COIN: A Live, Large-scale Benchmark for Semantics-to-Input Code Reasoning

Jiho Kim[1] **Cen Zhang**[2] **Youngjoon Kim**[1] **Dongkwan Kim**[2] **Brian Junekyu Lee**[1]
Dae R. Jeong[3] Taesoo Kim[1,2]

[1]Georgia Institute of Technology [2]Microsoft [3]Seoul National University

{jkim4050,czhang887,youngjoon.kim,hibrian827,taesoo}@gatech.edu
0xdkay@gmail.com dae.r.jeong@snu.ac.kr

Abstract

Constructing inputs that drive execution to chosen code locations underpins many dynamic-analysis tasks in software engineering and security. It is a generalized constraint-solving challenge that requires agents to reason about the semantic mapping between input and code. We built COIN, the first benchmark to systematically evaluate this ability in modern code agents. By leveraging OSS-Fuzz's long-running coverage data and maintainer-written harnesses, COIN constructs reachability challenges across over 1000 real-world projects in nine languages, and is contamination-resistant and self-refreshing as upstream code and OSS-Fuzz coverage evolve. To keep challenges non-trivial yet solvable, COIN samples two target families: lines OSS-Fuzz has covered but baselines miss, and lines never be reached but with static evidence of a reachable harness path. Across five independent runs on 137 targets from 15 projects in six languages, the strongest agent (Claude Code with Opus 4.6) reaches on average only 26.6% of the C/C++ targets and 39.7% of the Go/JVM/Python/Rust targets, never half of either set in any run, and the weakest (DeepSeek V3.2) stays below 9%, indicating substantial room for improvement.

1 Introduction

Code-capable language models are now widely deployed across software engineering and security tasks. Many of these tasks, especially those involving dynamic analysis (test case generation, vulnerability reproduction, exploit development, bug finding), require the same ability: constructing an input that drives execution to a chosen code location. For agents, this is intrinsically a generalized constraint-solving challenge that requires understanding the semantic mapping between input and code. A natural question follows: given a chosen code line in a real project, how well can the agent construct an input that reaches it?

Limitations of existing benchmarks. Existing benchmarks cannot soundly answer this question.

TestCase-Eval (Yang et al., 2025) is the only existing benchmark dedicated to LLM test-input generation, but it targets self-contained competitive programming problems, not real software, and its task is fault exposure on algorithm implementations instead of reaching arbitrary code lines.

Most input-generation related benchmarks work at project scope, but they evaluate other tasks in which reachability is only an intermediate step rather than the primary challenge: PoC reproduction in real OSS code (Wang et al., 2025c; Luo et al., 2026), real-world exploit generation (Zhu et al., 2025; Zhang et al., 2025a), CTF flag retrieval in constructed challenge environments (Zhang et al., 2025b; Shao et al., 2024), or finding planted vulnerabilities in competition codebases (Zhang et al., 2026). Each of these tasks bundles reachability with other capabilities the agent must bring: reading vulnerability descriptions, selecting exploit primitives, recognizing CTF tropes, or spotting injected patterns. As a result, scores on these benchmarks reflect the combined capability rather than reachability alone. Moreover, their targets come from a particular category (disclosed CVEs, bug-bounty findings, designed CTF challenges, and planted bugs); whether their results generalize to the broader question above is unclear.

Our observation. Despite these limits, our central observation is that, by proper design, benchmarking code understanding via reachability can scale, resist contamination, and run fully automatically, indicating a representative and sustainable way to evaluate agents' code reasoning.

First, OSS-Fuzz provides over 1000 actively maintained projects across nine languages, each shipping with maintainer-written fuzzing harnesses (entry-point programs that exercise specific parts of the codebase); at any given time, roughly 800 of

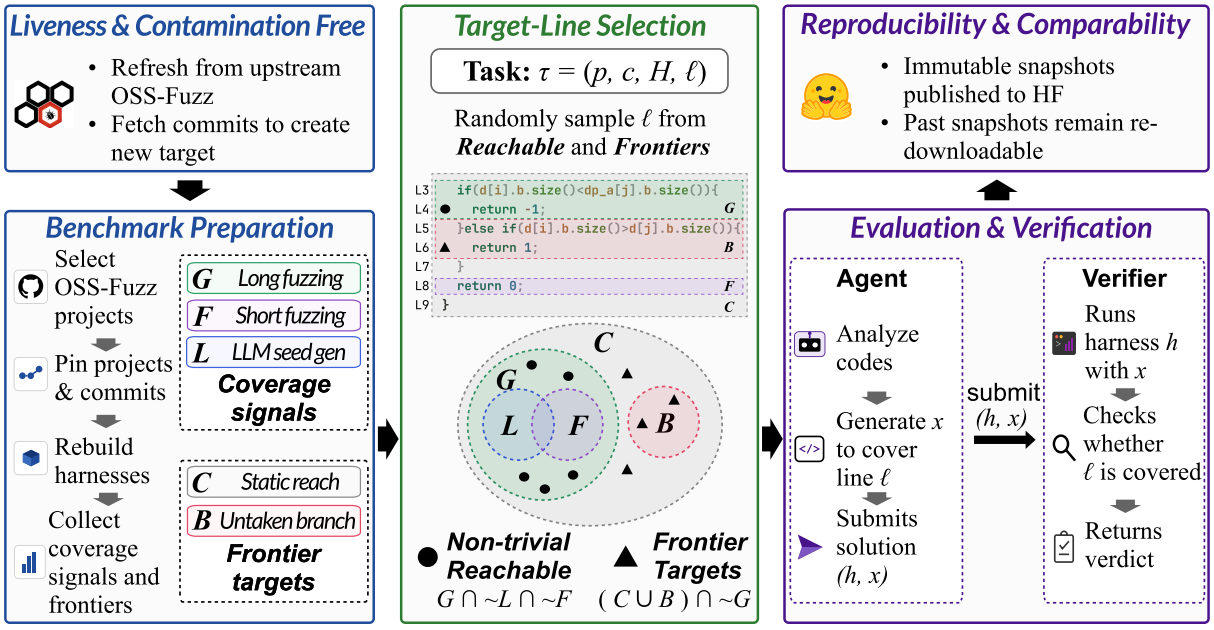


Figure 1: Overview of COIN. COIN prepares pinned OSS-Fuzz builds, extracts dynamic and static reachability signals, selects typed target lines, and evaluates agents through executable verification.

them pass COIN's eligibility gate (a fresh coverage report and a pinned source commit), a pool whose size has held stable since late 2023. Reachability challenges can be sourced by selecting candidate lines from the project code reachable through these harnesses. Second, the challenge supply is self-refreshing and contamination-resistant: lines that are not yet covered by default OSS-Fuzz fuzzing corpus means no public available input is known, thus contamination-free tasks can be drawn from these. Third, with existing coverage instrumentation, validating whether an input reaches the target line is straightforward and deterministic.

COIN From this observation, we built COIN, a benchmark of line-reachability tasks in real OSS-Fuzz projects. To set up a task, we pick a project at a specific commit and a target line of source code in it. Under a fixed wall-clock budget, the agent under evaluation then receives the project source and its existing fuzzing harnesses, and constructs an input for one of them. Finally, we run the input on the coverage-instrumented build and accept the submission if the target line is covered.

Target-line selection. Random lines could be trivial or infeasible. COIN avoids both by sampling two target families: ❶ non-trivial reachable targets, lines OSS-Fuzz's long-running fuzzing has covered but baselines (fuzzers and untargeted code agents) miss; and ❷ frontier targets, lines not yet covered, but with static evidence of a reachable harness path.

Self-refresh. COIN's construction pipeline automatically re-runs on upstream code and OSS-Fuzz coverage updates, producing fresh target lines and preventing benchmark saturation.

Findings. We evaluate eight state-of-the-art code agents on 137 challenges generated by COIN (70 from seven C/C++ projects, 67 from eight Go, JVM, Python, and Rust projects), repeating the complete evaluation five times to quantify run-to-run variance. The strongest agent (Claude Code with Opus 4.6) reaches on average only 26.6% of the C/C++ targets and 39.7% of the multi-language targets, never half of either set in any run, with the open-weight DeepSeek V3.2 at 4.0% and 8.7%. Precision is low as well: on the C/C++ targets, mean precision ranges from 8.3% to 47.0% across models, so even the best-calibrated agents submit inputs that miss their target more often than not. Within the OpenAI and Gemini families, flagships substantially outperform their lightweight counterparts, and open-weight models remain well below the flagship tier; together, these gaps leave substantial headroom for improving agent reachability.

Contributions. Our contributions are: (i) the observation that benchmarking code understanding via reachability on real OSS-Fuzz projects can scale, resist contamination, and run fully automatically, making it a representative and sustainable angle for evaluating agents' code reasoning; (ii) a frontier-target selection method combining long-running

OSS-Fuzz coverage with non-targeted baselines to retain only lines that require targeted reasoning; (iii) a fully automated pipeline that turns OSS-Fuzz into a continuously updated COIN, released with full artifacts and Croissant metadata; and (iv) state-of-the-art agent evaluation on COIN, ablated by model and tool access.

2 Benchmark

Figure 1 shows the benchmark pipeline. COIN evaluates code reasoning by framing the reasoning task as input generation: an agent must produce inputs whose execution reaches specified code or increases project coverage. The pipeline has three stages: ❶ preparation pins projects and versions from OSS-Fuzz and extracts coverage signals and static-analysis results, ❷ target line selection uses those signals to choose non-trivial reachable targets and frontier targets, and ❸ evaluation executes agents on the selected tasks, while verification replays their outputs on instrumented builds to compute the final task scores. COIN is live and reproducible: it refreshes daily against upstream projects, samples tasks after each refresh to reduce contamination, and snapshots the evaluation environment used for scoring.

2.1 Task formulation

Task. COIN evaluates code understanding through executable evidence: given (p, c, H, ℓ), the agent must submit a pair (h, x) where $h \in H$ and executing h on input x covers ℓ. The project source $p@c$ defines what the agent must reason about, including the semantic meaning of lines, constraints, control flow, and call chains. The fuzzing harness set H provides the input interfaces and runtime paths through which those code properties can be exercised. Each harness fixes the program entry point and accepted input format. The agent solves the task only through program execution: it selects a harness $h \in H$ and provides an input x that makes the run cover ℓ.

The benchmark has two task types. *Targeted reachability* is the primary task and asks the agent, as described above, to cover one specified line. *Coverage maximization* removes the single-line target and instead asks the agent to construct a corpus that explores as much reachable project code as possible through the same harness set.

Score. *Targeted reachability* reports a success rate over the sampled target-line set. For each sampled target line ℓ, the agent receives the project p, commit c, fuzzing harness set H, available tools, and task metadata. It submits a solution pair (h, x), where $h \in H$ and x is the byte string run through that harness. The solution is valid exactly when replaying $h(x)$ within the fixed time budget produces line coverage containing ℓ, written as $\ell \in \mathrm{Cov}\big(h(x)\big)$. The final targeted score is the fraction of sampled target lines for which the agent submits a valid solution.

Coverage maximization has no target line and no live score while the agent is running. The agent returns a finite input set X_h for each harness $h \in H$. After the run finishes, the verifier replays every submitted $h(x)$ and computes standard project-level line coverage:

$$\frac{\left|\bigcup_{h \in H} \bigcup_{x \in X_h} \mathrm{Cov}\big(h(x)\big)\right|}{\text{project executable lines}}. \tag{1}$$

The denominator is the number of executable lines in the project source, independent of coverage results. Thus the score measures how much of the project's executable lines the submitted corpus covers, not the fraction of all semantically reachable lines. Each covered line counts once, so redundant inputs do not help, and skipping a harness matters when that harness reaches code not reached by the others.

2.2 Benchmark preparation

COIN constructs tasks from projects that OSS-Fuzz already builds and covers successfully. The preparation pipeline first selects projects, then reproduces their builds at pinned commits, and finally computes the dynamic and static signals used for task construction.

Project selection. The pipeline starts from the latest synchronized OSS-Fuzz metadata, including coverage reports, source and OSS-Fuzz commits, build status, project size, and language. It keeps projects with successful upstream fuzz and coverage builds, known source and OSS-Fuzz commits, sizes within the configured range, and one of the nine supported languages. From the filtered projects, COIN samples projects randomly using a fixed seed. The resulting build batch records the OSS-Fuzz checkout and source commits used for every selected project, so the same projects can be rebuilt later under the same infrastructure state.

Pinned builds. For each selected project, COIN builds the harness set H twice: once for ordinary

fuzzing and once for coverage collection. Only project commits whose fuzzing and coverage builds both succeed advance to task extraction. We use these harness binaries to collect coverage evidence and to run and verify agent submissions.

Coverage and frontier evidence. COIN computes five line-level evidence sets on the same pinned project state for target-line selection. *Coverage evidence* record which lines are reached by non-targeted execution. $\mathcal{G}$ is the union of lines covered in the public OSS-Fuzz coverage data across the project's harnesses, giving long-running fuzzing evidence that those lines are reachable. $\mathcal{F}$ is the union of lines covered by fast non-targeted fuzzing. We ran the default harnesses under a fixed budget to compute $\mathcal{F}$. $\mathcal{L}$ is the union of lines covered by fast non-targeted execution from LLM-generated seeds. We generated those seeds with a target-blind prompt: the model sees harness context but not the target line. The seed-generation prompt is listed in §A.1. Together, these evidence sets identify lines that are reachable under long-running fuzzing or already covered by fast non-targeted procedures.

Frontier evidence identifies lines that long-running OSS-Fuzz coverage has not reached, but that still have evidence of a plausible harness path. $\mathcal{C}$ is static CodeQL (GitHub, Inc.) reachability evidence, recording which harnesses can reach a source line in the call graph; lines in $\mathcal{C} \setminus \mathcal{G}$ form a static frontier. $\mathcal{B}$ is untaken-branch frontier evidence derived from CodeQL branch extraction and the dynamic reached set $\mathcal{G} \cup \mathcal{F} \cup \mathcal{L}$. CodeQL first extracts the condition line and source span of each executable branch or loop body. The pipeline then overlays dynamic coverage on that structure. For example, consider a two-branch conditional. If the condition and one branch appear in $\mathcal{G} \cup \mathcal{F} \cup \mathcal{L}$, but the first executable line of the other branch does not, then that line receives the $\mathcal{B}$ label. The target is still a source line, not a branch identifier: $\mathcal{B}$ records that the line is the first executable line of an untaken frontier branch of a conditional whose sibling side has already been exercised.

2.3 Target-line selection

A useful target avoids two extremes. Trivially reachable lines fall to generic fuzzing, so reaching them reveals little about code reasoning. Lines with no valid harness path admit no input, so failing to reach them says little about the agent. COIN therefore partitions targets into typed pools and samples each with a fixed random seed.

Non-trivial reachable targets. The first target family is sampled from $\mathcal{G} \setminus (\mathcal{F} \cup \mathcal{L})$, namely historically reachable lines that are not covered by fast non-targeted procedures. $\mathcal{G}$ provides empirical evidence that a reaching input exists, while excluding $\mathcal{F}$ and $\mathcal{L}$ removes lines that fast target-blind procedures can reach without target-specific code reasoning. Reaching these targets therefore requires the agent to rediscover a witnessed path that cheap baselines miss.

Frontier targets. The second target family is sampled from $(\mathcal{B} \cup \mathcal{C}) \setminus (\mathcal{G} \cup \mathcal{F} \cup \mathcal{L})$. The sampler weights $\mathcal{B}$ higher so untaken-branch frontiers appear more often. $\mathcal{C}$ contributes lines that are statically reachable from a harness. $\mathcal{B}$ captures the first executable line of an untaken branch or loop body adjacent to already reached code. Removing $\mathcal{G} \cup \mathcal{F} \cup \mathcal{L}$ keeps only lines not reached by long-running coverage or fast non-targeted procedures. Frontier targets therefore give the agent lines to reach beyond the current dynamic coverage. Because a frontier target may in fact be unreachable from all valid harness inputs, the agent may record why it believes the frontier cannot be crossed.

2.4 Agent execution and result verification

Evaluation separates what the agent may do from how the benchmark scores it. During execution, the agent receives the task information and available tools, then produces a final targeted input, an abstention rationale, or a coverage corpus. Verification is deterministic: COIN replays the submitted artifacts on pinned harness builds and derives the final outcome from observed coverage.

Execution. *Targeted reachability* task runs one task for each sampled target line. The agent receives the task tuple (p, c, H, ℓ), the project source at $p@c$, the harness set H, and tools for code analysis and local execution. The prompt template is shown in §A.2. The agent's ordinary output is one final pair (h, x), where $h \in H$. If the agent believes the target is unreachable, it may instead abstain and return a short rationale r; this is mainly a debugging aid for understanding failures, not the central scoring path.

Coverage maximization task uses the same project and harness information but no target line. The agent receives (p, c, H) and the same tools, and returns a harness-indexed corpus $\{X_h\}_{h \in H}$. The initial prompt template is shown in §A.4. There is no official live coverage oracle during the

run. If the agent wants coverage feedback, it must measure coverage itself. To keep the agent working until the time budget expires, COIN relaunches the agent with the continuation prompt (§A.5).

