CHECKERBENCH: CAN LONG-HORIZON AGENTS SYNTHESIZE STATIC-ANALYSIS CHECKERS?

Anonymous authors
Paper under double-blind review

ABSTRACT

Static-analysis checker synthesis requires agents to interpret a defect specification, inspect a repository, implement analyzer-specific logic, and refine the checker through repeated compilation and analysis feedback. Existing coding-agent benchmarks focus on tasks such as patch generation or vulnerability detection and rarely assess whether an agent can develop a working checker in a repository from start to finish. We introduce CHECKERBENCH, an executable benchmark of 300 tasks derived from 297 CVEs across 167 repositories, 85 CWEs, and five language ecosystems. Each task includes vulnerable and fixed revisions, a pinned analysis environment, and a checker scaffold. We further introduce CHECKERLAB, a common evaluation framework that independently rebuilds submitted checkers and measures vulnerable–fixed diagnostic contrast, patch localization, false positives, and tool use. Across 21 model–harness configurations and three independent repeats per configuration, mean Pass@1 is 30.81%, while the best reaches 40.67%. These results show that reliable, reusable checker development remains challenging for current coding agents. **Code, Benchmark, and Leaderboard** are available at `https://anonymous.4open.science/r/CheckerBench-Dataset-6BBB`.

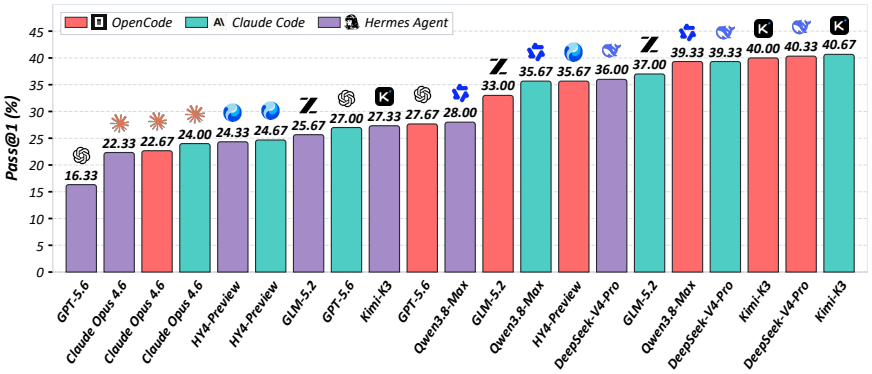

Figure 1: Pass@1 for 21 model–harness pairs on 300 CHECKERBENCH tasks (Table 2).

1 INTRODUCTION

Static analysis provides a scalable approach for detecting complex program defects without executing the target programs (Li et al., 2025b; He et al., 2025; Chen et al., 2025; Yang et al., 2025; Wang et al., 2026b). However, existing static analyzers typically rely on manually designed *checkers*, which require substantial expert effort to encode defect patterns and analysis strategies into tool-specific rules (Garg & Sengamedu, 2022; Yang et al., 2025; Wang et al., 2026b). Recent advances in large language models (LLMs) have opened new opportunities for automated code understanding and defect detection. Nevertheless, directly applying LLMs to code snippets remains challenging due to high inference costs and limited access to repository-wide context, which can lead to incomplete reasoning and hallucinated findings (Hao et al., 2023; Mohajer et al., 2023).

To overcome these limitations, recent coding agents extend LLMs with the ability to interact with software repositories and external tools, enabling iterative code modification, execution, and feedback-driven refinement. These capabilities provide a promising foundation for automating

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

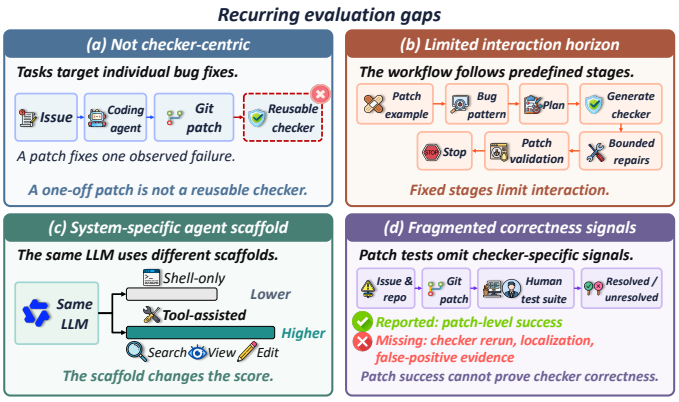

Figure 2: Four recurring evaluation gaps: (a) Not checker-centric (Jimenez et al., 2024); (b) Limited interaction horizon (Yang et al., 2025); (c) System-specific agent scaffold (Yang et al., 2024); and (d) Fragmented correctness signals (Pan et al., 2025).

static-analysis checker development, which traditionally requires substantial manual engineering effort. Given a defect specification, an agent can navigate analyzer infrastructures, implement checker logic, integrate the checker into the build system, and iteratively refine its behavior based on compiler and analyzer feedback (Jimenez et al., 2024; Wang et al., 2025; Ouyang et al., 2025; Xia et al., 2024; Wang et al., 2026a). However, synthesizing effective static-analysis checkers remains a challenging long-horizon task: agents must understand complex analyzer frameworks, make coordinated modifications across multiple files, and ensure that the resulting checker compiles, detects target defects, and maintains low false-positive rates (Li et al., 2025b; Wu et al., 2024; Chen et al., 2025). We refer to this task as ***long-horizon agentic static-analysis checker synthesis***.

However, as illustrated in Figure 2, existing benchmarks are insufficient for evaluating long-horizon agentic static-analysis checker synthesis. *First, current coding and vulnerability benchmarks are not checker-centric*: repository-level benchmarks typically evaluate code completion, patch generation, or feature implementation, while vulnerability benchmarks focus on detection, classification, or repair rather than reusable checker synthesis (Liu et al., 2024; Jimenez et al., 2024; Zhou et al., 2026; Wang et al., 2024; Wen et al., 2024; Yildiz et al., 2025). *Second, existing benchmarks provide limited task interaction*. They either evaluate isolated tasks or decompose workflows into predefined stages, preventing agents from performing the iterative planning, implementation, and debugging required for checker development (Garg & Sengamedu, 2022; Yang et al., 2025; Li et al., 2025a). *Third, benchmarks differ substantially in environments, tool interfaces, and evaluation protocols*, making results difficult to compare across systems (Yang et al., 2024; Wang et al., 2025; Xia et al., 2024). *Finally, current evaluation metrics mainly measure patch-level correctness or test-suite outcomes*, but do not capture checker-specific properties such as vulnerability coverage, false-positive control, and analyzer compatibility (Jimenez et al., 2024; Lee et al., 2025; Wang et al., 2026d). These limitations motivate a unified benchmark that evaluates agents' ability to autonomously synthesize reliable and reusable static-analysis checkers.

To address these gaps, we introduce CHECKERBENCH, the first benchmark for evaluating long-horizon static-analysis checker synthesis. CHECKERBENCH contains 300 real-world tasks collected from 297 CVEs, 167 repositories, 85 CWEs, and five programming-language ecosystems, covering diverse analyzer environments and defect patterns. Each task provides a reproducible repository, an analyzer-specific environment, and reusable skills, enabling agents to perform iterative development and validation rather than isolated code generation. We further develop CHECKERLAB, a unified evaluation framework that assesses coding-agent systems under a common task specification and scoring protocol. CHECKERLAB performs independent repeated generation and frozen-candidate verification: each synthesized checker is rebuilt, executed on vulnerable and fixed revisions, and evaluated based on diagnostic reduction, patch localization, residual reports, and computational cost. Across 21 model–harness configurations, the average Pass@1 score is only 30.81%, with the best-performing configuration reaching 40.67% (Figure 1). These results demonstrate that synthesizing reliable and reusable static-analysis checkers remains a challenging problem for current coding agents. In summary, our contributions are threefold.

- **To the best of our knowledge, we are the first to build a comprehensive executable benchmark for evaluating repository-level, long-horizon static-analysis checker synthesis by coding agents.** CHECKERBENCH contains 300 real-world tasks spanning multiple programming-

Table 1: Systems and benchmarks for agentic static-analysis checker synthesis. ✓ denotes full support, ✓̸ partial support, and ✗ no support. Appendix A defines the operational criteria.

System / Benchmark	Repository Context	Executable Environment	Multi-Harness Support	Long-Horizon Interaction	Checker Synthesis	Semantic Verification	False-Positive Evaluation
RhoSynth Garg & Sengamedu (2022)	✗	✗	✗	✗	✓	✓	✓
KNighter Yang et al. (2025)	✓̸	✓	✗	✗	✓	✓	✓
SWE-bench Jimenez et al. (2024)	✓	✓	✗	✓̸	✗	✓̸	✗
SWE-Gym Pan et al. (2025)	✓	✓	✗	✓	✗	✓̸	✗
R2E-Gym Jain et al. (2025b)	✓	✓	✗	✓	✗	✓̸	✗
ReposVul Wang et al. (2024)	✓̸	✗	✗	✗	✗	✓̸	✗
VulEval Wen et al. (2024)	✓̸	✗	✗	✗	✗	✓̸	✗
JitVul Yildiz et al. (2025)	✓̸	✗	✗	✓̸	✗	✓	✗
CHECKERBENCH	✓	✓	✓	✓	✓	✓	✓

language ecosystems and analyzer environments, enabling systematic evaluation of agentic checker development.

- **We develop a scalable benchmark construction pipeline that automatically reconstructs analyzer environments, packages synthesis tasks, and validates checker behavior.** This pipeline enables reproducible task creation beyond manually curated checker examples.
- **We present CHECKERLAB, a unified evaluation framework for coding-agent systems with a shared task specification and scoring protocol.** CHECKERLAB supports independent repeated generation and frozen-candidate verification, providing fine-grained measurements of checker correctness, patch localization, false-positive control, and computational cost.

2 RELATED WORK

2.1 STATIC ANALYSIS AND CHECKER SYNTHESIS

Static analyzers rely on expert-authored checkers that encode defect patterns and analyzer-specific logic (Garg & Sengamedu, 2022; Yang et al., 2025). RhoSynth learns graph-based code-quality rules from examples, while Gordon explores program-specific analyses from behavioral examples (Garg & Sengamedu, 2022; Gordon, 2018). LLM-based systems execute analysis pseudocode or filter bug reports (Hao et al., 2023; Mohajer et al., 2023). IRIS infers taint specifications for repository-wide analysis (Li et al., 2025b); MoCQ and CQLLM generate vulnerability patterns or CodeQL queries (Li et al., 2025a; Wang et al., 2026c). QLCoder synthesizes CodeQL queries through tool use and iterative feedback (Wang et al., 2026b). KNighter synthesizes and refines Clang Static Analyzer checkers from Linux kernel patches (Yang et al., 2025). These systems target specific analyzers or domains, motivating a common evaluation of general-purpose agents developing and validating checkers across analyzer environments.

2.2 CODING AGENTS FOR SOFTWARE ENGINEERING

Coding agents combine LLMs with repository navigation, code editing, tool execution, and feedback. SWE-agent and OpenHands support interactive development, while RepoGraph adds repository structure (Yang et al., 2024; Wang et al., 2025; Ouyang et al., 2025). RepoAudit applies an agent to repository-level bug detection (Guo et al., 2025). These systems produce patches or bug findings. Checker synthesis requires analyzer integration and validation through compiler and scan feedback on vulnerable and fixed revisions.

2.3 BENCHMARKS FOR REPOSITORY-LEVEL CODE AGENTS

Repository-level benchmarks cover issue resolution (SWE-bench and SWE-bench Pro), feature development (FeatureBench), and terminal tasks (Terminal-Bench) (Jimenez et al., 2024; Deng et al., 2025; Zhou et al., 2026; Merrill et al., 2026). SWE-Gym and R2E-Gym provide executable environments and verifiers for issue-resolution tasks (Pan et al., 2025; Jain et al., 2025b). ReposVul, VulEval, and JitVul target vulnerability detection; SEC-bench evaluates proof-of-concept generation and patching (Wang et al., 2024; Wen et al., 2024; Yildiz et al., 2025; Lee et al., 2025). These works score task completion, patches, or vulnerability detection. CHECKERBENCH evaluates reusable checker construction with held-out verification of builds, vulnerable–fixed diagnostics, patch localization, and false positives (Table 1).

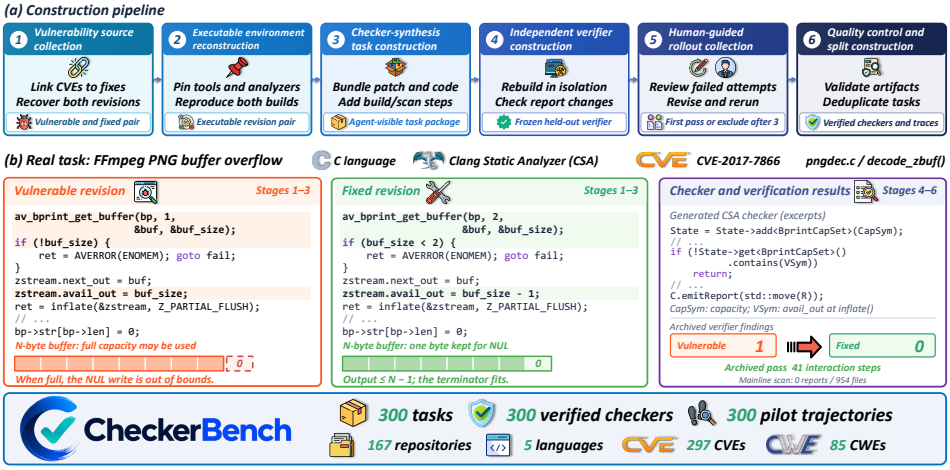

Figure 3: CHECKERBENCH construction.

3 THE CHECKERBENCH BENCHMARK

Problem formulation. Each agent-facing task $x_i = (p_i, \ell_i, \alpha_i, \mathcal{E}_i, \mathcal{Q}_i)$ combines a vulnerability-fix instance $p_i = (r_i, c_i^-, c_i^+, \Delta_i)$ (repository, vulnerable/fixed revisions, and patch) with its language, analyzer, reproducible version-pinned environment, and synthesis instruction. All runs share the initial skill library $\mathcal{K}$; agents may edit checker artifacts but must preserve both revisions.

Under an active generation time budget B, policy π_θ in harness h yields trajectory $\tau_i^{(h,\theta)} = (o_0, a_0, \ldots, a_{N_i-1}, o_{N_i})$ and candidate checker $q_i^{(h,\theta)}$. Here N_i counts interactions, a_k is an action, and o_{k+1} its resulting observation. A released record is $d_i^{(h,\theta)} = (\mathcal{E}_i, \mathcal{Q}_i, \mathcal{V}_i, \tau_i^{(h,\theta)}, q_i^{(h,\theta)})$, with executable verifier $\mathcal{V}_i$ hidden during synthesis. The environment, instruction, and verifier are fixed across model–harness configurations; trajectories and checkers vary by rollout.

3.1 BENCHMARK CONSTRUCTION

Figure 3 summarizes the six-stage pipeline: recover executable vulnerability-fix pairs, package synthesis tasks, freeze independent verifiers, and curate verified checker–trajectory pairs. All candidates start from the same harness-component baseline and shared skill library.

Stage I: Vulnerability Source Collection. We link public CVE records from the National Vulnerability Database[1] to fixing commits in open-source C/C++, Java, Python, JavaScript, and Go repositories. Each candidate must have a retrievable vulnerable–fixed revision pair and a patch that exposes a reusable static-analysis property. We exclude unresolved provenance and incompatible redistribution terms, and retain CVE/CWE mappings for auditing and stratified sampling.

Stage II: Executable Environment Reconstruction. We reconstruct both revisions in isolated workspaces with pinned toolchains and analyzers: Clang Static Analyzer (CSA) for C/C++ and CodeQL for the other four languages. Revision-compatible build snapshots and compilation databases recover the analyzable scope of CSA tasks; CodeQL tasks use pinned analysis environments. Each pair includes affected functions and reproducible checker compilation and scan commands. We reject pairs that cannot be reconstructed or do not support stable differential analysis.

