forward order: a problem is finalized either when one hidden submission passes all six private tests or when the three-submission cap is reached, after which the session advances to the next problem and finalized problems are not revisited (Figure 2 of the body). Each run has a separate workspace, and prompts prohibit reading sibling experiment folders or other runs' artifacts. These instructions alone do not establish operating-system-level isolation (Appendix A.11).

A.2 Task substrate details

The four languages we evaluate are drawn from EsoLang-Bench [Sharma and Chopra, 2026], which releases task statements, hidden tests, problem identifiers, difficulty tiers, and reference interpreters under a single specification. The task statements, hidden tests, and metadata used in this paper are loaded from the Lossfunk EsoLang-Bench Hugging Face dataset [Lossfunk, 2026]; we reproduce only a brief characterization of each language here so the appendix is self-contained; the canonical problem list, per-tier difficulty labels, and reference interpreters are documented in the EsoLang-Bench release. The supplementary archive ships the four interpreter Python sources used by the harness at `supplementary_code/benchmark_harness/interpreters/` (one file per language: Brainfuck, Befunge-98, Whitespace, Shakespeare), and the public problem statements at `benchmark_harness/public/esolang_full_public.json`.

Brainfuck. Eight-symbol minimal pointer machine over an unbounded tape of unsigned bytes. `>` `<` move the pointer, `+` `-` mutate the cell with byte wrap, `.` `,` do byte-level I/O, and `[` `]` form a conditional loop on the current cell. There are no variables or named functions; numeric I/O must be implemented digit by digit through ASCII conversion. Programs in this language tend to be long, fragile pointer-arithmetic sequences.

Befunge-98. Two-dimensional stack-based language whose instruction pointer moves over a grid of one-character commands and can be redirected by direction commands (`>` `<` `^` `v`), with stack arithmetic, string mode, an end token (`@`), and grid get / put primitives. The 2D control flow makes program structure positional rather than sequential, and even small grid edits can change which path the instruction pointer takes.

Whitespace. An assembly-style language whose lexicon consists only of the three characters Space, Tab, and Linefeed. Numbers are encoded as a sign bit followed by binary digits; control flow uses labels and jumps. Because the source is invisible to a human reader, programs must be produced by a generator script: any direct edit through standard text-editing tools tends to silently corrupt the program by altering whitespace.

Shakespeare. A natural-language shaped programming language in which programs are written as theatrical scripts with named characters and dialogue between exactly two on-stage characters at a time. Statements set, read, print, and stack the listener's value, expressions are built from positive and negative nouns scaled by adjectives, and Roman-numeral scenes act as jump targets. The syntax constraints are stylistic rather than minimal, but the interpretive rules around speaker, listener, and stage state are unfamiliar to typical software training data. EsoLang-Bench's fifth language, Unlambda, is excluded due to interpreter latency (Section 2); the canonical problem list, hidden-test contracts, and reference interpreters for the four languages we keep are released by EsoLang-Bench.

Task set and tier structure. Each language ships 80 problems split into four difficulty tiers of 20 each: easy (`E01`–`E20`), medium (`M01`–`M20`), hard (`H01`–`H20`), and extra-hard (`X01`–`X20`). The same

problem statements appear in every language, so cross-language differences come from the target syntax and execution model rather than from problem selection. The problems are short standard programming tasks that an introductory programming course could pose; the difficulty in our setting comes entirely from expressing them in an unfamiliar target language. The easy tier covers I/O and one-step arithmetic (echo a line, sum or multiply two integers, output a constant string, character-by-character echo). The medium tier covers list and string operations (sort a list of integers, count vowels, compute the length of a string, parity / odd–even check, integer absolute value, formatted multiplication tables, simple counters and accumulators). The hard tier introduces multi-step numeric manipulation (greatest common divisor, primality test, integer division and modulo, Fibonacci, leading-zero suppression, signed arithmetic with both inputs negative). The extra-hard tier exercises combined control flow and data manipulation (least common multiple, bracket-depth maximum, count inversions in a list, Roman-to-integer conversion, base conversions, signed average / halving, string rotation checks). EsoLang-Bench's own paper reports near-ceiling zero-shot and few-shot accuracy on these same statements when models answer in Python or JavaScript, which is the basis for our claim that the difficulty in this benchmark comes from the target language rather than the algorithmic problem. The full per-problem statements ship in the supplementary archive at `supplementary_code/benchmark_harness/public/esolang_full_public.json`.

A.3 Harness commands

The harness exposes a small command interface. `fetch` reveals the next problem statement. `run` executes a candidate program locally with a provided interpreter or verifier and user-specified input. `submit` evaluates the candidate against private tests and updates the problem state. `status` prints a progress dashboard and `export` dumps the full per-cell session state as JSON. Agents can edit local files and run local helper scripts inside their workspace. The full implementation is the single file `supplementary_code/benchmark_harness/harness.py` in the supplementary archive; the constant `MAX_SUBMISSIONS = 3` encodes the hidden-submission cap and is enforced inside `harness.py` rather than by the agent wrapper, so it cannot be bypassed.

Local execution limits. The four reference interpreters (`supplementary_code/benchmark_harness/interpreters/`) terminate non-halting candidate programs by capping the number of executed interpreter steps at `MAX_INTERPRETER_STEPS` $= 10^7$ instructions per local `run` (Brainfuck, Befunge-98, Whitespace, Shakespeare). Programs that exceed the cap return a `StepLimitExceeded` runtime error rather than hanging the session, which is important for Brainfuck and Befunge-98 where infinite loops are easy to author. The same cap applies inside hidden-test `submit` evaluation. The cap is generous enough that none of the headline solutions reported in the paper hit it; it functions as a watchdog, not as a difficulty knob.

A.4 Primary protocol

The protocol used for every result in the main body allows unlimited local interpreter calls and up to three hidden submissions per problem. The agent decides when it has converged on the local interpreter and then spends a hidden submission. The score is the number of problems solved out of 80. The full per-problem state machine is in Figure 2 of the body, and the operating parameters are summarized in Table 4.

A.5 Per-agent API endpoints, model identifiers, and harness invocations

Table 5 lists the API endpoint, model identifier, sampling configuration, and wrapper used for each of the six agents in the headline runs. We do not override sampling at the API call level: every agent runs at the wrapper's default temperature, top-p, and top-k settings, so the per-cell session-to-session variation we report comes from independent re-invocations of the wrapper rather than from explicit sampling changes. Example CLI invocations: `claude -model claude-opus-4-6` (Claude Code), `codex -model gpt-5.4 -reasoning xhigh` (Codex), `opencoderun--modelmoonshot/kimi-k2-thinking` (OpenCode); the exact wrapper versions, environment variables, and per-cell invocations used for the headline runs ship in the `supplementary_code/` archive.

Table 4: **Primary protocol parameters.** All main-text results use this single configuration unless explicitly stated otherwise (the diagnostic variants in Section 3.3–3.5 relax or tighten individual rows but leave hidden tests, scoring, and workspace isolation unchanged).

Item	Value
Languages	Brainfuck, Befunge-98, Whitespace, Shakespeare
Problems per language	80 (20 easy, 20 medium, 20 hard, 20 extra-hard)
Problem order	fixed forward; finalized problems not revisited
Hidden tests per problem	6 (private; agent never sees inputs or expected outputs)
Hidden submissions per problem	up to 3 (`MAX_SUBMISSIONS` = 3 in `harness.py`)
Local interpreter calls	unlimited (per-call step cap 10^7 instructions)
Output budget per assistant turn	32,000 tokens
Sampling settings	provider / wrapper defaults (temperature, top-p, top-k)
Workspace	separate per model×language run; sibling-folder reads prohibited by prompt
Submission feedback	aggregate `passed/total` only; no per-test diagnostics
Scoring rule	solved iff one submission returns $6/6$ on the hidden tests
Aggregation across runs	headline cell = Session 1; two further sessions per cell as sanity checks (Appendix B.7); ablation cells = mean over two sessions (Appendix A.7)
Uncertainty reporting	Wilson 95% binomial CI [Wilson, 1927] on $k/80$

Table 5: Per-agent API endpoint, model identifier, sampling configuration, and harness invocation. "Wrapper default" indicates that the agentic wrapper sets temperature, top-p, and top-k to its built-in defaults; we do not override at the API call level. The model strings are the production identifiers used by the listed wrapper at the time of the headline runs.

Agent	Wrapper	Endpoint	Model identifier	Sampling	Reasoning effort
Claude Opus 4.6	Claude Code	Anthropic API	`claude-opus-4-6`	wrapper default	—
Claude Sonnet 4.6	Claude Code	Anthropic API	`claude-sonnet-4-6`	wrapper default	—
Claude Haiku 4.5	Claude Code	Anthropic API	`claude-haiku-4-5`	wrapper default	—
GPT-5.4 xhigh	Codex	OpenAI API	`gpt-5.4`	wrapper default	xhigh
GPT-5.4 mini	Codex	OpenAI API	`gpt-5.4-mini`	wrapper default	medium
Kimi K2.5	OpenCode	Moonshot API	`kimi-k2-thinking`	wrapper default	—

A.6 Reporting and aggregation

A problem is counted as solved only when all six private hidden tests pass on a single submission. We do not aggregate per-test pass counts across submissions, because partial-credit aggregation can inflate the appearance of progress for runs whose submissions never fully pass. A problem with three failed hidden submissions is counted as 0, not as the best per-test count across the three. Cells that have not yet completed all 80 problems are explicitly marked partial and excluded from matched comparisons rather than imputed as zeros. Wilson 95% confidence intervals over solved counts are reported alongside the headline cells in Section B.6.

A.7 Robustness across independent sessions

What a run means in this paper. We run *independent sessions* of the underlying coding harness rather than seeded re-inferences. Each headline model×language cell in Table 1 is the solved count from Session 1; we additionally ran two further independent sessions per cell as session-to-session sanity checks (per-session counts in Appendix B.7). Each intervention cell (Section 3.3, Section 3.4, and Section 3.5) is the mean over two independent sessions. A session here is one fresh end-to-end invocation of the agent's deployed CLI (Claude Code for the Anthropic family, Codex for the GPT-5.4 family, OpenCode for Kimi K2.5) on a freshly initialized workspace, with no state shared across sessions.

Sessions, not seeds. The harnesses we evaluate are deployed CLI products and do not expose a deterministic seed at the inference layer; the underlying provider APIs do not return reproducible token sequences across requests at the default temperatures and sampling configurations these CLIs

use. Two re-runs of the same model on the same problem set therefore differ at the token level even with the same prompts, harness, and protocol parameters. This captures more variation than a seeded re-run inside a single inference call, because the spread also reflects session-level choices: which helper file the agent writes first, whether it tries direct target-language authoring before reaching for metaprogramming, how aggressively it batches problems within the session, and which debug-and-revise loops it falls into. We therefore use the terms *run* and *session* consistently throughout the paper to denote one such independent end-to-end CLI invocation, not a seeded RNG call inside a single request.

Why three sessions per headline cell and two per ablation cell. For headline cells in Table 1, we report Session 1 as the headline value and use Sessions 2 and 3 as session-to-session sanity checks; per-session counts for all three sessions are in Appendix B.7. Ablation cells are paired contrasts on top of an established headline (metaprogramming-allowed vs. direct authoring in Section 3.3, with vs. without a reference library in Section 3.4, and varying interpreter-call budgets in Section 3.5); the relevant signal is the within-agent, within-language shift induced by the intervention rather than the absolute score, so we ran two independent sessions per ablation cell and report the mean. In every reported ablation, the qualitative direction of the intervention is consistent across both sessions.

What this means for the headline claims. The inter-agent separations that drive Section 3.1 and Section 3.3 of the body are tens of problems out of 80 (for example 79 vs. 4 on Brainfuck and 80 vs. 4 on Befunge-98). Observed session-to-session variation across the three sessions per headline cell is small relative to these gaps (Appendix B.7), so the agent ordering reported in the body is stable under independent re-runs. The per-cell binomial Wilson interval on the solved-out-of-80 counts is in Appendix B.6.

A.8 Controlled-access (fixed-budget) protocol

For a stricter reproducibility-oriented control we also ran a controlled-access protocol that caps local runs and hidden submissions explicitly. Each problem allows at most three local interpreter or verifier calls before a single hidden submission is unlocked (the agent may submit sooner if it is confident, but cannot make a fourth local run before submitting), and at most one hidden submission per problem. Hidden tests, scoring rule, problem order, and workspace isolation are unchanged from the primary protocol; only the local-run and submission caps are tightened. The qualitative agent ordering under this stricter protocol matches the primary protocol; absolute scores are lower across the board because frontier agents lose access to the iterative repair loop they normally use. Per-cell numbers are recorded in the supplementary CSV.