Verification. Verification runs after the agent finishes. *Targeted reachability* task verification replays the submitted pair (h, x), and the verifier checks $\ell \in \mathrm{Cov}\big(h(x)\big)$. The agent does not receive verifier feedback during the run. The result contributes a hit or miss to the targeted success rate.

Coverage maximization task verification batch-replays the submitted corpus by running every reported input $x \in X_h$ on its harness h. It measures coverage under the pinned coverage build, unions the covered lines across harnesses, and reports the project-level score defined in Eq. 1.

2.5 Liveness and Reproducibility

COIN supports both liveness and reproducibility with daily refreshes and immutable snapshots. Each refresh follows upstream OSS-Fuzz, samples a fresh task set, and publishes the task data and scoring artifacts needed to rerun it.

Liveness by design. COIN is a benchmark of changing commits, not a fixed list of questions. The pipeline reruns daily against upstream OSS-Fuzz: it ingests current project state, discovers harnesses, extracts signals, selects targets, and packages tasks. Fresh commits create new line states, new frontier targets, and new coverage opportunities whose answers did not exist in older training corpora. An eight-year monthly reconstruction verifies that this supply is plentiful and stable (Appendix G).

Contamination resistance. The code, harnesses, target lines, and construction evidence are public or reproducible from public artifacts, but the answers are not published. Random sampling adds another protection: even for the same project commit, COIN can sample new target lines from the candidate pools, so one released snapshot does not determine the next task set.

Snapshot reproducibility. Random sampling also means that pinning commits alone is not enough to reproduce an evaluation. Each refresh therefore publishes a versioned snapshot, for example through Hugging Face, containing the exact selected targets, coverage-mode project tasks, evidence metadata, generated Docker images, and built harness binaries used for scoring. Re-evaluation loads this snapshot rather than reselecting targets, so results remain comparable even after the live pipeline has moved on or upstream build dependencies change or disappear (storage and rebuild requirements in Appendix H).

3 Evaluation

We evaluate COIN along three axes: the quality of target-line selection from OSS-Fuzz projects (§3.2), how state-of-the-art agents perform on the selected targeted-reachability tasks (§3.3), and how the same agents fare on coverage maximization (§3.4). §3.1 fixes the shared experimental setup.

3.1 Experiment Setup

Projects and target lines. We sample eight C/C++ OSS-Fuzz projects and draw ten target lines from each: five from the non-trivial reachable family and five from the frontier family. One project (`xerces-c`) exposes no eligible line and is dropped, leaving 70 target lines across seven projects. A second set of eight projects, randomly sampled two per language from Go, JVM, Python, and Rust, adds 67 further target lines (§3.2).

Agents and models. We evaluate eight models each driven by its provider's native agent scaffold: Claude Opus 4.6 and Claude Sonnet 4.6 under Claude Code; GPT-5.4 and GPT-5.4-mini under Codex; Gemini 3.1 Pro and Gemini 3 Flash under the Gemini CLI; and the open-weight DeepSeek-V3.2 and GLM-5, served through Vertex AI and driven under Claude Code.

Sandbox and tool access. Each agent runs inside a Docker container with full root privileges and unrestricted network egress. Within the container the agent may invoke arbitrary shell commands, read and write any file in the project tree, and access the open web. This gives the agent the same practical privileges as a human engineer. Open access does not create a shortcut: the ground-truth witness is produced only by execution and is not available on the web (§2.5). For the targeted-reachability coverage-tool ablation, we additionally mount the project's pre-replayed seed corpus and expose `corpusdb`, a coverage-measurement tool that reports which seeds touch a file or line range and exports those seed bytes for mutation.

Resource allocation. We assign only wall-clock time budgets. The exact time budget depends on the experiment: baseline construction, targeted reachability, and coverage maximization do not necessarily use the same wall-clock limit. We impose

Table 1: C/C++ benchmark projects and construction summary.

Project	Harn.	kLOC	GCS (%)	Fuzz (%)	LLM (%)	Cost ($)
`cups`	9	30.3	28.4	25.6	30.5	53.36
`cyclonedds`	6	81.3	26.3	13.8	25.3	43.41
`hdf5`	2	144.9	17.6	14.8	18.9	8.42
`karchive`	9	148.3	8.0	7.9	7.6	37.03
`liboqs`	4	165.2	47.7	48.5	48.5	18.75
`libraw`	4	38.0	80.3	38.4	35.3	31.78
`rdkit`	3	46.5	43.2	27.4	32.0	17.18
Total	**37**	**654.5**				**209.93**

no separate LLM token or cost cap, and do not otherwise limit CPU or RAM.

Evaluation metrics. For dataset creation, we report two construction metrics: the *creation cost*, the API spend to generate a benchmark (the fuzzing and LLM seed-generation baselines, CodeQL analysis, and target extraction), and the *benchmark size*, the number of evaluable target lines the pipeline yields.

For *targeted reachability*, we report *reach rate* and *precision*. *Reach rate* divides reached target lines by all attempted target lines. *Precision* divides reached target lines by submitted inputs. These differ because an agent may abstain: a cautious agent can submit few inputs and still have high *precision*, while its *reach rate* stays low. For *coverage maximization*, we report *line coverage*: the union of covered lines over the agent's corpus divided by the project's executable lines.

3.2 Dataset Generation

Table 1 shows the result of randomly selecting eight OSS-Fuzz projects and generating a benchmark from each. One (`xerces-c`) yields no eligible target line, not every project is guaranteed to expose the kind of target line we want, but it still serves the coverage-maximization task (§3.4). The seven remaining projects ship 37 fuzz harnesses. Table 1 reports executable kLOC, published OSS-Fuzz coverage (*GCS*), and the coverage from two local two-hour baselines: libFuzzer (*Fuzz*) and target-blind LLM seed generation (*LLM*), both measured by `corpusdb` replay. The *Cost* column is the LLM seed-generation API spend; the dropped `xerces-c` adds $6.65, bringing the construction total to $217.

Coverage spans a wide range: `karchive` near 8% by every measure, `libraw` 80% covered upstream but only ∼36% from the local baselines, and for several projects the two-hour local baselines already match the public OSS-Fuzz coverage.

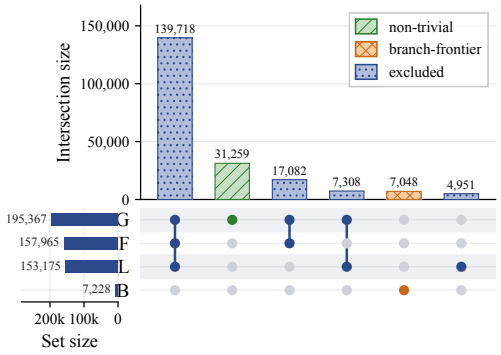


Figure 2: Signal-cube UpSet over candidate uncovered lines in the C/C++ projects (six largest intersections shown). Highlighted regions show the non-trivial reachable selection and untaken-branch frontier evidence.

Cost of construction. Constructing the benchmark is cheap. The eight-model seed-generation baseline cost $217 over 312 two-hour runs (per-project totals in Table 1; fuzzing adds only compute). Table 5 breaks this down by (project, harness, model): the costliest single run was Opus on `cups`'s `fuzz_ppd_gen_1` ($5.74), while lightweight models often spend under a cent per harness. Over the seven retained projects, Opus is the most expensive ($71.60) and Gemini 3 Flash the cheapest ($4.85). Seed coverage also varies: Gemini 3.1 Pro reaches the most (34.6% peak on `liboqs`), DeepSeek-V3.2 the least.

Target supply. Target lines are plentiful. Across the seven C/C++ projects (Figure 2), 639,410 candidate uncovered lines yield 31,259 non-trivial reachable lines ($\mathcal{G} \setminus (\mathcal{F} \cup \mathcal{L})$) and 7,048 branch-frontier targets. The eight multi-language projects add 192,961 candidates, with 1,564 non-trivial reachable and 884 branch-frontier lines (per-project breakdowns in Figure 5 and Figure 6). We sample five per family per project to bound wall-clock and API cost.

Beyond C/C++. To validate that construction generalizes across OSS-Fuzz's language ecosystem, we repeat it on eight further projects, randomly sampled two per language from Go, JVM, Python, and Rust (Table 2). All eight randomly drawn projects built on the first draw, and the pipeline ran unmodified end-to-end, yielding 67 target lines (37 frontier, 30 non-trivial reachable) at $324 construction cost (measured under the same max-of-counters spend rule as §3.3). Yields are reported as drawn: two projects under-fill the frontier family, and two yield no non-trivial reachable line because

Table 2: Multi-language construction summary: eight OSS-Fuzz projects, randomly sampled two per language. Targets = selected branch-frontier (F) and non-trivial reachable (R) lines; Cost = LLM seed-generation spend during benchmark construction, not agent evaluation.

Project	Lang	Harn.	kLOC	GCS (%)	Fuzz (%)	LLM (%)	Cost ($)	Targets (F+R)
`protocompile`	Go	1	22.1	13.1	5.8	21.0	5.61	5+5
`syzkaller`	Go	4	18.4	35.1	19.5	38.2	45.81	5+5
`jfreechart`	JVM	3	51.5	7.4	7.4	7.4	20.79	5+0
`xstream`	JVM	1	26.9	31.4	25.3	27.4	4.71	5+5
`libcst`	Python	1	16.8	63.5	57.0	60.4	3.85	5+5
`sqlalchemy-utils`	Python	3	75.9	43.0	42.9	43.0	15.27	5+0
`bson-rust`	Rust	7	44.5	16.9	15.7	13.8	37.21	3+5
`gitoxide`	Rust	35	113.5	20.9	18.4	18.2	191.18	4+5
Total		**55**	**369.6**				**324.43**	**37+30**

Table 3: Per-model reach, precision, and total LLM spend on the 70 C/C++ targets under the two targeted-reachability conditions (base = standard sandbox and tool access; tool = base plus `corpusdb`-based coverage measurement). Each cell is mean ± sd over five independent runs. Sorted by base reach rate.

	Reach %		Precision %		Cost ($)	
Model	base	tool	base	tool	base	tool
Opus 4.6	$26.6_{\pm 3.1}$	$29.7_{\pm 0.6}$	$46.7_{\pm 5.2}$	$54.5_{\pm 1.3}$	$354_{\pm 46}$	$350_{\pm 10}$
Sonnet 4.6	$25.7_{\pm 0.0}$	$23.4_{\pm 1.3}$	$46.3_{\pm 2.4}$	$45.1_{\pm 2.3}$	$388_{\pm 31}$	$342_{\pm 34}$
GPT-5.4	$25.7_{\pm 2.3}$	$25.1_{\pm 3.3}$	$44.3_{\pm 2.1}$	$48.9_{\pm 5.2}$	$104_{\pm 10}$	$106_{\pm 10}$
Gemini 3.1 Pro	$22.6_{\pm 2.6}$	$24.6_{\pm 2.1}$	$47.0_{\pm 3.7}$	$62.0_{\pm 6.7}$	$133_{\pm 17}$	$205_{\pm 29}$
GPT-5.4-mini	$15.1_{\pm 2.8}$	$13.4_{\pm 2.6}$	$24.9_{\pm 4.7}$	$23.3_{\pm 6.2}$	$46_{\pm 14}$	$42_{\pm 4}$
GLM-5	$14.6_{\pm 2.1}$	$16.0_{\pm 1.9}$	$29.8_{\pm 4.3}$	$34.1_{\pm 3.8}$	$128_{\pm 20}$	$137_{\pm 23}$
Gemini 3 Flash	$14.6_{\pm 1.2}$	$17.4_{\pm 3.3}$	$16.5_{\pm 1.1}$	$21.6_{\pm 4.7}$	$66_{\pm 6}$	$125_{\pm 18}$
DeepSeek-V3.2	$4.0_{\pm 3.1}$	$4.6_{\pm 1.9}$	$8.3_{\pm 5.1}$	$9.1_{\pm 3.5}$	$32_{\pm 9}$	$35_{\pm 8}$

their local baselines already saturate the public coverage, mirroring `xerces-c` in C/C++. Agent results on these 67 targets appear at the end of §3.3.

3.3 Targeted Reachability

We run all eight models on all 70 C/C++ target lines under the main *base* condition and a *coverage-tool* ablation that enables `corpusdb`-based coverage measurements over the project's seed corpus. To quantify run-to-run stochasticity, we repeat the complete evaluation five times end-to-end (the original run plus four independent replications) under the identical snapshot, harness binaries, agent versions, and prompts, and report each metric as mean ± sample standard deviation over the five runs. Reported spend is measured at our LLM gateway, taking per task the maximum of its billing counter and its request-log total, after cross-validating both against agent-side session logs (either source can drop events under load, but neither double-counts). Unless noted, prose figures use the *base* condition.

Reach rate per model. Claude Opus 4.6 leads at 26.6%, with Sonnet (25.7%) and GPT-5.4 (25.7%) statistically indistinguishable from it, and Gemini 3.1 Pro (22.6%) close behind; GPT-5.4-mini, GLM-5, and Gemini 3 Flash cluster near 15%, and DeepSeek-V3.2 trails at 4.0%. The flagship–lightweight tier separation is significant for every cross-tier pair (Welch's t-test, $p < 0.01$), while rankings *within* the flagship tier are not stable across runs. No model exceeds 30.0% reach in any of the five runs, and no single model reaches the union of solved target lines: per run, the best single model solves 19–21 of 70, while the union across models is 22–24. Successes also concentrate in the non-trivial reachable family. The best model averages 52.0% reach there, while only one of the 35 frontier targets is ever reached in any run (per-family results in Appendix D).

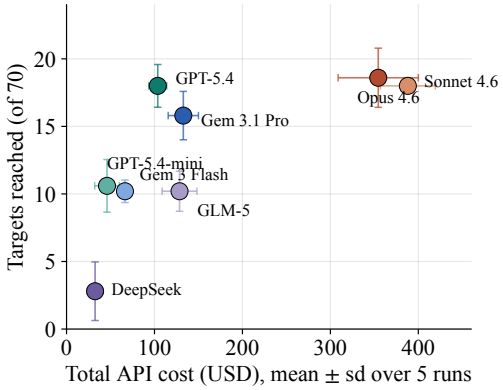


Figure 3: Per-model total API spend vs. C/C++ target lines reached (mean ± sd over five runs; one dot per model). Spend saturates: GPT-5.4 matches Opus and Sonnet within noise at roughly a quarter of their spend. Precision and multi-language counterparts: Figure 7 and Figure 8 in the appendix.

Precision per model. Gemini 3.1 Pro (47.0%), Opus (46.7%), and Sonnet (46.3%) are the best calibrated, roughly half of what they hand in lands, with GPT-5.4 (44.3%) just behind. At the other end, Gemini 3 Flash and DeepSeek-V3.2 submit aggressively but their answers rarely work (16.5% and 8.3%). `corpusdb` moves precision unevenly: it lifts Gemini 3.1 Pro to 62.0%, the table's top, Opus to 54.5%, and Flash from 16.5% to 21.6%, while Sonnet and GPT-5.4-mini are flat to slightly down.

Flagship vs lightweight. Pairing each family flagship-to-lightweight (Opus vs Sonnet, GPT-5.4 vs GPT-5.4-mini, Gemini 3.1 Pro vs Gemini 3 Flash), the three flagships average 25.0% reach versus 18.5% for their lightweights; the OpenAI and Gemini pairs degrade sharply (+10.6 and +8.0 points) while Sonnet essentially matches Opus (+0.9). Cost per success (per-run spend over

reached targets, averaged across runs) runs the *opposite* way for two of the three pairs: GPT-5.4-mini ($4.7 vs $5.8) and Flash ($6.5 vs $8.5) are cheaper per reach than their flagships, while Sonnet ($21.6) costs slightly more per success than Opus ($19.3). The commercial–open gap is larger: commercial models average 21.7% versus 9.3% for the two open-weights (DeepSeek-V3.2, GLM-5), and the open pair is *not* cheaper per success ($12.4 vs $11.1 per reached); their lower token price is cancelled by their lower precision.

Tool-use ablation. Enabling `corpusdb`-based coverage measurements over the mounted seed corpus leaves aggregate reach essentially unchanged (19.3% vs 18.6%, pooled over runs) but reshuffles which cells succeed: per run, 17–28 (model, target line) cells are solved only under *base* and 22–30 only with *coverage-tool*. The per-model effect is run-consistent only for three models: `corpusdb` never lowers Opus or Gemini 3.1 Pro in any run ($+3.1$ and $+2.0$ points on average, shrinking Opus's run-to-run sd from 3.1 to 0.6), and lowers Sonnet's reach in all five runs (-2.3); the remaining models move within noise (Flash $+2.9$, GLM-5 $+1.4$, GPT-5.4-mini -1.7, GPT-5.4 -0.6, DeepSeek $+0.6$, with mixed signs across runs). Either way, the set of distinct solvable target lines is essentially identical (22–24 per run).

Trajectory analysis of `corpusdb` use. Agent trajectories across the five repetitions explain the opposite per-model effects: benefiting from `corpusdb` depends on a multi-stage process, in which the model must select a seed that reaches the relevant upstream path, preserve its format and offset constraints while modifying it, and maintain the target's preconditions. Models fail at different stages of this chain, and the differences are not explained by distraction or tool-use frequency (the aggregate change is small, 521 of 2,800 attempts without the tool versus 540 with it, with Opus improving from 93 to 104 of its 350 attempts and Sonnet declining from 90 to 82). Appendix E walks through representative trajectories; we treat these observations as descriptive mechanistic evidence, not causal estimates.