Stage III: Checker-Synthesis Task Construction. Each task exposes the patch, pre- and post-fix functions, build metadata, compile/scan commands, and a minimal checker scaffold. Random workspace identifiers and omitted CVE/CWE labels reduce provenance shortcuts. The shared library $\mathcal{K}$ supplies harness-specific workflow skills and read-only analyzer documentation, example checker implementations, and semantic utilities; it excludes task-specific checker solutions. The agent must infer the defect mechanism, implement a reusable checker, and refine it through compilation and analysis feedback, producing checker artifacts and an interaction trace.

[1] https://nvd.nist.gov/vuln

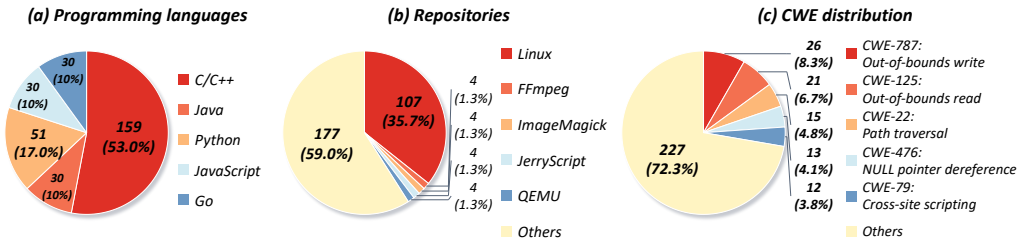

Figure 4: CHECKERBENCH distributions by (a) language, (b) repository, and (c) CWE.

Stage IV: Independent Verifier Construction. We freeze a held-out verifier before rollout collection. It rebuilds each submitted checker from source in a clean workspace and checks vulnerable–fixed diagnostic contrast and patch relevance under backend-specific rules. CSA requires a disappearing report in a patch-modified function. Passing additionally requires process and semantic quality, mainline false-positive control, and mechanism and anti-hardcoding checks (Appendix F).

Stage V: Human-Guided Rollout Collection and Sampling. Calibration uses fixed model, harness, tool, and budget settings. After a failed attempt, a human diagnoses the verification evidence and revises prompt guidance for a fresh rollout; the task, verifier, and thresholds stay fixed. Each task permits one initial attempt and at most two rerolls, stopping at the first pass and excluding tasks with no pass after three attempts. We retain all attempts and interventions for audit. Checker–trace pairs enter sampling only if the checker passes the frozen verifier and the trace contains at least 40 interaction steps (agent decisions). Eligible pairs are sampled jointly over repository, CWE, and horizon. The 40-step threshold is a curation rule; these pilot interventions are separate from autonomous main evaluation.

Stage VI: Quality Control, Deduplication, and Splits. Automated checks establish artifact integrity, reproducible builds and scans, verifier isolation, and package completeness; manual review confirms defect semantics and patch-localized diagnostics. We remove duplicate revision pairs and keep tasks connected by repository, CVE, shared revisions, or similar patches in one split. Task-specific reference checkers, calibration traces, and verifier details remain hidden from evaluation agents. Appendix C details these checks and split safeguards.

3.2 BENCHMARK STATISTICS

Projects and tasks. CHECKERBENCH contains 300 tasks from 167 open-source repositories, including Linux (Linux Kernel Community, 2026) (107 tasks), FFmpeg (FFmpeg Project, 2026) (4), and ImageMagick (ImageMagick Project, 2026) (4). Tasks cover C/C++ (159), Java (30), Python (51), JavaScript (30), and Go (30). Static analysis environments are CSA (159) and CodeQL (141).

Trajectories and interaction horizons. We release 300 accepted calibration trajectories, one per task. Calibration includes human intervention to diagnose failures and revise prompts. All trajectories pass executable verification and the construction-time semantic-score threshold. They are separate from independent main-experiment repeats. Interaction steps reflect workload under the calibration configuration (He et al., 2026). Appendices D.4 and D.5 report trajectory statistics.

Defect and task coverage. CHECKERBENCH covers 85 CWEs. The most frequent are CWE-787 (out-of-bounds write), CWE-125 (out-of-bounds read), and CWE-22 (path traversal), as shown in Figure 4(c). Each task also records *repository scale*, *integration scope*, *verification evidence*, and *calibration horizon*. Appendix D provides complete benchmark statistics.

4 CHECKERLAB: EVALUATION FRAMEWORK

CHECKERLAB separates interactive checker synthesis from independent evaluation of the frozen checker and its trajectory (Figure 5).

Evaluation workflow. Each run receives an isolated workspace with the task instruction, fixing patch, code context, analyzer tools, and enabled skills; the reference checker and held-out verifier remain hidden. Agents use native harness capabilities and compilation and analysis feedback within a common 50-minute generation budget. At termination, we freeze the submitted CSA checker or CodeQL query source and record the trajectory and tool invocations. The verifier then rebuilds the

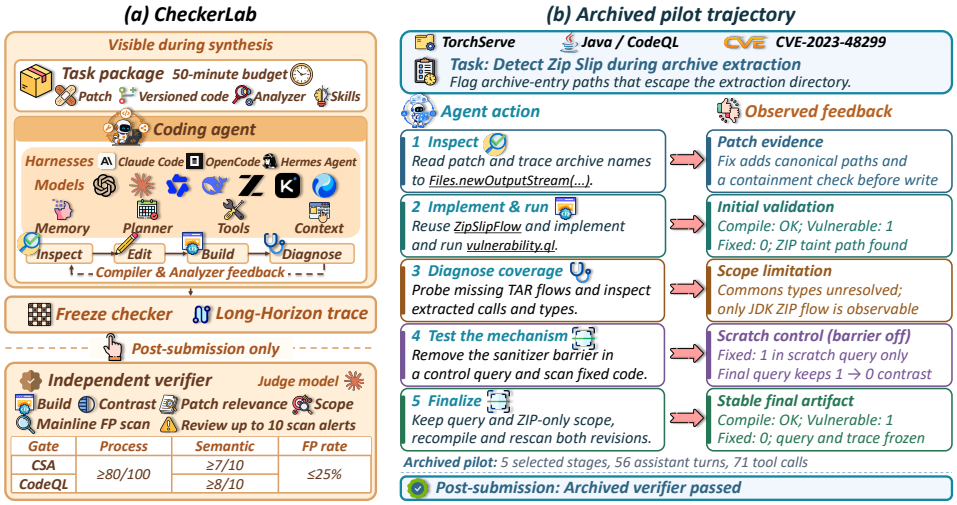

Figure 5: CHECKERLAB workflow and five selected stages from an archived TorchServe pilot.

frozen checker from source in a clean environment and checks vulnerable–fixed diagnostic contrast and patch relevance under backend-specific rules (Section 3.1). A broader mainline scan assesses false positives without checker edits. Claude Opus 5 (Anthropic, 2026c) judges process quality, vulnerability semantics, and sampled mainline alerts in separate stages. A run passes only after normal generation completion and satisfaction of all gates in Equation 5; Appendix F gives the full rubrics.

Metrics. Our primary metric, Pass@1, is the percentage of tasks whose checker passes the full evaluation after one independent synthesis run. For each of the 21 harness–model pairs, we evaluate the same 300 tasks in three independent repeats and average the three rates equally. Generation failures, refusals, and candidate-code errors count as failures. *Build* and *Diff. SR* are the percentages of all tasks passing independent compilation and differential verification, respectively. *Process* and *Semantic* are mean valid scores on 100-point and 10-point scales. The false-positive percentage $\mathrm{FP}_{\mathrm{pass}} = 100 \sum \mathrm{FP} / \sum (\mathrm{TP} + \mathrm{FP})$ pools fully adjudicated recorded mainline alerts from originally passed checkers. We compute each metric per repeat and average equally (Appendix H). Appendix E.6 details alert selection, recorded warning volume, and sensitivity to precision weighting and all-recorded-alert gates. For interaction efficiency, we average turns, tool calls, and explicit skill calls over successful trajectories within each repeat, including subagent activity. We then average these means equally across repeats and report success counts (Appendix E).

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Models and agent harnesses. We evaluate 7 models using 3 agent harnesses, including OpenCode (Anomaly Contributors, 2026), Claude Code (Anthropic, 2026a), and Hermes Agent (Nous Research, 2026). The models are GPT-5.6-Sol (OpenAI, 2026), Claude Opus 4.6 (Anthropic, 2026b), Qwen3.8-Max (Alibaba Cloud, 2026), DeepSeek-V4-Pro-0813 (DeepSeek-AI et al., 2026), GLM-5.2 (Z.ai, 2026a), Kimi-K3 (Kimi Team et al., 2026), and HY4-Preview (Tencent, 2026).

Task execution. Each CHECKERLAB run uses an isolated E2B sandbox and workspace containing the task instruction, fixing patch, relevant code context, and analyzer tools. Agents can inspect code, implement checkers, and use compilation and analysis feedback within a 50-minute active generation budget. The submitted source and execution records are frozen at termination for independent verification. Environment and execution details are provided in Appendix E.

Evaluation and judge. Each frozen checker is evaluated using backend-specific executable checks and scoring rubrics. We use Claude Opus 5 (Anthropic, 2026c) as the LLM judge for false-positive triage, process quality, and vulnerability semantics. Human validation gives a pooled

Table 2: Performance of seven models across three agent harnesses on CHECKERBENCH. **Bold** and underlined values mark the best and second-best models per metric within each harness. Model-row subscripts show numerical changes relative to GPT-5.6 in the same harness: ↑ increases and ↓ decreases (percentage points for percentage metrics). Header arrows indicate the preferred direction. Overall Average summarizes the 21 model–harness configurations.

Model / System	Pass@1 (%) ↑	Build (%) ↑	Diff. SR (%) ↑	FP_{pass} (%) ↓	Process ↑	Semantic ↑	Avg. Turns ↓	Avg. Tool Calls ↓
OpenCode								
GPT-5.6	27.67	**98.67**	88.33	9.09	78.01	7.41	**28.55**	58.54
Claude Opus 4.6	22.67$_{\downarrow 5.00}$	96.33$_{\downarrow 2.34}$	**93.67**$_{\uparrow 5.34}$	20.00$_{\uparrow 10.91}$	76.29$_{\downarrow 1.72}$	7.18$_{\downarrow 0.23}$	35.61$_{\uparrow 7.06}$	50.45$_{\downarrow 8.09}$
Qwen3.8-Max	39.33$_{\uparrow 11.66}$	69.33$_{\downarrow 29.34}$	69.00$_{\downarrow 19.33}$	24.66$_{\uparrow 15.57}$	**90.15**$_{\uparrow 12.14}$	**8.46**$_{\uparrow 1.05}$	36.31$_{\uparrow 7.76}$	47.84$_{\downarrow 10.70}$
DeepSeek-V4-Pro	**40.33**$_{\uparrow 12.66}$	92.67$_{\downarrow 6.00}$	90.67$_{\uparrow 2.34}$	**6.02**$_{\downarrow 3.07}$	84.89$_{\uparrow 6.88}$	8.12$_{\uparrow 0.71}$	50.96$_{\uparrow 22.41}$	65.04$_{\uparrow 6.50}$
GLM-5.2	33.00$_{\uparrow 5.33}$	77.33$_{\downarrow 21.34}$	75.33$_{\downarrow 13.00}$	20.00$_{\uparrow 10.91}$	86.73$_{\uparrow 8.72}$	8.11$_{\uparrow 0.70}$	41.23$_{\uparrow 12.68}$	61.19$_{\uparrow 2.65}$
Kimi-K3	40.00$_{\uparrow 12.33}$	83.33$_{\downarrow 15.34}$	81.67$_{\downarrow 6.66}$	9.46$_{\uparrow 0.37}$	88.48$_{\uparrow 10.47}$	8.30$_{\uparrow 0.89}$	32.58$_{\uparrow 4.03}$	**43.60**$_{\downarrow 14.94}$
HY4-Preview	35.67$_{\uparrow 8.00}$	72.67$_{\downarrow 26.00}$	71.33$_{\downarrow 17.00}$	19.83$_{\uparrow 10.74}$	86.90$_{\uparrow 8.89}$	8.32$_{\uparrow 0.91}$	41.25$_{\uparrow 12.70}$	50.17$_{\downarrow 8.37}$
Average	**34.10**	**84.33**	**81.43**	**15.58**	**84.49**	**7.99**	**38.07**	**53.83**
Claude Code								
GPT-5.6	27.00	**93.00**	79.00	**6.67**	78.85	7.57	35.59	60.93
Claude Opus 4.6	24.00$_{\downarrow 3.00}$	83.00$_{\downarrow 10.00}$	81.00$_{\uparrow 2.00}$	22.22$_{\uparrow 15.55}$	82.51$_{\uparrow 3.66}$	7.38$_{\downarrow 0.19}$	**26.39**$_{\downarrow 9.20}$	38.85$_{\downarrow 22.08}$
Qwen3.8-Max	35.67$_{\uparrow 8.67}$	67.33$_{\downarrow 25.67}$	67.00$_{\downarrow 12.00}$	19.12$_{\uparrow 12.45}$	89.58$_{\uparrow 10.73}$	8.38$_{\uparrow 0.81}$	33.98$_{\downarrow 1.61}$	41.55$_{\downarrow 19.38}$
DeepSeek-V4-Pro	39.33$_{\uparrow 12.33}$	91.33$_{\downarrow 1.67}$	**89.67**$_{\uparrow 10.67}$	7.35$_{\uparrow 0.68}$	85.25$_{\uparrow 6.40}$	8.27$_{\uparrow 0.70}$	46.97$_{\uparrow 11.38}$	62.31$_{\uparrow 1.38}$
GLM-5.2	37.00$_{\uparrow 10.00}$	76.33$_{\downarrow 16.67}$	74.67$_{\downarrow 4.33}$	21.82$_{\uparrow 15.15}$	89.67$_{\uparrow 10.82}$	**8.39**$_{\uparrow 0.82}$	38.70$_{\uparrow 3.11}$	49.92$_{\downarrow 11.01}$
Kimi-K3	**40.67**$_{\uparrow 13.67}$	83.67$_{\downarrow 9.33}$	82.33$_{\uparrow 3.33}$	24.87$_{\uparrow 18.20}$	87.79$_{\uparrow 8.94}$	8.34$_{\uparrow 0.77}$	32.08$_{\downarrow 3.51}$	**38.84**$_{\downarrow 22.09}$
HY4-Preview	24.67$_{\downarrow 2.33}$	49.33$_{\downarrow 43.67}$	47.33$_{\downarrow 31.67}$	20.00$_{\uparrow 13.33}$	**90.83**$_{\uparrow 11.98}$	8.35$_{\uparrow 0.78}$	50.05$_{\uparrow 14.46}$	61.66$_{\uparrow 0.73}$
Average	**32.62**	**77.71**	**74.43**	**17.44**	**86.35**	**8.10**	**37.68**	**50.58**
Hermes Agent								
GPT-5.6	16.33	**98.33**	50.00	22.22	77.81	7.47	**24.53**	44.27
Claude Opus 4.6	22.33$_{\uparrow 6.00}$	97.33$_{\downarrow 1.00}$	**93.00**$_{\uparrow 43.00}$	18.18$_{\downarrow 4.04}$	78.30$_{\uparrow 0.49}$	7.13$_{\downarrow 0.34}$	34.19$_{\uparrow 9.66}$	48.34$_{\uparrow 4.07}$
Qwen3.8-Max	28.00$_{\uparrow 11.67}$	44.00$_{\downarrow 54.33}$	42.33$_{\downarrow 7.67}$	20.35$_{\downarrow 1.87}$	**90.92**$_{\uparrow 13.11}$	**8.53**$_{\uparrow 1.06}$	40.82$_{\uparrow 16.29}$	52.43$_{\uparrow 8.16}$
DeepSeek-V4-Pro	**36.00**$_{\uparrow 19.67}$	94.00$_{\downarrow 4.33}$	90.67$_{\uparrow 40.67}$	21.46$_{\downarrow 0.76}$	85.29$_{\uparrow 7.48}$	8.03$_{\uparrow 0.56}$	39.47$_{\uparrow 14.94}$	62.28$_{\uparrow 18.01}$
GLM-5.2	25.67$_{\uparrow 9.34}$	79.67$_{\downarrow 18.66}$	78.00$_{\uparrow 28.00}$	**4.00**$_{\downarrow 18.22}$	82.87$_{\uparrow 5.06}$	7.81$_{\uparrow 0.34}$	42.19$_{\uparrow 17.66}$	61.94$_{\uparrow 17.67}$
Kimi-K3	27.33$_{\uparrow 11.00}$	87.00$_{\downarrow 11.33}$	79.67$_{\uparrow 29.67}$	24.47$_{\uparrow 2.25}$	82.62$_{\uparrow 4.81}$	8.08$_{\uparrow 0.61}$	32.65$_{\uparrow 8.12}$	**42.24**$_{\downarrow 2.03}$
HY4-Preview	24.33$_{\uparrow 8.00}$	57.67$_{\downarrow 40.66}$	52.67$_{\uparrow 2.67}$	24.00$_{\uparrow 1.78}$	88.42$_{\uparrow 10.61}$	8.44$_{\uparrow 0.97}$	50.33$_{\uparrow 25.80}$	62.51$_{\uparrow 18.24}$
Average	**25.71**	**79.71**	**69.48**	**19.24**	**83.75**	**7.93**	**37.74**	**53.43**
Overall Average	**30.81**	**80.59**	**75.11**	**17.42**	**84.86**	**8.00**	**37.83**	**52.61**

Cohen's $\kappa = 0.85$ for judge–human agreement. Appendix E.9 reports the sampling, labeling, adjudication, and stage-specific agreement. The scoring rubrics are detailed in Appendix F. For the 21 model–harness configurations, we report mean Pass@1 over three independent repeats on the same 300 tasks, with additional diagnostic metrics defined in Section 4. Efficiency metrics are averaged over successful trajectories only.