A.9 Interpreter-budget ablation

The interpreter-budget ablation (Section 3.5, Figure 4 of the body) caps the number of local interpreter calls per problem at $\{3, 5, 15, 30, \infty\}$ while holding the task substrate, hidden submissions, and scoring rule fixed. The full per-budget solved counts on both Brainfuck and Befunge-98 are the values plotted in body Figure 4; the unlimited-budget endpoints match the headline cells in Table 1 of the body (64 for Opus 4.6 on each of Brainfuck and Befunge-98, 12 for Sonnet 4.6 on Brainfuck and 64 on Befunge-98, 4 for Haiku 4.5 on each). The qualitative pattern reported in the body is that Opus 4.6 gains substantially as the budget grows on both languages and Sonnet 4.6 gains on Befunge-98, while Haiku 4.5 stays near the floor at every budget. Per-cell raw counts are emitted by the per-cell `export.json` files under `supplementary_code/experiments/01_main_experiments/` and its budget-restricted variants.

A.10 Metaprogramming constraints

In metaprogramming-allowed runs, the agent may write generator programs in a familiar language that emit the target esolang. In no-meta runs, the agent must write the target language directly. Cross-language generator runs constrain the generator language to Python, JavaScript, or Rust depending on the cell.

A.11 Isolation

Each run receives its own workspace. Agents are instructed not to inspect prior runs, sibling language folders, hidden tests, solved artifacts, or transcripts. This boundary is important because the benchmark is sequential: later problems may reuse notes and primitives developed earlier in the same run, but not artifacts from other runs. Prompt restrictions are not a sandbox guarantee. The retained trace audit flagged cross-workspace reads in some auxiliary Codex sessions, including the bounded APL mini control; those flags must be considered when interpreting retained diagnostics. They do not identify the exact snapshots of every separately reported headline or follow-up session.

A.12 System prompts and per-condition configuration

Each cell of the paper runs under one of three agentic harnesses (Claude Code, Codex, OpenCode), each of which automatically reads a project-level instruction file from the workspace at session start (`CLAUDE.md` for Claude Code, `AGENTS.md` for Codex and OpenCode) on top of its own internal default system prompt. We ship one such instruction file per language; its content is what the harness adds to its native default. The four prompts reproduced verbatim in the subsections below are the exact files shipped in the supplementary archive at `supplementary_code/prompts/<lang>/` (symlinked into every primary-protocol cell at `experiments/01_main_experiments/<harness>/<model>/<lang>/`). The harness's own internal default (turn-taking, tool-use formatting, file-editing rules) is left unchanged. We refer to the file we ship as the *language-reference prompt*.

The language-reference prompt has the same structure for every esolang: a one-line "start solving now" directive, the benchmark task description, the harness command list (`fetch`, `run`, `submit`, `status`, `skip`), the scoring rule (a problem is solved only if all six private hidden tests pass), the operating limits (up to three hidden submissions per problem, unlimited `run` calls, unlimited time, no skipping with submissions remaining), and a language-specific reference card. This single language-reference prompt is identical across the three Claude agents (Opus 4.6, Sonnet 4.6, Haiku 4.5) within a given language; for Codex and OpenCode we use the same content via the `AGENTS.md` mechanism. The four primary-protocol prompts are reproduced verbatim in Sections A.12.1–A.12.4 below.

Per-condition deviations from this primary-protocol prompt are as follows.

Primary protocol and main results (Section 3). The language-reference prompt described above (verbatim text in Sections A.12.1–A.12.4). No further additions. The agent receives each problem statement through `fetch`; everything else is handled by the harness.

Metaprogramming-allowed (Section 3.3). Same as the primary protocol above (the language-reference prompt is loaded as `CLAUDE.md` or `AGENTS.md`, on top of the harness's native default). The agent freely chooses whether to author the target esolang directly or via a host-language generator; no additional preamble is added.

No-meta direct authoring (Section 3.3). The harness removes the agent's bash access and any tool that could execute a host-language generator, and a short paper-authored preamble is added to the system prompt instructing the agent to author the target esolang directly. The preamble lists the file extensions accepted by the harness for that cell (e.g., `.bf` only) and explicitly notes that any generator script in the workspace at submission time will cause submission failure. Together, the bash removal and the prompt preamble prevent any host-language generation pathway.

Cross-language generator transfer (Section 3.3). A short preamble specifies which host language is permitted (Python, JavaScript, or Rust per cell) and lists the file extensions accepted for the generator. All other harness behavior is unchanged.

Distillation: text-only preamble (Section 3.4, condition (i)). A condensed prose version of Opus 4.6's strategies is preloaded as the system prompt. The full text is reproduced in Appendix G.1.

Distillation: reference library (Section 3.4, condition (ii)). The condition (i) preamble plus a `reference_lib/` directory containing code and notes from strong-agent traces (Claude Opus 4.6 and GPT-5.4 xhigh). The retained directories contain both reusable primitives and task-specific generator examples. The preamble points to the directory and its `README.md`. Appendix G.2 describes these files; Appendix J.5 reports their overlap with passing submissions and the limits of that check.

Interpreter-budget ablation (Section 3.5). Harness defaults plus a hard cap on local interpreter calls per problem (3, 5, 15, 30, or unlimited). The cap is enforced by the harness rather than the prompt.

Token-efficiency analysis (Section 3.5). Harness defaults; the cell is read out of the primary-protocol runs by counting API output tokens (including extended thinking) on E01–E20 only.

A.12.1 Primary-protocol language-reference prompt: Brainfuck

````
> **Start solving now.** The session is initialized. Begin by fetching your first problem:
> ```
> python harness.py fetch
> ```
> Work through all 80 problems sequentially. For each problem: read
> the description, generate your own test cases, use `run` to test,
> then `submit` when confident. Do NOT skip without attempting --
> see skip policy below.

# EsoLang-Bench: Brainfuck

You are solving 80 brainfuck programming problems sequentially.
Problems span 4 difficulty tiers: E01-E20 (easy), M01-M20 (medium),
H01-H20 (hard), X01-X20 (extra-hard).

Each problem has 6 hidden test cases you cannot see. No test cases
are provided -- generate your own test inputs from the problem
descriptions.

## Harness Commands

```bash
python harness.py fetch                          # Get next problem
python harness.py run <code_file> --input "..."  # Test with custom input (unlimited)
python harness.py submit <problem_id> <file>     # Hidden evaluation (max 3 per problem)
python harness.py status                         # Progress dashboard
python harness.py skip                           # Skip ONLY after attempting (see policy)
```

## Rules

- Solve problems in order. `fetch` advances to the next.
- Maximum 3 submissions per problem. Use `run` (unlimited) to test first.
- `submit` returns PASS/WRONG ANSWER/RUNTIME ERROR per hidden test
  -- no inputs or expected outputs shown.
- Problem is solved only if 6/6 hidden tests pass.
- Do not read private test files, reverse-engineer the harness, or
  hardcode outputs.

## CRITICAL: No Skipping -- Use All 3 Submissions

- You MUST work on every problem until you either score 6/6 OR use
  all 3 submissions. Do NOT move to the next problem with unused
  submissions remaining.
- You CANNOT revisit problems later -- once you move on, the problem
  is gone forever. Every unused submission is wasted.
- There is NO time limit. You have UNLIMITED time and UNLIMITED test
  runs (`run` command). The ONLY limit is 3 submissions per problem.
- For each problem: read the description, write a genuine solution,
  test it thoroughly with `run` using multiple inputs until it
  produces correct output, THEN submit.
- Do NOT submit untested code. Do NOT submit placeholder or stub
  solutions. Every submission must be a genuine, tested attempt.
- If a submission returns WRONG ANSWER or RUNTIME ERROR: analyze the
  failure, try a completely different approach, test it, then use
  your next submission.
- Even if a problem seems extremely difficult, you must use all 3
  submissions with genuinely different approaches before moving on.
  A partial score (1-5/6) is far better than 0/6.
````

```
- Do NOT call `fetch` or `skip` to move to the next problem while
  you still have submissions remaining on the current one.

## Scoring

- Per problem: tests passed / 6
- Overall: problems solved / 80 and total tests passed / 480

## Brainfuck Language Reference

### Commands

| Cmd | Description |
|-----|-------------|
| `>` | Move pointer right |
| `<` | Move pointer left |
| `+` | Increment cell (wraps 255->0) |
| `-` | Decrement cell (wraps 0->255) |
| `.` | Output cell as ASCII character |
| `,` | Read one byte into cell (0 if EOF) |
| `[` | Jump past matching `]` if cell is 0 |
| `]` | Jump back to matching `[` if cell nonzero |

All other characters are comments.

### Memory Model

- Unbounded tape of unsigned byte cells (0-255), initialized to 0.
- Pointer starts at cell 0; cannot go below 0 (runtime error).

### Essential Patterns

- Zero a cell:        [-]
- Move (destructive): [->+<] (cell 0 to cell 1)
- Copy:               [->+>+<<]>>[-<<+>>]
- Add cell 1 into 0:  >[-<+>]<
- Subtract 1 from 0:  >[-<->]<
- Read loop:          ,[...,] reads until EOF (0)
- Print decimal:      Divide by 10, store remainders, print in
                      reverse + ASCII 48

### ASCII Quick Reference

`0`=48, `9`=57, `A`=65, `Z`=90, `a`=97, `z`=122, space=32,
newline=10, `-`=45

### Common Pitfalls

- Number I/O: Input arrives as ASCII chars ('5'=53), not raw
  numbers. Output must also be ASCII digits. This is the #1 error
  source.
- Forgetting to zero cells before reuse.
- Cell overflow: 255+1=0, 0-1=255. Can cause infinite loops.
- Pointer tracking: Always keep a written map of which cell holds
  what.
- Multi-digit numbers: Parsing and printing require digit-by-digit
  handling.
```

A.12.2 Primary-protocol language-reference prompt: Befunge-98

The header, harness commands, rules, no-skipping policy, and scoring section are identical to the Brainfuck prompt above (with "brainfuck" replaced by "befunge-98"). Only the language reference card differs and is reproduced below.

```
## Befunge-98 Language Reference

### Program Structure
```

```
2D grid. Instruction pointer starts at (0,0) moving right. Wraps at
edges. Ends at `@`.

### Instructions

Stack:    0-9 push digit; a-f push 10-15; " toggle string mode;
          : dup; \ swap; $ pop; n clear stack
Arith:    + - * / %  (pop b, pop a, push a op b)
Compare:  ! (NOT); ` (greater-than)
Direction: > < ^ v ?(random) [(turn left) ](turn right) r(reverse)
           #(skip next) j(jump n)
Branch:   _ (right if 0, left if nonzero); | (down if 0, up if
          nonzero)
I/O:      . print int+space; , print char; & read int;
          ~ read char (-1 at EOF)
Grid:     g get; p put; s store next; ' fetch next
Flow:     @ end; q quit; ; comment toggle; space=nop

### Essential Patterns

- Print string: "!dlroW olleH">:#,_@ (push reversed, loop print)
- Print loop:   >:#,_@ (dup, skip print if 0, print char, repeat)
- Read int:     & (built-in)
- Conditional:  !#v_ (branch on value)

### Common Pitfalls

- `.` outputs number + space. For clean output, convert to digit
  chars and use `,`.
- Stack order: 52- = 5-2=3, not 2-5.
- String mode: "abc" pushes a,b,c in order; they pop as c,b,a.
- Missing `@`: IP wraps and re-executes, hitting step limit.
- `~` returns -1 at EOF, not 0.
```

A.12.3 Primary-protocol language-reference prompt: Whitespace

The header, harness commands, rules, no-skipping policy, and scoring section are identical to the Brainfuck prompt above (with "brainfuck" replaced by "whitespace"). Only the language reference card differs and is reproduced below.

```
## Whitespace Language Reference

### Encoding

Programs use only: Space (S, ASCII 32), Tab (T, ASCII 9), Linefeed
(L, ASCII 10). All other chars are ignored.

### Instruction Encoding (selected)

Stack:    SS<num>  push number; SLS dup; SLT swap; SLL discard
Arith:    TSSS add; TSST sub; TSSL mul; TSTS div; TSTT mod
Heap:     TTS store; TTT retrieve
I/O:      TLSS out_char; TLST out_num; TLTS read_char; TLTT read_num
Flow:     LSS<lbl> label; LST<lbl> call; LSL<lbl> jump;
          LTS<lbl> jz; LTT<lbl> jn (jump if negative);
          LTL ret; LLL end

### Key Concepts

- Number encoding: Sign bit (S=+, T=-) + binary digits
  (S=0, T=1) + L terminator.
- Heap-based I/O: read_char/read_num store to heap at a popped
  address, not the stack. Push address first, then read, then
  push address + retrieve to get value onto stack.
- Labels: Sequences of S and T terminated by L. Must be unique.
```

```
### Common Pitfalls

- Heap I/O: Read operations store to heap, not stack. Must
  retrieve afterward.
- Sign bit required even for positive numbers.
- Always include the end instruction (LLL) or execution runs
  off the end.
- Stack order: sub pops b then a, computes a-b.
```