Multi-language targeted reachability. We run the same five-repetition protocol on the 67 multi-language targets (Table 4). The C/C++ tier structure carries over. Opus leads at 39.7% mean reach, GPT-5.4 (35.2%) and Sonnet (30.4%) follow, and no model reaches half the targets in any run under either condition (the maximum is 44.8%). Run-to-run variance is larger than on C/C++. Gemini 3.1 Pro solves between 6 and 22 targets across the five base runs (18.8 ± 8.9%), through abstention-heavy trajectories rather than scaffold failures. The `corpusdb` effect also differs from C/C++. The tool never raises Sonnet's reach in any run (-4.8 points on average), helps DeepSeek-V3.2 in four of five runs ($+3.6$), and moves the remaining models within noise (Appendix E). The five multi-language runs cost $10,576 in total, and Gemini 3.1 Pro accounts for the largest share by far, driven by long, context-heavy trajectories (Appendix C). On the multi-language set, agents reach 7 of the 37 frontier targets at least once (versus 1 of 35 on C/C++), while frontier reach remains far below the non-trivial reachable family for every model (Appendix D).

Table 4: Multi-language targeted reachability: per-model reach, precision, and total LLM spend over the 67 Go/JVM/Python/Rust targets (mean $\pm$ sd over five independent runs; conditions as in Table 3).

	Reach %		Precision %		Cost ($)	
Model	base	tool	base	tool	base	tool
Opus 4.6	$39.7_{\pm 4.0}$	$37.9_{\pm 2.3}$	$73.6_{\pm 7.8}$	$71.3_{\pm 3.7}$	$186_{\pm 4}$	$203_{\pm 10}$
GPT-5.4	$35.2_{\pm 2.9}$	$32.5_{\pm 2.2}$	$69.4_{\pm 5.3}$	$65.7_{\pm 3.3}$	$63_{\pm 3}$	$73_{\pm 5}$
Sonnet 4.6	$30.4_{\pm 4.6}$	$25.7_{\pm 2.2}$	$57.6_{\pm 8.0}$	$49.9_{\pm 2.8}$	$184_{\pm 18}$	$191_{\pm 24}$
GPT-5.4-mini	$19.4_{\pm 2.1}$	$16.4_{\pm 1.5}$	$38.0_{\pm 3.4}$	$30.0_{\pm 3.6}$	$20_{\pm 4}$	$23_{\pm 2}$
Gemini 3.1 Pro	$18.8_{\pm 8.9}$	$16.7_{\pm 6.7}$	$37.2_{\pm 21.2}$	$32.0_{\pm 20.4}$	$263_{\pm 37}$	$623_{\pm 140}$
GLM-5	$15.8_{\pm 3.4}$	$16.4_{\pm 4.2}$	$27.3_{\pm 5.5}$	$27.2_{\pm 7.7}$	$67_{\pm 12}$	$73_{\pm 11}$
Gemini 3 Flash	$13.4_{\pm 3.7}$	$13.1_{\pm 5.1}$	$16.0_{\pm 4.0}$	$15.8_{\pm 5.6}$	$37_{\pm 5}$	$65_{\pm 11}$
DeepSeek-V3.2	$8.7_{\pm 1.9}$	$12.2_{\pm 2.7}$	$20.1_{\pm 4.3}$	$25.7_{\pm 5.5}$	$22_{\pm 2}$	$22_{\pm 3}$

3.4 Coverage Maximization

We also run the coverage-maximization experiment on the same projects, with a two-hour budget per model and project.

Coverage over time. The primary signal is coverage over elapsed time (Figure 4): models differ not only in final coverage, but in how quickly they turn time into new covered lines. Claude Opus 4.6 and Claude Sonnet 4.6 reach high coverage early and remain ahead, while Gemini 3.1 Pro rises more slowly but finishes third. Lightweight and open-weight models plateau lower, even when they continue producing inputs.

Final coverage and cost. Table 12 (appendix) reports the final endpoint: the strongest agents cluster near 30% mean coverage, while the weakest agents remain near half that level. This mirrors the targeted-reachability ranking and cross-validates it on an independent task. The coverage-maximization runs cost $1,011 in total: the

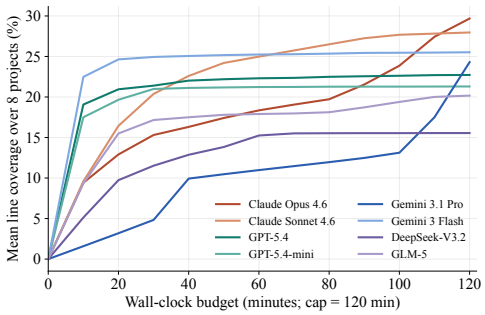


Figure 4: Coverage vs. elapsed time, per model (mean over the eight C/C++ projects).

strongest agents are also the most expensive, but the cheapest agents are not proportionally worse, so cost ranking does *not* track coverage ranking.

Iteration dynamics. Most runs obtain the bulk of their final coverage early, then add little in later iterations, which explains the flattening coverage-time curves; Sonnet is the cleanest exception, with no zero-gain iterations in our runs. Figure 9 and Table 13 give the full breakdowns.

Coverage is not a target proxy. Coverage is not a substitute for target-line evaluation: on `liboqs` the coverage-maximization task touches ∼47.7% of executable lines yet hits 0 of the 10 selected target lines, the central argument for scoring whether specific hard lines are reached, not aggregate coverage.

4 Related Work

Test Input Generation Benchmarks. The benchmark most closely related to ours is TestCase-Eval, which evaluates agent performance on input generation tasks for algorithmic problems (Yang et al., 2025). Owing to the nature of algorithmic challenges, the benchmark adopts fault coverage and fault exposure as evaluation metrics, requiring generated test inputs to thoroughly assess the correctness and robustness of submitted solutions.

Security Task Benchmarks. In security test benchmarks, input generation plays an important role but is evaluated jointly with other capabilities. These benchmarks commonly assess LLM agents on vulnerability exploitation (Zhu et al., 2025), CTF solving (Zhang et al., 2025b; Shao et al., 2024), fuzzing (Metzman et al., 2021; Li et al., 2021), or end-to-end security workflows (Wang et al., 2025c; Lee et al., 2025). Other benchmarks focused on vulnerability analysis (Zhang et al., 2024), vulnerability detection (Fan et al., 2020), and automated patch generation (Byun et al., 2025; Wang et al., 2025a) do not require input generation in their evaluation.

Software Engineering Benchmarks. Other software engineering benchmarks evaluate agents' code reasoning capabilities through a variety of tasks beyond test input generation, including code completion (Zhuo et al., 2025; Liu et al., 2023), test case or test suite generation (Wang et al., 2025b; Jain et al., 2025a), solving algorithmic challenges (Jain et al., 2025b), and answering code-related questions (Chen et al., 2025; Xu et al., 2025). More recent benchmarks further broaden the evaluation scope to include increasingly complex tasks, such as full repository generation (Ding et al., 2025), resolving GitHub issues (Jimenez et al., 2024; Mündler et al., 2024), and end-to-end software engineering workflows (Just et al., 2014; Qiu et al., 2025).

5 Discussion

What COIN does and does not measure. COIN measures one underexplored capability: backward reasoning from a target line to an input that reaches it, in real codebases, under tool-permissive conditions. It is not a measure of code generation, patching, or end-to-end engineering skill, and OSS-Fuzz, its harnesses, and static analysis serve only as construction infrastructure, not as the task itself. A high COIN score is necessary but not sufficient for downstream code work, while a low score identifies a reasoning gap that other benchmarks do not isolate.

Implications. Examples of "here is an input that exercises this branch" are rare in pretraining corpora outside fuzzing corpora themselves. COIN's refresh cycle provides a continuous source of (target, witness) pairs that downstream training could consume. We do not pursue this here, keeping the benchmark and any model trained on its outputs strictly separated.

6 Conclusion

We presented COIN, the first large-scale, contamination-resistant benchmark to systematically evaluate language-model agents' code semantic reasoning through reachability challenges. Even the strongest agent reaches on average only 26.6% (C/C++) and 39.7% (multi-language) of the targets over five runs, leaving substantial room for improvement.

Limitations

Challenge difficulty control. COIN controls challenge difficulty by removing lines that our libFuzzer and LLM seed-generation baselines can already cover, keeping the rest as the evaluation pool. This is empirical rather than principled: the retained pool depends on the specific baseline runs, its difficulty distribution is uncontrolled, and no per-challenge difficulty label is produced that downstream analysis could use to attribute agent failures or compare agents along a difficulty axis. Structural descriptors we examined (call depth, branch conditions, and related features) show no statistically significant association with solve outcomes. We attribute this to the construction itself. The baselines remove targets that are easy along exactly these axes, so the surviving pool is truncated in those dimensions and their residual variation no longer predicts difficulty. We leave the difficulty-aware challenge construction as future work.

OSS-Fuzz constraints. COIN is built on OSS-Fuzz, which both supplies and bounds its substrate. Although the coverage already spans nine languages and over 1000 projects, challenges cannot be drawn from projects outside OSS-Fuzz, and even within OSS-Fuzz they are restricted to a project's existing harnesses. Automatic OSS-Fuzz integration and automatic fuzz-harness generation are complementary research directions; progress on either would directly broaden COIN's evaluation coverage and challenge diversity.

Contamination on private data. COIN's contamination claim covers only public OSS-Fuzz data: no triggering input for a frontier target is publicly known at construction time. This guarantee does not extend to what enters an agent's training data. Agents likely have been trained on the project source code and harnesses themselves, which already gives them prior knowledge of the codebase. Their training corpora may also include inputs produced by dynamic techniques (fuzzing, symbolic execution, automated input generation) that happen to match a frontier target's triggering input.

Frontier-target feasibility. The frontier predicate can in principle select a *defensively dead* arm (e.g. `if(bad) return -1;`), infeasible rather than hard, which conflates "the model cannot reach this line" with "no input can." Manual per-target audits do not scale to a daily refresh, so COIN relies on CodeQL's untaken-branch and call-graph reachability (§2.2) to filter candidates; this signal is sound on the call graph but path-insensitive on conditional feasibility, and therefore admits a residual false-positive rate on the frontier pool. The pipeline can stay fully automated by chaining a complementary feasibility filter, e.g. an LLM-as-judge over the controlling predicate or light-weight symbolic execution along its path, to tighten the reported frontier reach.

Evaluation. The frontier-target pool is too large to enumerate, so we evaluate on a sample under fixed wall-clock budgets. We repeat the full evaluation five times and report mean $\pm$ sd, which stabilizes tier-level conclusions, but rankings within the flagship tier remain within noise, and reported rates still reflect agent behavior on this particular target sample, not the full frontier pool.

Ethical considerations

A reachability witness implies that an attacker-controlled input can drive the program to a particular line; whether that line also constitutes a vulnerability depends on what the line does. We embargo sanitizer-detected conditions, do not redistribute concrete witnesses, and report results in aggregate, all of which keep COIN within the existing OSS-Fuzz disclosure norms.

Acknowledgments

We disclose that generative AI tools were used during the preparation of this paper. Specifically, Claude Code (Anthropic) and Codex (OpenAI) were used to assist with coding and with drafting, editing, and reviewing the text. All AI-generated content was reviewed and verified by the authors.

References

TJ Byun, Cornelius Aschermann, Kai Yuan Thng, Wu Zhou, Yupeng Yang, Lauren Deason, and Joshua Saxe. 2025. Introducing AutoPatchBench: A Benchmark for AI-Powered Security Fixes.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2025. Reasoning runtime behavior of a program with llm: How far are we? In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1869–1881.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, Weiran Shi, Zaiyuan Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Qizhi Chen, Xianfu Cheng, Bo Deng, Qingshui Gu, and 30 others. 2025. NL2Repo-Bench: Towards Long-Horizon Repository Generation Evaluation of Coding Agents. *arXiv preprint*. ArXiv:2512.12730 [cs.SE].

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, pages 508–512, New York, NY, USA. Association for Computing Machinery.

GitHub, Inc. CodeQL. https://codeql.github.com/. Accessed: 2026-05-26.

Kush Jain, Gabriel Synnaeve, and Baptiste Roziere. 2025a. Testgeneval: A real world unit test generation and test completion benchmark. In *The Thirteenth International Conference on Learning Representations*.

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

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

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA. Association for Computing Machinery.

Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. 2025. Sec-bench: Automated benchmarking of llm agents on real-world software security tasks. In *Advances in Neural Information Processing Systems*, volume 38, pages 116342–116378. Curran Associates, Inc.

Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2777–2794. USENIX Association.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, volume 36, pages 21558–21572. Curran Associates, Inc.

Xianzhen Luo, Jingyuan Zhang, Shiqi Zhou, Jinyang Huang, Chuan Xiao, Qingfu Zhu, Zhiyuan Ma, Xing Yue, Yang Yue, Wencong Zeng, and Wanxiang Che. 2026. Cve-factory: Scaling expert-level agentic tasks for code security vulnerability. *Preprint*, arXiv:2602.03012.

Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pages 1393–1403, New York, NY, USA. Association for Computing Machinery.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. 2024. Swt-bench: Testing and validating real-world bug-fixes with code agents. In *Advances in Neural Information Processing Systems*, volume 37, pages 81857–81887. Curran Associates, Inc.

Jielin Qiu, Zuxin Liu, Zhiwei Liu, Rithesh Murthy, Jianguo Zhang, Haolin Chen, Shiyu Wang, Ming Zhu, Liangwei Yang, Juntao Tan, Zhepeng Cen, Cheng Qian, Shelby Heinecke, Weiran Yao, Silvio Savarese, Caiming Xiong, and Huan Wang. 2025. LoCoBench: A Benchmark for Long-Context Large Language Models in Complex Software Engineering. *arXiv preprint*. ArXiv:2509.09614 [cs.SE].

Minghao Shao, Sofija Jancheska, Meet Udeshi, Brendan Dolan-Gavitt, Haoran Xi, Kimberly Milner, Boyuan Chen, Max Yin, Siddharth Garg, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Muhammad Shafique. 2024. Nyu ctf bench: A scalable open-source benchmark dataset for evaluating llms in offensive security. In *Advances in Neural Information Processing Systems*, volume 37, pages 57472–57498. Curran Associates, Inc.

Peiran Wang, Xiaogeng Liu, and Chaowei Xiao. 2025a. CVE-Bench: Benchmarking LLM-based Software Engineering Agent's Ability to Repair Real-World CVE Vulnerabilities. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4207–4224, Albuquerque, New Mexico. Association for Computational Linguistics.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025b. TestEval: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3547–3562, Albuquerque, New Mexico. Association for Computational Linguistics.

Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. 2025c. CyberGym: Evaluating AI Agents' Cybersecurity Capabilities with Real-World Vulnerabilities at Scale. *arXiv preprint*. ArXiv:2506.02548 [cs].

Ruiyang Xu, Jialun Cao, Yaojie Lu, Ming Wen, Hongyu Lin, Xianpei Han, Ben He, Shing-Chi Cheung, and Le Sun. 2025. CRUXEVAL-X: A benchmark for multilingual code reasoning, understanding and execution. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23762–23779, Vienna, Austria. Association for Computational Linguistics.

Zheyuan Yang, Zexi Kuang, Xue Xia, and Yilun Zhao. 2025. Can LLMs generate high-quality test cases for algorithm problems? TestCase-eval: A systematic evaluation of fault coverage and exposure. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1050–1063, Vienna, Austria. Association for Computational Linguistics.

Andy Zhang, Joey Ji, Celeste Menders, Riya Dulepet, Thomas Qin, Ron Wang, Junrong Wu, Kyleen Liao, Jiliang Li, Jinghan Hu, Sara Hong, Nardos Demilew, Shivatmica Murgai, Jason Tran, Nishka Kacheria, Ethan Ho, Denis Liu, Lauren McLane, Olivia Bruvik, and 15 others. 2025a. Bountybench: Dollar impact of ai agent attackers and defenders on real-world cybersecurity systems. In *Advances in Neural Information Processing Systems*, volume 38. Curran Associates, Inc.

Andy K Zhang, Neil Perry, Riya Dulepet, Joey Ji, Celeste Menders, Justin W Lin, Eliot Jones, Gashon Hussein, Samantha Liu, Donovan Julian Jasper, Pura Peetathawatchai, Ari Glenn, Vikram Sivashankar, Daniel Zamoshchin, Leo Glikbarg, Derek Askaryar, Haoxiang Yang, Aolin Zhang, Rishi Alluri, and 6 others. 2025b. Cybench: A framework for evaluating cybersecurity capabilities and risks of language models. In *The Thirteenth International Conference on Learning Representations*.