5.2 OVERALL PERFORMANCE

Across 21 model–harness configurations, average Pass@1 is 30.81%; even the best combination, Kimi-K3 with Claude Code, reaches only 40.67% (Figure 1; Table 2). OpenCode has the highest harness average (34.10%), followed by Claude Code (32.62%) and Hermes Agent (25.71%). Qwen3.8-Max with Hermes Agent leads Process and Semantic scores, yet achieves only 28.00% Pass@1 and 44.00% Build, showing that strong diagnostic scores can coexist with low end-to-end success. Model strengths also vary by task composition (Figure 6): DeepSeek-V4-Pro leads C/C++ (27.04%), while Qwen3.8-Max leads Python (65.36%), Java (61.11%), and JavaScript (56.67%). Qwen3.8-Max also outperforms DeepSeek-V4-Pro on CWE-79, with the ordering reversed on CWE-787. Repository-equal weighting reverses their overall ordering relative to task-equal weighting (Appendix Figure 10). These descriptive comparisons reflect joint language, analyzer, and task composition differences; full breakdowns appear in Appendix H.4. With Claude Code, Kimi-K3 achieves a higher Pass@1 than DeepSeek-V4-Pro (40.67% vs. 39.33%), although its successful runs use fewer turns (32.08 vs. 46.97) and tool calls (38.84 vs. 62.31). In this comparison, the higher success rate is not accompanied by more interaction.

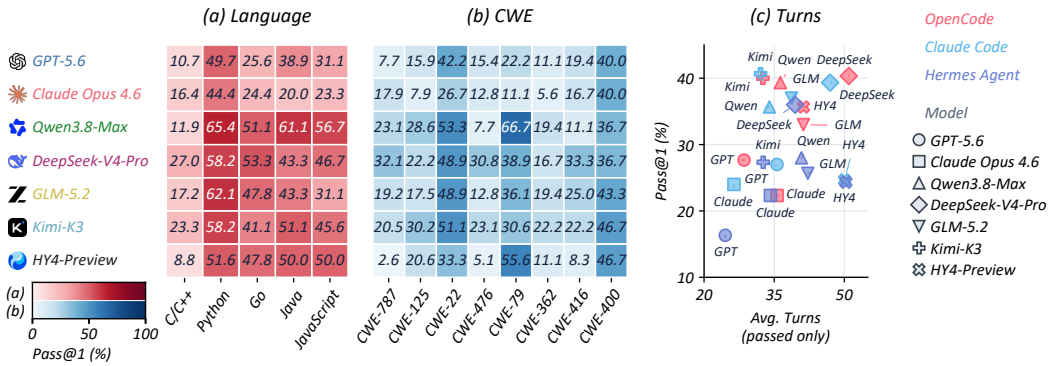

Figure 6: Pass@1 by (a) language and (b) CWE, averaged equally across three harnesses. (c) Pass@1 versus average turns on successful runs; colors denote harnesses and shapes denote models.

5.3 Comparison with Baselines

Table 3: Checker validity on KNighter's 61 Linux-kernel tasks.

Method	Valid	Rate
Ours	**49/61**	**80.3%**
KNighter	**41/61**	**67.2%**
Ours *w/o skills*	**18/61**	**29.5%**

We compare our skill-guided checker-synthesis method, implemented with OpenCode, against KNighter on the 61 Linux-kernel tasks from KNighter's original benchmark. All methods use DeepSeek-V4-Pro. Our method achieves an 80.3% valid-checker rate, compared with 67.2% for KNighter (Table 3). Removing the skill library lowers our method's rate to 29.5%. KNighter organizes synthesis through stage-specific prompts and a checker template, while our method integrates the shared reusable guidance as skills. Its lead on KNighter's original tasks suggests that this skill-guided workflow transfers well to an independently defined task set. Details in Appendix H.1.

5.4 Ablation Study

We compare DeepSeek-V4-Pro on the same 300 tasks, using its original OpenCode run as Full. As summarized by Table 4, Pass@1 is 40.33% for Full, 8.33% without skills and strategy references, 13.33% with execution feedback hidden, and 0.33% for one-shot generation without tools or a harness. These gaps persist after excluding Process and normal-termination requirements: artifact pass is 48.67%, 18.67%, 20.33%, and 1.33%, respectively. One-shot performance also reflects invalid or incomplete submissions in 237/300 tasks. Differences between historical and new run conditions prevent attributing the gaps solely to the ablated components. See details in Appendix H.2.

5.5 Time-Budget Sensitivity

We assess time-budget sensitivity by retrospectively applying 10, 20, 30, 40, and 50-minute active generation cutoffs to the existing trajectories across all 21 model–harness configurations. Only final artifacts completed by each cutoff contribute, with all 300 tasks retained per configuration. Figure 7 shows that aggregate Pass@1 rises from 7.05% at 10 minutes to 24.84% at 30 minutes and 30.81% at 50 minutes, with diminishing gains: the 40-minute cutoff captures 93.35% of the successes observed by 50 minutes, and the final 10 minutes add 2.05 percentage points. These late gains are largest for Qwen3.8-Max and HY4-Preview. The curves characterize completion timing under the original 50-minute protocol; performance under separately imposed shorter deadlines remains untested. Full metrics and timing rules appear in Appendix H.3.

5.6 Analysis of Checker Synthesis Failures

We analyze 4,360 selected failed runs across seven models and three harnesses using tool traces, independent verification logs, and existing judge records. The leading failure categories are budget exhaustion (25.32%), inadequate synthesis process (22.96%), and inadequate semantic modeling (21.15%). These records expose gaps between analysis plans and implementations: for example, a planned path-sensitive analysis becomes AST matching that omits locking and control-flow con-

Table 4: Performance comparison of DeepSeek-V4-Pro ablations on 300 CHECKERBENCH tasks. **Bold** and underlined values mark the best and second-best configurations per metric. Subscripts show numerical changes relative to Full: ↑ increases and ↓ decreases (percentage points for percentage metrics). Header arrows indicate the preferred direction. N/A denotes no alerts.

Configuration	Skill lib.	Exec. feedback	Pass@1 (%) ↑	Build (%) ↑	Diff. SR (%) ↑	FP_{pass} (%) ↓	Process ↑	Semantic ↑	Avg. Turns ↓	Avg. Tool Calls ↓
Full	✓	✓	**40.33**	**92.67**	**90.67**	**6.02**	**84.89**	**8.12**	50.96	65.04
w/o Skills	✗	✓	8.33 $_{\downarrow 32.00}$	79.33 $_{\downarrow 13.34}$	52.67 $_{\downarrow 38.00}$	6.25 $_{\uparrow 0.23}$	65.31 $_{\downarrow 19.58}$	7.39 $_{\downarrow 0.73}$	59.56 $_{\uparrow 1.93}$	80.53 $_{\uparrow 7.80}$
w/o Execution Feedback	✓	✗	13.33 $_{\downarrow 27.00}$	67.33 $_{\downarrow 25.34}$	41.67 $_{\downarrow 49.00}$	25.00 $_{\uparrow 18.98}$	79.93 $_{\downarrow 4.96}$	8.00 $_{\downarrow 0.12}$	53.58 $_{\downarrow 4.05}$	72.16 $_{\downarrow 0.57}$
w/o Harness	✗	✗	0.33 $_{\downarrow 40.00}$	3.33 $_{\downarrow 89.34}$	3.33 $_{\downarrow 87.34}$	N/A	73.64 $_{\downarrow 11.25}$	7.45 $_{\downarrow 0.67}$	**1.00** $_{\downarrow 56.63}$	**0.00** $_{\downarrow 72.73}$

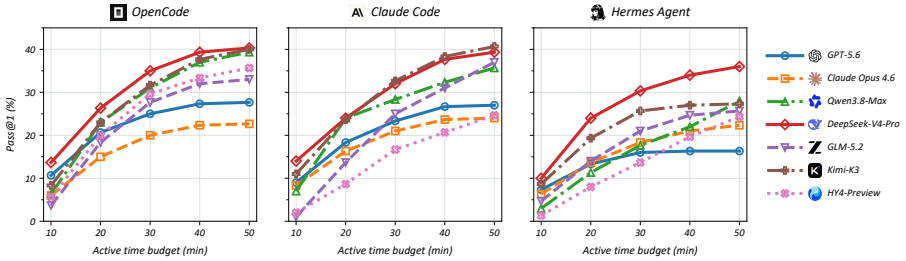

Figure 7: Retrospective Pass@1 versus active generation time for three agent harnesses.

ditions. Excess false positives account for 15.96% of failures, underscoring the need to evaluate precision beyond the original vulnerable–fixed pair. The remaining failures involve differential detection or localization (8.17%), patch-specific overfitting or anchoring (3.56%), missing or invalid artifacts (1.83%), and compilation or registration (1.05%). Key challenges for checker synthesis include timely completion, faithful implementation of analysis plans, and precise semantic modeling.

5.7 DATA CONTAMINATION AND PRIOR EXPOSURE

CVE publication postdates release for 28/300 tasks with Claude Opus 4.6, one with GLM-5.2, and none with other models (Figure 8(a,b)); code or patches may have appeared earlier. On 100 language-stratified tasks, we compare original inputs (O), paraphrases retaining identifiers (P), and versions of P with meaning-preserving identifier substitutions (D) (Fang et al., 2025). Figure 8(c) shows P–D Pass@1 gaps of +3.00 and +2.00 percentage points for Claude Opus 4.6 and DeepSeek-V4-Pro. Exact CVE recall falls from 10.00% to 1.00% and from 8.00% to 1.00%, respectively. The 7–9-point recall decline accompanies only a 2–3-point Pass@1 drop, suggesting that recognizing familiar vulnerabilities yields only limited gains under this intervention (Appendix C.3).

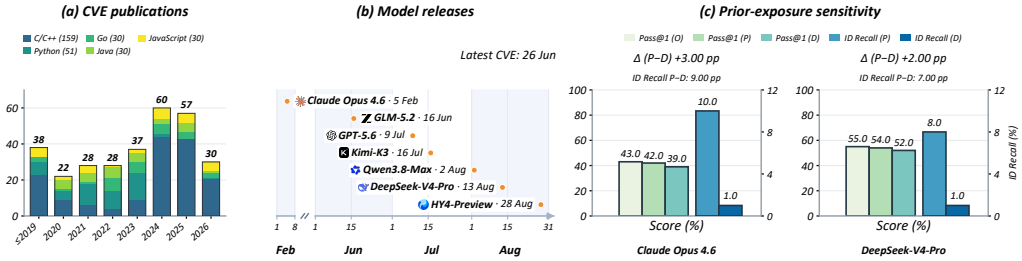

Figure 8: Temporal coverage and prior exposure: (a) CVE publication counts; (b) model release dates; (c) Pass@1 (O/P/D; left axes, 0–100%) and exact CVE recall (P/D; right axes, 0–12%).

6 CONCLUSION

We introduced CHECKERBENCH, an executable benchmark for long-horizon static-analysis checker synthesis with 300 tasks across five programming-language ecosystems. CHECKERLAB independently rebuilds submitted checkers and assesses diagnostic behavior on vulnerable and fixed revisions, patch localization, and false positives. Across seven models and three agent harnesses, the best configuration achieves 40.67% Pass@1, leaving substantial room for improvement within the evaluated repositories and analyzer configurations. CHECKERBENCH provides a reproducible testbed for agents that turn vulnerability specifications into reliable, reusable checkers.

AI USE STATEMENT

We used AI tools to assist with writing and polishing the manuscript. Details of AI use in this work are described in the paper.

REFERENCES

AgentRE-Bench. AgentRE-Bench. Project website, 2026. URL https://www.agentre-bench.ai/.

Alibaba Cloud. Qwen3.8-Max: Model studio release log. https://www.alibabacloud.com/help/en/model-studio/newly-released-models, 2026. Release entry dated August 2, 2026; accessed September 19, 2026.

Anomaly Contributors. OpenCode. https://github.com/anomalyco/opencode, 2026. Accessed August 25, 2026.

Anonymous. BIN-BENCH: Can LLM Agents Reason Through Long-Horizon Binary Analysis? OpenReview, 2026. URL https://openreview.net/pdf?id=KCBDPLjVrV. Anonymous ACL submission.

Anthropic. How Claude Code works. https://code.claude.com/docs/en/how-claude-code-works, 2026a. Accessed August 2, 2026.

Anthropic. Introducing Claude Opus 4.6. https://www.anthropic.com/news/claude-opus-4-6, 2026b. Released February 5, 2026; accessed September 20, 2026.

Anthropic. Introducing Claude Opus 5. https://www.anthropic.com/news/claude-opus-5, 2026c. Accessed September 20, 2026.

Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng Cheng, Fan Yang, Shuvendu K. Lahiri, Tao Xie, and Lidong Zhou. Automated proof generation for Rust code via self-evolution. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/b2e20d7402c9985eae4ba924c65370a8-Abstract-Conference.html.

Cilium eBPF Maintainers. BTF: Fixed panics during parsing of malformed input. https://github.com/cilium/ebpf/pull/2021, 2026. Pull request 2021; merged May 27, 2026.

Clang Project. Clang Static Analyzer. https://clang.llvm.org/docs/ClangStaticAnalyzer.html, 2026. Accessed August 4, 2026.

CVE Program. CVE List V5: Official CVE records. https://github.com/CVEProject/cvelistV5, 2026. Per-record JSON snapshots archived September 15, 2026.

Isaac David and Arthur Gervais. CrackMeBench: Binary Reverse Engineering for Agents, 2026. URL https://arxiv.org/abs/2605.10597.

DeepSeek-AI. DeepSeek Harness. https://github.com/deepseek-ai/deepseek-harness, 2026a. Accessed August 25, 2026.

DeepSeek-AI. DeepSeek-V4-Pro GA release. https://api-docs.deepseek.com/news/news260813/, 2026b. DeepSeek-V4-Pro-0813 released August 13, 2026; accessed September 20, 2026.

DeepSeek-AI et al. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

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, Kai Hua, Juntao Lin, Pai Liu, Mingchen Li, Xuanguang Pan, Zifan Peng, Yujia Qin, Yong Shan, Zhewen Tan, Weihao Xie, Zihan Wang, Yishuo Yuan, Jiayu Zhang, Enduo Zhao, Yunfei Zhao, He Zhu, Liya Zhu, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyang Tao, Jian Yang, Tong Yang, Zhaoxiang Zhang, Xinjie Chen, Wenhao Huang, and Ge Zhang. NL2Repo-Bench: Towards Long-Horizon Repository Generation Evaluation of Coding Agents, 2025. URL https://arxiv.org/abs/2512.12730.