A.12.4 Primary-protocol language-reference prompt: Shakespeare

The header, harness commands, rules, no-skipping policy, and scoring section are identical to the Brainfuck prompt above (with "brainfuck" replaced by "shakespeare"). Only the language reference card differs and is reproduced below.

```
## Shakespeare Language Reference

### Program Structure

Title.
Character, description.
Act I: Description.
Scene I: Description.
[Enter Character1 and Character2]
Character1:
Statement.

### Value System

- Positive nouns (+1): angel, cat, day, flower, hero, joy, king,
  rose, summer, sun
- Negative nouns (-1): bastard, beast, coward, death, devil,
  famine, hell, pig, plague
- Zero: nothing, zero
- Each adjective doubles: "a big cat"=2, "a big big cat"=4,
  "a big big big cat"=8
- Arithmetic: the sum of X and Y, the difference between X and Y,
  the product of X and Y, the quotient between X and Y, the
  remainder of the quotient between X and Y, the square of X,
  twice X
- Pronouns: you / thou / thee = listener's value;
            I / me           = speaker's value

### Statements (all target the LISTENER)

- Assign:        You are EXPR. / Thou art EXPR.
- Output char:   Speak your mind.        (listener's value as char)
- Output number: Open your heart.        (listener's value as int)
- Read char:     Open your mind.         (-1 at EOF)
- Read int:      Listen to your heart.
- Stack push:    Remember EXPR.          (onto listener's stack)
- Stack pop:     Recall.                 (pop listener's stack into
                                          listener's value)

### Comparisons and Flow Control

- Am I better than EXPR?  (speaker > EXPR)
- Am I worse than EXPR?   (speaker < EXPR)
- Am I as good as EXPR?   (speaker == EXPR)
- If so, let us proceed to Scene X.   (jump if true)
- If not, let us proceed to Scene X.  (jump if false)
- Let us proceed to Scene X.          (unconditional jump)

### Stage Rules

- Exactly 2 characters on stage for dialogue.
- [Enter X and Y], [Exit X], [Exeunt] (remove all)
```

```
- Scene labels use Roman numerals and must be globally unique.

### Common Pitfalls

- I/O targets the LISTENER, not the speaker. This is the #1 bug
  source.
- No direct numbers: "You are 72." is INVALID. Use adjective-noun
  expressions.
- Stage management: Must have exactly 2 characters on stage.
- Roman numeral scenes must be globally unique across all Acts.
```

B Additional results

Every per-cell number reported in this appendix is reproducible from the per-cell `export.json` files emitted by `python harness.py export`; the supplementary archive ships the harness, the 48 ready-to-run cell directories, and a rigorous end-to-end test (`scripts/rigorous_test.sh`) that exercises the export path without requiring any provider API key.

B.1 Terminal-Bench 2.0 and SWE-Bench Verified vs EsoLang-Bench scatter (cliff visualization)

Figure 6 below visualizes two columns of Table 2 of the body, Terminal-Bench 2.0 and SWE-Bench Verified, against each agent's EsoLang-Bench mean score. The shaded vertical band in each panel is the agent cluster on the mainstream benchmark (33.3 pp wide on Terminal-Bench 2.0 and 6.6 pp wide on SWE-Bench Verified), while the vertical extent of the markers shows the 88.4-pt EsoLang-Bench spread of the same six agents.

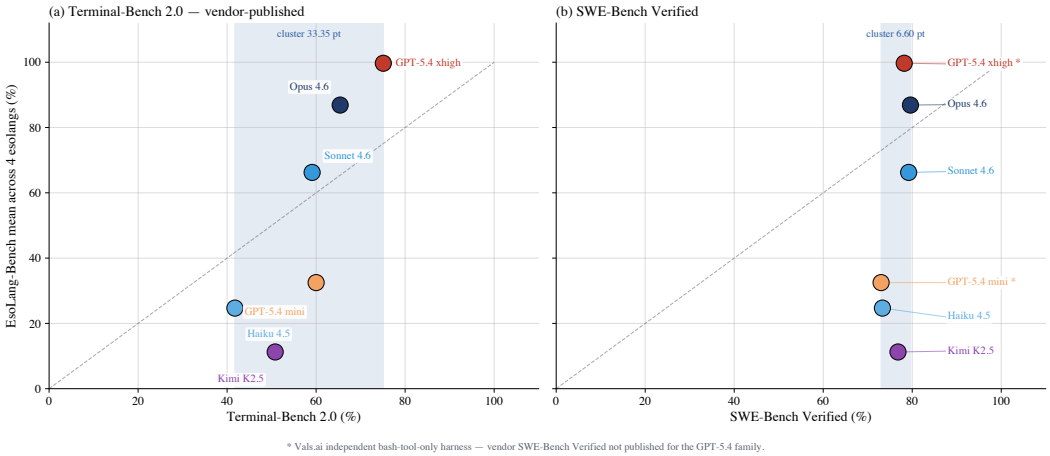

Figure 6: **Mainstream coding scores cluster while unfamiliar-language scores separate, on both Terminal-Bench 2.0 and SWE-Bench Verified.** Each marker is one of the six evaluated coding agents. (a) Terminal-Bench 2.0 (vendor-published) on the x-axis; mean EsoLang-Bench score under our protocol on the y-axis; shaded vertical band is the 33.3-pt TB-2.0 cluster. (b) SWE-Bench Verified on the x-axis with the same y-axis; shaded vertical band is the 6.6-pt SWE-V cluster. Asterisks mark Vals.ai bash-tool-only-harness numbers used where vendor SWE-V scores are not published for the GPT-5.4 family.

B.2 Main results in raw problems-solved counts

Table 1 in the body reports the four esolang columns as percentages out of 80 problems per language for readability. Table 7 reproduces the same six-agent results in the underlying problems-solved-out-of-80 format.

Mainstream-benchmark sourcing. The SWE-Bench Verified and Terminal-Bench 2.0 rows of Table 2 use vendor-published numbers wherever those exist. We did not re-run SWE-Bench Verified

or Terminal-Bench 2.0 ourselves for the headline numbers; every cell in those two rows is sourced from the public reports listed in Table 6 below. Where no vendor SWE-V score is published, we use the Vals.ai third-party leaderboard [Vals AI, 2026b], which evaluates SWE-Bench Verified under a published bash-tool-only harness; this applies only to the GPT-5.4 family. As a sanity check on the third-party numbers, we additionally re-ran SWE-Bench Verified on GPT-5.4 mini and GPT-5.4 xhigh under our own harness and recovered scores within a few points of the Vals.ai cells, so the headline values are stable under independent replication; we use the Vals.ai numbers in the table for transparency about the source.

Table 6: **Per-agent sourcing for the SWE-Bench Verified and Terminal-Bench 2.0 rows of Table 2.** Every cell is a public report from the listed source; we did not re-score any of the mainstream cells ourselves for the headline tables. The Vals.ai entries are used only because OpenAI does not publish vendor SWE-V numbers for the GPT-5.4 family.

Agent	SWE-V (%)	TB-2.0 (%)	Source
Claude Opus 4.6	79.6	65.4	Anthropic system card / model overview [Anthropic, 2026]
Claude Sonnet 4.6	79.2	59.1	Anthropic system card / model overview [Anthropic, 2026]
Claude Haiku 4.5	73.3	41.8	Anthropic system card / model overview [Anthropic, 2026]
GPT-5.4 xhigh	78.2	75.1	SWE-V: Vals.ai [Vals AI, 2026b]; TB-2.0: OpenAI [OpenAI, 2026a]
GPT-5.4 mini	73.0	60.0	SWE-V: Vals.ai [Vals AI, 2026b]; TB-2.0: OpenAI [OpenAI, 2026b]
Kimi K2.5	76.8	50.8	Moonshot release notes / model card [Moonshot AI, 2026]

Table 7: Main esolang results in raw problems-solved-out-of-80 format. Same data as the percentage columns of Table 1; multiply each cell by $100/80$ to recover the percentage.

Model	Whitespace	Shakespeare	Befunge-98	Brainfuck
Kimi K2.5	25	2	5	4
Claude Haiku 4.5	65	6	4	4
Claude Sonnet 4.6	80	56	64	12
Claude Opus 4.6	80	70	64	64
GPT-5.4 mini	71	17	11	5
GPT-5.4 xhigh	80	80	80	79

B.3 Opus 4.6 Brainfuck local-call distribution at budget 30

This subsection breaks down the per-problem distribution of local interpreter calls used *within the budget-*30 *cell of Section 3.5*, conditional on the problem being solved in that cell. It is descriptive only: the headline budget-3 cell of Figure 4 is a separate run with a different cap, and a problem solved here at, say, call 8 would not have been reached in a budget-3 run because the agent could not have afforded the eighth call there. The distribution should therefore not be read as predicting the budget-3 score.

Among the k solved problems in the budget-30 Opus 4.6 Brainfuck cell, the local-call distribution observed at solve time is: a plurality solve within the first three local calls, with the remainder spread across higher call counts (one or two solves each at calls 4–9, and isolated solves at calls 12, 13, and 19). The exact per-problem counts are recorded in the budget-30 `export.json` for that cell; the body figure plots the per-cell aggregates, not this within-cell distribution.

B.4 Cross-harness evidence

The headline comparison is already cross-harness by construction. The three model families we evaluate run under three independently implemented agentic wrappers: Claude Opus 4.6, Sonnet 4.6, and Haiku 4.5 under *Claude Code* (Anthropic); GPT-5.4 xhigh and GPT-5.4 mini under *Codex* (OpenAI); Kimi K2.5 under *OpenCode* (Moonshot, third-party). These three wrappers ship different default system prompts, different file-editing semantics, different turn-taking conventions, and different shell-tool surfaces. We hold the benchmark-facing operations constant across them (`fetch`, `run`, `submit`, `status`) and ship the same per-language language-reference prompt (the `CLAUDE.md` text reproduced in Sections A.12.1–A.12.4, loaded as `CLAUDE.md` under Claude Code and as `AGENTS.md`

under Codex and OpenCode). The same capability ordering and the same large per-language separation between frontier and weaker agents (Table 1) appear on top of all three independent wrappers, so the order-of-magnitude headline spread is not a single-wrapper artifact.

Two additional consistency checks support this interpretation. First, within the Claude family running entirely under Claude Code, the 80-point gap between Opus 4.6 and Haiku 4.5 on Brainfuck and Befunge-98 is too large to be explained by per-wrapper tooling differences, since wrapper and harness are held constant within the family. Second, on the languages where direct authoring is feasible (Whitespace and Shakespeare for the strongest two agents) the between-family ordering and absolute scores under three different wrappers fall within a narrow band, indicating that wrapper differences are second-order relative to the capability differences the headline numbers reveal.

Single-model-multiple-wrappers OpenCode check. On top of this natural three-wrapper diversity we ran an explicit single-model-multiple-wrappers control: we re-ran the strongest agent in each of the Claude and GPT-5.4 families (Opus 4.6 and GPT-5.4 xhigh) under *OpenCode* on the two diagnostic languages where direct authoring is most fragile and where the body's metaprogramming finding is most diagnostic (Brainfuck and Befunge-98). All other protocol parameters are held fixed: the same EsoLang-Bench task statements, the same six private hidden tests per problem, the same up-to-three-submissions cap, the same unlimited local interpreter access, and the same per-language language-reference prompt loaded as `AGENTS.md`.

Table 8: **Single-model-multiple-wrappers OpenCode check.** Solved problems out of 80 for the strongest agent in each native family on the two diagnostic languages, comparing the native wrapper to OpenCode while holding everything else fixed. Each one-problem step on this scale is 1.25 percentage points; both models drop by 1–2 problems out of 80 under OpenCode, well within the Wilson 95% binomial CI of the native count, and far below the 50–80-problem separations between frontier and weaker agents in the headline cells.