Cen Zhang, Younggi Park, Fabian Fleischer, Yu-Fu Fu, Jiho Kim, Dongkwan Kim, Youngjoon Kim, Qingxiao Xu, Andrew Chin, Ze Sheng, Hanqing Zhao, Michael Pelican, David J. Musliner, Jeff Huang, Jon Silliman, Mikel Mcdaniel, Jefferson Casavant, Isaac Goldthwaite, Nicholas Vidovich, and 2 others. 2026. Sok: Darpa's ai cyber challenge (aixcc): Competition design, architectures, and lessons learned. *Preprint*, arXiv:2602.07666.

Jian Zhang, Chong Wang, Anran Li, Wenhan Wang, Tianlin Li, and Yang Liu. 2024. Vuladvisor: Natural language suggestion generation for software vulnerability repair. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pages 1932–1944, New York, NY, USA. Association for Computing Machinery.

Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, Jet Geronimo, Avi Dhir, Sudhit Rao, Kaicheng Yu, Twm Stone, and Daniel Kang. 2025. CVE-bench: A benchmark for AI agents' ability to exploit real-world web application vulnerabilities. In *Forty-second International Conference on Machine Learning*.

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, James Hoang, Armel Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, and 14 others. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, volume 2025, pages 66602–66656.

A Prompt templates

This appendix collects the prompt templates used by COIN. They are ordered as the design introduces them: the target-blind LLM seed baseline from §2.3, then the targeted reachability task and the coverage-maximization task from §2.4. Template variables use Jinja-style delimiters and are substituted from the pinned snapshot before the prompt is mounted as `/workspace/TASK.md`.

A.1 Seed-generation prompt

Seed-Generation Prompt

```
You have **{{ time_budget }}**. Your goal is to generate
  ↪ a diverse set of seed inputs that collectively
  ↪ maximize code coverage of the fuzzing harness `{{
  ↪ harness }}` in the **{{ project }}** OSS-Fuzz
  ↪ project.

## Environment

This is an **OSS-Fuzz** project. You have internet
  ↪ access, Docker access via `$DOCKER_HOST`, and can
  ↪ clone the project source code to understand the
  ↪ harness and its target library.

## Steps

1. **Understand the harness**: find and read the source
  ↪ code of `{{ harness }}` in the OSS-Fuzz repo
  ↪ (`https://github.com/google/oss-fuzz`, under
  ↪ `projects/{{ project }}/`) and the project's own
  ↪ repo. Understand what input format the harness
  ↪ expects.
2. **Generate seeds**: create a variety of inputs that
  ↪ exercise different code paths: valid inputs, edge
  ↪ cases, malformed inputs, boundary values, etc.
3. **Write seeds**: save each seed as a separate file
  ↪ under `/workspace/seeds/{{ harness }}/` (e.g.
  ↪ `input_000.bin`, `input_001.bin`, ...).

## Tips

- Focus on breadth: many small, structurally distinct
  ↪ inputs are better than a few large ones.
- Aim for at least 10-20 seeds covering different code
  ↪ paths.
- You can use Docker to build and test inputs if needed.

## Output

All seed files must be written to `/workspace/seeds/{{
  ↪ harness }}/` before you exit. The runner will collect
  ↪ them and measure coverage automatically.
```

A.2 Targeted reachability prompt

Targeted Reachability Prompt

```
Choose a harness, craft an input that reaches the target,
  ↪ and submit it.

## Target

- **Project:** {{ project }} (built with OSS-Fuzz)
- **Function:** `{{ function }}`
- **File:** `{{ file }}`
- **Lines:** {{ line_start }}-{{ line_end }}

```
{{ target_snippet }}
```

{% if is_frontier %}
Branch frontier

The controlling conditional is at `{{ cond_file }}:{{
  ↪ cond_line }}`:

```
{{ condition_snippet }}
```

A baseline already executed this conditional and took the
  ↪ **{{ reached_arm }}** path; the **{{ unreached_arm
  ↪ }}** path was never taken by any baseline. Your goal:
  ↪ produce an input that makes execution take the **{{
  ↪ unreached_arm }}** path, so line {{ line_start }}
  ↪ (the target below) executes. Reaching the conditional
  ↪ is not enough: you must invert which way it goes.
{% endif %}

Source

{% for source in sources %}
- `git clone {{ source.url }} && cd {{ source.repo_name
  ↪ }} && git checkout {{ source.rev }}`
{% endfor %}
- OSS-Fuzz: `{{ oss_fuzz_commit }}`

Clone the repo to read the source code.

Harnesses

Available at `/workspace/harnesses/` (read-only):

{% for harness_name in harness_names %}
- `{{ harness_name }}`
{% endfor %}

Submission
```

```
When you are ready, write your final answer to the
  ↪ `/answer/` directory:

1. `/answer/blob.bin`: the raw input bytes you want fed
  ↪ to the harness.
2. `/answer/blob.harness`: plain text, one line: the
  ↪ harness binary name from the list above (e.g. `{{
  ↪ example_harness }}`).

You have **no live reachability feedback**: there is no
  ↪ verification endpoint to call. Whatever is in
  ↪ `/answer/` when you exit is your final answer; it
  ↪ will be replayed against the harness later. Only the
  ↪ last write of each file counts.

## If the target is unreachable

Difficulty is **not** a reason to abstain: most targets
  ↪ are hard and you are expected to attempt them.
  ↪ Abstain **only** if, after genuine analysis, you have
  ↪ positive evidence the target line cannot be reached
  ↪ by *any* input through *any* listed harness, e.g. the
  ↪ guarding condition is provably never false for valid
  ↪ inputs, the code is dead, a `dynamic_cast`/check can
  ↪ never fail on this path, or the line is unreachable
  ↪ from every harness entry.

In that case, write **only** `/answer/UNREACHABLE.md`
  ↪ explaining the concrete evidence, and do **not**
  ↪ write `/answer/blob.bin`. If `blob.bin` exists it is
  ↪ treated as a normal attempt and `UNREACHABLE.md` is
  ↪ ignored. A wrong abstention (the target was in fact
  ↪ reachable) counts against you, so only abstain when
  ↪ the evidence is solid.

## Environment

You're running as **root** inside a Docker container with
  ↪ full internet access and Docker access (`DOCKER_HOST`
  ↪ is set). You can do anything.
```

A.3 Seed-corpus hint

Seed-Corpus Hint

```
## Seed corpus tool (hint, optional)

A pre-replayed per-harness seed corpus is bind-mounted at
  ↪ `/workspace/corpusdb/`, and the `corpusdb` command is
  ↪ on your PATH. The corpus combines libfuzzer baseline
  ↪ seeds + LLM seed-generation seeds across every model,
  ↪ deduped by content. Each seed already ran through the
  ↪ coverage build, so `corpusdb` can tell you which
  ↪ seeds touch any given line/range.

Seeds are tied to the harness that produced them, so
  ↪ every per-seed query takes the harness name as the
  ↪ first positional.

Discover which harnesses have indexed corpora:

```bash
corpusdb harnesses
```

Examples for your target ({{ file }}:{{ line_start }}-{{
  ↪ line_end }}, harness `{{ example_harness }}`):

```bash
# List all seeds we have for this harness:
corpusdb {{ example_harness }} list | head -20

# Which seeds touch any line in your target range?
corpusdb {{ example_harness }} seeds-touching {{ file
  ↪ }}:{{ line_start }}:{{ line_end }}

# Same query, but require the seed to also touch a second
  ↪ range
# (e.g. a guarding predicate at line G):
corpusdb {{ example_harness }} seeds-touching --all {{
  ↪ file }}:{{ line_start }}:{{ line_end }} {{ file
  ↪ }}:G:G

# How close does a specific seed get?
corpusdb {{ example_harness }} seed-executes-range
  ↪ <seed_name> {{ file }} {{ line_start }} {{ line_end
  ↪ }}

# Dump a seed's bytes so you can mutate it:
corpusdb {{ example_harness }} seed-bytes <seed_name> >
  ↪ /tmp/start.bin
```

This is a hint, not a requirement. You can ignore it and
  ↪ craft from scratch; the submission contract is
  ↪ unchanged.
```

A.4 Coverage-maximization prompt

Coverage-Maximization Prompt

```
Produce a corpus that, taken together, exercises as much
  ↪ of the project's source code as possible through the
  ↪ available harness binaries.

## Project

- **Project:** {{ project }} (built with OSS-Fuzz)

## Goal: exactly how you're scored

For **each** harness, every input you submit for that
  ↪ harness is run through the coverage-instrumented
  ↪ binary, and the set of source lines those runs touch
  ↪ is unioned together:

```
harness_A_covered = lines hit by (harness_A on input_A1)
  ↪ union
                    lines hit by (harness_A on input_A2)
                      ↪ union
                    ...
harness_B_covered = lines hit by (harness_B on input_B1)
  ↪ union
                    lines hit by (harness_B on input_B2)
                      ↪ union
                    ...
...
```

Your final score is the **union of all per-harness
  ↪ covered sets**, divided by the project's executable
  ↪ line count:

```
score = |harness_A_covered union harness_B_covered union
  ↪ ...| / total_lines
```

Consequences of how this is aggregated:

- A line covered by **any** input through **any** harness
  ↪ counts **once**. Hitting the same line via two
  ↪ different harnesses, or via two different inputs of
  ↪ the same harness, does **not** double-count.
- An extra input is worthless if every line it covers is
  ↪ already covered by some other input you submitted
  ↪ (anywhere across the corpus).
- An input is valuable when it hits **new** files /
  ↪ functions / branches not yet covered by anything
  ↪ else.
- Skipping a harness entirely is a real cost if that
  ↪ harness reaches code no other harness can reach.

Source

{% for source in sources %}
- `git clone {{ source.url }} && cd {{ source.repo_name
  ↪ }} && git checkout {{ source.rev }}`
{% endfor %}
- OSS-Fuzz: `{{ oss_fuzz_commit }}`

Clone the repo to read the source code and understand
  ↪ what each harness parses.

Harnesses
```

```
Available at `/workspace/harnesses/` (read-only). Each is
  ↪ a libFuzzer-style binary that accepts a single input
  ↪ file as its only positional argument.

{% for harness_name in harness_names %}
- `{{ harness_name }}`
{% endfor %}

## Submission

For each harness you want to contribute inputs for,
  ↪ create a directory under `/answer/` named exactly
  ↪ like the harness binary, and drop one input file per
  ↪ line of coverage you intend to hit:

```
/answer/{{ example_harness }}/input_001.bin
/answer/{{ example_harness }}/input_002.bin
/answer/{{ example_harness }}/input_003.bin
/answer/<another_harness>/...
```

Filenames are arbitrary: only the parent directory name
  ↪ (the harness binary name) matters. Empty files are
  ↪ rejected. You may submit any number of inputs per
  ↪ harness; you may skip harnesses entirely.

You have **no live coverage feedback** during your run.
  ↪ Whatever is in `/answer/` when you exit is your final
  ↪ corpus; it will be replayed against each harness's
  ↪ coverage build and the total line coverage will be
  ↪ recorded. Spend time understanding the harnesses +
  ↪ source first, then craft inputs that exercise diverse
  ↪ parsing paths.

Environment

You're running as **root** inside a Docker container with
  ↪ full internet access and Docker access (`DOCKER_HOST`
  ↪ is set). You can do anything.
```

A.5 Coverage continuation prompt

Coverage Continuation Prompt

```
Continue. Goal: maximize line coverage across all
  ↪ harnesses.

Reminder of how scoring works:
- Inputs you've already written under /answer/<harness>/
  ↪ are kept; their coverage already counts.
- Only NEW lines beyond what your existing corpus already
  ↪ hits add to your score.
- Re-creating an input that covers lines you've already
  ↪ covered is worth zero.

Look at what's already in /answer/<harness>/, identify
  ↪ uncovered code paths, and ADD inputs that exercise
  ↪ those paths. Use distinct filenames.
```

B Dataset Generation: per-(project, harness, model) construction

Table 5 expands the per-project totals in Table 1 to the cell level. Each cell is **cost (USD) / coverage % / seeds** for that harness × model (coverage = corpusdb per-seed union ∩ GCS executable set ÷ project executable lines; *not* the `-merge` covreport). The **all** row is total cost / mean coverage / total seeds per model; "–" marks runs with no indexed seeds. Figure 5 then breaks the signal-cube partition of Figure 2 out by project, and Figure 6 does the same for the multi-language projects. Table 6 reports the same construction breakdown for the eight multi-language projects.

C Targeted Evaluation: per-(model, project) breakdown

Figure 7 complements the body Figure 3 by plotting per-model precision against total API spend on C/C++, and Figure 8 shows both cost-efficiency views for the multi-language benchmark.

Gemini 3.1 Pro spend. Gemini 3.1 Pro's multi-language spend is the outlier in Figure 8. Its trajectories average 122 API calls per attempt at roughly 89k input tokens per call (total input tokens over total calls, both conditions, runs 2–5), versus 32 calls for GPT-5.4, and because every turn re-sends a longer conversation, per-attempt input grows quadratically with conversation length. Its most expensive attempt re-sent 154M input tokens over 694 calls (roughly $130), and six of its ten most expensive attempts ran the full two-hour budget, nine of the ten without reaching the target. Table 7 and Table 8 report the full per-(model, project) evaluation log underlying the summary in Table 3, and Table 9 and Table 10 do the same for the multi-language benchmark (Table 4). Each cell aggregates all five runs of that (model, project) pair, 50 attempts on C/C++ (ten targets per project) and five attempts per target on the multi-language projects, whose target counts vary (Table 2). Columns: **R** = reached, **S** = submitted (any blob), **A** = abstain (`UNREACHABLE.md`), **NS** = no submission written, **T** = 2 h timeout, **E** = scaffold error (all summed over the five runs); **$** = mean per-run LLM spend in USD, measured as described in §3.3; **tk in / tk out** = per-run total input / output tokens; **turns** = mean API calls per attempt; **min** = mean wall-clock minutes per attempt. Token and API-call counts derive from the gateway's request logs. On C/C++ they come from the original run, whose logs dropped entries under load for parts of `cups`, `cyclonedds`, and `hdf5`, so **tk** and **turns** are lower bounds for those projects. On the multi-language benchmark, run 1 has no gateway telemetry, so token and turn columns average runs 2–5.

D Targeted Evaluation: per-family results

Table 11 splits base-condition reach by target family for the C/C++ and multi-language benchmarks. The two families behave differently. On C/C++, one of the 35 frontier targets is reached in any of the ten evaluations, and per-model frontier reach never exceeds 1.7%. On the multi-language set, agents reach 7 of the 37 frontier targets at least once, and Opus averages 17.3% frontier reach. Reach on the non-trivial reachable family is several times higher for every model, up to 52.0% on C/C++ and 67.3% on the multi-language set (both Opus). Frontier reach thus measures expansion beyond the current coverage frontier and remains the benchmark's headroom, while the reachable family shows the retained tasks are solvable but far from saturated.

E Targeted Evaluation: representative `corpusdb` trajectories

This appendix illustrates the multi-stage select-preserve-edit process behind the per-model `corpusdb` effects (§3.3) with representative trajectories on two structured-file cases.

Canon metadata target. On the `libraw` Canon target (`canon.cpp:370`), GPT-5.4-mini reaches the target in four of five `corpusdb` runs, submitting final inputs that differ from their retrieved seeds by only two bytes. Opus and Gemini 3.1 Pro complete the same chain more consistently: both rise from two of five base runs to five of five with `corpusdb`. Sonnet, after reaching this target in four of five base runs, fails all five `corpusdb` runs: two runs edit a 10,943-byte seed whose coverage does not include the relevant `canon.cpp` region, and three abandon the full-size retrieved files for 108–116-byte synthetic inputs, failures at the seed-selection and structure-preservation stages, respectively.

HDF5 targets. The same model can fail at a different stage on a different format: GPT-5.4-mini's combined reach across three `hdf5` targets falls from 10 of 15 attempts without `corpusdb`

Table 5: Per-(project, harness, model) LLM-seed construction for the C/C++ benchmark. Each cell is *cost (USD) / coverage % / seeds*. Models: Opus/Sonnet (Claude Code), GPT-5.4/GPT-mini (Codex), Gem-Pro/Gem-Fl (Gemini 3.1 Pro / 3 Flash), DSeek (DeepSeek-V3.2), GLM-5. The **All** row includes the dropped `xerces-c`; 13 runs logged no per-run data (shown "–"), so column totals fall $0.19 short of the audited $217.