Yixiong Fang, Tianran Sun, Yuling Shi, Min Wang, and Xiaodong Gu. LastingBench: Defend benchmarks against knowledge leakage. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 18304–18317. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-emnlp.993. URL https://aclanthology.org/2025.findings-emnlp.993/.

Yukang Feng, Jianwen Sun, Zelai Yang, Jiaxin Ai, Chuanhao Li, Zizhen Li, Fanrui Zhang, Kang He, Rui Ma, Jifan Lin, Jie Sun, Yang Xiao, Sizhuo Zhou, Wenxiao Wu, Yiming Liu, Pengfei Liu, Yu Qiao, Shenglin Zhang, and Kaipeng Zhang. LongCLI-Bench: A Preliminary Benchmark and Study for Long-horizon Agentic Programming in Command-Line Interfaces, 2026. URL https://arxiv.org/abs/2602.14337.

FFmpeg Project. FFmpeg Source Repository. https://github.com/FFmpeg/FFmpeg, 2026. Accessed August 28, 2026.

Pranav Garg and Srinivasan H. Sengamedu. Synthesizing code quality rules from examples. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):1757–1787, 2022. doi: 10.1145/3563350. URL https://doi.org/10.1145/3563350.

Colin S. Gordon. Synthesizing program-specific static analyses, 2018. URL https://arxiv.org/abs/1810.06600.

Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing. In *International Conference on Machine Learning*, pp. 21083–21100, 2025. URL https://proceedings.mlr.press/v267/guo25n.html.

Yu Hao, Weiteng Chen, Ziqiao Zhou, and Weidong Cui. E&V: Prompting large language models to perform static analysis by pseudo-code execution and verification, 2023. URL https://arxiv.org/abs/2312.08477.

Hang He, Yixing Luo, Chengcheng Wan, Ting Su, Haiying Sun, and Geguang Pu. Automated detection of atomicity violations in large-scale systems, 2025. URL https://arxiv.org/abs/2504.00521.

ImageMagick Project. ImageMagick Source Repository. https://github.com/ImageMagick/ImageMagick, 2026. Accessed August 28, 2026.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/94074dd5a072d28ff75a76dabed43767-Paper-Conference.pdf.

Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environment generation and hybrid verifiers for scaling open-weights SWE agents. In *Conference on Language Modeling*, 2025b. URL https://openreview.net/forum?id=7evvwwdo3z.

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

Kimi Team et al. Kimi K3: Open frontier intelligence, 2026. URL https://arxiv.org/abs/2607.24653.

Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. SEC-bench: Automated Benchmarking of LLM Agents on Real-World Software Security Tasks. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2506.11791.

Penghui Li, Songchen Yao, Josef Sarfati Korich, Changhua Luo, Jianjia Yu, Yinzhi Cao, and Junfeng Yang. Neuro-symbolic Static Analysis with LLM-generated Vulnerability Patterns, 2025a. URL https://arxiv.org/abs/2504.16057v5. Version 5, revised April 2026.

Ziyang Li, Saikat Dutta, and Mayur Naik. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In *International Conference on Learning Representations*, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/582d4e27fa24168f3af1f4582655034b-Abstract-Conference.html.

Zongxia Li, Zhongzhi Li, Yucheng Shi, Ruhan Wang, Junyao Yang, Zhichao Liu, Xiyang Wu, Anhao Li, Yue Yu, Ninghao Liu, Lichao Sun, Haotao Mi, and Leowei Liang. Long-Horizon-Terminal-Bench: Testing the Limits of Agents on Long-Horizon Terminal Tasks with Dense Reward-Based Grading, 2026. URL https://arxiv.org/abs/2607.08964.

Linux Kernel Community. Linux Kernel Source Repository. https://github.com/torvalds/linux, 2026. Accessed August 28, 2026.

Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking repository-level code auto-completion systems. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/d191ba4c8923ed8fd8935b7c98658b5f-Abstract-Conference.html.

LLVM Project. Clang Static Analyzer Checker Implementations. https://github.com/llvm/llvm-project/tree/main/clang/lib/StaticAnalyzer/Checkers, 2026. Accessed August 25, 2026.

Haotian Luo, Huaisong Zhang, Xuelin Zhang, Haoyu Wang, Zeyu Qin, Wenjie Lu, Guozheng Ma, Haiying He, Yingsha Xie, Qiyang Zhou, Zixuan Hu, Hongze Mi, Yibo Wang, Naiqiang Tan, Hong Chen, Yi R. Fung, Chun Yuan, and Li Shen. UltraHorizon: Benchmarking Agent Capabilities in Ultra Long-Horizon Scenarios, 2025. URL https://arxiv.org/abs/2509.21766.

Mike A. Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Daniel Miessler and PAI Contributors. Personal AI Infrastructure. https://github.com/danielmiessler/Personal_AI_Infrastructure, 2026. Accessed August 25, 2026.

MITRE Corporation. Common Weakness Enumeration (CWE). https://cwe.mitre.org/, 2026. Accessed August 25, 2026.

Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. SkipAnalyzer: A tool for static code analysis with large language models, 2023. URL https://arxiv.org/abs/2310.18532.

Moonshot AI. Kimi K3: Model release in Kimi Code. https://www.kimi.com/code/docs/en/kimi-code/whats-new.html, 2026. Release entry dated July 16, 2026; accessed September 19, 2026.

Nous Research. Hermes Agent. https://github.com/NousResearch/hermes-agent, 2026. Accessed August 2, 2026.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. https://openai.com/index/gpt-5-6/, 2026. Accessed September 14, 2026.

OpenClaw Contributors. OpenClaw. https://github.com/openclaw/openclaw, 2026. Accessed August 2, 2026.

Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouthi. SlopCodeBench: Benchmarking How Coding Agents Degrade Over Long-Horizon Iterative Tasks, 2026. URL https://arxiv.org/abs/2603.24755.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. RepoGraph: Enhancing AI software engineering with repository-level code graph. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/4a4a3c197deac042461c677219efd36c-Abstract-Conference.html.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym, 2025. URL https://arxiv.org/abs/2412.21139.

Weihan Peng, Yuling Shi, Yuhang Wang, Xinyun Zhang, Beijun Shen, and Xiaodong Gu. SWE-QA: Can language models answer repository-level code questions?, 2026. URL https://arxiv.org/html/2509.14635v2. Version 2, April 26, 2026; accepted to Findings of ACL 2026.

Tencent. Tencent releases and open-sources Tencent Hy4 preview. https://www.tencent.com/tencent-releases-and-open-sources-tencent-hy4-preview/, 2026. Released August 28, 2026; accessed September 19, 2026.

Hang He, Chuhuai Yue, Chengqi Dong, Chengcheng Wan, Ting Su, Haiying Sun, Jiajun Chai, Xiaohan Wang, and Guojun Yin. VistaHop: Benchmarking long-horizon visual DeepSearch. 2026. URL https://arxiv.org/abs/2606.03273.

Boshi Wang, Weijian Xu, Yunsheng Li, Xuemei Gao, Yujia Xie, Huan Sun, and Dongdong Chen. Improving code localization with repository memory. In *International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=8yjWLJy2eX.

Claire Wang, Ziyang Li, Saikat Dutta, and Mayur Naik. QLCoder: A query synthesizer for static analysis of security vulnerabilities. In *International Conference on Learning Representations*, 2026b. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/2adf01ab15adde8820622f7f24bd516b-Abstract-Conference.html.

Le Wang, Chan Chen, Junyi Zhu, Rufeng Zhan, and Weihong Han. CQLLM: A Framework for Generating CodeQL Security Vulnerability Detection Code Based on Large Language Model. *Applied Sciences*, 16(1):517, 2026c. doi: 10.3390/app16010517. URL https://www.mdpi.com/2076-3417/16/1/517.

Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. ReposVul: A repository-level high-quality vulnerability dataset, 2024. URL https://arxiv.org/abs/2401.13169.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a4b6ad6b48850c0c331d1259fc66a69c-Abstract-Conference.html.

Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. CyberGym: Evaluating AI Agents' Real-World Cybersecurity Capabilities at Scale. In *International Conference on Learning Representations*, 2026d. URL https://openreview.net/forum?id=2YvbLQEdYt.

Xin-Cheng Wen, Xinchen Wang, Yujia Chen, Ruida Hu, David Lo, and Cuiyun Gao. VulEval: Towards repository-level evaluation of software vulnerability detection, 2024. URL https://arxiv.org/abs/2404.15596.

Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/0c86142265c5e2c900613dd1d031cb90-Abstract-Conference.html.

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

Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. KNighter: Transforming static analysis with LLM-synthesized checkers, 2025. URL https://arxiv.org/abs/2503.09002.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Alperen Yildiz, Sin G. Teo, Yiling Lou, Yebo Feng, Chong Wang, and Dinil M. Divakaran. Benchmarking LLMs and LLM-based agents in practical vulnerability detection for code repositories, 2025. URL https://arxiv.org/abs/2503.03586.

Z.ai. GLM-5.2: Built for long-horizon tasks. https://z.ai/blog/glm-5.2, 2026a. Accessed August 2, 2026.

Z.ai. GLM-5.2: Developer release notes. https://docs.z.ai/release-notes/new-released, 2026b. Release entry dated June 16, 2026; accessed September 19, 2026.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, Dandan Tu, and Zhaoxiang Zhang. FeatureBench: Benchmarking agentic coding for complex feature development. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=41xrZ3uGuI.

APPENDIX CONTENTS

A Extended Related Work 16

B Dataset Construction Details 17
- B.1 Source Collection and Provenance 17
- B.2 Build Recovery and Executable Scope 17
- B.3 Task Packaging and Shared References 17
- B.4 Human-Guided Pilot Rerolls 18

C Quality-Control and Contamination Audits 18
- C.1 Temporal Coverage and Prior-Exposure Audit 19
- C.2 Time-Window Evaluation Protocol 20
- C.3 Paired Prior-Exposure Sensitivity Protocol 20

D Dataset Composition 22
- D.1 Repository-Level Task Counts 22
- D.2 Programming Languages and Static Analysis Frameworks 25
- D.3 Complete CWE Label Distribution 25
- D.4 Calibration Horizon and Difficulty Distributions 27
- D.5 Pilot Trajectory Statistics 28
- D.6 CVE Publication-Time Distribution 29

E Evaluation and Inference Protocol 29
- E.1 System Configurations and Reproducibility 29
- E.2 Task Admission and Independent Repeats 30
- E.3 Source and Evidence Freezing 30
- E.4 Backend Execution 30
- E.5 Frozen Mainline Scan and Single-Judge Decisions 30
- E.6 Mainline Alert Selection and Sensitivity 31
- E.7 Failure Accounting, Recovery, and Aggregation 32
- E.8 Statistical Reporting 32
- E.9 Human Validation of the LLM Judge 33

F Post-Synthesis Verifier Rubric 34
- F.1 Independent Replay and Diagnostic Predicates 34
- F.2 CSA Deterministic Objective Gate 34
- F.3 CSA Process and Semantic Quality 35
- F.4 CodeQL Quality Gates 35
- F.5 Mainline False Positives and Final Verdict 36

G Representative CSA Checker Patterns 36

H Additional Experimental Results 37
- H.1 KNighter Comparison Details 37
- H.2 Ablation Outcomes and Coverage 38
- H.3 Time-Budget Sensitivity 39
- H.4 Language, Repository, and CWE Performance 41

A Extended Related Work

Table 1 applies the following operational criteria. *Repository Context* requires the evaluated agent or system to receive a complete repository snapshot together with general-purpose navigation capabilities. Isolated functions, patch-only inputs, pre-extracted dependency views, and repositories used solely by an external validation pipeline receive partial credit. *Executable Environment* requires a reproducible repository or analyzer state with automated build and execution support. *Multi-Harness Support* is assessed against the six coding-agent harnesses integrated through CheckerBench's adapters: Claude Code (Anthropic, 2026a), Hermes Agent (Nous Research, 2026), OpenClaw (OpenClaw Contributors, 2026), PAI (Miessler & PAI Contributors, 2026), OpenCode (Anomaly Contributors, 2026), and DS Harness (DeepSeek-AI, 2026a). Full support requires documented, reproducible integrations for all six harnesses under a shared task and scoring contract; support for a subset receives partial credit. A generic patch-submission format, compatibility that depends on an unreleased wrapper, or a system-specific internal agent does not count as released harness support.

Long-Horizon Interaction requires a repeated observation–tool–edit–execution loop. Protocols that permit multi-step agent execution or bounded retrieval without specifying the complete loop receive partial credit. *Checker Synthesis* requires the task output to be an executable, reusable static-analysis rule or checker. *Semantic Verification* requires an independent rerun of the submitted checker on vulnerable and fixed code together with patch-localized diagnostic evidence; self-reported counts, generic repository tests, and vulnerability labels receive partial credit. *False-Positive Evaluation* requires residual diagnostic control on the fixed revision, a production scan, or an equivalent procedure that measures overly broad reporting. The audit reflects the task definitions, released artifacts, and evaluation protocols documented by each work, not capabilities that could be supplied by an external agent implementation.

Adjacent agent benchmarks. Recent benchmarks extend agentic software engineering beyond short issue resolution tasks. NL2Repo-Bench starts with a natural-language specification and an empty workspace, requiring agents to construct an installable, multi-module Python library (Ding et al., 2025). SWE-Bench Pro instead draws complex issues from maintained repositories, where solutions often require coordinated changes across multiple files (Deng et al., 2025). LongCLI-Bench covers from-scratch development, feature addition, bug fixing, and refactoring in a command-line environment, using requirement and regression tests together with step-level scoring (Feng et al., 2026). SlopCodeBench follows agents as they repeatedly extend their own solutions under evolving requirements, measuring checkpoint success alongside structural erosion and code verbosity (Orlanski et al., 2026).

Other evaluations focus on the trajectory itself. Long-Horizon-Terminal-Bench uses graded subtasks across software engineering and other terminal workflows to measure partial progress during extended runs (Li et al., 2026). UltraHorizon tests sustained exploration in partially observable environments, probing planning, memory, and tool use over long interactions (Luo et al., 2025). These benchmarks expose failure modes in sustained execution that also matter when an agent must implement, compile, and refine a checker.

Binary-analysis benchmarks address a related security workflow. BIN-BENCH evaluates long chains of tool-assisted binary analysis and examines reasoning traces in addition to final outcomes (Anonymous, 2026). AgentRE-Bench asks agents to investigate unfamiliar binaries without source code and scores their structured behavioral claims against fixed ground truth (AgentRE-Bench, 2026). CrackMeBench requires agents to recover a binary's validation logic and submit an input or generator accepted by the executable (David & Gervais, 2026). Together, these works measure software construction, intermediate progress, and binary reasoning. CheckerBench evaluates a distinct artifact: a reusable analyzer checker, independently rebuilt and tested for vulnerable–fixed diagnostic contrast, patch localization, and false positives.

B Dataset Construction Details

B.1 Source Collection and Provenance

We collect public CVE records from the National Vulnerability Database and link them to fixing commits in open-source repositories written in C/C++, Java, Python, JavaScript, and Go. A candidate must identify an exact fix, a retrievable parent revision, and a patch that exposes a defect-bearing program change. CVE identifiers provide instance-level provenance; CWE labels support coverage analysis and stratified sampling. We discard candidates with unresolved provenance, unavailable source, incompatible redistribution terms, or no reusable static-analysis property. A stable instance identifier binds each retained repository, revision pair, patch, and downstream artifact.

B.2 Build Recovery and Executable Scope

The following build-anchor and compilation-database details describe the CSA construction adapter. CodeQL tasks retain their own pinned query-analysis environment and reconstruction metadata. The construction catalog links each vulnerability-fixing commit pair to its repository and to one build anchor per revision. Each anchor records a compilation database, a build snapshot, and build-completeness metadata. Construction first extracts the unified diff, commit message, and pre- and post-fix functions. The vulnerable-side compilation database then filters changed files to translation units for which the original include paths, preprocessor definitions, language mode, and generated dependencies can be recovered. Unmatched files remain in the provenance record as skipped inputs; pairs with no analyzable changed translation unit do not enter synthesis.