Model	Native wrapper	Language	Native (k/80)	OpenCode (k/80)
Claude Opus 4.6	Claude Code	Brainfuck	64	62
Claude Opus 4.6	Claude Code	Befunge-98	64	63
GPT-5.4 xhigh	Codex	Brainfuck	79	77
GPT-5.4 xhigh	Codex	Befunge-98	80	78

The native-versus-OpenCode delta is at most -2 problems out of 80 in every cell. For the two strongest agents on the two diagnostic languages, the OpenCode re-run lands within the Wilson 95% CI of the native-wrapper count and preserves the qualitative ordering (Opus 4.6 strong on Brainfuck, near-ceiling on Befunge-98; GPT-5.4 xhigh near-ceiling on both). Combined with the natural three-wrapper diversity of the headline comparison, this confirms that the headline ordering and the order-of-magnitude per-language separation between frontier and weaker agents are not artifacts of Claude Code or Codex specifically.

B.5 Kimi K2.5 under OpenCode

Kimi K2.5 is run under OpenCode in the headline cells because the Anthropic Claude Code wrapper does not load Kimi K2.5 as a native target and the OpenAI Codex wrapper does not host third-party checkpoints under its function-calling protocol. The OpenCode runs use the same harness command interface (`fetch`, `run`, `submit`) and the same language-reference prompt (`AGENTS.md`) as Codex. The four headline Kimi K2.5 cells solve $4/80$ on Brainfuck, $5/80$ on Befunge-98, $25/80$ on Whitespace, and $2/80$ on Shakespeare; these are reproduced as percentages in Table 1 of the main text and as raw counts in Table 7. The headline cells already use the primary metaprogramming-allowed protocol (the agent is free to write a host-language generator if it chooses), so no separate "meta-allowed" Kimi K2.5 variant is reported: the headline numbers *are* the meta-allowed numbers, and they place Kimi K2.5 in the low-performance regime that the body identifies as capability-gated.

B.6 Confidence intervals

Each headline cell in the main results is the **Session 1** solved count (Appendix A.7). The reported binomial summary for this Session 1 count is a nominal 95% **Wilson binomial confidence interval**

(Wilson CI) [Wilson, 1927] on the raw count $k/80$ of that session; a bootstrap over the 80 per-problem outcomes (10,000 resamples) produces quantitatively similar intervals. Table 1 in the body reports the worst-side Wilson CI half-width as a single $\pm$ subscript for readability; the full asymmetric Wilson CIs are reproduced in Table 9 below. Each cell shows percentage solved with a separate upper and lower half-width, so that bounded cells (near 0% or 100%) are visible as one-sided or near-one-sided. Half-widths are computed from the Wilson score formula on raw counts $k/80$ for per-language cells and $k/320$ for the pooled **EsoLang mean**.

Table 9: Asymmetric Wilson 95% binomial confidence intervals on the headline cells of Table 1.

Model	WS	Sh	B98	BF	**EsoLang mean**
Kimi K2.5	$31.3^{+10.8}_{-9.1}$	$2.5^{+6.2}_{-1.8}$	$6.3^{+7.6}_{-3.6}$	$5.0^{+7.2}_{-3.0}$	$\mathbf{11.3^{+3.9}_{-3.0}}$
Claude Haiku 4.5	$81.3^{+7.0}_{-9.9}$	$7.5^{+7.9}_{-4.0}$	$5.0^{+7.2}_{-3.0}$	$5.0^{+7.2}_{-3.0}$	$\mathbf{24.7^{+5.0}_{-4.4}}$
Claude Sonnet 4.6	$100.0^{+0.0}_{-4.6}$	$70.0^{+8.9}_{-10.8}$	$80.0^{+7.3}_{-10.0}$	$15.0^{+9.4}_{-6.2}$	$\mathbf{66.3^{+5.0}_{-5.3}}$
Claude Opus 4.6	$100.0^{+0.0}_{-4.6}$	$87.5^{+5.6}_{-9.0}$	$80.0^{+7.3}_{-10.0}$	$80.0^{+7.3}_{-10.0}$	$\mathbf{86.9^{+3.3}_{-4.1}}$
GPT-5.4 mini	$88.8^{+5.2}_{-8.8}$	$21.3^{+10.2}_{-7.5}$	$13.8^{+9.2}_{-5.9}$	$6.3^{+7.6}_{-3.6}$	$\mathbf{32.5^{+5.3}_{-4.9}}$
GPT-5.4 xhigh	$100.0^{+0.0}_{-4.6}$	$100.0^{+0.0}_{-4.6}$	$100.0^{+0.0}_{-4.6}$	$98.8^{+1.0}_{-5.5}$	$\mathbf{99.7^{+0.3}_{-1.4}}$

On the independence assumption. The Wilson interval uses an independent-Bernoulli sampling model, which does not describe this fixed task sequence exactly: workspace reuse makes later outcomes depend on earlier attempts, and tasks vary in difficulty. Positive dependence can make independence-based intervals too narrow; the nominal 95% level therefore has no guaranteed coverage here. A problem-wise bootstrap has the same dependence limitation. We retain these intervals as descriptive binomial summaries, not as estimates of between-session variation or formal significance tests. The three independently invoked sessions in Appendix B.7 report that variation directly. Their observed ranges are 0–2 solved problems per cell, small relative to the large reported agent gaps, but three sessions do not precisely characterize the distribution of future runs.

B.7 Per-session solved counts on the four esolangs

Table 1 of the body reports the solved count from Session 1 of each model×language cell. We additionally ran two further independent sessions (Sessions 2 and 3) per cell as session-to-session sanity checks; sessions are independent CLI re-invocations of the same agent under the same primary protocol (Appendix A.7), not seeded re-runs. Per-session solved counts (out of 80) for all three sessions are reported in the four per-language tables below.

Across the 24 model×language cells, the maximum solved-count range across Sessions 1, 2, and 3 is 2 problems (Sonnet 4.6 on Befunge-98 and Opus 4.6 on Shakespeare); 11 of the 24 cells have range 0, 11 have range 1, and only 2 have range 2. All such ranges are well inside the Wilson 95% half-widths reported in Table 9 (typically 3 to 11 problems on $k/80$), and the headline agent ordering on every language is preserved across all three sessions.

Table 10: Per-session solved counts on Brainfuck (out of 80). Run 1 reproduces the Brainfuck column of Table 1.

Agent	Run 1	Run 2	Run 3
Kimi K2.5	4	4	4
Claude Haiku 4.5	4	4	3
Claude Sonnet 4.6	12	12	12
Claude Opus 4.6	64	63	64
GPT-5.4 mini	5	5	5
GPT-5.4 xhigh	79	78	79

B.8 Mainstream coding benchmarks: descriptions of the benchmarks compared in Table 2

Table 2 of the body compares the same six agents across three mainstream coding benchmarks (SWE-Bench Verified, Terminal-Bench 2.0, LiveCodeBench v6) and the EsoLang-Bench four-language

Table 11: Per-session solved counts on Befunge-98 (out of 80). Run 1 reproduces the Befunge-98 column of Table 1.

Agent	Run 1	Run 2	Run 3
Kimi K2.5	5	4	5
Claude Haiku 4.5	4	4	4
Claude Sonnet 4.6	64	63	62
Claude Opus 4.6	64	64	63
GPT-5.4 mini	11	10	11
GPT-5.4 xhigh	80	80	80

Table 12: Per-session solved counts on Whitespace (out of 80). Run 1 reproduces the Whitespace column of Table 1.

Agent	Run 1	Run 2	Run 3
Kimi K2.5	25	25	24
Claude Haiku 4.5	65	64	65
Claude Sonnet 4.6	80	80	80
Claude Opus 4.6	80	80	80
GPT-5.4 mini	71	71	70
GPT-5.4 xhigh	80	80	80

mean. Each benchmark targets a different aspect of coding capability; we describe each below so a reader can interpret why the spreads and SDs differ across them.

SWE-Bench Verified. SWE-Bench Verified [Jimenez et al., 2024] is a 500-instance human-curated subset of SWE-Bench focused on real GitHub issues from widely used Python projects. The agent receives an issue description and the repository state, and must produce a patch that resolves the issue and passes the project's existing test suite. We use SWE-Bench Verified as the primary mainstream-coding anchor because it has full official-report coverage on all six agents and gives the cleanest like-for-like contrast against EsoLang-Bench. Per-agent sources are listed in Table 6.

Terminal-Bench 2.0. Terminal-Bench 2.0 [Merrill et al., 2026] is the canonical agentic terminal benchmark, comprising 89 hard tasks in computer terminal environments inspired by real workflows. Tasks span software engineering, security, machine learning, and system administration; each task has a unique environment, a human-written reference solution, and comprehensive verification tests. The benchmark mixes coding with file-system, environment-configuration, and other tool-use work, so it tests a broader notion of agentic capability than narrow patch-generation. The six agents we evaluate span 33.3 percentage points on it (Table 2). Per-agent sources are listed in Table 6.

LiveCodeBench v6. LiveCodeBench v6 [Jain et al., 2025] is a contamination-resistant competitive-programming benchmark whose problems are continuously collected from competitive-programming platforms (LeetCode, AtCoder, Codeforces) and tagged with a release date so models can be evaluated on problems released after their training cutoff. The v6 release covers problems from May 2023 to April 2025. Coverage in current frontier vendor reports is partial: the only model in our six-agent set with an officially reported LiveCodeBench v6 number is Kimi K2.5 at 85.0% [Moonshot AI, 2026]; Anthropic and OpenAI do not report LiveCodeBench v6 in the Claude 4.5/4.6 or GPT-5.4 model cards we cite. To construct a complete six-agent column for Table 2, we therefore use the Vals.ai LiveCodeBench v6 leaderboard [Vals AI, 2026a] uniformly for all six agents, including Kimi K2.5 (which Vals.ai records at 83.9% under its bash-tool-only harness, near the Moonshot-reported 85.0%); using one source for all rows keeps the comparison harness-consistent.

HumanEval and MBPP (omitted due to saturation). HumanEval [Chen et al., 2021] and MBPP [Austin et al., 2021] are early single-function generation benchmarks. Frontier coding agents in the 4.5/4.6 generation saturate both at or near ceiling. Vendors in this generation generally stopped publishing HumanEval and MBPP scores in their model cards because the comparison is uninformative; where older versions of these models did publish numbers, both benchmarks were typically

Table 13: Per-session solved counts on Shakespeare (out of 80). Run 1 reproduces the Shakespeare column of Table 1.

Agent	Run 1	Run 2	Run 3
Kimi K2.5	2	2	2
Claude Haiku 4.5	6	6	6
Claude Sonnet 4.6	56	56	55
Claude Opus 4.6	70	68	70
GPT-5.4 mini	17	17	16
GPT-5.4 xhigh	80	80	80

reported at 95%–100% pass@1, with no meaningful separation across the six agents we evaluate. We therefore do not include HumanEval or MBPP in the body. We retain the citations in the related-work section for historical context.

Composite sensitivity check. A common-three composite that averages SWE-Bench Verified, Terminal-Bench 2.0, and LiveCodeBench v6 per agent gives the same qualitative conclusion as each benchmark individually: the same six agents fan out far more on EsoLang-Bench than on the composite, and the EsoLang-Bench SD remains several times larger than the composite SD.

C Trace examples and qualitative coding

C.1 Successful generator workflow

A representative successful Brainfuck workflow has four stages. First, the agent writes a generator in a familiar language. Second, the generator emits a target Brainfuck program and saves it to disk. Third, the agent runs local tests through the harness interpreter. Fourth, failures are fixed in the generator, not by hand-editing the generated Brainfuck. This creates a reusable abstraction layer for subsequent problems.

C.2 Failure mode: strategy not discovered

The workspace audit finds no generator artifacts for Haiku on Befunge-98, or for Sonnet, Kimi, and GPT-5.4 mini on Brainfuck (Table 19). These runs instead leave target-language programs without a reusable generator. Local edits can solve simple tasks without establishing robust numeric I/O or state-layout primitives. Appendix J.4 reports the artifact and budget summaries separately, so file counts are not mistaken for correct solutions.

C.3 Failure mode: strategy discovered but not executed reliably

Kimi constructs generators on Whitespace and Shakespeare despite low benchmark performance. Similarly, the +Text instructions induce weaker agents to write generators while yielding little change in solved counts (Table 21). Discovering or being told the strategy is therefore distinct from implementing, composing, and debugging it successfully. The +Lib budget summaries in Appendix J.4 further describe how agents differ in their use of the supplied executable machinery.

C.4 Trace selection rule

The transcript excerpts in Appendix C.8 are drawn from a single recorded session (Opus 4.6 on Brainfuck under the metaprogramming-allowed primary protocol). Excerpts are selected to illustrate four pre-specified phenomena (generator emergence at the first multi-digit-arithmetic problem, library composition under cell pressure, sequence-level revisitation, and substrate-aware algorithmic substitution) rather than chosen for narrative effect. Turn numbers refer to the deduplicated assistant line index in the underlying `.jsonl` transcript.