Project	Harness	Opus	Sonnet	GPT-5.4	GPT-mini	Gem-Pro	Gem-Fl	DSeek	GLM-5
cups	fuzz_array	0.77/1.1/46	0.85/1.1/25	0.33/1.1/17	0.12/1.1/17	0.51/1.1/17	0.12/1.1/18	0.51/1.1/70	0.89/1.1/51
	fuzz_cups	1.11/2.5/105	0.96/2.5/104	0.27/2.3/21	0.18/0.8/18	0.42/2.5/37	0.22/2.4/21	0.75/1.5/26	0.60/2.4/97
	fuzz_http_core	1.25/3.3/79	0.71/3.2/104	0.27/3.4/20	0.23/3.6/26	0.80/3.5/39	0.06/3.2/32	0.38/3.2/51	0.74/3.3/128
	fuzz_ipp	1.74/4.9/101	2.26/4.8/80	0.50/4.9/35	0.36/4.5/16	0.71/4.3/58	0.07/4.6/32	0.44/4.1/26	0.64/4.6/85
	fuzz_ipp_gen	1.47/3.1/79	0.59/3.0/89	0.69/5.0/54	0.21/3.0/16	0.56/3.0/48	0.16/3.0/24	0.33/3.1/135	0.40/3.1/427
	fuzz_ppd_gen_1	5.74/8.9/71	1.37/7.9/38	0.88/8.4/20	0.13/7.8/14	0.64/7.9/59	0.14/7.7/23	0.52/6.5/21	0.90/7.6/58
	fuzz_ppd_gen_cache	2.56/16.1/50	0.54/15.5/43	1.06/16.3/20	0.14/15.0/16	0.18/16.0/37	0.13/15.7/23	0.26/14.2/30	0.62/15.8/90
	fuzz_ppd_gen_conflicts	3.62/8.5/62	0.55/7.1/28	0.64/7.8/20	0.11/7.8/16	0.53/6.9/31	0.13/7.2/18	–	1.62/7.8/41
	fuzz_raster	1.43/1.6/86	1.47/1.5/53	0.75/1.6/27	0.24/1.5/18	0.77/1.5/32	0.11/1.6/20	0.50/1.3/39	0.82/1.4/70
cyclonedds	fuzz_config_init	4.76/3.6/159	2.68/3.3/141	0.86/3.4/27	0.38/2.9/20	0.61/3.3/54	0.15/3.3/26	0.37/3.1/23	0.96/3.1/124
	fuzz_handle_rtps_message	2.21/7.4/132	1.56/7.1/67	0.52/7.1/44	0.21/6.5/14	–	0.13/6.3/24	0.47/6.0/23	0.70/6.3/102
	fuzz_handshake	1.54/0.2/49	0.57/16.8/40	0.65/17.1/17	0.11/17.1/15	0.38/0.2/79	0.03/17.1/14	0.30/0.2/18	0.51/17.1/51
	fuzz_sample_deser	2.85/1.8/141	3.27/1.8/68	0.56/1.7/111	0.12/1.0/12	–	0.13/1.6/22	0.42/1.7/52	0.50/1.7/83
	fuzz_security_deser	1.53/0.5/46	1.72/0.5/44	0.40/0.5/22	0.11/0.5/16	0.67/0.5/40	0.12/0.5/19	0.49/0.3/29	–
	fuzz_type_object	2.76/5.3/99	2.25/7.3/48	1.60/8.0/11	0.07/8.0/13	0.98/8.4/216	0.42/8.0/67	0.53/5.5/58	1.23/5.3/53
hdf5	h5_extended_fuzzer	1.11/12.8/111	0.62/12.2/54	0.23/13.5/31	0.17/16.4/26	0.19/13.0/24	0.14/11.5/19	0.19/11.8/55	0.34/6.4/316
	h5_read_fuzzer	3.16/9.5/123	0.80/10.0/61	0.23/13.4/60	0.08/13.1/28	0.19/9.4/52	0.11/9.1/12	0.34/10.6/49	0.53/9.6/101
karchive	k7z_fuzzer	1.04/5.2/100	0.74/5.3/66	0.29/2.4/6	0.09/2.5/7	0.60/5.3/26	0.08/4.8/17	0.31/5.3/167	0.70/3.0/82
	kar_fuzzer	0.97/3.4/47	0.98/3.4/46	0.37/3.4/26	0.15/3.3/17	0.78/3.4/24	0.16/3.3/20	0.24/3.4/37	0.41/3.4/66
	ktar_bz2_fuzzer	0.92/4.0/63	1.03/4.0/66	0.23/4.0/17	0.19/4.0/12	0.72/4.2/37	0.08/3.9/19	0.17/4.0/32	0.45/4.1/94
	ktar_fuzzer	2.28/3.5/77	1.03/3.2/48	0.27/3.3/24	0.20/3.1/13	0.60/3.5/56	0.13/3.4/30	0.47/3.2/39	0.50/3.6/79
	ktar_gz_fuzzer	1.29/4.1/88	1.00/4.2/78	0.39/4.4/30	0.21/3.9/22	0.32/4.4/40	0.08/4.0/17	0.20/3.9/36	0.97/4.3/66
	ktar_lz_fuzzer	1.70/3.7/121	1.26/3.7/79	0.55/3.8/38	0.18/3.6/17	0.33/3.7/34	0.21/3.6/18	0.32/3.6/41	0.78/3.7/60
	ktar_xz_fuzzer	1.11/3.4/64	0.83/3.3/52	–	–	–	–	0.35/3.3/31	0.60/3.7/97
	ktar_zst_fuzzer	–	0.63/3.6/57	0.40/3.6/20	0.19/3.6/22	0.50/3.7/35	0.09/3.6/22	0.35/3.6/24	0.46/3.7/105
	kzip_fuzzer	1.17/4.4/63	0.86/4.3/53	0.43/4.0/16	0.17/4.6/19	0.34/4.1/20	0.08/4.3/25	0.43/4.1/25	1.09/4.2/37
liboqs	fuzz_test_kem	1.44/13.2/122	1.02/13.2/108	0.53/11.8/53	0.13/6.8/23	0.21/13.2/472	0.32/13.2/245	0.26/11.9/77	0.29/5.2/230
	fuzz_test_sig	1.42/21.1/99	1.22/19.5/57	0.67/14.3/23	0.15/10.6/16	0.19/34.6/3612	0.13/19.5/44	0.43/15.3/82	0.39/19.8/172
	fuzz_test_sig_stfl_lms	1.33/0.1/161	1.64/0.1/66	0.36/0.1/26	0.27/0.1/26	0.48/0.1/8995	0.06/0.1/17	0.32/0.1/43	0.34/0.1/58
	fuzz_test_sig_stfl_xmss	1.39/0.1/75	1.76/0.1/25	0.42/0.1/21	0.15/0.1/10	0.48/0.1/1764	0.14/0.1/26	–	0.79/0.1/71
libraw	libraw_cr2_fuzzer	2.25/13.5/76	1.96/6.2/63	0.46/2.4/20	0.13/2.5/14	0.47/24.1/71	0.08/3.2/33	0.40/3.0/44	0.75/2.5/99
	libraw_fuzzer	1.93/31.5/104	2.42/23.8/70	0.81/2.4/20	0.07/2.4/13	0.53/16.0/78	0.17/2.5/59	0.39/2.5/57	0.51/2.5/128
	libraw_nef_fuzzer	2.88/12.9/211	3.22/6.6/80	0.52/21.2/24	0.28/2.4/15	0.76/19.7/14	0.15/2.5/68	0.44/10.3/38	0.65/2.5/190
	libraw_raf_fuzzer	3.64/3.8/79	2.46/3.6/70	0.90/2.4/20	0.37/2.5/16	0.88/9.7/20	0.18/2.4/20	0.36/3.3/74	0.77/2.5/197
rdkit	mol_data_stream_to_mol_fuzzer	1.96/6.4/90	1.91/14.3/169	0.18/5.1/28	0.16/6.4/23	0.56/20.1/654	0.09/4.1/21	0.61/0.6/32	0.63/0.7/95
	mol_deserialization_fuzzer	2.32/2.7/125	2.57/4.2/107	0.48/3.9/33	0.14/3.5/30	0.66/4.7/295	0.10/4.5/62	0.32/3.0/61	1.36/2.8/31
	smiles_string_to_mol_fuzzer	0.97/12.9/304	0.67/12.2/210	0.38/13.5/54	0.04/7.8/6	0.50/16.5/1468	0.11/10.6/215	0.13/8.1/123	0.34/10.8/178
xerces-c	parse_target	0.50/13.6/128	0.47/13.6/117	0.18/11.5/35	0.14/10.0/22	0.15/13.1/47	–	–	0.34/13.4/118
	parse_target_proto	2.18/0.2/123	1.07/0.2/97	0.39/0.2/29	0.20/0.2/19	0.52/0.2/81	0.12/0.2/16	0.31/0.2/30	–
All		**74.3/6.7/3859**	**53.5/6.6/2871**	**19.6/6.1/1120**	**6.6/5.1/663**	**18.7/7.9/18666**	**5.0/5.3/1408**	**13.6/4.6/1798**	**25.1/5.2/4130**

to 3 of 15 with it, and on the two `H5T` targets its median submitted-input size decreases from 6,032 to 2,078 bytes, consistent with, but not by itself proving, the loss of necessary file structure.

Multi-language usage. On the multi-language set, `corpusdb` usage is sparser. The agent never queries the tool in 1,352 of 2,680 tool-condition attempts, and where seeds are exported, only 142 of 749 attempts obtain a seed whose coverage touches the target's file. The run-consistent effects also differ from C/C++. `corpusdb` never raises Sonnet's reach in any run and helps DeepSeek-V3.2 in four of five runs, while the remaining models' deltas change sign across runs.

Interpretation. Across the examined cases, `corpusdb` amplifies model- and format-specific seed-selection and structure-preservation behavior rather than being uniformly helpful or distracting. These are descriptive observations over agent trajectories, not causal estimates.

F Coverage maximization: per-(model, project) breakdown

Table 12 reports the final per-project line coverage and cost summarized in §3.4, Figure 9 breaks coverage-maximization out by iteration, and Table 13 expands Table 12 to one row per coverage-track run. **Iters** = completed iterations; **Inputs** = total submitted blobs at the run's last checkpoint; **Waste%** = share of inputs in iterations that added zero new lines; **Cov%** = final line coverage from the corpusdb replay; **$** = total LLM API spend ("-" where the LiteLLM proxy logged a zero ledger); **Min** = wall-clock minutes. Two cells log broken per-iteration coverage (Opus on `rdkit`, Gemini 3.1 Pro on `cyclonedds`), so the time-series views (Figure 4, Figure 9) slightly under-report those two models' endpoints relative to Table 12.

G Live substrate: longitudinal evidence

To verify that the supply behind COIN's daily refresh is plentiful and stable, we reconstructed the stage-1 eligibility gate monthly from the public OSS-Fuzz archive (coverage buckets and the `oss-fuzz` git history) over Sep 2018 – Jul 2026, 95 snapshots in total, applying the pipeline's own eligibility logic offline. We validated the reconstruction against a live stage-1 run on 2026-07-09: 816 usable projects reconstructed versus 814 in the live run, and 1,046 versus 1,047 projects with a fresh coverage report.

Table 14 shows yearly anchor months. Configured projects grew from 149 to 1,357 over eight years, and the pool passing the eligibility gate (a fresh coverage report within the staleness window plus an extractable source commit) has held between 793 and 822 in every reconstructed month since Dec 2023. The substrate also absorbs upstream incidents. On 2025-07-01, a partial-upload outage published only 282 same-day coverage reports (roughly 950 is typical), yet the usable pool stayed at 805, because the pipeline reads each project's latest report within a staleness window rather than requiring a same-day build.

H Reproduction requirements

The benchmark itself is the snapshot metadata: it pins the commits, harnesses, target lines, and construction evidence, and every larger artifact is derived from it. Table 15 lists the measured tiers for the two evaluated snapshots. The Docker images and harness binaries are a convenience, rebuildable from the pinned commits and public OSS-Fuzz images in roughly 40 (C/C++) and 29 (multi-language) minutes, and

Table 6: Per-(project, harness, model) LLM-seed construction for the multi-language benchmark. Cell format and model abbreviations as in Table 5.