For each retained pair, the builder creates isolated Git worktrees for the two source revisions and overlays revision-compatible generated files from the corresponding snapshots. Paths in compilation-database entries are rewritten from the original repository and build roots to the worktree and snapshot roots. Output directives, warnings-as-errors, disabled system includes, and known incompatible GCC-specific options are removed or normalized before CSA execution. Worktrees permit the two revisions to coexist and allow pairs to be processed concurrently without changing the shared repository mirror. They are removed after scanning, and stale worktrees from interrupted tasks are pruned before subsequent construction runs.

After recovering both revisions, we extract the affected pre- and post-fix functions and generate deterministic commands for compiling a checker and scanning the pair. We reject an instance if either revision remains unreconstructable, affected translation units cannot be analyzed after bounded build recovery, or the pair does not support stable differential analysis. The resulting source–build correspondence, build metadata, and scan context remain available for auditing failures.

B.3 Task Packaging and Shared References

Each task is packaged in a randomly named workspace containing the patch, affected functions, a minimal checker template, compile and scan scripts, and read-only reference assets. The visible configuration contains repository, revision, build, and scan-scope metadata but omits CVE and CWE identifiers. Static templates and reference assets are hard-linked when possible and copied otherwise; in either case they are made read-only. A separate provenance ledger maps the random identifier back to the source pair. This design exposes all inputs required for synthesis while preventing vulnerability labels from becoming task shortcuts.

The shared library $\mathcal{K}$ has two layers. Harness-specific skills support patch interpretation, checker design, build recovery, functional validation, and false-positive triage. A harness-independent, read-only reference layer provides analyzer API documentation, official checker or rule implementations (including CSA resources for C/C++ tasks (Clang Project, 2026; LLVM Project, 2026)), reusable language-specific AST, data-flow, and state utilities, CWE mechanism cards (MITRE Corporation, 2026), and repository-specific conventions when available. These assets provide implementation knowledge but do not encode the task's defect-specific state-transition logic or diagnostic reporting predicate. The agent must infer defect semantics, select analyzer extension points and semantic or state representations, implement and register the checker, and refine it using compiler and analyzer feedback. A completed run produces checker source and plugin or query artifacts, a structured defect

analysis, and a scope statement. The declarations support analysis; only independent execution establishes correctness.

B.4 Human-Guided Pilot Rerolls

Calibration fixes model and harness versions, each task's analyzer container and language toolchain, workspace schema, enabled tools, shared skills, and time, token, and interaction budgets for each attempt. Pilot runs cover the inspect–implement–compile–scan–repair cycle. We retain observations, edits, tool calls, and responses. One *interaction step* is one agent decision, independent of token count or low-level tool events.

For each candidate task, the construction protocol permits one initial rollout and at most two additional rollouts after human review. Each attempt uses an isolated workspace and the same task, analyzer configuration, verification predicates, and acceptance thresholds. After a failed checker submission, the reviewer identifies the failure from execution logs and verification evidence, revises the prompt guidance, and records the reason for the change before launching the next rollout. A prompt revision guides the agent's next attempt; every resulting checker is independently checked.

The ledger links the attempt index (1–3), prompt version and changes, human diagnosis, source checksum, complete trace, verification evidence, verdict, and termination status. The first passing checker advances to quality control and sampling. A task with no passing checker after attempt 3 is excluded. The released checker–trace pairs therefore all pass the construction-time checker-admission criteria. These pilot retries are recorded separately from independent main-evaluation repeats and their infrastructure-recovery attempts.

Human diagnosis can address compiler errors, missed or mislocalized diagnostics, semantic mismatches, or false positives. Verifier internals and task-specific reference checker source remain held out from the calibration agent. Each submitted checker undergoes independent contrastive and patch-relevance checks; accepted executable checkers additionally undergo a subsystem scan on a prebuilt main-branch worktree and sampled LLM warning triage (Appendix C). All applicable executable, semantic, scope, and false-positive gates must pass.

Final selection requires reproducible artifacts, complete traces, and independent verification records. We retain passing trajectories with at least 40 interaction steps and sample jointly over repository, CWE category, observed horizon, and difficulty. This threshold selects sustained interaction under the calibration configuration; it is not an intrinsic lower bound on task difficulty. The construction ledger retains checker source and traces for every attempt, independent verification and false-positive evidence, human diagnoses and prompt revisions, attempt counts, and either a passing checker–trace pair eligible for curation or an exclusion record after the cap.

C Quality-Control and Contamination Audits

Automated admission checks cover revision ancestry, patch and artifact integrity, the presence of build anchors, compilation-database coverage, clean checker compilation, registration and plugin loading, deterministic vulnerable–fixed scans, verifier isolation, and package completeness. The CSA contrastive gate requires $N_i^- > N_i^+$ and $N_i^+ < 50$; CodeQL retains its own frozen differential predicate. Passing the CSA gate is insufficient by itself: code verification recomputes diagnostics and requires at least one disappearing vulnerable-revision report whose source location maps to a function modified by the patch. Hence empty scans, unrelated diagnostic reductions, and agent-reported counts cannot satisfy the hard gate.

For accepted executable checkers, a broader subsystem scan on a prebuilt main-branch worktree supplies false-positive evidence. This worktree is neither the vulnerable nor the fixed task revision. During construction-time calibration, a triage model assesses sampled diagnostics. Failed false-positive checks enter the same human-diagnosis and prompt-revision loop as other admission failures, within the shared three-attempt cap (Appendix B.4). Each new checker is rebuilt and independently rechecked. The main evaluation uses the frozen-candidate scan and single ✳ Claude Opus 5 judge in Section 4, without returning scores for repair. Manual audit samples passing and failing submissions across projects, CWE families, and horizon strata, and inspects both defect-

mechanism fidelity and the patch-localized true positive. Final removal counts, auditor agreement, and contamination-overlap statistics will be reported with the frozen release.

Deduplication and split assignment. Manual review confirms that the patch supports the stated defect and that disappearing reports lie in changed defect-bearing functions. We remove exact duplicate revision pairs; separable units from the same CVE retain a shared provenance link. Connected groups formed by repository identity, CVE identity, shared revisions, and patch similarity are each assigned to one split. Repository-disjoint and weakness-family-disjoint subsets are additionally marked for generalization analysis.

Artifact isolation. Reference states, task-specific checker implementations, calibration trajectories, verifier thresholds, and adjudication records remain outside the agent-visible snapshot. The vulnerability patch is an intentional task input. Potential training exposure to public vulnerability information is distinct from evaluation-time leakage of checker implementations, calibration traces, or verifier artifacts. The following audits examine public dates and model release boundaries; artifact isolation addresses the separate evaluation-time leakage risk.

C.1 Temporal Coverage and Prior-Exposure Audit

Public evidence and date provenance. We archive CVE Program JSON records for the fixed task inventory, recording the source URL, retrieval provenance, and SHA-256 checksum alongside each task (CVE Program, 2026). The audit snapshot of September 15, 2026 covers all 300 tasks and 297 canonical CVEs. The original inventory identifier CVE-2024-7773 is a rejected duplicate of CVE-2024-45436; the audit retains the original identifier and uses the canonical record for dating. No task is removed by this normalization. Publication dates range from June 14, 2010, to June 26, 2026. Appendix D.6 gives the complete year distribution. The machine-readable ledger is `figures/temporal/task_dates.csv`, accompanied by the archived records and a public-evidence index.

The ledger separates CVE publication, the CNA's explicit public-disclosure date, dated public events, and independently checked patch or checker publication evidence. Discovery dates, CVE reservation dates, packaging dates, and CVE identifier years do not establish public availability. Git author and committer timestamps alone also do not establish when a patch became public. For 48 tasks, the CVE record explicitly dates a public event before its own publication. The earliest documented event establishes that information was public by that date; it does not prove the absence of still earlier sources. The full first-public patch and checker audit is unresolved, so the ledger does not certify any task as previously unseen on the basis of publication date alone.

Official model release dates. Table 5 uses public release dates documented in provider announcements or release logs, checked on September 19–20, 2026. GPT-5.6 Sol and Claude Opus 4.6 were released on July 9 and February 5, respectively (OpenAI, 2026; Anthropic, 2026b). The official records date Qwen3.8-Max to August 2, DeepSeek-V4-Pro (the evaluated 0813 version) to August 13, GLM-5.2 to June 16, and Kimi-K3 to July 16 (Alibaba Cloud, 2026; DeepSeek-AI, 2026b; Z.ai, 2026b; Moonshot AI, 2026). HY4-Preview was released on August 28 (Tencent, 2026). Dates follow the named model version. Public service or API availability counts as release even if weights follow later; named public previews are included and restricted partner previews excluded. The ledger `figures/temporal/model_releases.json` records each source and date convention. Counts distinguish CVE publication before, on, and after the reported release day. These boundaries describe public availability, not a model's training-data cutoff. The actual model version must be pinned and matched to its release before temporal scoring.

A patch-before-CVE example. CVE-2026-10722, a task in `cilium/ebpf`, was published on June 3, 2026. Its CVE record references pull request #2021, whose fix was already merged on May 27, 2026 (Cilium eBPF Maintainers, 2026). Thus the patch was public a week before the CVE publication. A CVE-only filter can overstate the evidence for novelty relative to a model release between those dates. We retain both dates and require review of the earlier source; we do not infer whether the model actually trained on it.

Table 5: Official model release dates and task counts by CVE publication date. The CVE snapshot is September 15, 2026; model release sources were checked September 19–20, 2026. "On date" counts publications on the release day. Later publication alone does not certify an unseen task.

Model	Release date	Before	On date	After
GPT-5.6	2026-07-09	300	0	0
Claude Opus 4.6	2026-02-05	272	0	28
Qwen3.8-Max	2026-08-02	300	0	0
DeepSeek-V4-Pro	2026-08-13	300	0	0
GLM-5.2	2026-06-16	299	0	1
Kimi-K3	2026-07-16	300	0	0
HY4-Preview	2026-08-28	300	0	0

C.2 Time-Window Evaluation Protocol

Following LiveCodeBench (Jain et al., 2025a), the temporal analysis groups eligible tasks by audited public-availability windows and compares Pass@1 for each harness–model pair using the same three independent generation repeats as the main comparison. Eligibility requires a pinned model version and its official release date together with an audit of public vulnerability descriptions, patches, and task-specific checker sources. Tasks with unresolved public-availability dates remain explicitly unclassified. A common post-release comparison requires the same audited task subset after all included models' releases. The current inventory has no CVE publications after the latest of the seven release dates, so it supplies no common post-release subset for all seven models. Model-specific comparisons report their own eligible task counts and remain conditional on the earlier-source audit.

Windows are fixed from date and task metadata before inspecting outcomes. We begin with half-year windows and merge adjacent windows until each contains at least 20 distinct CVEs, without crossing a model's release date. A remaining smaller subset is reported with its sample count and interval, without fitting a temporal trend. Each point reports its distinct task count and a 95% confidence interval. For each window, we compute Pass@1 from the final-pass verdicts of the same eligible tasks separately in each repeat and report the arithmetic mean of the three rates, weighting repeats equally. Resampling keeps the three repeat outcomes for each task and all tasks sharing a CVE together. We report language and static analysis framework strata wherever both periods have support, and disclose repository and CWE composition for every window; unsupported strata are not treated as matched comparisons. Construction-time horizon information serves only as a workload proxy. A score decline across time is interpreted jointly with these composition differences, rather than as proof of memorization.

The patch remains an intended synthesis input. Reference checker code, calibration trajectories, and verifier artifacts remain held out from the agent-visible package. The audit of generation traces separately records any external retrieval of task-specific solutions or verification material; time-window evidence alone cannot rule out such evaluation-time leakage. The CVE publication counts in Figure 8(a) describe coverage only and do not supply performance estimates. The construction pilot trajectories do not supply independent temporal performance estimates. Performance curves are populated only from completed main-evaluation verdicts and audited task windows.

C.3 Paired Prior-Exposure Sensitivity Protocol

Sampling. The sampling protocol was fixed before collecting supplementary model outcomes. The initial candidate roster contains 100 distinct canonical CVEs: 53 C/C++, 17 Python, and 10 each in Go, Java, and JavaScript. Within each language, task IDs are sorted and placed in a fixed pseudorandom order; canonical-CVE duplicates are skipped across strata. The full candidate order and original metadata hashes are retained. A task that fails environment validation is replaced by the next eligible task in its frozen language order, with the failure evidence and replacement recorded before any generation or recognition probe. The final roster is admitted only after all 100 tasks pass validation. Selection and replacement never use model outcomes or probe recognition.

Conditions and execution. Claude Opus 4.6 and DeepSeek-V4-Pro each run with OpenCode under original (O), ordinary-paraphrase (P), and identity-cue-reduced (D) conditions. P

rewrites non-executable task prose while preserving requirements, source identifiers, and the complete patch. D starts from P and consistently substitutes safely modifiable internal identifiers with meaningful alternatives, and removes residual provenance cues from visible metadata. Standard-library and required external API names, types, constants, control/data flow, and the vulnerability and fix mechanisms are preserved. Unsafe transformations involving reflection, macros, or dynamic name lookup are rejected. This is a semantics-preserving intervention, distinct from the answer-changing counterfactuals studied by LastingBench (Fang et al., 2025).

There is one independent generation per task–model–condition combination, for 600 scheduled generations. This supplementary experiment does not reuse historical Full trajectories. All three conditions share the same skill library, executable feedback, 3,000-second active budget, model decoding settings within each model, and frozen verification rules with Claude Opus 5 as judge. Condition order is randomized within task blocks. Workspaces and sessions are independent, and probes never enter generation context. Model API IDs, returned version metadata, access dates, harness version, and environment fingerprints are archived. Mutable API aliases cannot establish that a provider's underlying weights remained unchanged.

Transformation and admission checks. The before, after, and mainline source trees, patch, extracted context, metadata, and tool-visible paths must agree with the same transformation. For CodeQL, all three corresponding databases must be rebuilt from transformed source; changing only prompts, source-archive text, or database display labels does not qualify. Original Git history, source copies, private reverse maps, reference checkers, and verifier material must be inaccessible to generation. All conditions use preinstalled dependencies and local documentation, with network access restricted to the model gateway.

Admission requires reverse-mapped syntax checks, human review of defect and fix equivalence, successful builds or database extraction, and independent reference-checker replay covering differential behavior, patch localization, and mainline false positives. Validation logs and human review records bind to the exact variant hashes. Missing evidence blocks admission. Infrastructure failures follow the existing bounded recovery policy and are reported separately; they are not imputed as model failures. Valid model failures and budget exhaustion remain failures, without score-based retries.

Claude Opus 4.6 drafts transformation proposals and DeepSeek-V4-Pro provides preliminary critiques. These preparation calls are separate from the 600 synthesis runs and 400 recognition probes and request neither checker solutions nor CVE identification. A human reviewer checks defect and fix equivalence in the complete transformed environments; proposal review does not admit an unbuilt environment. Final semantic approval must reference the exact transformed environment hashes and retain the reviewer identity, review date, supporting evidence, and decision. Independent executable checks remain mandatory. Human review records are retained with the experiment artifacts.

Closed-book recognition probes. Each task–model pair receives separate P and D probes, giving 400 scheduled short calls. Each probe contains pre-fix code only: no fixing patch, CVE/CWE label, source URL, or identifying document heading. A fixed extraction rule selects at most 16,000 characters of code at complete line boundaries; P and D use corresponding source spans, with equal clipping boundaries rather than independently selected excerpts. Probe requests use temperature 0 and an output limit of 256 tokens, with no tools or retrieval. The common instruction is:

> Identify the publicly documented CVE associated with the supplied pre-fix code only if you recognize this specific instance. Do not infer an identifier from the general vulnerability type. Return exactly one JSON object with the key `cve_id`: a single CVE identifier, or `"unknown"` if you do not know. Do not use tools or external retrieval.