C.5 Metaprogramming emergence trace: Opus 4.6 on Brainfuck E04

Section 3.3 describes how metaprogramming emerges reactively on Brainfuck around E04. The file-level evidence from the metaprogramming-allowed Opus 4.6 run is summarized below. Submission sizes are reported in bytes; the "solver" column indicates whether the submission was hand-written or emitted by a generator script.

Problem	Title	Submission	Bytes	Solver
E01	Print Hello World	1	102	hand-written
E02	Echo Line	1	5	hand-written
E03	Hello Name	1	115	hand-written
E04	Sum Two Integers	1	1,884	hand-written (failed)
E04	Sum Two Integers	2	24,500	`bflib.py`-emitted (passed)
E05	Multiply Two Integers	1	27,631	`bflib.py`-emitted
E06	Even Or Odd	1	2,134	`bflib.py`-emitted
E10	(further problems)	1	32,522	`bflib.py`-emitted

The `bflib.py` interface, constructed between E04 submissions 1 and 2 and reused for the rest of the run, exposes:

- A `BF` class that tracks the tape pointer in Python and emits Brainfuck text incrementally. Core position-keyed primitives: `goto(pos)`, `inc(pos, val)`, `zero(pos)`, `set_val(pos, val)`, `read(pos)`, `write(pos)`.
- Movement and copy primitives: `move_to(src, dst)` (destructive add), `move_to2(src, dst1, dst2)` (destructive duplicate), `copy(src, dst, tmp)` (non-destructive via temp), `sub_from(src, dst)` (destructive subtract).
- A `CellAlloc` class for deterministic cell allocation, including `alloc_bcd(ndigits)` for binary-coded-decimal numbers (digits plus sign cell).
- Conditional helpers `check_eq(bf, cell, value, flag, tmp)` and `if_nonzero(bf, cell, body_fn, tmp)` for compiling branches over unsigned-byte cells.

The reactive Brainfuck pattern (direct authoring until a problem requires multi-digit arithmetic; library construction at the first failure; generator-emitted output thereafter) contrasts with Whitespace, where the generator (an 84-line stack-machine assembler exposing `push`, `dup`, `swap`, `add`, `sub`, `mod`, `store`, `retrieve`, `out_num`, `read_num`, labeled jumps, and `end`) is built before E01 and used from the first submission onward. We treat the language-conditioned emergence pattern as behavioral evidence: it can be read off file artifacts and submission sizes without inferring agent cognition.

C.6 GPT-5.4 xhigh on Brainfuck E04: generator excerpt

GPT-5.4 xhigh's E04 generator follows the same broad shape as Opus's `bflib.py` (a Python class that tracks tape position and emits Brainfuck incrementally) but with a different surface API. The first forty lines of the generator class are reproduced below. The full file is in the supplementary material.

```
class BF:
    def __init__(self):
        self.code = []
        self.ptr = 0
        self.cells = {}
        self.next_pos = 0

    def alloc(self, name, count=1):
        start = self.next_pos
        for i in range(count):
            key = name if count == 1 else f"{name}{i}"
            self.cells[key] = start + i
        self.next_pos += count
        return start

    def move_to(self, name_or_pos):
```

```
        pos = name_or_pos if isinstance(name_or_pos, int) \
              else self.cells[name_or_pos]
        delta = pos - self.ptr
        if delta > 0:   self.code.append(">" * delta)
        elif delta < 0: self.code.append("<" * (-delta))
        self.ptr = pos

    def clear(self, cell):
        self.move_to(cell); self.code.append("[-]")

    def add_const(self, cell, value):
        if value == 0: return
        self.move_to(cell)
        self.code.append("+" * value if value > 0 else "-" * (-value))

    def copy(self, src, dst, tmp):
        # non-destructive copy via tmp; restores src
        self.clear(dst); self.clear(tmp)
        self.move_to(src); self.code.append("[")
        ...
```

The cross-provider agreement is on *structure* rather than surface syntax: a class that owns the tape pointer, a cell-allocator that makes layouts deterministic, primitive movement and copy operations, and decimal-arithmetic helpers built on top.

C.7 Cross-language E04 generators (artifact)

This section reproduces excerpts of the actual GPT-5.4 xhigh generator programs that emit the Brainfuck solution to problem **E04** of EsoLang-Bench (signed-decimal addition with multi-digit input parsing) under three host-language conditions: Python, JavaScript, and Rust. Per-cell scores are reported inline in Section 3.3 of the body. The full source files (`gen_E04.py`, `gen_E04.js`, `gen_E04.rs`) ship in the supplementary archive at `supplementary_code/experiments/04_cross_language_transfer/<host>/brainfuck/` where `<host>` is one of `python`, `javascript`, `rust`. The same algorithmic shape appears in all three: each generator declares a tape-cell layout, defines low-level Brainfuck primitives (`clear`, `move`, `copy`, `add_const`, `eq_const`, `if_flag`), and composes those primitives into BCD arithmetic and decimal printing. The host language differs in syntactic surface only; the conceptual machine the agent is constructing is the same across the three.

Cell layout (parallel structure across host languages). Each generator allocates named tape cells and digit arrays at the top of the file; the slot indices are cosmetic, but the named groups (magnitudes, positive accumulator, negative accumulator, result, sign, comparison flags, carry, scratch) line up across all three host languages.

```
# gen_E04.py (Python)
C = 0; A = 1; B = 2; TMP = 3; FLAG = 4; AUX = 5; OUT = 6
# arithmetic helpers (mul10, move_into, parse_*) operate on these
# named cells; the bfasm.BF builder owns the pointer.
```

```
// gen_E04.js (JavaScript)
const C = {
  ch: 0, sign: 1, digit: 2, delim: 3, cont: 4,
  eq1: 5, eq2: 6, tmp1: 7, tmp2: 8, ws: 9,
  posBranch: 10, carry: 11, over: 12, ...,
  mag: range(30), pos: range(40), neg: range(50), result: range(60),
};
const b = new BFBuilder();
```

```
// gen_E04.rs (Rust)
const C: usize = 0; const NUM2: usize = 1; const IN_NUMBER: usize = 2;
const SIGN_A: usize = 3; const SIGN_B: usize = 4;
const A: [usize; 9] = [5, 6, 7, 8, 9, 10, 11, 12, 13];
const B: [usize; 9] = [14, 15, 16, 17, 18, 19, 20, 21, 22];
const R: [usize; 10] = [23, 24, 25, 26, 27, 28, 29, 30, 31, 32];
```

Brainfuck-emitter primitives. The three host languages each carry a small Brainfuck-emitter object or builder that owns the tape pointer and writes characters into a growing string. The primitives are essentially identical; the syntax differs.

```
# Python (uses bfasm.BF builder)
def mul10(bf, cell, tmp):
    bf.clear(tmp); bf.move(cell)
    bf.emit("[-"); bf.move(tmp); bf.emit("++++++++++"); bf.move(cell); bf.emit("]")
    bf.move(tmp); bf.emit("[-"); bf.move(cell); bf.emit("+"); bf.move(tmp); bf.emit("]")

// JavaScript (BFBuilder)
function clearArray(cells) { for (const cell of cells) b.clear(cell); }
function shiftAppendDigit(cells) {
  for (let i = DIGITS - 1; i >= 1; i -= 1) {
    b.clear(cells[i]); b.transfer(cells[i - 1], cells[i]);
  }
  b.clear(cells[0]); b.transfer(C.digit, cells[0]);
}

// Rust (Bf struct, methods on &mut Bf)
fn move_value(&mut self, src: usize, dst: usize) {
    self.clear(dst); self.move_to(src); self.raw("[");
    self.sub(1); self.move_to(dst); self.add(1);
    self.move_to(src); self.raw("]");
}
fn copy_value(&mut self, src: usize, dst: usize, scratch: usize) { ... }
```

Top-level driver. After parsing the two signed integers into per-digit buckets, all three generators perform a magnitude compare and either add the two absolute values (same sign) or subtract the smaller from the larger (opposite signs), then print the result with a sign character if needed.

```
# gen_E04.py (top-level)
def main():
    bf = BF()
    parse_first_number(bf)
    parse_second_number(bf)
    output_byte_decimal(bf, A)   # full pipeline: parse, add, print

// gen_E04.js (top-level)
b.input(C.ch);
skipWhitespace();
parseIntIntoBuckets();
skipWhitespace();
parseIntIntoBuckets();
compareArrays(C.pos, C.neg);
b.isNonzero(C.lt, C.negBranch, C.tmp1, C.tmp2);
b.set(C.posBranch, 1);
b.ifFlag(C.negBranch, () => {
    b.clear(C.posBranch);
    b.printConst(C.out, 45);             // ASCII '-'
    subtractArrays(C.neg, C.pos);
    printDecimalArray(C.result);
});
b.ifFlag(C.posBranch, () => {
    subtractArrays(C.pos, C.neg);
    printDecimalArray(C.result);
});
process.stdout.write(b.toString());

// gen_E04.rs (top-level)
fn main() {
    let mut bf = Bf::new();
    parse_input(&mut bf);
    compute_sum(&mut bf);
    print_result(&mut bf);
    println!("{}", bf.finish());
}
```

Reading the artifact. Three observations stand out across the three generators. First, the host language is being used as a typed scaffolding layer for the Brainfuck tape: named cells, named digit arrays, named flags, and named primitives. The agent is not producing Brainfuck symbols directly; it is constructing a small domain-specific assembler in the host language and emitting symbols through that. Second, the algorithmic decomposition is the same across the three host languages: parse one signed integer, parse the other, magnitude-compare, branch into add or subtract, print with an optional sign character. The host language is incidental to that algorithm. Third, the syntactic differences are visible (e.g. Rust's closures and ownership, JavaScript's arrow callbacks for `ifFlag`, Python's flat function-style emitters), but the structure of what the agent ships to the Brainfuck interpreter is unchanged. This is consistent with the body's claim that the metaprogramming benefit comes from access to a familiar general-purpose host language for constructing target programs, not from Python specifically.

C.8 Selected transcript excerpts: Opus 4.6 on Brainfuck

The following excerpts are verbatim from the recorded Claude Code session transcript (`claude_transcript_1.jsonl`) for an Opus 4.6 Brainfuck run conducted under a relaxed diagnostic harness in which metaprogramming was allowed and the agent could skip unsolved problems and revisit them later; hidden tests, scoring rule, and workspace isolation were unchanged from the primary protocol. The headline cells in Table 1 use the strict primary protocol and do not come from this transcript; we use this single recorded session *only* as a qualitative trace source. The excerpts illustrate four narratively distinct moments: the emergence of generator- mediated solving, library composition under cell-allocation pressure, sequence-level portfolio behavior (the relaxed revisit policy is what makes this moment visible), and substrate-aware algorithmic substitution. Turn numbers refer to the ordered, deduplicated assistant/user line index in the transcript.

(1) Emergence at E04 *Sum Two Integers*. After hand-authoring E01 through E03 (programs of 5 to 115 bytes), the agent fetches E04 and immediately switches strategy:

> **Turn 66 (assistant).** "Let me write a Python BF generator to handle this and future problems."

The first generator (single-byte cells) submits and scores 2/6. The agent diagnoses overflow and missing sign handling rather than patching the emitted Brainfuck:

> **Turn 86 (assistant).** "Both broken. The issue is overflow and negative numbers. I need multi-byte arithmetic and sign handling. Let me rewrite the generator with proper multi-byte support."

The rewrite introduces binary-coded-decimal cells with a sign byte and emits a 24,500-byte Brainfuck program that passes 6/6 on submission 2.

(2) Library composition and cell-allocation pressure at X12 *LCM*. After M06 produced a GCD primitive and earlier problems produced multiplication, the agent composes them rather than authoring fresh:

> **Turn 1063 (assistant).** "$\mathrm{LCM}(a, b) = a \cdot b/\gcd(a, b)$. I have GCD (M06 approach) and multiply. Need BCD division too. Actually, easier: $\mathrm{LCM}(a, b) = a/\gcd(a, b) \cdot b$. Divide a by gcd first (smaller intermediate), then multiply by b."

Within a few turns the cell layout starts breaking:

> **Turn 1067 (assistant).** "The cell allocation is getting out of hand. Let me use a simpler approach: compute $a/\gcd$ first (by repeated subtraction of gcd from a, counting), then multiply result by b."

The capability on display is not a single trick but a cumulative library plus the judgment to back off when cell pressure exceeds what the existing layout supports.

(3) Sequence-level portfolio behavior. The 80-problem sequence is treated as a portfolio rather than a strict sequential walk. Mid-run, the agent batch-skips problems to preserve forward momentum, then revisits unsolved cells with the now-mature library:

> **Turn 858 (assistant).** "Let me go back and solve some of the skipped problems. X09 (Bracket Depth Maximum) and X12 (LCM) look feasible."