Project	Harness	Opus	Sonnet	GPT-5.4	GPT-mini	Gem-Pro	Gem-Fl	DSeek	GLM-5
bson-rust	iterate	0.57/73.6/131	0.86/69.8/89	0.52/65.4/33	0.31/56.3/21	0.56/65.6/54	0.54/69.6/33	0.41/48.5/56	0.59/71.3/111
	raw_deserialize	1.80/54.5/178	0.66/52.8/93	1.06/54.0/34	0.17/51.8/216	1.29/80.3/1034	0.48/53.6/34	0.36/44.0/41	0.53/55.8/152
	decode	1.55/88.6/144	0.59/88.1/96	1.40/77.6/27	0.41/77.9/25	0.51/84.4/59	0.25/81.6/17	0.37/88.0/841	0.57/97.9/245
	encoding	1.19/86.6/162	0.91/84.2/111	0.94/94.8/199	0.21/76.0/28	1.12/81.7/85	0.16/76.6/23	0.40/83.5/50	0.50/85.0/152
	raw_deserialize_utf8_lossy	1.08/58.3/111	0.88/54.2/86	0.90/53.4/39	0.22/52.5/23	1.16/32.7/26	0.08/48.3/20	0.25/35.1/185	0.34/55.6/102
	string_handling	2.50/92.6/100	0.67/88.0/80	0.87/65.2/34	0.23/54.3/21	0.78/94.1/86	0.13/47.4/24	0.26/77.8/159	0.39/82.4/135
	type_markers	0.89/71.7/107	0.60/67.9/65	0.76/57.4/44	0.27/56.6/19	1.13/58.6/315	0.18/62.6/26	0.51/69.3/92	0.36/70.9/120
gitoxide	gix-actor-actors	0.44/96.3/100	0.43/95.1/76	0.60/92.5/33	0.16/88.5/20	0.50/97.5/152	0.35/94.8/44	0.40/96.1/96	0.23/95.4/55
	gix-bitmap-ewah	0.61/98.8/88	1.63/100.0/42	0.31/99.4/29	0.15/98.8/20	0.46/100.0/29	0.23/98.8/23	0.14/99.4/42	0.49/100.0/158
	gix-attributes-fuzz_search	3.66/90.5/361	1.07/69.3/99	0.55/85.4/30	0.16/76.5/21	0.41/98.5/1637	0.39/58.5/92	0.43/54.3/68	0.75/69.5/1275
	gix-command-prepare	0.83/97.6/112	2.12/97.2/74	0.38/95.8/30	0.09/80.1/21	0.52/97.2/70	0.12/95.8/41	0.36/97.6/202	0.78/82.9/310
	gix-commitgraph-fuzz_file	4.58/90.9/99	1.62/77.0/43	0.85/93.6/41	0.21/19.8/27	1.06/95.0/31	0.22/94.4/24	0.16/18.3/20	0.53/18.3/54
	gix-config-fuzz_file	1.40/89.7/120	1.02/89.7/121	0.35/88.6/27	0.21/78.8/26	0.59/90.7/94	0.34/92.8/32	0.73/79.6/32	0.18/83.1/50
	gix-config-parse	1.00/92.2/142	1.50/87.6/89	0.76/81.3/24	0.35/81.7/16	0.80/80.1/57	0.12/93.2/15	0.89/86.3/24	0.47/84.4/101
	gix-config-value-fuzz_value	2.86/61.2/152	1.17/67.5/52	0.47/73.1/24	0.60/72.0/18	0.44/95.1/605	1.59/65.9/30	0.47/56.2/30	0.47/67.2/161
	gix-credentials-context	0.66/87.9/100	–	0.80/87.7/32	0.10/80.4/17	0.47/86.2/85	0.18/87.7/45	0.23/79.8/28	0.37/90.0/105
	gix-date-parse	0.53/80.7/200	0.68/76.7/90	0.41/67.5/53	0.12/69.2/50	0.76/89.2/164	0.16/61.6/60	–	0.27/80.2/287
	gix-ignore-ignore	–	0.59/84.9/60	0.42/85.8/30	0.15/82.3/22	0.34/88.2/37	0.22/88.7/44	0.15/86.5/72	0.28/83.5/100
	gix-imara-diff-comprehensive_diff	2.26/90.1/85	0.83/91.4/41	0.68/90.1/27	0.32/98.2/22	0.31/99.7/2047	0.26/93.5/23	–	–
	gix-imara-diff-diff_compute_with	3.61/97.8/95	0.85/97.6/44	0.33/99.1/33	0.16/97.9/20	0.72/99.1/101	0.16/98.6/31	0.13/93.3/25	0.27/94.5/80
	gix-imara-diff-postprocess_heuristics	1.61/89.1/75	1.63/66.3/51	0.59/84.3/26	0.34/84.9/19	0.71/99.5/1772	1.10/86.4/20	0.09/71.6/29	1.01/81.3/97
	gix-imara-diff-unified_diff_printer	1.72/91.0/115	2.15/93.9/82	0.81/90.2/25	0.26/74.5/21	0.94/99.7/1403	0.41/79.1/61	–	1.21/82.5/106
	gix-index-index_file	4.83/92.6/209	1.27/82.4/51	0.53/87.9/45	0.13/82.2/39	1.34/88.2/86	0.16/82.2/28	–	0.52/62.3/71
	gix-mailmap-mailmap	0.98/95.7/91	0.49/93.5/46	1.05/92.1/28	0.10/92.6/16	0.77/95.8/95	0.12/94.5/26	–	0.50/93.9/80
	gix-merge-blob	2.90/77.7/68	1.49/75.5/40	0.67/79.9/29	0.14/68.3/18	0.83/89.6/79	0.25/84.8/39	–	0.46/92.1/101
	gix-object-fuzz_commit	1.71/93.6/120	0.65/90.6/50	0.58/90.6/28	0.00/86.8/22	0.51/93.2/68	0.34/92.1/26	–	0.42/92.7/101
	gix-object-fuzz_tag	0.52/94.5/70	0.77/91.8/40	0.49/93.4/26	0.13/90.9/21	0.55/91.8/32	0.13/93.4/40	–	0.54/93.5/89
	gix-object-fuzz_tree	1.34/97.5/93	0.75/94.2/50	0.69/94.2/36	0.16/77.2/24	0.60/91.2/44	0.25/94.2/28	–	0.94/95.3/721
	gix-pack-data_file	1.47/82.7/143	1.99/90.2/55	0.65/75.5/20	0.12/53.6/20	0.91/93.5/32	0.13/75.2/24	0.34/81.8/52	0.32/80.3/65
	gix-pack-index_file	3.13/96.3/106	0.91/96.5/57	0.36/87.5/20	0.14/88.7/29	0.56/92.5/31	0.09/90.1/23	–	0.73/96.8/80
	gix-pack-multi_index_file	1.95/89.3/70	2.57/89.5/35	1.28/84.8/31	0.17/87.7/23	0.54/92.3/27	0.16/78.8/22	–	0.68/96.3/58
	gix-packetline-packet_lines	0.53/99.7/103	0.89/99.2/71	0.45/96.8/40	0.13/96.3/23	0.50/97.1/33	0.22/98.3/37	0.31/99.2/158	0.26/99.5/139
	gix-pathspec-parse	1.96/93.4/326	1.13/88.4/98	0.35/81.3/40	0.13/85.8/28	0.65/90.0/75	0.19/86.2/67	0.49/89.0/106	0.38/91.2/105
	gix-ref-fuzz_log	4.83/89.8/110	2.16/48.4/35	0.97/90.5/20	0.35/90.1/18	0.71/95.1/31	0.14/46.2/20	–	0.64/47.9/102
	gix-ref-fuzz_names	4.42/94.2/123	1.33/59.4/48	0.77/90.7/30	0.25/89.4/23	0.52/99.4/330	0.10/64.0/64	0.33/47.3/73	0.93/91.3/81
	gix-ref-fuzz_packed_buffer	1.84/83.7/60	2.07/89.6/65	0.53/62.7/28	0.58/61.7/18	0.58/81.5/23	0.27/60.9/23	0.44/44.7/79	2.01/86.0/83
	gix-refspec-parse	1.05/36.1/139	0.83/24.3/121	0.54/37.8/60	0.14/19.4/37	0.15/18.7/41	0.08/22.1/54	0.39/21.2/91	0.38/35.1/200
	gix-revision-parse	0.75/55.4/162	1.16/57.9/150	0.35/21.0/72	0.14/45.9/54	0.79/80.4/658	0.08/37.8/58	0.27/47.6/52	0.52/66.1/400
	gix-transport-capabilities_from_bytes	0.59/97.5/50	0.45/96.1/40	0.31/94.7/28	0.06/87.3/20	0.52/94.2/49	0.12/94.2/20	0.20/98.1/24	0.31/95.3/60
	gix-transport-capabilities_from_lines	1.05/94.9/93	0.99/95.7/43	0.40/94.8/24	0.05/91.0/19	0.62/94.8/46	0.11/91.9/26	0.16/93.0/32	0.48/93.8/51
	gix-transport-capabilities_handshake	1.35/98.7/73	0.89/96.4/40	0.41/97.4/32	0.12/95.6/20	0.42/96.0/27	0.08/90.2/20	0.43/98.4/50	0.57/96.6/49
	gix-url-parse	0.71/88.1/206	0.71/80.8/100	0.36/78.6/80	0.12/72.5/50	0.76/85.8/136	0.11/79.1/45	0.40/89.2/256	0.71/87.4/305
jfreechart	DefaultCategoryDatasetChartFuzzer	1.14/100.0/50	2.96/100.0/36	0.76/100.0/24	0.36/99.9/16	0.52/100.0/698	0.16/99.9/20	0.41/99.9/32	0.83/100.0/65
	PieDatasetChartFuzzer	1.62/100.0/50	2.46/100.0/45	0.67/100.0/24	0.26/98.3/16	1.12/100.0/43	0.03/100.0/13	0.36/100.0/45	0.69/100.0/106
	TimeSeriesChartFuzzer	0.77/99.9/69	1.62/100.0/56	1.09/99.9/23	0.13/99.6/22	0.91/99.7/61	0.54/99.6/25	0.40/99.6/52	0.97/100.0/51
libcst	fuzz_transformer	1.41/87.3/166	0.74/90.0/66	0.57/96.2/36	0.12/89.8/32	0.32/90.3/486	0.07/86.7/27	0.24/86.4/37	0.38/88.1/301
protocompile	fuzz_protocompile	1.56/96.9/506	0.78/80.9/84	0.90/73.7/27	0.21/61.7/22	0.23/79.3/417	0.02/65.9/24	0.46/66.5/21	1.45/71.1/141
sqlalchemy-utils	fuzz_db	0.77/99.9/132	0.63/100.0/176	0.28/99.7/32	0.09/99.9/39	0.37/99.9/53	0.07/99.7/44	0.36/100.0/81	0.25/99.9/104
	fuzz_orm	1.35/99.3/54	1.19/99.3/69	0.76/99.3/27	0.07/99.3/21	0.60/99.3/23	0.21/99.3/27	0.27/99.3/22	0.16/99.3/40
	fuzz_type	1.96/99.8/51	2.92/99.9/64	0.61/99.1/25	0.34/99.0/16	0.73/100.0/161	0.08/98.9/20	0.17/99.8/116	1.06/99.0/30
syzkaller	compiler_fuzzer	3.95/95.8/200	1.14/77.2/100	0.90/83.1/23	0.27/74.2/20	0.48/81.0/119	2.31/70.3/40	0.45/70.3/52	0.84/75.7/125
	prog_deserialize_fuzzer	3.19/95.0/250	1.53/90.3/204	1.47/66.2/37	0.20/80.4/47	1.09/87.6/761	0.26/61.7/36	0.49/80.1/51	0.34/88.0/122
	prog_parselog_fuzzer	2.03/94.2/120	1.95/92.8/91	1.15/83.5/34	0.19/74.2/20	0.57/59.7/23	0.13/56.5/25	0.37/85.4/44	0.45/73.1/71
	report_fuzzer	9.42/95.8/362	7.00/94.5/76	0.85/91.9/43	0.14/75.0/28	0.80/83.0/42	0.22/94.2/121	0.39/90.2/30	1.24/83.1/200
xstream	XmlFuzzer	1.65/85.4/155	0.52/56.5/61	1.28/86.5/57	0.14/57.4/18	0.43/91.1/256	0.08/57.8/120	0.34/73.1/36	0.28/58.5/136
All		**104.6/88.5/7357**	**72.5/83.7/3947**	**37.8/83.5/2003**	**10.9/77.8/1532**	**36.6/88.1/15029**	**15.0/79.6/1994**	**15.4/77.3/3684**	**31.8/82.6/8389**

the seed corpora are needed only for the `corpusdb` condition. Verifying a submitted input requires only the coverage build and a single deterministic harness execution, so checking reported scores is far cheaper than re-running the agents.

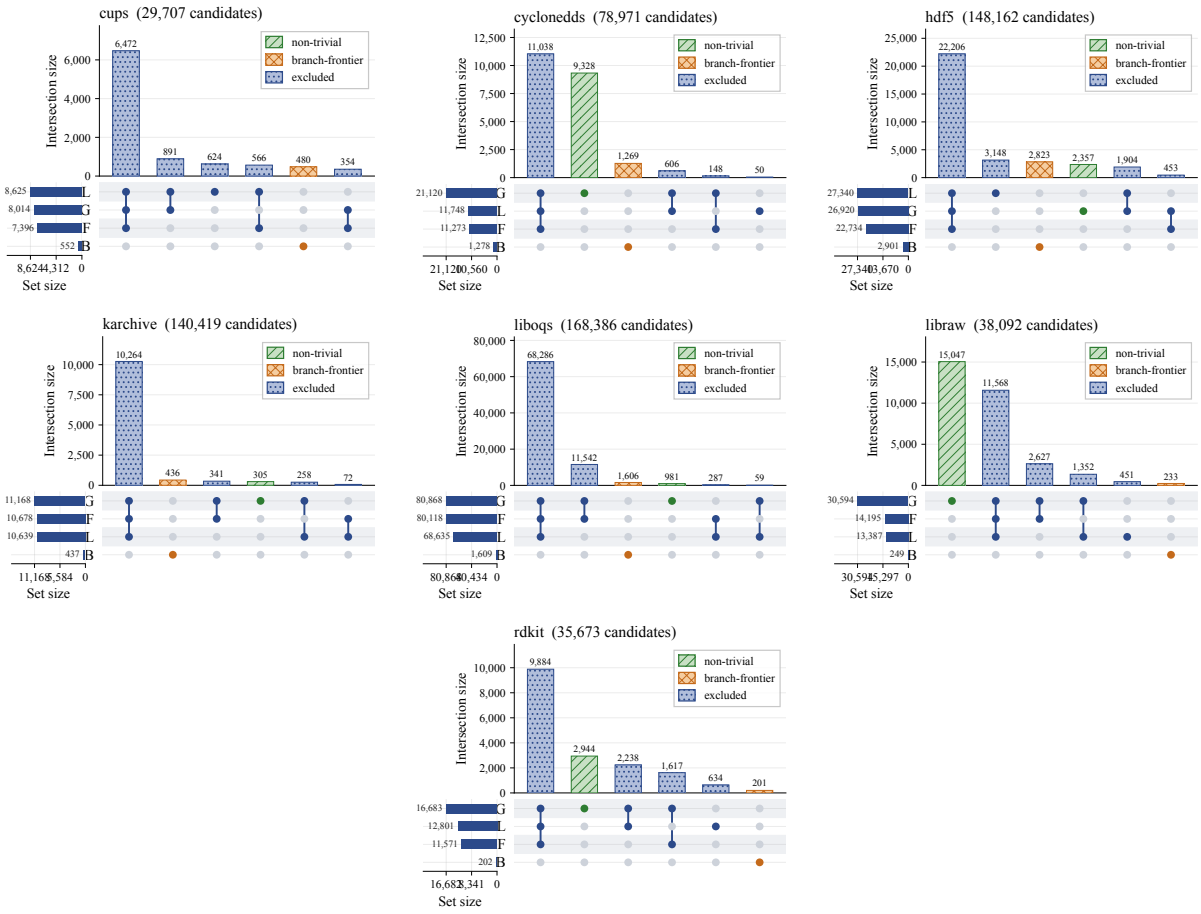


Figure 5: Per-project $\{\mathcal{G}, \mathcal{F}, \mathcal{L}, \mathcal{B}\}$ signal UpSets for the C/C++ benchmark (one panel per project, alphabetical; six largest intersections per panel; green = $\mathcal{G} \cap \overline{\mathcal{F} \cup \mathcal{L}}$ selection, orange = branch-frontier $\mathcal{B}$, navy = excluded; `xerces-c` omitted: no signal-bearing candidates).

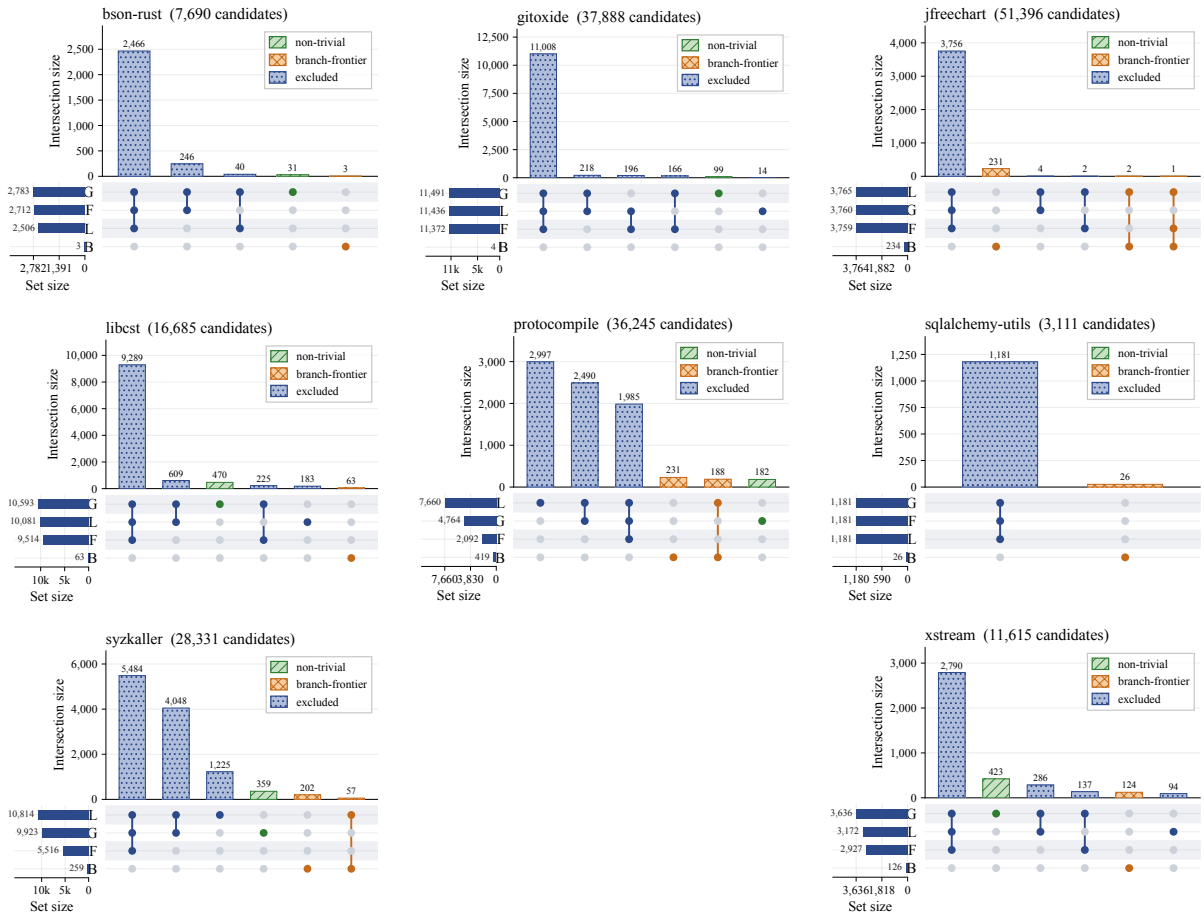


Figure 6: Per-project signal UpSets for the multi-language benchmark (one panel per project, alphabetical; six largest intersections per panel; colors and set alphabet as in Figure 5; the static call-graph signal $\mathcal{C}$ is folded out of the shown combinations for comparability).

Table 7: C/C++ targeted-track per-(model, project) full statistics summed over the five runs (50 attempts per cell; token and turn columns from the original run).