ID Recall is the fraction of valid completed calls returning the canonical CVE; documented duplicate identifiers are normalized through the frozen ledger. Incorrect, unknown, refused, or malformed model answers do not receive repair attempts; infrastructure-incomplete calls remain unobserved. A correct identifier indicates instance familiarity, not memorization of a checker solution. Failure to recall an identifier does not establish non-exposure.

Estimation and interpretation. The primary outcome is the existing final `full_pass` verdict. For each model, the primary effect is

$$\Delta_{\text{identity}} = \frac{100}{N}\sum_{i=1}^{N}\big(Y_{i,P} - Y_{i,D}\big), \qquad N = 100,$$

in percentage points. O–P and O–D differences separate general paraphrase sensitivity from the total intervention. The prespecified uncertainty analysis uses 10,000 paired bootstrap draws, resampling canonical-CVE clusters jointly across conditions, to obtain percentile 95% intervals. A supplementary analysis resamples repositories while retaining within-repository task weights. The intervals describe variation across sampled tasks/clusters under one realized generation per condition; they do not separately estimate repeated-generation variability. The two models are reported separately.

Probe P–D recall differences and descriptive P–D effects among tasks recognized and unrecognized under P provide supporting evidence. A decline in recognition together with a positive P–D effect supports sensitivity to identifying cues; ordinary paraphrase effects and residual difficulty changes remain alternative explanations. We prespecify a practical margin of ± 5 percentage points. Only an entire 95% interval inside that margin supports a bounded effect under this intervention. A nonsignificant difference alone does not support equivalence or absence of training contamination. SWE-QA similarly interprets its older-versus-newer source-cohort gap as a possible exposure association (Peng et al., 2026).

Build and Diff. SR retain the scheduled task denominator. Semantic scores report their own valid-score counts. Incomplete conditions and contrasts remain unavailable in the main comparison rather than silently changing denominators. Figure 8(c) in the main text reports aggregate Pass@1 and exact CVE recall for both models. The P–D effects are derived from O/P/D Pass@1 of 43.00/42.00/39.00% for Claude Opus 4.6 and 55.00/54.00/52.00% for DeepSeek-V4-Pro, yielding $+3.00$ and $+2.00$ percentage points, respectively. The corresponding P–D recall declines are 9.00 and 7.00 percentage points.

D Dataset Composition

D.1 Repository-Level Task Counts

CheckerBench contains 300 tasks from 167 repositories: 159 C/C++ tasks use CSA, while the 141 CodeQL tasks comprise Java (30), Python (51), JavaScript (30), and Go (30). Table 6 lists every repository, its target language, static analysis framework, task count, and share.

Projects are counted at the repository level. Dataset prefixes are removed, and repository identifiers are compared without case sensitivity. The historical `python-imaging/Pillow` identifier is merged with `python-pillow/Pillow`; `ImageMagick/ImageMagick` and `ImageMagick/ImageMagick6` remain separate repositories. Target language denotes the language analyzed by the task, which can differ from the repository's dominant implementation language. The counts are taken from the frozen task inventory and its repository-level aggregation.

Across the full corpus, 151 repositories contribute one task each, 7 contribute two, 4 contribute three, and 4 contribute four; the Linux kernel repository contributes 107 tasks (35.7%).

Table 6: Complete repository-level task counts. Shares use all 300 tasks as the denominator and are rounded to one decimal place.

Repository	Target language	Static analysis framework	Tasks	Share
`torvalds/linux`	C/C++	CSA	107	35.7%
`ffmpeg/ffmpeg`	C/C++	CSA	4	1.3%
`imagemagick/imagemagick`	C/C++	CSA	4	1.3%
`jerryscript-project/jerryscript`	C/C++	CSA	4	1.3%
`qemu/qemu`	C/C++	CSA	4	1.3%
`airlift/aircompressor`	Java	CodeQL	3	1.0%
`freerdp/freerdp`	C/C++	CSA	3	1.0%
`gpac/gpac`	C/C++	CSA	3	1.0%
`libredwg/libredwg`	C/C++	CSA	3	1.0%

Continued on next page

Table 6 continued

Repository	Target language	Static analysis framework	Tasks	Share
ikus060/rdiffweb	Python	CodeQL	2	0.7%
libarchive/libarchive	C/C++	CSA	2	0.7%
mlflow/mlflow	Python	CodeQL	2	0.7%
openexr/openexr	C/C++	CSA	2	0.7%
python-pillow/pillow	Python	CodeQL	2	0.7%
rweather/noise-java	Java	CodeQL	2	0.7%
verdammelt/tnef	C/C++	CSA	2	0.7%
adaptivecomputing/torque	C/C++	CSA	1	0.3%
alex/rply	Python	CodeQL	1	0.3%
andialbrecht/sqlparse	Python	CodeQL	1	0.3%
ansible/ansible	Python	CodeQL	1	0.3%
apache/incubator-answer	JavaScript	CodeQL	1	0.3%
apache/netbeans-html4j	Java	CodeQL	1	0.3%
apache/sling-org-apache-sling-jcr-base	Java	CodeQL	1	0.3%
apache/thrift	JavaScript	CodeQL	1	0.3%
appleboy/gorush	Go	CodeQL	1	0.3%
argoproj/argo-events	Go	CodeQL	1	0.3%
argoproj/argo-workflows	Go	CodeQL	1	0.3%
auth0/nextjs-auth0	JavaScript	CodeQL	1	0.3%
avo-hq/avo	JavaScript	CodeQL	1	0.3%
aws/aws-sdk-go	Go	CodeQL	1	0.3%
bettercap/bettercap	Go	CodeQL	1	0.3%
bitly/oauth2_proxy	Go	CodeQL	1	0.3%
bonigarcia/webdrivermanager	Java	CodeQL	1	0.3%
bonitasoft/bonita-connector-webservice	Java	CodeQL	1	0.3%
briancappello/flask-unchained	Python	CodeQL	1	0.3%
brokercap/bifrost	JavaScript	CodeQL	1	0.3%
bspkrs/mcpmappingviewer	Java	CodeQL	1	0.3%
caronc/apprise	Python	CodeQL	1	0.3%
ccxvii/mujs	C/C++	CSA	1	0.3%
cesnet/libyang	C/C++	CSA	1	0.3%
chaosblade-io/chaosblade	Go	CodeQL	1	0.3%
churchcrm/crm	JavaScript	CodeQL	1	0.3%
cilium/ebpf	Go	CodeQL	1	0.3%
codehaus-plexus/plexus-utils	Java	CodeQL	1	0.3%
containernetworking/plugins	Go	CodeQL	1	0.3%
corazawaf/coraza	Go	CodeQL	1	0.3%
cowtowncoder/java-merge-sort	Java	CodeQL	1	0.3%
craftcms/cms	JavaScript	CodeQL	1	0.3%
cryptpad/cryptpad	JavaScript	CodeQL	1	0.3%
darrenofficial/dpaste	Python	CodeQL	1	0.3%
datadog/guarddog	Python	CodeQL	1	0.3%
decidim/decidim	JavaScript	CodeQL	1	0.3%
decolua/9router	JavaScript	CodeQL	1	0.3%
dedsecinside/torbot	Python	CodeQL	1	0.3%
deeplook/svglib	Python	CodeQL	1	0.3%
dhowden/tag	Go	CodeQL	1	0.3%
dojo/dojo	JavaScript	CodeQL	1	0.3%
duke-git/lancet	Go	CodeQL	1	0.3%
dwisiswant0/apkleaks	Python	CodeQL	1	0.3%
eladnava/mailgen	JavaScript	CodeQL	1	0.3%
electerm/electerm	JavaScript	CodeQL	1	0.3%
encode/starlette	Python	CodeQL	1	0.3%
ethyca/fides	Python	CodeQL	1	0.3%
fonttools/fonttools	Python	CodeQL	1	0.3%
foxcpp/maddy	Go	CodeQL	1	0.3%
gomarkdown/markdown	Go	CodeQL	1	0.3%
googleapis/google-oauth-java-client	Java	CodeQL	1	0.3%
goreleaser/goreleaser	Go	CodeQL	1	0.3%
gphper/ginadmin	Go	CodeQL	1	0.3%
gruntjs/grunt	JavaScript	CodeQL	1	0.3%
hedgedoc/hedgedoc	JavaScript	CodeQL	1	0.3%
horazont/xmpp-http-upload	Python	CodeQL	1	0.3%
horovod/horovod	Python	CodeQL	1	0.3%
huggingface/transformers	Python	CodeQL	1	0.3%
hybridgroup/gobot	Go	CodeQL	1	0.3%
hyperadev/dragonfly	Java	CodeQL	1	0.3%
hyperledger/fabric	Go	CodeQL	1	0.3%
ifmeorg/ifme	JavaScript	CodeQL	1	0.3%
imagemagick/imagemagick6	C/C++	CSA	1	0.3%

Continued on next page

Table 6 continued

Repository	Target language	Static analysis framework	Tasks	Share
jberet/jsr352	Java	CodeQL	1	0.3%
jdhwpgmbca/pcapture	Java	CodeQL	1	0.3%
jedisct1/pure-ftpd	C/C++	CSA	1	0.3%
jenkinsci/aws-global-configuration-plugin	Java	CodeQL	1	0.3%
jenkinsci/azure-ad-plugin	JavaScript	CodeQL	1	0.3%
jenkinsci/codedx-plugin	Java	CodeQL	1	0.3%
jenkinsci/dependency-track-plugin	Java	CodeQL	1	0.3%
jenkinsci/environment-manager-tools-plugin	Java	CodeQL	1	0.3%
jenkinsci/xcode-plugin	Java	CodeQL	1	0.3%
jupyter-server/jupyter_server	Python	CodeQL	1	0.3%
keycloak/keycloak	JavaScript	CodeQL	1	0.3%
keylime/keylime	Python	CodeQL	1	0.3%
knative/func	Go	CodeQL	1	0.3%
kyverno/kyverno	Go	CodeQL	1	0.3%
langchain-ai/langchain	Python	CodeQL	1	0.3%
libical/libical	C/C++	CSA	1	0.3%
libssh2/libssh2	C/C++	CSA	1	0.3%
libvnc/libvncserver	C/C++	CSA	1	0.3%
line/centraldogma	JavaScript	CodeQL	1	0.3%
mansuf/mangadex-downloader	Python	CodeQL	1	0.3%
matrix-org/matrix-react-sdk	JavaScript	CodeQL	1	0.3%
matrix-org/sydent	Python	CodeQL	1	0.3%
matrix-org/synapse	Python	CodeQL	1	0.3%
mde/ejs	JavaScript	CodeQL	1	0.3%
meetecho/janus-gateway	JavaScript	CodeQL	1	0.3%
mholt/archiver	Go	CodeQL	1	0.3%
michaelforney/samurai	C/C++	CSA	1	0.3%
misp/misp-maltego	Python	CodeQL	1	0.3%
mity/md4c	C/C++	CSA	1	0.3%
mommyheather/advancedbackups	Java	CodeQL	1	0.3%
moodle/moodle	JavaScript	CodeQL	1	0.3%
mrvautin/expresscart	JavaScript	CodeQL	1	0.3%
mwarning/kadnode	C/C++	CSA	1	0.3%
nautobot/nautobot-plugin-device-onboarding	Python	CodeQL	1	0.3%
netplex/json-smart-v2	Java	CodeQL	1	0.3%
netty/netty-incubator-codec-quic	Java	CodeQL	1	0.3%
nltk/nltk	Python	CodeQL	1	0.3%
nvbn/thefuck	Python	CodeQL	1	0.3%
nvidia/nvflare	Python	CodeQL	1	0.3%
nyuccl/psiturk	Python	CodeQL	1	0.3%
oblac/jodd-http	Java	CodeQL	1	0.3%
ofirdagan/cross-domain-local-storage	JavaScript	CodeQL	1	0.3%
ollama/ollama	Go	CodeQL	1	0.3%
openstack/ceilometer	Python	CodeQL	1	0.3%
opensuse/libsolv	C/C++	CSA	1	0.3%
ossrs/srs	Go	CodeQL	1	0.3%
outray-tunnel/outray	JavaScript	CodeQL	1	0.3%
overleaf/overleaf	JavaScript	CodeQL	1	0.3%
pallets/werkzeug	Python	CodeQL	1	0.3%
panva/jose	JavaScript	CodeQL	1	0.3%
paramiko/paramiko	Python	CodeQL	1	0.3%
petl-developers/petl	Python	CodeQL	1	0.3%
pgadmin-org/pgadmin4	Python	CodeQL	1	0.3%
phpmyadmin/phpmyadmin	JavaScript	CodeQL	1	0.3%
pomerium/pomerium	Go	CodeQL	1	0.3%
prismjs/prism	JavaScript	CodeQL	1	0.3%
prometheus/client_golang	Go	CodeQL	1	0.3%
pytorch/serve	Java	CodeQL	1	0.3%
pytorchlightning/pytorch-lightning	Python	CodeQL	1	0.3%
rasahq/rasa	Python	CodeQL	1	0.3%
redis/redis	C/C++	CSA	1	0.3%
revel/revel	Go	CodeQL	1	0.3%
saitoha/libsixel	C/C++	CSA	1	0.3%
sajari/docconv	Go	CodeQL	1	0.3%
sass/libsass	C/C++	CSA	1	0.3%
satori/go.uuid	Go	CodeQL	1	0.3%
seccomp/libseccomp	C/C++	CSA	1	0.3%
sehmaschine/django-grappelli	Python	CodeQL	1	0.3%
skvadrik/re2c	C/C++	CSA	1	0.3%
smallrye/smallrye-config	Java	CodeQL	1	0.3%

Continued on next page

Table 6 continued

Repository	Target language	Static analysis framework	Tasks	Share
snowflakedb/snowflake-connector-python	Python	CodeQL	1	0.3%
spiral-project/ihatemoney	Python	CodeQL	1	0.3%
stranger6667/pyanyapi	Python	CodeQL	1	0.3%
strukturag/libde265	C/C++	CSA	1	0.3%
strukturag/libheif	C/C++	CSA	1	0.3%
tankywoo/simiki	Python	CodeQL	1	0.3%
tdunning/pig-vector	Java	CodeQL	1	0.3%
tenable/integration-jira-cloud	Python	CodeQL	1	0.3%
txthinking/brook	Go	CodeQL	1	0.3%
uclouvain/openjpeg	C/C++	CSA	1	0.3%
vadz/libtiff	C/C++	CSA	1	0.3%
vyperlang/vyper	Python	CodeQL	1	0.3%
weaveworks/tf-controller	Go	CodeQL	1	0.3%
wildfly/jboss-ejb-client	Java	CodeQL	1	0.3%
xsuchy/templated-dictionary	Python	CodeQL	1	0.3%
xuxueli/xxl-job	Java	CodeQL	1	0.3%
zopefoundation/products.genericsetup	Python	CodeQL	1	0.3%
zwczou/weixin-python	Python	CodeQL	1	0.3%
Total: 167 repositories	**5 ecosystems**	CSA / CodeQL	**300**	**100.0%**

D.2 Programming Languages and Static Analysis Frameworks

Table 7 gives the complete task distribution by target programming language and static analysis framework. Each task contributes once. Language denotes the code analyzed by the task, using the same definition and frozen inventory as Table 6.

Table 7: Complete task counts by target language and static analysis framework. Shares use all 300 tasks.

Target language	CSA	CodeQL	Tasks	Share
C/C++	159	0	159	53.0%
Java	0	30	30	10.0%
Python	0	51	51	17.0%
JavaScript	0	30	30	10.0%
Go	0	30	30	10.0%
Total	**159**	**141**	**300**	**100.0%**

D.3 Complete CWE Label Distribution

Table 8 lists all 85 CWEs and the separate *Other* group. The source retains original task annotations, supplements three previously unlabeled tasks with explicit CWE labels from saved public CVE records, and assigns Other to the 14 tasks for which no explicit CWE label was found. Labels are deduplicated within each task. Of the 300 tasks, 272 have one CWE label, 14 have two, and 14 have none; 286 tasks therefore have an explicit CWE label. The table's Other row contains the 14 tasks without an explicit CWE label.