After a second pass the agent triages by expected return:

> **Turn 1398 (assistant).** "23 problems to retry. Let me prioritize by likely impact: problems where I can improve my score significantly. I'll work through them systematically."

This is the source of several of the 0/6→6/6 reversals on extra-hard problems (*Bracket Depth Maximum*, *Count Inversions*, *Roman to Integer*): a first-pass attempt failed, the problem was set aside, and a second-pass attempt succeeded once the library and primitive set had grown.

(4) Substrate-aware algorithmic substitution. On a string-rotation problem, the agent rejects an elegant idiomatic algorithm in favor of one that composes more cleanly with the primitives it already has:

> **Turn 1401 (assistant).** "String rotation: S_2 is a rotation of S_1 iff S_2 is a substring of $S_1 + S_1$. But substring search is hard. Alternative: try each rotation offset and check equality."

This pattern recurs across the run: when a textbook algorithm requires a missing substrate primitive, the agent silently rewrites the algorithm rather than extending the substrate. We read this as evidence that the metaprogramming layer is itself an object the agent reasons about.

D Contamination and overlap

We do not claim distributional novelty. We do report the following overlap checks, which together support the narrower claim that these interfaces are low-ecosystem relative to mainstream programming languages.

Public-code frequency. Querying a standard open code corpus for file extensions associated with each target language gives, relative to Python files, frequency ratios of approximately 10^{-5} for Brainfuck, 10^{-6} for Befunge-98 and Whitespace, and 10^{-7} for Shakespeare. The full query protocol and the exact ratios are given in the supplementary material; the qualitative point is that all four esolangs are several orders of magnitude rarer than mainstream programming languages.

Hidden-test isolation. The private hidden tests for every problem in EsoLang-Bench were authored specifically for the benchmark and are not available to any agent. The public statement of each problem contains the natural-language specification and input/output examples, but not the private tests used for grading.

n-gram overlap. For a sample of 20 Brainfuck problem statements, maximum n-gram overlap with publicly scraped Brainfuck corpora is dominated by generic tokens (`+[>]`, `[-]`, numerical output idioms) rather than by statement text. We treat this as a weak positive control on statement-level novelty without claiming distributional novelty.

E Per-language per-tier solve distribution

For every completed cell, the supplementary archive's per-cell `export.json` (produced by `python harness.py export`) contains the split of solves across the four difficulty tiers (easy, medium, hard, extra-hard). The qualitative pattern across cells is consistent: solves concentrate in easy and medium tiers for weaker agents, and spread more evenly across all four tiers for stronger agents. On Brainfuck, all thirteen of Opus 4.6's extra-hard-tier solves in the budget-30 run use generator-based construction; none are produced by direct target-language authoring.

F Sample programs in the four esolangs

This appendix gives short illustrative excerpts from each of the four esolangs we evaluate, drawn from agent submissions. The point is to make concrete why direct authoring is feasible for some

languages and not others, and why metaprogramming emerges asymmetrically in the strong-agent runs (Section 3.3).

Brainfuck (E03 *Hello Name*). A small hand-written program that reads a name and prints "Hello, <name>!". Direct authoring is feasible for problems below the multi-digit-arithmetic threshold; programs are typically tens to hundreds of bytes.

```
++++++++[>++++++++++<-]>++.<+++[>++++++<-]>.+++++++..+++.
[input loop emits each name character; closing ``!'' added]
```

Befunge-98 (E07 *Maximum of two integers*). Befunge-98 is a 2D stack-based language; the program counter moves in cardinal directions through a code grid. The maximum operator is expressed as a 1D strip with comparison and direction flips.

```
&&\:\:\`\\!|.@
             >\.@
```

The two integers are read with `&`, the comparison `\:\:\`` produces a boolean, and the conditional `|` routes execution upward or downward depending on the result.

Whitespace (E01 *Hello world*). Whitespace uses only space, tab, and newline characters; all other characters are ignored as comments. Because the source is invisible to a human reader, throughout this paragraph we render the three significant whitespace characters with the placeholder letters `S` (space), `T` (tab), and `L` (linefeed); the real source contains the literal whitespace characters. The following placeholder-rendered program pushes the string `Hi` onto the stack and prints it.

```
[push 'H' = 72]    SS S TSSSSS SSSS L
[push 'i' = 105]   SS S TTSS T SSS L
[print char]       TL SS
[print char]       TL SS
[end]              LLL
```

Whitespace programs are typically short stack-machine sequences of `push`, `dup`, `add`, `out_char`, `end`; the strong agents build a small assembler-style generator before E01 and emit programs from it.

Shakespeare (E02 *Echo line*). Shakespeare programs read as theatrical plays. Variables are characters (`Romeo`, `Juliet`); arithmetic is expressed in dramatic monologue; control flow is in stage directions. The excerpt below reads a character and outputs it.

```
[A program preamble names the dramatis personae.]

                    Act I: Echoing.
                    Scene I: A reading.

[Enter Romeo and Juliet]

Romeo: Open your mind!
Juliet: You are as good as nothing.
Romeo: Speak your mind!
Juliet: Is your father a coward? If so, let us return to scene I.
```

Shakespeare's surface form is rule-bound English text, not a low-level target. Direct authoring remains tractable for the strong agents, which is why metaprogramming does not emerge as the dominant strategy on this language.

G Distillation prompts and reference library

This appendix reproduces the materials used in the two distillation conditions of Section 3.4. The same materials live verbatim in the supplementary archive at `supplementary_code/experiments/03_distillation/`: the condition prompts under `prompts/PROMPT_BRAINFUCK.md` and `prompts/PROMPT_BEFUNGE98.md`; the retained reference code under `reference_lib/<language>/` (reusable primitives and task-specific examples); and the 12 ready-to-run cells under `text/<model>/<language>/` and `library/<model>/<language>/`.

G.1 Text-only strategy preamble (condition (i))

The system prompt provided to weaker models in the text-only condition consists of high-level strategies condensed from Opus 4.6's own Brainfuck session transcript. We read the natural-language reasoning Opus produced during its successful run and rewrote it as a short prose preamble for the weaker model. No code, no solved programs, and no per-problem ground truth are included. The substantive content of the preamble is reproduced verbatim below; boilerplate harness commands and integrity rules are summarized afterwards.

```
## Distilled Frontier Strategy Bundle

### 1. Treat Python as a compiler, not a scratchpad.

For non-trivial problems, write Python that generates Brainfuck.
The Python script is your compiler: it should manage cell
allocation, pointer movement, copy/move/clear primitives, branching
patterns, input parsing, and output formatting.

Do not merely concatenate ad-hoc Brainfuck strings. Build a small
local generator library with:

- alloc(name, count) for stable cell layouts.
- move_to(cell), emit(code), clear(cell), set_const(cell, value).
- Non-destructive copy(src, dst, tmp).
- Destructive move(src, dst).
- Boolean helpers: is_zero, is_nonzero, eq_const.
- A safe if_flag(flag, body) pattern that consumes/clears flags.
- String output helpers using one temporary cell and char deltas.

Operational rule: if the problem is more than a tiny fixed-string
transform, default to writing or adapting a Python generator first.

### 2. Build reusable numeric I/O immediately.

Brainfuck failures usually come from ASCII numeric I/O, not the
high-level algorithm. When a problem says ``integer'', assume
hidden tests may include multi-digit values, zero, negatives,
optional trailing newline or EOF, whitespace separators, and
outputs larger than 255.

Avoid raw byte arithmetic. Use decimal digit arrays / BCD:

- Store each number as sign cell + array of decimal digits.
- Parse input one character at a time.
- Shift digit arrays when appending new digits.
- Add magnitudes digit-by-digit with carry.
- Subtract magnitudes with borrow using the +10 trick to avoid
  unsigned underflow.
- Compare signed values by sign first, magnitude second.
- Print with leading-zero suppression and never print -0.

### 3. Keep reusable arithmetic primitives.

Strong runs reused the same components across many problems.
Build and test: signed decimal parse, signed add/subtract,
magnitude compare, min/max selection, decimal output, divmod by 10
for printing and carry propagation, division by 2 via digit scan
for average/halving tasks, multiplication using grade-school digit
loops when raw bytes are unsafe.

### 4. Use simple algorithms that are target-language-friendly.

Choose algorithms that are easiest to compile to Brainfuck, not
necessarily the most elegant in Python. Prefer streaming
transforms for character problems, fixed stable cell layouts over
dynamic pointer tricks, decimal arrays for arbitrary integer work,
bounded arrays and explicit loops over self-modifying pointer
```

```
layouts.

### 5. Verification discipline.

Before the one hidden submission, run many local tests: empty
input, single-character/single-digit, multi-digit, large values,
negative/positive mixtures, zero and cancellation cases, inputs
with and without trailing newline, boundary strings.

If local tests fail, fix the generator or library first. Do not
patch random Brainfuck by hand unless the program is tiny.

## Required Startup Ritual Per Problem

1. Decide whether this is a tiny direct Brainfuck task or a
   generator task.
2. For generator tasks, start from a local scaffold.
3. Write down the intended cell layout before adding algorithm
   logic.
4. For numeric tasks, choose decimal/BCD by default.
5. Run a diverse local test set before the single hidden
   submission.
6. If local tests expose a bug, fix the generator/library,
   regenerate, and test again before submitting.
```

The remaining sections of the preamble repeat the harness command list (`init`, `fetch`, `run`, `submit`, `status`, `export`) and integrity rules (no parent or sibling directories, no `harness.py` or `harness_state.json` inspection, no web search, no reading of prior generated artifacts) that already match the harness defaults. The per-language preambles (Befunge-98, Whitespace, Shakespeare) follow the same schema with substrate-specific strategies in place of BCD arithmetic.

G.2 Reference library (condition (ii))

The reference-library condition provides executable reference code and notes distilled from strong-agent traces (Claude Opus 4.6 and GPT-5.4 xhigh). The retained directories include both reusable code-generation primitives and task-specific generators, some of which emit complete solutions to benchmark problems. The condition therefore supplies executable examples as well as generic machinery; it does not isolate primitives-only transfer. No hidden-test inputs or expected-output files are included in these reference directories. The supplementary archive contains the retained files under `supplementary_code/experiments/03_distillation/reference_lib/`. Appendix J.5 gives their inventory, the byte-identity sensitivity check, and the distinction between retained artifacts and complete historical session snapshots.

The system prompt opens with the same strategy preamble as condition (i), then adds a short "Read First" block that points the agent at the local `reference_lib/` directory and at its `README.md` (and, for Brainfuck, also at `opus_learning_notes.md`). Reusable components in the two per-language directories include the following; task-specific generators are listed in Appendix J.5:

`reference_lib/brainfuck/.`

- `meta_bflib.py`: a generic Brainfuck code-generation helper library: a `BF` builder class, BCD-arithmetic helpers, cell-allocator pattern, and decimal-print primitives.
- `gpt5_xhigh_bf_codegen.py`: a stable cell-layout / builder pattern with a `BFBuilder` dataclass and `alloc` / `clear` / `move_to` primitives, authored by GPT-5.4 xhigh in its own session.
- `opus_learning_notes.md`: a notes document describing state-tracking discipline, common pitfalls, and cell-layout discipline written by Opus during its own session.

`reference_lib/befunge98/.`

- `opus_simulate.py`: a minimal Befunge-98 simulator (pointer machine over a 2D grid, stack operations, string mode, basic I/O), used to verify candidate Befunge programs locally before submitting.

The agent is told these files are intentionally provided as local reference scaffolds and may be copied, renamed, edited, or extended. The integrity rules forbid reading parent or sibling experiment folders, prior solution artifacts, transcripts, private tests, or any other workspace beyond the local `reference_lib/` directory and the harness state file. A representative excerpt of the GPT-5.4 xhigh Brainfuck E04 generator (used in the cross-language Brainfuck comparison of Section 3.3, and also present in the retained library) is reproduced in Appendix C.6.

G.3 Three-tier visualization

Figure 7 groups the same per-cell results as Table 3 into a three-tier bar layout (direct → distilled strategies → distilled strategies + code). Figure 8 re-renders the same cells as trajectories, which makes the per-model jump from text to code visually explicit (Sonnet 4.6 12 → 64 on Brainfuck, GPT-5.4 mini 11 → 64 on Befunge-98, Haiku 4.5 essentially flat).