Model	Project	R	S	A	NS	T	E	$	tk in	tk out	turns	min
Claude Opus 4.6	cups	21	34	13	0	3	0	46.50	11.9M	134k	28	26.6
	cyclonedds	8	20	25	1	4	0	77.43	47.3M	448k	126	40.5
	hdf5	23	26	15	2	7	0	90.31	49.1M	674k	147	48.7
	karchive	5	25	25	0	0	0	37.97	32.4M	374k	97	22.9
	liboqs	0	39	11	0	0	0	33.81	41.7M	399k	116	22.4
	libraw	13	30	20	0	0	0	32.57	45.7M	428k	158	20.6
	rdkit	23	25	24	1	0	0	35.87	42.8M	280k	98	18.7
Claude Sonnet 4.6	cups	20	29	15	0	6	0	45.70	16.4M	260k	32	36.7
	cyclonedds	8	21	19	0	10	0	92.04	90.2M	825k	167	60.9
	hdf5	23	31	5	0	14	0	85.64	87.1M	1.0M	177	59.3
	karchive	0	28	17	0	5	0	53.89	136.4M	1.2M	331	40.2
	liboqs	0	31	16	0	3	0	28.97	32.1M	656k	107	29.1
	libraw	15	29	20	1	0	0	41.36	79.6M	720k	200	28.4
	rdkit	24	26	22	0	2	0	40.49	77.1M	671k	225	29.1
GPT-5.4	cups	19	28	22	0	0	0	11.33	3.5M	13k	6	7.1
	cyclonedds	7	36	14	0	0	0	26.77	31.1M	112k	35	15.8
	hdf5	20	31	19	0	0	0	17.70	33.7M	121k	49	10.1
	karchive	5	18	32	0	0	0	10.25	19.4M	103k	39	8.0
	liboqs	0	28	20	2	0	0	13.64	20.3M	105k	46	9.9
	libraw	15	34	16	0	0	0	14.65	31.8M	165k	51	8.6
	rdkit	24	28	22	0	0	0	9.38	8.7M	58k	22	5.9
GPT-5.4-mini	cups	10	33	15	2	0	0	3.68	4.2M	20k	6	7.1
	cyclonedds	0	36	14	0	0	0	14.89	19.5M	119k	26	21.4
	hdf5	12	33	17	0	0	0	8.80	23.5M	206k	42	14.3
	karchive	3	23	24	3	0	0	3.86	22.2M	185k	36	8.3
	liboqs	0	28	21	0	1	0	5.64	14.8M	135k	37	11.5
	libraw	9	30	19	1	0	0	5.14	22.5M	194k	42	9.7
	rdkit	19	30	19	1	0	0	3.99	18.9M	148k	32	7.0
Gemini 3.1 Pro	cups	20	23	27	0	0	0	12.54	1.3M	16k	4	9.2
	cyclonedds	0	21	29	0	0	0	31.72	25.9M	118k	42	22.0
	hdf5	19	27	21	2	0	0	27.63	31.0M	204k	56	17.9
	karchive	3	17	31	2	0	0	14.21	25.1M	204k	66	11.4
	liboqs	0	25	22	2	1	0	13.74	27.1M	198k	47	15.0
	libraw	14	29	20	1	0	0	18.02	29.4M	198k	66	15.4
	rdkit	23	26	24	0	0	0	14.91	27.3M	242k	59	9.8
Gemini 3 Flash	cups	17	43	2	4	1	0	7.82	3.7M	26k	6	15.3
	cyclonedds	1	46	1	3	0	0	11.11	43.4M	371k	46	13.8
	hdf5	3	44	3	3	0	0	9.34	42.5M	250k	64	18.3
	karchive	3	46	2	2	0	0	6.99	56.0M	421k	86	12.4
	liboqs	0	46	1	0	3	0	10.12	65.4M	490k	67	20.7
	libraw	3	40	7	3	0	0	9.43	108.1M	633k	104	13.5
	rdkit	24	43	6	1	0	0	11.63	71.0M	938k	84	14.7
DeepSeek-V3.2	cups	9	24	4	12	1	9	3.56	1.2M	6k	4	17.5
	cyclonedds	0	11	6	23	0	10	4.23	12.7M	74k	33	20.8
	hdf5	0	25	4	18	3	0	5.62	22.2M	112k	71	29.5
	karchive	0	18	7	19	1	5	4.35	21.1M	136k	78	22.0
	liboqs	0	24	4	21	0	1	4.61	23.6M	145k	76	15.0
	libraw	1	27	8	15	0	0	4.97	25.2M	183k	88	16.1
	rdkit	4	23	8	19	0	0	5.13	20.0M	116k	64	18.3
GLM-5	cups	18	23	13	12	0	2	16.11	8.2M	91k	16	23.7
	cyclonedds	0	22	6	17	0	5	23.34	17.5M	298k	34	28.1
	hdf5	6	27	8	14	0	1	28.27	44.5M	448k	94	33.6
	karchive	4	21	11	16	1	1	17.12	27.7M	590k	75	27.9
	liboqs	0	27	12	7	0	4	10.37	23.6M	504k	71	19.4
	libraw	7	26	19	5	0	0	20.20	38.1M	349k	97	21.0
	rdkit	16	25	8	13	0	4	13.07	34.6M	449k	72	16.6

Table 8: C/C++ tool-track per-(model, project) full statistics summed over the five runs (50 attempts per cell; token and turn columns from the original run).

Model	Project	R	S	A	NS	T	E	$	tk in	tk out	turns	min
Claude Opus 4.6	cups	20	29	18	0	3	0	45.31	12.4M	128k	24	29.0
	cyclonedds	10	17	33	0	0	0	69.59	49.5M	384k	107	37.2
	hdf5	22	30	14	1	5	0	89.38	78.1M	900k	200	53.1
	karchive	5	25	25	0	0	0	34.46	38.2M	352k	113	29.1
	liboqs	0	35	13	0	2	0	45.99	36.6M	340k	102	28.2
	libraw	23	30	20	0	0	0	37.53	8.4M	81k	32	18.0
	rdkit	24	25	25	0	0	0	28.00	1.6M	11k	6	16.1
Claude Sonnet 4.6	cups	19	30	16	0	4	0	36.32	13.9M	324k	30	31.3
	cyclonedds	8	20	23	0	7	0	72.49	72.3M	672k	142	53.5
	hdf5	22	31	4	0	15	0	79.13	98.7M	1.2M	221	59.2
	karchive	5	21	25	0	4	0	49.24	87.5M	947k	230	39.5
	liboqs	0	26	20	1	3	0	29.35	38.4M	537k	129	27.3
	libraw	4	29	20	0	1	0	36.20	16.5M	229k	50	24.6
	rdkit	24	25	23	0	2	0	38.80	53.0M	409k	92	31.3
GPT-5.4	cups	20	30	20	0	0	0	13.10	3.1M	10k	5	9.6
	cyclonedds	5	31	19	0	0	0	23.33	7.7M	25k	11	13.1
	hdf5	16	27	22	1	0	0	19.74	4.7M	22k	9	13.0
	karchive	5	16	34	0	0	0	10.92	3.4M	15k	7	7.4
	liboqs	0	20	30	0	0	0	14.47	1.7M	4k	3	11.1
	libraw	20	31	19	0	0	0	14.90	5.1M	12k	4	7.5
	rdkit	22	25	24	1	0	0	9.21	1.6M	6k	2	5.7
GPT-5.4-mini	cups	10	31	18	1	0	0	3.80	1.6M	10k	3	8.5
	cyclonedds	3	36	12	2	0	0	11.26	14.4M	56k	16	18.7
	hdf5	7	32	16	2	0	0	8.74	10.1M	55k	15	17.3
	karchive	2	19	24	7	0	0	4.14	6.7M	28k	7	8.5
	liboqs	0	24	23	3	0	0	4.31	2.9M	4k	3	11.3
	libraw	7	35	13	2	0	0	5.23	2.6M	9k	2	9.1
	rdkit	18	28	20	2	0	0	4.55	2.5M	26k	4	7.9
Gemini 3.1 Pro	cups	20	20	29	1	0	0	18.90	4.0M	22k	6	13.0
	cyclonedds	1	13	37	0	0	0	45.92	71.0M	243k	68	23.5
	hdf5	23	28	18	0	4	0	52.96	54.4M	290k	62	36.5
	karchive	3	13	35	0	2	0	19.28	38.5M	191k	57	21.7
	liboqs	0	14	34	1	1	0	21.37	40.4M	190k	45	16.5
	libraw	15	27	20	1	2	0	26.55	20.1M	91k	33	18.8
	rdkit	24	25	25	0	0	0	19.65	1.3M	7k	3	12.3
Gemini 3 Flash	cups	14	44	2	3	1	0	11.08	8.0M	40k	8	19.3
	cyclonedds	1	42	2	3	3	0	26.25	233.6M	547k	113	36.0
	hdf5	14	41	2	3	4	0	25.97	70.4M	219k	55	41.3
	karchive	2	42	4	0	4	0	15.55	117.6M	661k	84	33.9
	liboqs	0	38	5	1	6	0	18.66	90.1M	411k	66	36.0
	libraw	9	39	7	3	1	0	12.69	42.4M	125k	32	17.3
	rdkit	21	38	3	9	0	0	15.30	29.9M	348k	15	21.5
DeepSeek-V3.2	cups	6	24	4	7	0	15	4.28	1.6M	9k	6	18.7
	cyclonedds	0	25	12	10	1	2	5.59	16.3M	88k	46	23.1
	hdf5	0	21	7	18	0	4	4.87	19.9M	112k	66	22.4
	karchive	0	20	13	17	0	0	4.65	26.1M	171k	103	22.7
	liboqs	0	29	15	6	0	0	5.59	18.6M	114k	60	20.1
	libraw	2	35	11	4	0	0	5.13	6.2M	41k	23	19.3
	rdkit	8	31	11	8	0	0	5.28	2.0M	10k	6	21.8
GLM-5	cups	18	25	9	16	0	0	15.80	5.5M	71k	12	24.4
	cyclonedds	0	19	14	13	1	3	26.96	29.8M	390k	72	34.0
	hdf5	12	29	12	7	0	2	27.29	36.2M	463k	91	39.1
	karchive	2	16	23	10	0	1	21.75	47.2M	665k	129	33.7
	liboqs	0	21	25	3	0	1	15.39	22.1M	457k	68	22.7
	libraw	6	26	20	4	0	0	16.43	8.7M	95k	28	16.9
	rdkit	18	29	13	6	0	2	13.84	5.4M	32k	11	17.9

Table 9: Multi-language targeted-track per-(model, project) full statistics summed over the five runs (five attempts per target per cell; token and turn columns average runs 2–5).

Model	Project	R	S	A	NS	T	E	$	tk in	tk out	turns	min
Claude Opus 4.6	bson-rust	7	14	26	0	0	0	39.89	57.1M	343k	92	18.9
	gitoxide	17	33	12	0	0	0	36.99	33.4M	534k	48	22.8
	jfreechart	0	1	24	0	0	0	18.01	23.0M	195k	61	16.4
	libcst	19	25	25	0	0	0	18.72	21.6M	135k	48	7.3
	protocompile	40	40	10	0	0	0	14.27	12.1M	88k	33	5.4
	sqlalchemy-utils	0	0	25	0	0	0	7.47	6.0M	73k	29	6.7
	syzkaller	31	39	11	0	0	0	26.06	34.4M	191k	56	10.9
	xstream	19	29	21	0	0	0	24.70	26.2M	274k	44	11.2
Claude Sonnet 4.6	bson-rust	9	19	20	0	1	0	37.33	77.4M	684k	116	33.9
	gitoxide	9	24	18	1	2	0	30.67	29.9M	873k	46	38.0
	jfreechart	0	0	24	0	1	0	15.62	29.3M	318k	72	24.5
	libcst	7	25	25	0	0	0	18.16	30.2M	285k	57	12.4
	protocompile	36	40	10	0	0	0	13.81	16.3M	154k	37	6.1
	sqlalchemy-utils	0	0	25	0	0	0	9.63	16.2M	170k	53	13.0
	syzkaller	20	40	9	0	1	0	33.45	65.5M	585k	85	23.9
	xstream	21	29	21	0	0	0	25.31	39.9M	484k	57	18.2
GPT-5.4	bson-rust	5	11	29	0	0	0	8.03	18.3M	95k	28	4.0
	gitoxide	16	30	15	0	0	0	12.61	33.8M	181k	48	6.6
	jfreechart	0	0	25	0	0	0	3.87	7.7M	52k	24	3.3
	libcst	6	24	26	0	0	0	7.05	13.3M	81k	24	3.3
	protocompile	39	40	10	0	0	0	7.14	13.8M	74k	22	2.5
	sqlalchemy-utils	0	0	25	0	0	0	2.26	2.8M	31k	17	2.0
	syzkaller	34	39	10	0	1	0	12.01	32.6M	118k	41	7.7
	xstream	18	26	24	0	0	0	10.18	17.4M	118k	32	4.7
GPT-5.4-mini	bson-rust	2	15	25	0	0	0	1.89	10.8M	92k	19	3.0
	gitoxide	5	36	9	0	0	0	4.25	34.0M	248k	38	6.2
	jfreechart	0	0	23	2	0	0	0.95	4.7M	54k	20	2.6
	libcst	0	24	25	1	0	0	2.49	15.9M	124k	27	4.1
	protocompile	26	41	9	0	0	0	2.14	13.0M	93k	20	2.4
	sqlalchemy-utils	0	1	24	0	0	0	0.96	5.0M	54k	24	2.7
	syzkaller	21	33	15	2	0	0	4.36	39.7M	179k	40	6.6
	xstream	11	21	26	3	0	0	2.74	12.6M	126k	27	3.8
Gemini 3.1 Pro	bson-rust	3	16	23	0	1	0	24.83	43.6M	86k	104	34.9
	gitoxide	7	27	15	1	2	0	61.72	108.6M	238k	147	43.9
	jfreechart	0	0	24	0	1	0	13.99	17.8M	71k	83	29.8
	libcst	3	29	17	3	1	0	28.57	46.6M	135k	104	38.1
	protocompile	26	38	7	1	4	0	36.50	71.3M	127k	91	35.8
	sqlalchemy-utils	0	1	24	0	0	0	9.88	8.0M	46k	57	18.6
	syzkaller	13	32	12	2	4	0	57.93	96.1M	147k	123	52.4
	xstream	11	33	17	0	0	0	29.56	37.9M	121k	90	21.7
Gemini 3 Flash	bson-rust	3	33	1	2	4	0	5.54	64.6M	117k	165	24.2
	gitoxide	3	40	1	4	0	0	3.98	36.9M	95k	106	8.6
	jfreechart	0	19	3	2	1	0	3.52	36.8M	63k	130	16.3
	libcst	0	42	0	7	1	0	5.54	56.6M	83k	103	19.9
	protocompile	22	41	1	2	6	0	4.59	39.3M	58k	75	19.8
	sqlalchemy-utils	0	23	1	1	0	0	2.15	16.2M	55k	97	13.1
	syzkaller	8	37	3	9	1	0	7.34	87.4M	107k	165	13.9
	xstream	9	46	2	2	0	0	4.63	30.7M	70k	92	7.8
DeepSeek-V3.2	bson-rust	0	15	6	19	0	0	3.00	24.3M	131k	78	31.8
	gitoxide	1	19	8	18	0	0	2.86	22.5M	157k	62	31.1
	jfreechart	0	4	4	17	0	0	1.63	12.6M	79k	72	33.9
	libcst	0	22	6	22	0	0	3.21	26.6M	132k	79	41.7
	protocompile	17	27	4	19	0	0	3.21	29.3M	159k	81	37.9
	sqlalchemy-utils	0	17	2	6	0	0	1.36	9.6M	76k	52	20.3
	syzkaller	7	27	9	14	0	0	3.41	28.8M	141k	74	41.0
	xstream	4	14	10	26	0	0	3.60	29.7M	171k	98	37.8
GLM-5	bson-rust	2	23	14	3	0	0	11.46	46.5M	453k	109	28.5
	gitoxide	4	28	15	2	0	0	8.25	27.5M	348k	65	18.7
	jfreechart	0	13	10	2	0	0	6.64	26.3M	287k	101	27.4
	libcst	0	26	23	1	0	0	8.72	32.5M	281k	77	18.7
	protocompile	24	34	10	6	0	0	7.44	23.2M	178k	59	10.8
	sqlalchemy-utils	0	6	17	2	0	0	4.36	14.4M	189k	70	17.0
	syzkaller	15	39	8	3	0	0	9.96	36.4M	304k	84	19.3
	xstream	8	25	17	8	0	0	10.61	30.4M	290k	88	16.8

Table 10: Multi-language tool-track per-(model, project) full statistics summed over the five runs (five attempts per target per cell; token and turn columns average runs 2–5).