Each task contributes once to each of its labels. Table percentages use all 300 tasks, so rows can overlap and their shares sum to 104.7% including Other. Figure 4(c) instead partitions the 314 task–label pairs into pie sectors. Its five leading CWEs are shown with their names. The pie's Other sector combines 213 pairs from the remaining 80 CWEs with the 14 unlabeled tasks, giving 227 pairs (72.3%). Ties are ordered by ascending CWE number.

The CWEs comprise 82 Weakness entries and three upstream Category entries (CWE-189, CWE-264, and CWE-399), as recorded in the supplied CWE v4.20 catalog (MITRE Corporation, 2026). Category annotations are retained without remapping them to more specific weaknesses; CWE-264 is marked obsolete in that catalog. CSA and CodeQL columns report task counts for the respective static analysis frameworks.

Table 8: Complete distribution of 85 CWEs and Other. Each task counts once per distinct label; shares use all 300 tasks. Multiple-label tasks appear in more than one row. † marks an upstream Category entry.

CWE	Name	CSA	CodeQL	Tasks	Share
CWE-787	Out-of-bounds Write	25	1	26	8.7%
CWE-125	Out-of-bounds Read	14	7	21	7.0%
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	2	13	15	5.0%
CWE-476	NULL Pointer Dereference	13	0	13	4.3%
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	0	12	12	4.0%
CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	9	3	12	4.0%
CWE-416	Use After Free	12	0	12	4.0%
CWE-400	Uncontrolled Resource Consumption	2	8	10	3.3%
CWE-908	Use of Uninitialized Resource	9	0	9	3.0%
CWE-667	Improper Locking	8	0	8	2.7%
CWE-190	Integer Overflow or Wraparound	6	1	7	2.3%
CWE-415	Double Free	7	0	7	2.3%
CWE-20	Improper Input Validation	1	5	6	2.0%
CWE-611	Improper Restriction of XML External Entity Reference	0	6	6	2.0%
CWE-617	Reachable Assertion	6	0	6	2.0%
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	5	0	5	1.7%
CWE-191	Integer Underflow (Wrap or Wraparound)	5	0	5	1.7%
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	0	5	5	1.7%
CWE-377	Insecure Temporary File	0	5	5	1.7%
CWE-401	Missing Release of Memory after Effective Lifetime	5	0	5	1.7%
CWE-601	URL Redirection to Untrusted Site ('Open Redirect')	0	5	5	1.7%
CWE-770	Allocation of Resources Without Limits or Throttling	3	2	5	1.7%
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	0	4	4	1.3%
CWE-129	Improper Validation of Array Index	3	1	4	1.3%
CWE-674	Uncontrolled Recursion	3	1	4	1.3%
CWE-94	Improper Control of Generation of Code ('Code Injection')	0	3	3	1.0%
CWE-189†	Numeric Errors	1	2	3	1.0%
CWE-287	Improper Authentication	0	3	3	1.0%
CWE-326	Inadequate Encryption Strength	0	3	3	1.0%
CWE-1333	Inefficient Regular Expression Complexity	0	3	3	1.0%
CWE-23	Relative Path Traversal	0	2	2	0.7%
CWE-59	Improper Link Resolution Before File Access ('Link Following')	0	2	2	0.7%
CWE-74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	0	2	2	0.7%
CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	0	2	2	0.7%
CWE-284	Improper Access Control	0	2	2	0.7%
CWE-338	Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)	0	2	2	0.7%
CWE-352	Cross-Site Request Forgery (CSRF)	0	2	2	0.7%
CWE-502	Deserialization of Untrusted Data	0	2	2	0.7%
CWE-532	Insertion of Sensitive Information into Log File	0	2	2	0.7%
CWE-672	Operation on a Resource after Expiration or Release	2	0	2	0.7%
CWE-772	Missing Release of Resource after Effective Lifetime	2	0	2	0.7%
CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')	2	0	2	0.7%
CWE-863	Incorrect Authorization	0	2	2	0.7%
CWE-29	Path Traversal: '\..\filename'	0	1	1	0.3%
CWE-88	Improper Neutralization of Argument Delimiters in a Command ('Argument Injection')	0	1	1	0.3%
CWE-91	XML Injection (aka Blind XPath Injection)	0	1	1	0.3%
CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	1	0	1	0.3%
CWE-122	Heap-based Buffer Overflow	1	0	1	0.3%
CWE-131	Incorrect Calculation of Buffer Size	1	0	1	0.3%
CWE-193	Off-by-one Error	1	0	1	0.3%
CWE-201	Insertion of Sensitive Information Into Sent Data	0	1	1	0.3%
CWE-203	Observable Discrepancy	1	0	1	0.3%
CWE-209	Generation of Error Message Containing Sensitive Information	0	1	1	0.3%
CWE-250	Execution with Unnecessary Privileges	0	1	1	0.3%
CWE-256	Plaintext Storage of a Password	0	1	1	0.3%
CWE-264†	Permissions, Privileges, and Access Controls	0	1	1	0.3%
CWE-266	Incorrect Privilege Assignment	0	1	1	0.3%
CWE-295	Improper Certificate Validation	0	1	1	0.3%

Continued on next page

Table 8 continued

CWE	Name	CSA	CodeQL	Tasks	Share
CWE-312	Cleartext Storage of Sensitive Information	0	1	1	0.3%
CWE-327	Use of a Broken or Risky Cryptographic Algorithm	0	1	1	0.3%
CWE-331	Insufficient Entropy	0	1	1	0.3%
CWE-347	Improper Verification of Cryptographic Signature	0	1	1	0.3%
CWE-364	Signal Handler Race Condition	1	0	1	0.3%
CWE-366	Race Condition within a Thread	0	1	1	0.3%
CWE-369	Divide By Zero	1	0	1	0.3%
CWE-399†	Resource Management Errors	1	0	1	0.3%
CWE-407	Inefficient Algorithmic Complexity	0	1	1	0.3%
CWE-459	Incomplete Cleanup	1	0	1	0.3%
CWE-460	Improper Cleanup on Thrown Exception	0	1	1	0.3%
CWE-522	Insufficiently Protected Credentials	0	1	1	0.3%
CWE-538	Insertion of Sensitive Information into Externally-Accessible File or Directory	0	1	1	0.3%
CWE-552	Files or Directories Accessible to External Parties	0	1	1	0.3%
CWE-613	Insufficient Session Expiration	0	1	1	0.3%
CWE-639	Authorization Bypass Through User-Controlled Key	0	1	1	0.3%
CWE-668	Exposure of Resource to Wrong Sphere	0	1	1	0.3%
CWE-670	Always-Incorrect Control Flow Implementation	0	1	1	0.3%
CWE-680	Integer Overflow to Buffer Overflow	1	0	1	0.3%
CWE-692	Incomplete Denylist to Cross-Site Scripting	0	1	1	0.3%
CWE-789	Memory Allocation with Excessive Size Value	0	1	1	0.3%
CWE-826	Premature Release of Resource During Expected Lifetime	1	0	1	0.3%
CWE-843	Access of Resource Using Incompatible Type ('Type Confusion')	1	0	1	0.3%
CWE-862	Missing Authorization	0	1	1	0.3%
CWE-918	Server-Side Request Forgery (SSRF)	0	1	1	0.3%
CWE-1188	Initialization of a Resource with an Insecure Default	0	1	1	0.3%
CWE-1336	Improper Neutralization of Special Elements Used in a Template Engine	0	1	1	0.3%
Other	Other (no explicit CWE found)	8	6	14	4.7%
Sum across label rows		**165**	**149**	**314**	**104.7%**

D.4 Calibration Horizon and Difficulty Distributions

Table 9 expands the horizon and difficulty statistics reported in Section 3.2. The minimum horizon is a curation constraint: only accepted calibration trajectories with at least 40 interaction steps are eligible for the final 300-instance sample. Difficulty scores are used by the dataset-selection pipeline. The median horizon is 60 steps and the maximum is 123. The associated difficulty score has median 0.684 and range 0.414–0.846. These retained construction-time fields use the original calibration metadata.

Table 9: Interaction-horizon and difficulty distributions of the calibration trajectories.

Interval	Count	Share
Interaction steps		
$[40, 60)$	144	48.0%
$[60, 80)$	104	34.7%
$[80, \infty)$	52	17.3%
Difficulty score		
$[0.0, 0.5)$	32	10.7%
$[0.5, 0.6)$	74	24.7%
$[0.6, 0.7)$	59	19.7%
$[0.7, 0.8)$	130	43.3%
$[0.8, 1.0)$	5	1.7%

Table 10 reports every one-step bin underlying the horizon intervals above, including zero-count bins. These counts use construction-time interaction-step metadata; the message-level pilot counts follow below.

Table 10: Complete one-step frequency counts in the retained construction-time interaction-step metadata. Read down each three-column block. All 84 bins, including zero-count bins, are shown; shares use 300 trajectories.

Steps	Traces	Share	Steps	Traces	Share	Steps	Traces	Share
40	4	1.3%	68	2	0.7%	96	2	0.7%

Continued on next page

Table 10 continued

Steps	Traces	Share	Steps	Traces	Share	Steps	Traces	Share
41	4	1.3%	69	7	2.3%	97	0	0.0%
42	4	1.3%	70	8	2.7%	98	1	0.3%
43	3	1.0%	71	5	1.7%	99	1	0.3%
44	2	0.7%	72	5	1.7%	100	0	0.0%
45	6	2.0%	73	6	2.0%	101	1	0.3%
46	7	2.3%	74	5	1.7%	102	0	0.0%
47	8	2.7%	75	5	1.7%	103	0	0.0%
48	6	2.0%	76	5	1.7%	104	0	0.0%
49	11	3.7%	77	6	2.0%	105	2	0.7%
50	3	1.0%	78	1	0.3%	106	0	0.0%
51	12	4.0%	79	4	1.3%	107	1	0.3%
52	7	2.3%	80	1	0.3%	108	1	0.3%
53	8	2.7%	81	1	0.3%	109	0	0.0%
54	11	3.7%	82	9	3.0%	110	0	0.0%
55	6	2.0%	83	2	0.7%	111	1	0.3%
56	13	4.3%	84	3	1.0%	112	0	0.0%
57	9	3.0%	85	1	0.3%	113	0	0.0%
58	12	4.0%	86	3	1.0%	114	0	0.0%
59	8	2.7%	87	3	1.0%	115	0	0.0%
60	7	2.3%	88	3	1.0%	116	1	0.3%
61	8	2.7%	89	4	1.3%	117	0	0.0%
62	4	1.3%	90	4	1.3%	118	0	0.0%
63	5	1.7%	91	0	0.0%	119	0	0.0%
64	3	1.0%	92	1	0.3%	120	0	0.0%
65	5	1.7%	93	1	0.3%	121	1	0.3%
66	6	2.0%	94	2	0.7%	122	0	0.0%
67	7	2.3%	95	1	0.3%	123	1	0.3%

D.5 Pilot Trajectory Statistics

Figure 9 summarizes the original pilot traces for the 300 selected tasks (159 CSA and 141 CodeQL), with one admitted trajectory per task. Selection follows the original checker-admission records, including 13 trajectories whose checkers passed admission while their agent runs ended at the spending limit. These statistics describe the construction pilot; independent main-evaluation runs follow the final-pass criteria in Section 4.

An *assistant turn* is a unique assistant message, deduplicated by session and message identifier. A *tool call* is a unique tool-use invocation, deduplicated by session and invocation identifier; failed calls and calls without a matching result are included. The tool-call total includes explicit skill invocations. Assistant-turn counts are computed from message events and use a different definition from the construction-time interaction-step metadata in Table 9 and the runner's native turn counter.

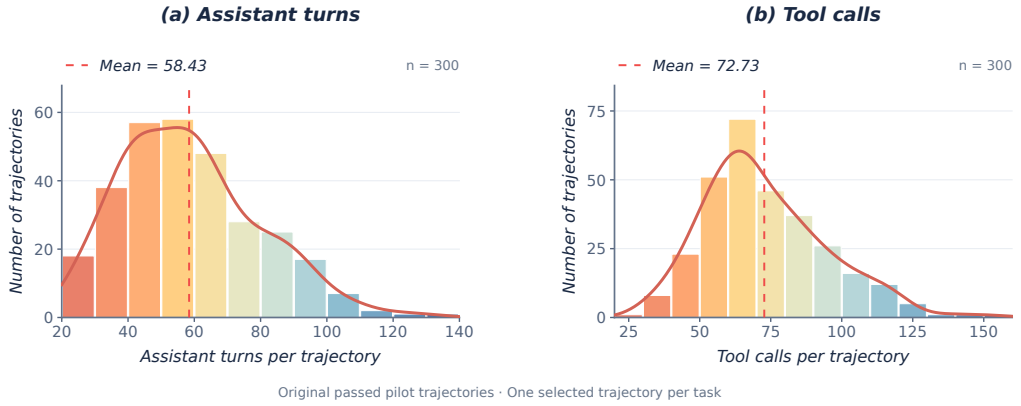

Figure 9: Distributions of (a) assistant turns and (b) tool calls across 300 admitted pilot trajectories in CheckerBench. Histogram bins have width 10. Dashed red lines mark sample means; solid red curves show Gaussian kernel density estimates with Scott's bandwidth, scaled to histogram counts.

Table 11 gives all histogram bins for both pilot metrics. The two columns each account for the same 300 admitted pilot trajectories under their respective event-counting definitions.

Table 11: Complete bin counts for the pilot distributions in Figure 9. Intervals are left-inclusive and right-exclusive. Each share uses 300 pilot trajectories.

Interval	Assistant turns	Share	Tool calls	Share
[20, 30)	18	6.0%	1	0.3%
[30, 40)	38	12.7%	8	2.7%
[40, 50)	57	19.0%	23	7.7%
[50, 60)	58	19.3%	51	17.0%
[60, 70)	48	16.0%	72	24.0%
[70, 80)	28	9.3%	46	15.3%
[80, 90)	25	8.3%	37	12.3%
[90, 100)	17	5.7%	26	8.7%
[100, 110)	7	2.3%	16	5.3%
[110, 120)	2	0.7%	12	4.0%
[120, 130)	1	0.3%	5	1.7%
[130, 140)	1	0.3%	1	0.3%
[140, 150)	0	0.0%	1	0.3%
[150, 160)	0	0.0%	1	0.3%
Total	**300**	**100.0%**	**300**	**100.0%**

D.6 CVE Publication-Time Distribution

Table 12 gives every publication year, including empty years. Counts use the 300 task records, so separable tasks sharing a CVE contribute separately; uncertainty estimates for temporal performance keep those tasks grouped. The 2026 bin is observed through the September 15 metadata snapshot. The complete dating and boundary policies are in Appendix C.1.

Table 12: CVE publication years for the fixed 300 tasks. Dates use the canonical CVE record; the CVE identifier year is not used.

Publication year	CSA	CodeQL	Tasks	Share
2010	0	1	1	0.3%
2011	1	0	1	0.3%
2012	0	0	0	0.0%
2013	1	1	2	0.7%
2014	2	2	4	1.3%
2015	0	0	0	0.0%
2016	2	0	2	0.7%
2017	7	3	10	3.3%
2018	5	2	7	2.3%
2019	5	6	11	3.7%
2020	9	13	22	7.3%
2021	6	22	28	9.3%
2022	4	24	28	9.3%
2023	9	28	37	12.3%
2024	44	16	60	20.0%
2025	43	14	57	19.0%
2026	21	9	30	10.0%
Total	159	141	300	100.0%

E Evaluation and Inference Protocol

E.1 System Configurations and Reproducibility

The seven evaluated models are GPT-5.6-Sol (OpenAI, 2026), Claude Opus 4.6 (Anthropic, 2026b), Qwen3.8-Max (Alibaba Cloud, 2026), DeepSeek-V4-Pro-0813 (DeepSeek-AI et al., 2026), GLM-5.2 (Z.ai, 2026a), Kimi-K3 (Kimi Team et al., 2026), and HY4-Preview (Tencent, 2026). We record harness and model versions, context and decoding settings, reasoning effort, and enabled tools. All systems share pinned repositories, toolchains, containers,

resource limits, and the verifier. Released artifacts record prompts, skills, tool schemas, commands, workspace diffs, diagnostics, termination reasons, and submissions.