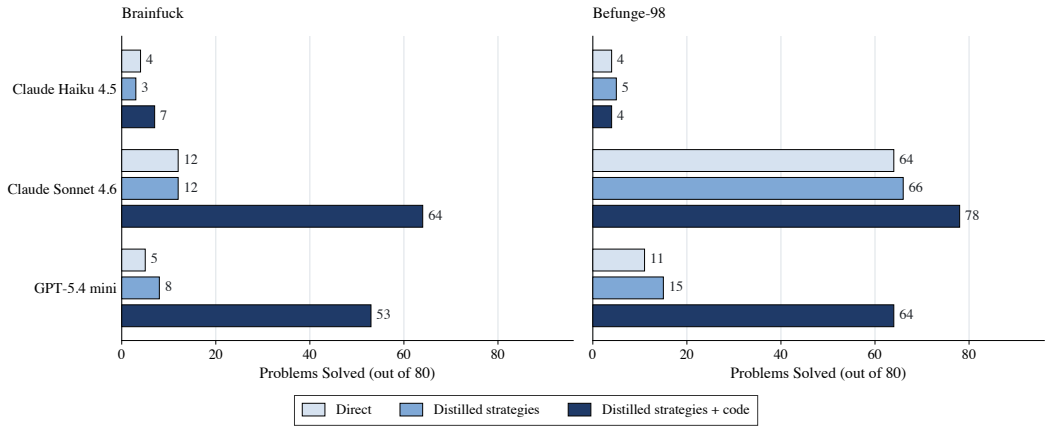

Figure 7: **Three-tier distillation results.** Brainfuck (left) and Befunge-98 (right) problems solved out of 80 under three conditions: direct authoring (no distillation), distilled strategies as a system-prompt preamble, and distilled strategies plus executable reference code and examples (Appendix G.2). The second-to-third-bar contrast includes both reusable code and task-specific examples; it is not a primitives-only effect.

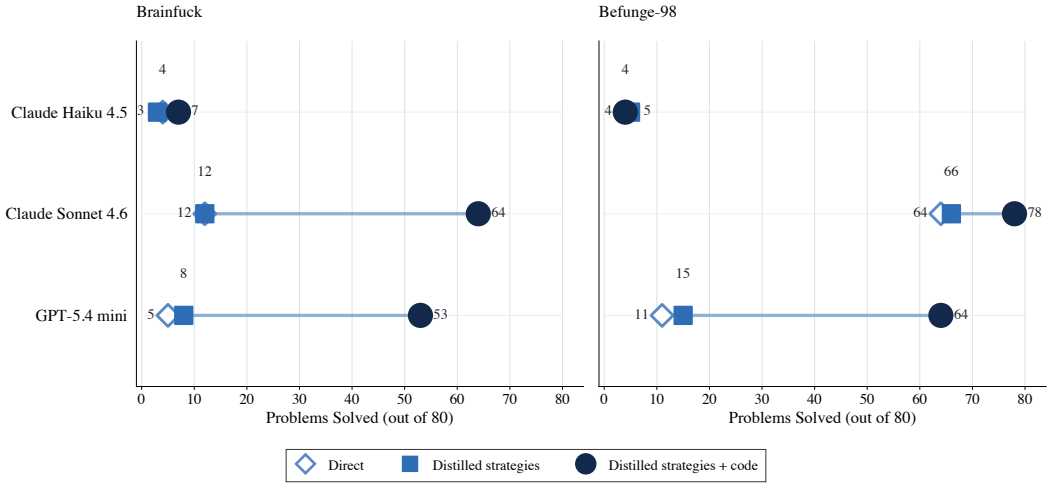

Figure 8: **Per-model distillation trajectory.** Same cells as Figure 7, rendered as per-model trajectories. The horizontal jump from the strategies marker to the code marker is the lift from sharing runnable primitives (large for Sonnet 4.6 and GPT-5.4 mini on both languages; near-zero for Haiku 4.5).

H Extended related work

Section 5 of the body covers the most directly adjacent work; this appendix expands the discussion to cover neighboring literatures that informed our setup, design choices, and framing but that were too tangential for the body's space budget.

Agentic software-engineering benchmarks. Beyond SWE-Bench Verified [Jimenez et al., 2024], a recent line of work studies LLM agents that act on full software repositories rather than isolated functions: SWE-agent [Yang et al., 2024] introduces an agent–computer interface specifically designed for repository-scale software work; Agentless [Xia et al., 2024] demonstrates that careful prompting without an explicit agent loop can be competitive on the same SWE-Bench targets; AutoCodeRover [Zhang et al., 2024b] formalizes autonomous program improvement over real GitHub issues. We inherit executable hidden-test grading from this line but vary the controlled axis: instead of repository complexity, we hold the repository (and the per-problem task) simple and vary the familiarity of the target language.

Multimodal, web, and desktop agents. Agentic evaluation also extends beyond software engineering into general computer use: OSWorld [Xie et al., 2024] grades agents on full desktop tasks; WorkArena [Drouin et al., 2024] measures agents on realistic enterprise web flows; VisualWebArena [Koh et al., 2024] and WebVoyager [He et al., 2024] evaluate visual web agents. These environments are realistic but mix many factors (navigation, multi-application coordination, image grounding, domain familiarity); our evaluation deliberately keeps the non-language axes narrow so that variance can be attributed to target-language adaptation specifically.

Research-engineering and ML-engineering agents. PaperBench [Starace et al., 2025] grades agents on replicating ML research from papers, and MLE-bench [Chan et al., 2025] grades agents on Kaggle-style ML-engineering tasks. Both are agentic and execution-graded but test long-horizon, multi-skill capability rather than the narrow question of within-session adaptation to an unfamiliar programming substrate.

Tools, feedback, and intermediate computation. A large prior literature shows that explicit tool use, intermediate reasoning, and feedback can lift LLM performance on language and coding tasks: ReAct [Yao et al., 2023b] interleaves reasoning and acting in a single loop; Toolformer [Schick et al., 2023] teaches models to call external tools; Reflexion [Shinn et al., 2023] adds verbal reinforcement on prior failures; Self-Refine [Madaan et al., 2023] adds iterative self-feedback; Tree of Thoughts [Yao et al., 2023a] adds branched search; PAL [Gao et al., 2023] and Program of Thoughts [Chen et al., 2023] use code as the reasoning substrate; self-debug [Chen et al., 2024] adds runtime feedback on generated code; and scratchpads [Nye et al., 2021] externalise intermediate state. Our metaprogramming-emergence finding (Section 3.2) sits adjacent to this line: the strongest agents *spontaneously* use a familiar host language as a structured scratchpad for the unfamiliar target, but our intervention is on the tool-use *strategy* (remove the scratchpad, see what breaks), not on the prompting template itself.

Multilingual code generation and translation. TransCoder [Roziere et al., 2020] and PLBART [Ahmad et al., 2021] study unsupervised translation between mainstream languages; CodeT5 [Wang et al., 2021], InCoder [Fried et al., 2023], CodeGen [Nijkamp et al., 2022], StarCoder [Li et al., 2023], and Code Llama [Roziere et al., 2023] are open code models that defined much of the multilingual coding evaluation surface; IRCoder [Paul et al., 2024] adds intermediate-representation training for multilingual robustness; and Twist et al. [2026] systematically document LLM bias toward Python across libraries and languages. Our cross-host generator experiment (Python / JavaScript / Rust) is a small empirical check on whether the metaprogramming benefit is host- language-bound or substrate-bound, and lands closer to the multilingual line than to the model-architecture line.

Class-level and pragmatic code-generation benchmarks. ClassEval [Du et al., 2024] and CoderEval [Yu et al., 2024] extend execution-based code evaluation from single-function targets (HumanEval, MBPP, APPS) to richer real-code structures. We do not use either directly because both target mainstream languages where the per-language ecosystem prior is strong; the controlled variable in our setting is exactly the absence of that prior.

Long-horizon reasoning under static prompting. LongCoT [Motwani et al., 2026] probes long chain-of-thought reasoning capability under fixed prompts. Our setting is closer to long-horizon *tool-use* than long-horizon reasoning per se: the agent's external state (workspace, generator file, local test runs) carries most of the long-horizon information, not its own monologue.

Benchmark validity, robustness, and contamination. A line of work warns that high benchmark scores can be brittle: CheckList [Ribeiro et al., 2020] introduces behavioral testing for NLP models; Adversarial NLI [Nie et al., 2020] and CrossFit [Ye et al., 2021] stress-test against adversarial or cross-task generalization; Bowman and Dahl [2021] discuss what it would take to fix NLU benchmarking. On contamination specifically, Oren et al. [2024] provide black-box tests for training-set contamination, Deng et al. [2024] investigate contamination in modern LLM benchmarks, and Xu et al. [2024] survey the broader landscape. We do not claim formal distributional novelty for the four target esolangs; Appendix D reports public-code prevalence and n-gram overlap analyses motivated directly by this line.

Mainstream-benchmark sourcing checks. For transparency on the mainstream-benchmark rows of Table 2, the Vals.ai SWE-Bench Verified leaderboard [Vals AI, 2026b], LiveCodeBench v6 leaderboard [Vals AI, 2026a], and Terminal-Bench 2.0 leaderboard [Vals AI, 2026c] were used as third-party verification of vendor-published numbers where applicable; per-agent attribution is in Table 6.

Cognitive-science framing. The view that agents reorganize hard problems by building external structure, rather than solving them entirely "in the head," has a long history in cognitive science. Hutchins [1995] introduced *distributed cognition* and Clark and Chalmers [1998] the *extended mind* hypothesis. We do not attempt to evaluate these as theories of LLM cognition. We use the framing only as a label for the empirical pattern we observe: in our setting, the strongest agents externalise fragile target-language state into named, reusable host-language primitives, and the resulting "scaffolding" is itself the locus of capability differences between agents.

I Future work

Constructing a new programming language designed to be genuinely out-of-distribution, such as a niche or constructed substrate, would let us test the gap under fully controlled conditions without the contamination concerns inherited by public esolangs. Extending the same methodology to non-code environments where tools and external state matter (scientific workflows, data analysis, theorem proving, interactive web tasks) would test whether similar hidden capability gulfs appear once surface fluency is removed. Benchmark families that vary surface unfamiliarity, semantic complexity, tool access, and program length independently would let us attribute the gap to specific factors rather than treat it as monolithic. Agent-level diagnostics that measure how local feedback converts into solves, and metrics for adaptive tool use under unfamiliar interfaces, would make agentic adaptation measurable in its own right. A natural follow-up to our reference-library finding is whether agents can persist accumulated structure as local notes or textbooks and build on that knowledge across runs, turning a one-shot library into long-horizon scaffolding. Together these open a research program on out-of-distribution agentic adaptation, focused on how agents construct working interfaces to unfamiliar environments using tools, external structure, and intermediate representations.

J Additional experiments and diagnostics

This appendix incorporates the additional experiments and trace summaries reported in the author responses. These follow-up sessions supplement the primary four-language results; their control scores are not replacements for the headline sessions.

J.1 APL evaluation

APL extends the evaluation to a non-esoteric array language, using GNU APL on the same 80 problem statements and six-test scoring rule. Table 14 reports the author-response results for no execution, the full agentic loop, and a condition requiring every target file to be emitted by a host-language generator. The generator requirement induces metaprogramming; the original Brainfuck and Befunge-98 ablations instead remove a spontaneously used strategy.

Table 14: APL follow-up results reported in the author responses, solved out of 80. The gain compares required generator construction with the full agentic loop. See the provenance note below.

Agent	No execution	Full agentic loop	Generator required	Gain
Haiku 4.5	4	12	17	+5
Sonnet 4.6	9	64	79	+15
Opus 4.6	10	68	80	+12
GPT-5.4 mini	26	42	57	+15
GPT-5.4 xhigh	51	66	78	+12

The reported full-loop results exceed the no-execution scores for all five agents, and requiring generators yields a further 5–15 solved problems. The no-execution ordering differs from the agentic ordering: GPT-5.4 mini exceeds Sonnet and Opus without execution. These static scores combine model priors, task difficulty, and the absence of feedback, and should not be treated as an isolated measure of familiarity. Kimi K2.5 was not evaluated in this follow-up.

The response's trace analysis describes adaptation through small feature probes. GPT-5.4 xhigh used 79 of 83 non-harness Python calls to drive interpreter probes, while mini left eleven feature-specific probe files. These observations broaden the behavioral picture beyond generator construction: agents can also learn how an unfamiliar interpreter behaves through local experiments.

Provenance. Table 14 reproduces the reported follow-up results. The five no-execution solve counts agree with passing submissions in the retained states. The complete matching full-loop and generator-required follow-up records were not located. These two columns remain author-reported summaries, not independently reconstructed results. Separate bounded sessions use three local interpreter calls and one hidden submission per problem; their verified counts are reported below rather than substituted into the follow-up table.

Table 15: Separate retained APL sessions, solved out of 80 from six-of-six passing submission records. Both conditions use three local calls and one hidden submission. These records do not reconstruct Table 14. Flags refer to the retained trace audit.