Model	Project	R	S	A	NS	T	E	$	tk in	tk out	turns	min
Claude Opus 4.6	bson-rust	8	13	27	0	0	0	39.58	46.5M	251k	94	18.7
	gitoxide	16	30	15	0	0	0	41.45	36.5M	534k	58	28.0
	jfreechart	0	3	22	0	0	0	20.97	29.2M	197k	79	19.4
	libcst	21	25	25	0	0	0	21.46	25.7M	146k	54	8.4
	protocompile	38	40	10	0	0	0	16.47	16.2M	96k	40	6.2
	sqlalchemy-utils	0	0	25	0	0	0	7.80	5.7M	73k	29	8.7
	syzkaller	29	37	13	0	0	0	29.88	35.9M	175k	61	11.5
	xstream	15	30	20	0	0	0	24.99	26.0M	254k	44	11.8
Claude Sonnet 4.6	bson-rust	9	14	25	1	0	0	40.64	91.9M	681k	130	29.6
	gitoxide	7	25	18	1	1	0	31.56	33.8M	907k	56	47.8
	jfreechart	0	1	24	0	0	0	13.23	23.5M	262k	63	20.6
	libcst	3	25	25	0	0	0	22.80	41.9M	350k	71	17.5
	protocompile	35	40	10	0	0	0	14.69	21.2M	182k	44	7.6
	sqlalchemy-utils	0	0	25	0	0	0	7.97	10.5M	130k	42	13.2
	syzkaller	24	39	10	0	1	0	36.56	81.0M	566k	115	28.4
	xstream	8	28	22	0	0	0	23.45	36.3M	461k	53	17.7
GPT-5.4	bson-rust	2	7	33	0	0	0	7.75	11.9M	64k	27	3.9
	gitoxide	17	30	15	0	0	0	16.55	42.7M	159k	61	9.3
	jfreechart	0	0	25	0	0	0	4.45	6.1M	40k	26	3.3
	libcst	5	25	25	0	0	0	8.40	10.2M	65k	25	3.2
	protocompile	38	40	10	0	0	0	9.18	14.0M	64k	27	3.2
	sqlalchemy-utils	0	0	25	0	0	0	3.54	3.1M	27k	22	2.6
	syzkaller	33	40	10	0	0	0	12.51	29.7M	109k	42	5.7
	xstream	14	24	26	0	0	0	10.29	15.5M	108k	30	3.8
GPT-5.4-mini	bson-rust	0	17	23	0	0	0	2.52	10.0M	73k	21	3.5
	gitoxide	4	33	11	1	0	0	4.21	28.6M	172k	39	6.8
	jfreechart	0	0	23	2	0	0	1.29	4.6M	43k	20	2.8
	libcst	2	29	20	1	0	0	3.40	19.2M	114k	35	5.0
	protocompile	25	40	9	1	0	0	2.70	10.6M	82k	23	2.7
	sqlalchemy-utils	0	2	23	0	0	0	1.61	8.3M	52k	35	4.7
	syzkaller	17	38	11	1	0	0	4.84	36.6M	196k	44	6.7
	xstream	7	25	21	4	0	0	2.92	11.8M	115k	25	3.7
Gemini 3.1 Pro	bson-rust	2	18	20	0	2	0	64.39	66.3M	84k	114	46.8
	gitoxide	5	26	15	0	4	0	78.78	95.8M	148k	119	55.7
	jfreechart	0	4	20	0	1	0	74.06	106.1M	119k	179	40.9
	libcst	1	34	11	2	3	0	84.75	124.4M	142k	156	54.8
	protocompile	26	33	6	3	8	0	90.81	135.8M	240k	146	53.2
	sqlalchemy-utils	0	12	13	0	0	0	33.91	43.1M	46k	118	61.0
	syzkaller	12	35	11	2	2	0	129.85	197.7M	156k	187	56.0
	xstream	10	28	21	0	1	0	66.24	65.4M	128k	103	32.1
Gemini 3 Flash	bson-rust	3	33	0	6	1	0	8.90	79.8M	94k	204	16.0
	gitoxide	2	38	4	2	1	0	6.80	34.7M	53k	75	10.5
	jfreechart	0	10	6	8	0	1	7.42	57.5M	611k	140	17.4
	libcst	1	45	0	4	1	0	11.10	91.2M	86k	117	27.2
	protocompile	23	45	1	1	3	0	6.93	39.8M	82k	77	12.9
	sqlalchemy-utils	0	21	1	0	3	0	4.26	19.8M	19k	70	35.2
	syzkaller	8	39	4	6	0	1	10.03	99.0M	129k	139	11.9
	xstream	7	46	1	3	0	0	9.50	76.1M	85k	101	8.3
DeepSeek-V3.2	bson-rust	0	13	13	14	0	0	3.03	20.4M	99k	79	28.3
	gitoxide	3	17	11	17	0	0	2.58	14.6M	96k	55	27.7
	jfreechart	0	6	6	13	0	0	1.59	10.7M	77k	76	31.1
	libcst	0	31	4	15	0	0	3.31	20.6M	101k	83	36.0
	protocompile	19	32	2	16	0	0	3.23	22.5M	138k	88	38.4
	sqlalchemy-utils	0	13	4	8	0	0	1.37	9.2M	58k	69	45.5
	syzkaller	10	28	11	11	0	0	3.24	31.5M	145k	80	39.6
	xstream	9	21	11	18	0	0	3.32	28.3M	155k	94	37.7
GLM-5	bson-rust	0	15	24	1	0	0	11.65	38.0M	301k	114	26.6
	gitoxide	3	27	18	0	0	0	10.49	30.7M	281k	91	22.6
	jfreechart	0	11	13	1	0	0	6.32	15.3M	173k	89	27.2
	libcst	0	28	18	4	0	0	10.26	27.4M	208k	86	19.4
	protocompile	29	41	9	0	0	0	8.90	21.6M	136k	65	12.8
	sqlalchemy-utils	0	13	12	0	0	0	4.72	10.8M	126k	74	23.5
	syzkaller	12	39	11	0	0	0	10.54	37.6M	310k	88	21.8
	xstream	11	30	18	2	0	0	10.23	28.3M	239k	81	17.2

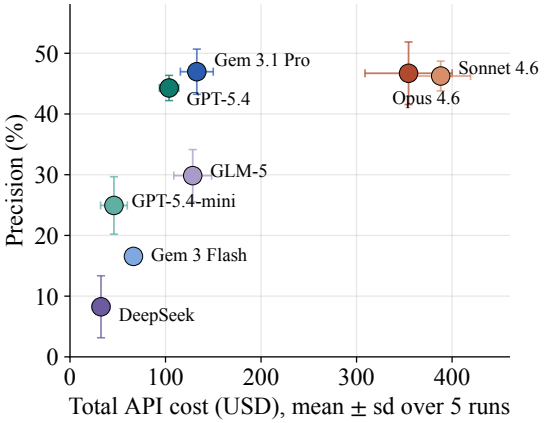


Figure 7: Per-model total API spend vs. precision on the C/C++ benchmark (mean ± sd over five runs; one dot per model). Precision decouples from spend: Gemini 3.1 Pro matches Opus's precision (47.0% vs 46.7%) at just over a third of the spend.

	C/C++		Multi-language		Total	
Model	R (35)	F (35)	R (30)	F (37)	R (65)	F (72)
Opus 4.6	$52.0_{\pm 5.5}$	$1.1_{\pm 1.6}$	$67.3_{\pm 9.2}$	$17.3_{\pm 1.5}$	$59.1_{\pm 5.2}$	$9.4_{\pm 1.2}$
GPT-5.4	$49.7_{\pm 3.3}$	$1.7_{\pm 1.6}$	$58.0_{\pm 7.7}$	$16.8_{\pm 1.2}$	$53.5_{\pm 3.5}$	$9.4_{\pm 1.2}$
Sonnet 4.6	$49.7_{\pm 1.6}$	$1.7_{\pm 1.6}$	$49.3_{\pm 7.6}$	$15.1_{\pm 3.1}$	$49.5_{\pm 3.7}$	$8.6_{\pm 1.2}$
Gemini 3.1 Pro	$44.0_{\pm 5.6}$	$1.1_{\pm 1.6}$	$31.3_{\pm 14.3}$	$8.6_{\pm 4.8}$	$38.2_{\pm 6.7}$	$5.0_{\pm 3.0}$
GPT-5.4-mini	$30.3_{\pm 5.6}$	$0.0_{\pm 0.0}$	$26.0_{\pm 2.8}$	$14.1_{\pm 2.3}$	$28.3_{\pm 3.5}$	$7.2_{\pm 1.2}$
GLM-5	$28.0_{\pm 5.1}$	$1.1_{\pm 1.6}$	$20.0_{\pm 6.2}$	$12.4_{\pm 2.4}$	$24.3_{\pm 4.9}$	$6.9_{\pm 1.0}$
Gemini 3 Flash	$28.6_{\pm 2.0}$	$0.6_{\pm 1.3}$	$18.0_{\pm 5.6}$	$9.7_{\pm 2.4}$	$23.7_{\pm 1.8}$	$5.3_{\pm 1.2}$
DeepSeek-V3.2	$8.0_{\pm 6.2}$	$0.0_{\pm 0.0}$	$12.0_{\pm 3.8}$	$5.9_{\pm 2.3}$	$9.8_{\pm 4.0}$	$3.1_{\pm 1.2}$

Table 11: Base-condition reach % by target family (mean ± sd over five runs). R = non-trivial reachable, F = frontier; family sizes in parentheses. The Total columns pair run i of the two benchmarks.

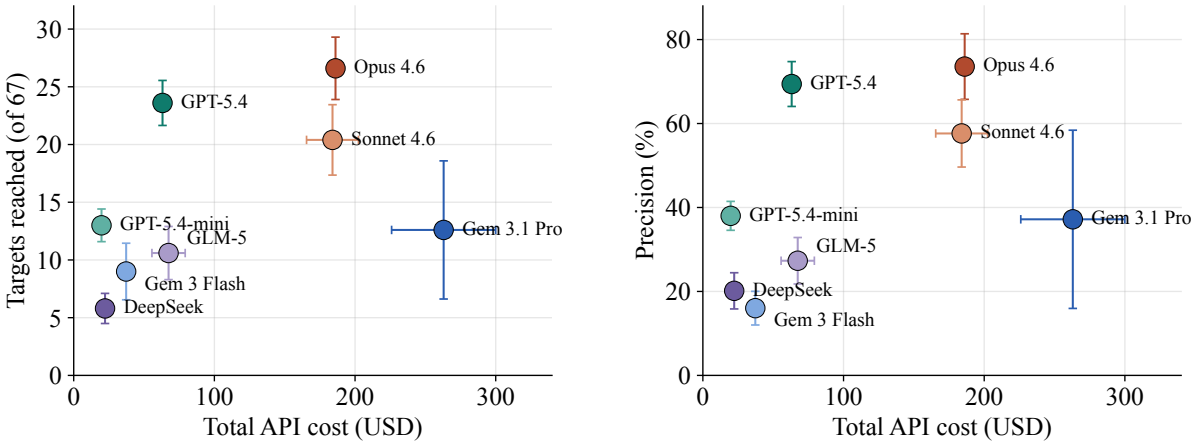


Figure 8: Multi-language cost efficiency: per-model total API spend vs. targets reached (left) and precision (right), base condition, mean ± sd over five runs. The C/C++ picture inverts here: GPT-5.4 approaches Opus's reach at roughly a third of the spend, while Gemini 3.1 Pro spends the most with the widest run-to-run spread.

Table 12: Final line coverage (%) and total LLM cost in the coverage-maximization task on the eight C/C++ projects, sorted by mean. Bold marks each model's mean coverage.

Model	`cups`	`cyclonedds`	`hdf5`	`karchive`	`liboqs`	`libraw`	`rdkit`	`xerces-c`	**Mean**	**Cost ($)**
Claude Opus 4.6	31.9	25.1	19.3	7.8	47.7	48.8	40.1	18.3	**29.9**	211.39
Claude Sonnet 4.6	29.1	24.5	20.2	7.3	47.6	46.1	31.8	17.1	**28.0**	189.24
Gemini 3.1 Pro	28.8	25.0	18.9	7.8	47.7	27.5	37.6	19.2	**26.6**	146.20
Gemini 3 Flash	25.6	13.9	19.8	7.1	47.7	47.4	25.3	17.4	**25.5**	84.26
GPT-5.4	28.2	24.9	19.7	7.7	47.2	2.5	34.7	16.8	**22.7**	198.73
GPT-5.4-mini	25.5	24.1	19.3	7.3	47.2	2.5	27.0	17.5	**21.3**	67.43
GLM-5	25.0	13.9	18.7	7.3	47.7	15.3	19.6	13.9	**20.2**	82.10
DeepSeek-V3.2	12.7	11.1	10.6	7.0	47.6	16.0	5.9	13.5	**15.6**	31.27

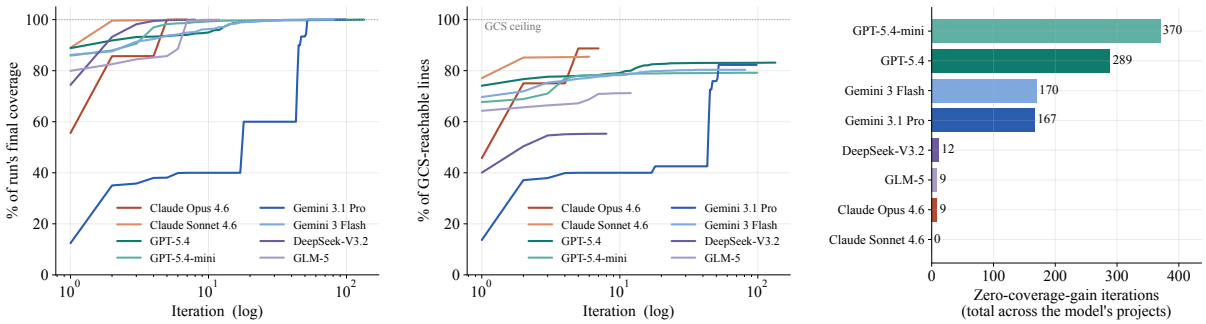


Figure 9: Coverage-maximization dynamics on the C/C++ projects, per iteration. *Left:* mean coverage at iteration k as a percentage of each run's final coverage: Sonnet/Opus reach most of their final coverage by iter 1–2, while GPT-5.4-mini and Gemini 3.1 Pro only approach 100% after tens of iterations. *Middle:* mean coverage as a fraction of GCS-reachable lines (the gap below the dotted 100%-line is the upstream-reachable area still uncovered); curves plateau well below GCS, and Gemini 3.1 Pro shows a distinct late surge around iter 30–40. *Right:* total zero-coverage-gain iterations across the model's 8 coverage runs: GPT-5.4-mini burns 370 and GPT-5.4 burns 289, while Sonnet has none.

Table 13: C/C++ coverage-track per-(model, project) full statistics (one 2-hour run per cell). **Cov%** is the final corpusdb replay, matching Table 12. The remaining columns derive from iteration checkpoints and request logs, which dropped entries under load for some cells, and are therefore lower bounds; corrected per-model cost totals appear in Table 12.

Model	Project	Iters	Inputs	Waste%	Cov%	$	Min
Claude Opus 4.6	cups	2	5,524	0	31.9	3.81	107
	cyclonedds	3	9	0	25.1	7.56	102
	hdf5	1	2,203	0	19.3	1.72	24
	karchive	2	3,641	0	7.8	6.62	116
	liboqs	7	66,885	8	47.7	28.23	93
	libraw	2	27,545	0	48.8	17.36	65
	rdkit	3	0	0	40.1	24.47	80
	xerces-c	5	3,793	0	18.3	8.08	113
Claude Sonnet 4.6	cups	2	477	0	29.1	0.66	41
	cyclonedds	3	1,882	0	24.5	6.07	118
	hdf5	2	501	0	20.2	1.11	49
	karchive	1	260	0	7.3	5.55	45
	liboqs	1	2,182	0	47.6	5.59	26
	libraw	1	418	0	46.1	–	95
	rdkit	1	1,011	0	31.8	3.76	22
	xerces-c	6	924	0	17.1	10.49	108
GPT-5.4	cups	134	593	40	28.2	–	120
	cyclonedds	39	2,194	19	24.9	5.43	120
	hdf5	83	1,442	77	19.7	–	120
	karchive	27	8,354	10	7.7	–	119
	liboqs	15	2,570	2	47.2	–	119
	libraw	88	631	90	2.5	–	120
	rdkit	27	114,114	6	34.7	17.55	119
	xerces-c	35	2,420	25	16.8	13.27	119
GPT-5.4-mini	cups	84	1,529	74	25.5	–	120
	cyclonedds	81	5,032	53	24.1	–	120
	hdf5	82	1,518	67	19.3	–	121
	karchive	79	496	42	7.3	–	121
	liboqs	17	2,086	29	47.2	–	120
	libraw	99	41,988	99	2.5	–	120
	rdkit	48	4,244	20	27.0	7.53	121
	xerces-c	13	5,729	0	17.5	2.92	118
Gemini 3.1 Pro	cups	52	7,621	0	28.8	11.57	120
	cyclonedds	10	0	0	25.0	8.06	118
	hdf5	7	2,475	0	18.9	15.65	117
	karchive	98	3,334	3	7.8	26.10	121
	liboqs	8	116,503	8	47.7	7.02	50
	libraw	–	–	–	27.5	–	–
	rdkit	19	796	0	37.6	12.73	115
	xerces-c	–	–	–	19.2	–	–
Gemini 3 Flash	cups	27	862	0	25.6	–	121
	cyclonedds	47	1,949	7	13.9	–	119
	hdf5	57	15,970	13	19.8	3.39	120
	karchive	40	382	26	7.1	3.45	30
	liboqs	81	781,099	64	47.7	–	120
	libraw	39	1,922	56	47.4	–	119
	rdkit	28	1,614	3	25.3	4.98	89
	xerces-c	16	2,546	2	17.4	7.92	120
DeepSeek-V3.2	cups	8	252	12	12.7	0.36	75
	cyclonedds	4	161	0	11.1	0.26	69
	hdf5	5	230	0	10.6	0.79	87
	karchive	8	282	34	7.0	3.75	101
	liboqs	4	7,340	5	47.6	3.00	68
	libraw	7	385	77	16.0	3.20	70
	rdkit	5	134	0	5.9	4.70	76
	xerces-c	6	105	0	13.5	1.96	81
GLM-5	cups	12	2,724	2	25.0	0.24	114
	cyclonedds	7	3,908	0	13.9	2.85	117
	hdf5	8	2,528	6	18.7	3.24	121
	karchive	7	738	9	7.3	8.21	113
	liboqs	6	209,332	7	47.7	9.70	118
	libraw	7	1,680	30	15.3	6.94	110
	rdkit	9	1,576	0	19.6	8.42	117
	xerces-c	6	538	0	13.9	3.64	65

Snapshot	Configured	Cov. builds	Usable
2018-09	149	108	–
2019-09	240	195	–
2020-07	335	257	–
2021-07	520	393	–
2022-07	648	477	418
2023-08	1,140	910	770
2024-08	1,260	945	797
2025-08	1,309	929	812
2026-01	1,328	959	803

Table 14: Monthly reconstruction of the eligibility gate over the public OSS-Fuzz archive (yearly anchors). Cov. builds = projects with a same-day coverage report; Usable = projects passing the eligibility gate (computable from Feb 2022, when per-harness coverage data begins).

Artifact	C/C++	Multi-lang.
Snapshot metadata	130 KB	13 KB
Agent Docker images	6 GB	6.5 GB
Project runtime / base images	48 GB	3.7 GB
Harness binaries (fuzz + cov.)	39 GB	8.4 GB
Seed corpora (`corpusdb` only)	19 GB	21.3 GB
Harness rebuild time	$\sim$40 min	$\sim$29 min

Table 15: Storage and rebuild requirements per snapshot. The multi-language runtime layers on the shared OSS-Fuzz base images instead of per-project runtime images; its corpora row includes the 18.6 GB per-seed coverage store used by `corpusdb`.