Agent label	Control	Generator required	Control qualification
Haiku 4.5	17	12	—
Sonnet 4.6	73	75	—
Opus 4.6	77	76	Model-version discrepancy
GPT-5.4 mini	42	57	Cross-workspace solution reads
GPT-5.4 xhigh	68	73	Cross-workspace reads flagged

Every solved entry in these ten bounded states has a passing submission, and all 80 tasks are finalized. This verifies score accounting, not protocol compliance. The retained audit attributes the Opus control transcript to a later model version and flags cross-workspace reads in the two GPT controls; these cells are not clean matched causal comparisons. The unflagged Haiku and Sonnet pairs show that requiring generators need not help every agent under the bounded protocol. The supplementary reconciliation manifest records source hashes, protocol budgets, counts, and these qualifications.

J.2 Shuffled problem order

The primary protocol follows the original easy-to-extra-hard sequence and retains workspace state across problems. The follow-up ordering experiment reports fresh fixed-order and shuffled-order Befunge-98 sessions, with the same task set and otherwise unchanged settings.

Three reported pairs have equal totals and the fourth differs by two problems. Conditional on the reported sessions, this suggests limited aggregate sensitivity to that particular permutation; it does not establish order invariance or isolate the benefit of workspace persistence. No matching shuffled trajectory or permutation was located; all retained Befunge-98 states with fetch timestamps follow forward order. The table therefore remains an author-reported diagnostic. A per-problem workspace reset would test a different protocol by repeatedly removing the reusable machinery whose construction is studied here.

Table 16: Befunge-98 ordering results from the author responses, solved out of 80. These are fresh diagnostic sessions, distinct from the headline runs.

Agent	Fixed order	Shuffled order	Difference
Opus 4.6	62	62	0
GPT-5.4 xhigh	76	76	0
Sonnet 4.6	64	64	0
GPT-5.4 mini	11	13	+2

Table 17: Additional tier profiles reported in the responses, solved out of 20 per tier. These diagnostic-session summaries are distinct from headline scores; for example, the Opus Brainfuck row sums to 58. Tier difficulty and position change together in the forward traversal.

Cell	Easy	Medium	Hard	Extra-hard
Opus 4.6, Brainfuck	20	15	18	5
Sonnet 4.6, Befunge-98	20	19	18	7
Haiku 4.5, Befunge-98	4	0	0	0
Kimi K2.5, Whitespace	13	9	0	0
Sonnet 4.6, Brainfuck +Lib	20	18	19	7

J.3 API cost and latency

Table 18 gives the four Claude-family examples reported in the responses. Output includes extended-thinking tokens. Elapsed time runs from launch to the final recorded submission; the active-time proxy sums consecutive transcript gaps shorter than five minutes. Neither measures human operator time. The finalized counts state the coverage of each example, without extrapolating to a complete 80-problem session.

Table 18: Author-reported resource summaries. Tokens are in millions; finalized problems are not solved problems. Costs are approximate historical accounting estimates, not current prices.

Cell	Finalized /80	Output (M)	Cache write (M)	Cache read (M)	Cost (USD)	Active (h)	Elapsed (h)
Opus, B98	77	2.46	2.63	93.5	374	8.9	34.0
Opus, BF	75	2.54	2.60	61.1	331	6.9	34.0
Sonnet, B98	67	3.84	3.68	86.8	97	9.4	60.7
Haiku, B98	79	0.30	0.70	36.6	6	1.3	15.5

The source calculations use rates per million tokens of $(15, 75, 18.75, 1.50)$ for Opus, $(3, 15, 3.75, 0.30)$ for Sonnet, and $(1, 5, 1.25, 0.10)$ for Haiku, ordered as uncached input, output, cache creation, and cache read. These are the response's accounting assumptions. The retained summaries do not provide a complete separate input/reasoning-token breakdown or comparable measurements for every model and language, so the table is an illustrative set rather than a full cost ranking.

J.4 Weak-agent behavior and budget use

The behavioral audit distinguishes failure to discover generator construction from failure to use it successfully. Table 19 records target files and generator files in the audited workspaces. These counts can include intermediate candidates and partial sessions; they are not counts of independently correct solutions.

The first four rows have no observed generator artifacts, whereas Kimi constructs generators in Whitespace and Shakespeare despite low benchmark performance. Sonnet's contrasting behavior across Brainfuck and Befunge-98 also shows that discovery depends on the target. Generator construction alone does not ensure correct implementation, composition, or debugging.

The Brainfuck +Lib responses further report 19.1, 16.9, and 0.7 local runs per problem for mini, Sonnet, and Haiku, respectively; Haiku wrote 17 additional generator files. This supports distin-

Table 19: Artifact counts reported for selected audited workspaces. BF, B98, WS, and Sh denote Brainfuck, Befunge-98, Whitespace, and Shakespeare.

Agent	Language	Target files	Generator files
Haiku 4.5	B98	232	0
Sonnet 4.6	BF	25	0
Kimi K2.5	BF	74	0
GPT-5.4 mini	BF	134	0
Sonnet 4.6	B98	68	4
Kimi K2.5	WS	243	290
Kimi K2.5	Sh	350	8
Opus 4.6	BF	105	67

Table 20: Budget-use summaries from the responses. Output includes thinking. The audited sessions need not coincide with the headline scoring sessions.

Cell	Local runs/problem	Submissions/problem	Output (M)
Opus 4.6, BF	9.2	0.85	2.54
Opus 4.6, B98	12.8	0.82	2.46
Haiku 4.5, B98	1.5	2.90	0.30

guishing access to a strategy from the ability to compose and debug it. The statistics are descriptive, not matched-budget causal estimates.

Table 21: Brainfuck +Text behavior reported in the responses. Solve counts are out of 80; generator counts are artifact counts. Dashes denote conditions not reported. Matching complete trajectories are needed to reconstruct the pairing of scoring and artifact summaries.

Agent	Base solves	Base generators	+Text solves	+Text generators
Opus 4.6	64	64	—	—
GPT-5.4 xhigh	79	79	—	—
Sonnet 4.6	12	0	12	105
GPT-5.4 mini	5	0	8	20
Haiku 4.5	4	0	3	15
Kimi K2.5	4	0	4	35

The revised response to Reviewer 1Qfu reports 64 Opus generator files in this comparison, whereas the workspace audit in Table 19 reports 67. These are source-specific artifact summaries, not interchangeable score counts. The reported +Text condition changes whether weaker agents write generators, while yielding little change in solved counts. This separates instruction to use a strategy from successful execution of that strategy. The relationship to supplied metaprogramming representations [Zhang et al., 2024a], executable skill libraries [Wang et al., 2023], and code actions [Wang et al., 2024] concerns this distinction between spontaneous use and externally supplied structure.

Mechanism scope across languages. The responses discuss the no-metaprogramming comparisons already shown in Figure 3: 80 versus 80 for Opus on Whitespace, 72 versus 67 on Shakespeare, and 77 versus 28 on Befunge-98. The Whitespace comparison is at ceiling, while the Shakespeare difference is smaller than the Brainfuck and Befunge-98 effects. These are ablation-session results, distinct from the headline sessions. Generator construction is therefore not a complete explanation of adaptation across targets. One proposed explanation for its spontaneous use on Brainfuck is that generators provide explicit names and reusable operations for otherwise implicit tape state; this remains a conjecture, not a separately tested intervention.

J.5 Reference-library provenance check

The retained reference-library directories contain both reusable primitives and task-specific generators; the main text and Appendix G.2 describe +Lib accordingly. The Brainfuck files include generators for E04, E10, E12, E19, and X06; the Befunge-98 files include E13, M01, H10, X01, and

X10 examples. The Brainfuck README explicitly permits consulting example generators. The supplementary manifest records the files and hashes for each agent. Retained Befunge-98 libraries agree byte-for-byte across the three agents; the Brainfuck libraries differ in mini's E10 generator. These are retained workspace artifacts, not authenticated initial snapshots of every reported session.

A byte-identity check executed each agent's retained file-producing generators in a scratch directory and compared their output with files referenced by passing submissions. Subtracting each identified problem once changes Brainfuck +Lib totals from 7 to 4 for Haiku, 64 to 60 for Sonnet, and 53 to 51 for mini. The matched problems are E04/E10/E12, E04/E10/E12/E19, and E10/E19, respectively. No byte-identical Befunge-98 matches were found among available passing files; one mini passing file was missing. The supplementary audit records file hashes and match coverage.

These counts retain substantial gains for Sonnet and mini over their original baselines, but do not estimate performance with examples removed. The check does not cover modified copies, semantic overlap, or stdout-only generators, and overlap in a final workspace does not establish the direction of copying. The retained states reproduce all six +Lib solve counts, but the Sonnet and mini Brainfuck states finalize only 73 and 63 of the 80 tasks; completed historical snapshots are not available in those directories. The denominator remains the full 80-task set. Original main-table values are retained, and the sensitivity analysis is limited to recorded passing submissions.

J.6 Broader impacts and software ecosystems

Evaluation on uncommon executable interfaces can help practitioners assess agents for internal DSLs, legacy integration, and unfamiliar APIs. The reported rankings are specific to the tested tasks and protocols; applying them directly to production systems could mislead deployment or budget decisions. This work releases an evaluation harness rather than a new generative model, and does not establish reliability in safety-critical use.

Gu et al. [2026] examine bias in AI programming assistance across software ecosystems. Generating a less familiar language through a common host offers one possible response to such asymmetry, but may also concentrate reusable tooling on the host side. Our experiments measure program-construction behavior, not long-term software adoption or ecosystem change.

Reproducibility and record coverage. The supplementary archive includes a deterministic audit script, sanitized state extracts, retained reference libraries, and available passing programs. The audit reconstructs the stated retained counts and exact-file matches without model calls; it is not a new model evaluation or hidden-test regrade. Missing final APL follow-up states, shuffled permutations, and complete historical library snapshots limit independent reconstruction, as detailed above. Reproduction instructions and source hashes accompany the archive.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The paper studies deployed coding agents under a specified sequential protocol. Sections 3.4 and 4 distinguish the generator intervention from unresolved mechanisms; the library description includes examples.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Section 4 discusses exposure, artificial targets, wrapper differences, and causal scope. Appendices B.6 and J describe statistical dependence and incomplete follow-up records.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: This is an empirical study with no new theoretical results.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Section 2 and Appendix A specify the tasks, scoring, budgets, prompts, and session protocol. The public code supports rerunning the main conditions; Appendix J explicitly identifies follow-up summaries and historical snapshots that cannot be reconstructed from the retained records.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The code is public at `https://github.com/Lossfunk/esolang-metaprogramming` and the dataset at `https://huggingface.co/datasets/Lossfunk/Esolang-Bench`. The accompanying archive documents the main experiment configurations and a separate retained-record audit; it does not contain complete historical trajectories for every reported follow-up.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 2 and Appendix A document the evaluation settings and per-condition deviations. These experiments use existing API models without training or parameter tuning; provider/wrapper sampling defaults and the limits of workspace restrictions are stated.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Table 1 gives nominal Wilson binomial summaries, with the dependence limitation in Appendix B.6; three-session counts are reported in Appendix B.7. These are not formal significance tests, and follow-up point estimates are explicitly labeled with their limited replication and record coverage.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [No]

   Justification: The local interpreter and API budgets are specified, and Appendix J.3 gives four historical cost/latency examples with coverage and measurement definitions. Complete per-cell input/reasoning accounting, operator time, and vendor-side hardware information are unavailable, so full resource reporting for every experiment is not claimed.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research evaluates existing coding agents on programming tasks and does not involve human-subject experiments, private personal data, or model training. This retains the authors' submitted ethics declaration.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Appendix J.6 discusses improved evaluation for uncommon interfaces, overgeneralization of rankings, and possible concentration of tools around common host languages; the latter is a potential implication rather than a measured ecosystem effect.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The released assets are an evaluation harness, programming tasks, prompts, and code examples, rather than new model weights or a high-risk dataset.

    Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: EsoLang-Bench is credited and released under CC BY 4.0; the harness has an MIT license, and the third-party Shakespeare interpreter retains its own license. Benchmark and model sources are cited in Appendix B; the official NeurIPS style file is unmodified.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The supplementary README and HOWTO document the harness, interpreter setup, main experiment cells, prompts, and reference code. The additional audit includes sanitized inputs, source hashes, and explicit coverage limitations.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The work does not involve crowdsourcing or human-subject experiments.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The work does not involve human-subject experiments.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: The core evaluated systems are LLM-based coding agents. Section 2 identifies the models, wrappers, and interactions, and the appendix supplies prompts and intervention details.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.