E.2 Task Admission and Independent Repeats

The evaluation manifest fixes 300 environments (159 CSA and 141 CodeQL), three harnesses, seven models, and three independent generation repeats per environment–harness–model combination. All three repeats cover the same 300 tasks for each of the 21 harness–model pairs, yielding 6,300 logical runs per repeat and 18,900 in total. The reported results are the arithmetic mean of the three repeat-level estimates, with equal weight for each repeat. Recovery attempts are tracked separately. A held-out reference checker is used to verify environment usability before runs are admitted. Only environment–model combinations that pass preflight are released, and blocked combinations remain visible in the coverage ledger. The reference checker is never part of the synthesis inputs.

Each independent repeat uses a separate workspace and a 50-minute generation budget. It receives the task instruction, patch, relevant code context, tools, and enabled skills. Visible compilation and analysis feedback can support multiple inspect–write–compile–repair cycles within this budget. The final source is `checker.cpp` for CSA or CodeQL query source. Generation completion status is recorded independently of downstream verification.

E.3 Source and Evidence Freezing

At termination, the ledger records the submitted source, SHA-256 digest, complete trajectory, tool-call events, explicit skill invocations, elapsed time, usage, and termination reason. Task, harness, model, container, tool, and scorer versions are retained with independent-repeat and recovery-attempt indices. Subsequent scans and judges consume this exact candidate. Compiled artifacts can be regenerated from it, but verification recovery and a valid low quality score cannot replace its source. Raw evidence is retained alongside any normalized summaries and referenced by path and content hash.

E.4 Backend Execution

The reference execution image pins LLVM/Clang 18 and mounts the task workspace at `/work`. Checker source is compiled with `/opt/llvm/bin/clang++` in C++17 mode as a position-independent shared object using `-fPIC`, `-shared`, `-fno-rtti`, `-fno-exceptions`, the pinned LLVM include directory, and `-Wl,--allow-shlib-undefined`. The resulting object is a CSA plugin, not a standalone executable. Each scan invokes `/opt/llvm/bin/clang --analyze`, loads the plugin through the analyzer, enables its registered checker name, and appends the translated compilation-database flags and source file. Diagnostics are attributed to the checker through its registered diagnostic label. The task-local validator and the held-out verifier call the same generated scripts, but the verifier rebuilds from source in a clean workspace.

For CodeQL, the verifier independently compiles and executes the submitted query using the task's pinned CodeQL environment and revision-specific analysis inputs. It enforces the frozen CodeQL rules for target and patch relevance, vulnerable–fixed differential behavior, query scope, diagnostic quality, and hardcoding. CSA-specific compiler commands, report signatures, count limits, and plugin gates are not reused as CodeQL acceptance rules. The backend and rule version accompany every reported gate result.

E.5 Frozen Mainline Scan and Single-Judge Decisions

After paired-revision verification, the same checker is scanned over a broader mainline scope. The mainline code is distinct from the vulnerable and fixed task revisions. At most ten warnings requiring judgment are sampled; the selected warning IDs and supporting code context are retained. Appendix E.6 specifies the selection rule, recorded warning counts, and the distinction between the sampled gate and aggregate precision. ✳ Claude Opus 5 (Anthropic, 2026c) performs false-positive triage and, in separate stages, assesses process quality and vulnerability semantics. All stages use this one judge model and record the exact model configuration and rubric version. Judge decisions cannot override a failed executable or risk gate.

Table 13: Recorded warning volume in the historical, originally passed repeat-1 cohort. Counts precede the ten-alert triage selection.

Backend	Checkers	Zero alerts	> 10 alerts	Recorded alerts	Sampled alerts
CSA	550	348	11	880	596
CodeQL	1,390	1,218	5	455	382
Total	1,940	1,566	16	1,335	978

Each stage commits its first valid decision, including valid low scores and negative verdicts. Only unavailable or invalid evidence follows recovery; quality-based resampling is prohibited. Mainline "refinement" is an assessment of the frozen candidate, with no follow-up edits by the synthesis model. Empty warning sets and missing or invalid judgments follow each backend's frozen rules; an undefined sample statistic is not fabricated.

E.6 Mainline Alert Selection and Sensitivity

Selection unit and ordering. The frozen implementation selects individual alert records from each checker's saved mainline output. Selection is deterministic conditional on that list; it is not uniform random sampling over alerts, files, or alert types, and the selector does not rank by severity. All records are judged when the list has at most ten entries. For a longer list of length N, the zero-based selected indices are

$$\begin{aligned} I_{\mathrm{CSA}} &= \{k\lfloor N/10 \rfloor : k = 0, \ldots, 9\}, \\ I_{\mathrm{CodeQL}} &= \{\mathrm{round}(k(N-1)/9) : k = 0, \ldots, 9\}. \end{aligned} \tag{1}$$

CSA retains checker-tagged warnings and deduplicates identical path–line–column locations within each compiler invocation. Its collected list follows scan-worker completion order, so a fresh scan need not reproduce the same ordering. In particular, for 11–19 CSA warnings the rule takes the first ten. CodeQL uses the normalized finding order and first persists at most 100 findings; its ten-alert selector operates on that retained list. Neither backend allocates a fixed quota per file or alert type. Replaying a sample therefore requires the saved list and selected warning identities.

Warning volume before triage. Table 13 reports the archived audit of 1,940 originally passed checkers from retained repeat 1, covering all 21 harness–model combinations. This historical cohort predates the subsequent aggregate corrections to Table 2; it is not a three-repeat total. Of its 1,335 recorded alerts, 978 were initially sampled and 357 required supplementary judgment. Only 16 checkers had more than ten recorded alerts; 1,566 had none. These counts cover the originally passed cohort, not all generated checkers. CSA stops collecting after a compiler invocation brings the collected total to at least 100 warnings, and one scan in this cohort reached that cap. Consequently, recorded counts do not establish the number of alerts an uncapped scan of the entire repository would produce.

Alert-weighted and checker-weighted precision. The ten-alert sample supplies a per-checker acceptance gate. The reported $\mathrm{FP}_{\mathrm{pass}}$ instead pools recorded alerts from originally passed checkers, including supplementary judgments beyond that sample. For a fixed system and repeat, let $\mathcal{C}_+$ contain those checkers with $N_i = \mathrm{TP}_i + \mathrm{FP}_i > 0$. The corresponding alert-weighted and checker-weighted precision percentages are

$$P_{\mathrm{micro}} = 100 \frac{\sum_{i \in \mathcal{C}_+} \mathrm{TP}_i}{\sum_{i \in \mathcal{C}_+} N_i}, \qquad P_{\mathrm{macro}} = \frac{100}{|\mathcal{C}_+|} \sum_{i \in \mathcal{C}_+} \frac{\mathrm{TP}_i}{N_i}. \tag{2}$$

Thus $\mathrm{FP}_{\mathrm{pass}} = 100 - P_{\mathrm{micro}}$; larger alert sets receive more weight in this metric. P_{macro} gives every alert-bearing checker equal weight. Zero-alert checkers are reported separately and enter neither precision denominator; a clean scan does not establish perfect precision.

Observed sensitivity in the archived cohort. We recompute both estimators using the same post hoc Opus 5 audit of all 1,335 saved alerts. It contains 1,063 TP and 272 FP labels. Table 14 averages the seven system-level estimates equally within each harness. The precision ordering changes from OpenCode, Claude Code, Hermes Agent under alert weighting to Claude Code,

Table 14: Precision weighting and additional FP-gate sensitivity in the archived repeat-1 cohort (percent). Each value averages seven model-specific estimates equally. Both Pass@1 columns use the same 300 tasks per model; the additional gate requires per-checker FP $\leq 25\%$ across all recorded alerts and retains all other original failures. Table 2 reports the three-repeat mean with later aggregate corrections.

	Precision		Pass@1 (repeat 1)	
Harness	Alert weighted	Checker weighted	Original	Additional gate
OpenCode	84.42	81.92	34.05	32.76
Claude Code	82.14	88.00	32.62	31.38
Hermes Agent	72.04	77.34	25.71	24.38

OpenCode, Hermes Agent under checker weighting. Within Claude Code, the precision leader changes from GPT-5.6 (93.33% micro) to HY4-Preview (96.11% macro). These are conditional precision rankings, distinct from Pass@1 rankings.

Adding the all-recorded-alert gate removes 81 of the 1,940 original passes. The harness ordering by Pass@1 remains unchanged, but the leading harness–model pair changes from Claude Code/Kimi-K3 (122/300 originally) to OpenCode/DeepSeek-V4-Pro (119/300 after the added gate). The archived sampled gates and supplementary alert audit both used Claude Opus 5 judgments. The additional gate expands coverage from the ten-alert sample to all recorded alerts while retaining the same judge. Moreover, this is a passed-only audit of saved output: it does not re-adjudicate originally failed checkers or recover alerts beyond scan caps. A representative-subset comparison would need a common task subset spanning repositories and backends, include checkers reaching the FP stage regardless of their original verdict, and compare the sampled and complete outputs under the same judge and rubric. That analysis is not established by this audit; stability of the current full-evaluation rankings under exhaustive adjudication remains unverified.

E.7 Failure Accounting, Recovery, and Aggregation

Final `passed` requires normal generation completion and every backend-specific executable, mainline, process, semantic, mechanism, scope, and anti-hardcoding gate. Valid generation failures, refusals, and candidate code errors are failures. Infrastructure failures permit at most two additional attempts under the recovery policy. When source already exists, verification recovery reuses it and preserves prior valid stage decisions. Each additional attempt is attached to its original logical run and does not contribute an additional independent value to the reported estimate.

Unresolved infrastructure outcomes are recorded separately from valid model failures, and evaluation coverage is reported with the scheduled manifest. Incomplete repeat groups are reported as incomplete coverage; they are not silently treated as completed failures or removed from the scheduled task set. For each harness–model pair, we compute Pass@1 as the percentage of the 300 tasks whose checker passes the full evaluation in each independent repeat. We report the arithmetic mean of these three rates, giving each repeat equal weight (Section 4). Each independent run contributes its own final verdict; a successful repeat does not replace a failed repeat. Final-success efficiency summaries average interaction turns, tool calls, and explicit skill calls over passed trajectories in that repeat, counting recorded invocation events including subagent activity. The conditional means from the three repeats are averaged equally. Each repeat's passed-trajectory count is retained. If a retained repeat has no successful trajectory, its conditional efficiency and the aggregate efficiency mean are `N/A`, not zero. Full failure and recovery records remain archived for audit and resource accounting.

E.8 Statistical Reporting

The reported main comparison, time-budget levels, and task subsets average the same three independent repeats equally. The separate ablation study uses one generation per task for each new variant and reuses the original Full cohort (Appendix H.2). The subgroup figure and tables report descriptive estimates without significance claims. Ablation intervals resample paired task IDs jointly across configurations; they reflect task composition and do not estimate uncertainty over repeated model

generations or independent CVE clusters. Valid-score sample sizes and backend-specific breakdowns accompany diagnostic means; passed-trajectory counts accompany efficiency summaries.

Build and Diff. SR use all 300 tasks as their denominator. Process and Semantic average available valid scores on their respective scales. For originally passed checkers, each recorded mainline alert is labeled TP or FP. The reported $\mathrm{FP}_{\mathrm{pass}}$ pools these alerts as $100 \sum \mathrm{FP} / \sum(\mathrm{TP} + \mathrm{FP})$. Checkers with no alerts contribute no denominator. One retained mainline scan reached its warning cap. Judge labels are not human ground truth. The Average and Overall Average rows in Table 2 are unweighted means of the reported metric values over the seven models within each harness and all 21 harness–model combinations, respectively.

E.9 Human Validation of the LLM Judge

We validate Claude Opus 5's false-positive triage, process-quality assessment, and vulnerability-semantic assessment against independent human annotations under the backend-specific rubrics in Appendix F. The three annotators have backgrounds in computer science graduate study and software development.

Sampling and judgment units. We sampled 100 of the 159 CSA tasks and 100 of the 141 CodeQL tasks. The CodeQL allocation comprised 25 tasks each for Java, Python, JavaScript, and Go. The task allocation followed these backend and language strata, with one frozen checker submission per task and no score-boundary or outcome oversampling. Each of the 200 submissions received one process and one semantic judgment. For each backend, 80 of the 100 sampled tasks contributed 100 individual mainline warnings: 60 tasks contributed one warning each, and 20 contributed two each. The remaining 20 tasks contributed none. This yielded 200 warning judgments, 400 submission-stage judgments, and 600 judge–human pairs. No item was excluded or weighted. Model, harness, repeat, and pass/fail status were retained for audit, without allocation quotas.

Human labels and reference. All three annotators labeled every item independently, giving 1,800 initial human labels. Their view contained the warning and source context for false-positive triage; checker source and trajectory evidence for process quality; and the fixing patch plus vulnerable and fixed code for semantic quality. Judge labels and rationales, model and harness identities, and final verdicts were hidden during initial labeling. A majority of the three binary labels defined the human reference. For process and semantics, the binary label records whether the score and applicable discrete gates meet the backend's rubric: process at least 80/100, CSA semantics at least 7/10, and CodeQL semantics at least 8/10. The table covers these stage decisions; numeric-score error and individual rubric-criterion agreement are outside its scope.

Label distribution and agreement. Table 15 gives separate judge and majority-human positive counts. Positive means a false positive at the warning stage and an accepted decision at the process and semantic stages. Each row contains 100 valid pairs. Judge–human agreement uses unweighted Cohen's κ; human–human agreement uses three-rater Fleiss' κ on the original independent binary labels. Exact matches range from 89 to 95 per row. The 2×2 cells in table order are $(21, 3, 2, 74)$, $(23, 2, 5, 70)$, $(67, 3, 5, 25)$, $(76, 3, 2, 19)$, $(59, 7, 4, 30)$, and $(65, 5, 3, 27)$, with cells ordered as judge-positive/human-positive, judge-positive/human-negative, judge-negative/human-positive, and judge-negative/human-negative. The minority vote is positive on $(3, 5, 4, 3, 7, 5)$ split items in table order; the remaining split items have a negative minority vote.

Adjudication. The annotation set has 57 nonunanimous human items. Majority voting resolved all 57; no item was tied or removed from the denominator. The majority label changed only the human audit reference and left benchmark verdicts intact. Fleiss' κ uses the 1,800 initial labels before majority resolution.

CSA versus CodeQL and interpretation of $\kappa = 0.85$. Across the six rows, the 556 matching labels among 600 pairs give exact agreement of 0.9267. The judge-positive total is 334 and the human-positive total is 332, so chance agreement is 0.5060 and pooled Cohen's $\kappa = (0.9267 - 0.5060)/(1 - 0.5060) = 0.852$ to three decimal places. These calculations give the pooled $\kappa = 0.85$ reported in the main text after rounding to two decimal places. Reproducing the CSA and CodeQL results requires the frozen task IDs, checker and warning IDs, judge labels, all three independent human labels, and the adjudicated references. Task-clustered uncertainty intervals likewise require those records.

Table 15: Human validation of the three categorical judge stages, with 100 items per row. J+/H+ gives judge and majority-human positive counts; exact agreement is a count. Split gives items with a nonunanimous initial human vote.

Backend	Stage	J+/H+	Exact	J–H κ	H–H κ	Split
CSA	FP triage	24/23	95	0.861	0.867	7
CodeQL	FP triage	25/28	93	0.821	0.835	10
CSA	Process	70/72	92	0.806	0.852	9
CodeQL	Process	79/78	95	0.852	0.883	6
CSA	Semantic	66/63	89	0.760	0.800	14
CodeQL	Semantic	70/68	92	0.813	0.832	11
Pooled	**All stages**	**334/332**	**556**	**0.852**	**0.872**	**57**

F Post-Synthesis Verifier Rubric

CSA and CodeQL retain separate frozen executable and scoring policies. Their common final requirements are normal generation completion, independent executable evidence, mainline false-positive control, process and semantic quality, and backend-specific mechanism, scope, and anti-hardcoding checks. Figure 5 gives the numerical thresholds. The CSA implementation details below describe the `post-synthesis-v4` rubric; CodeQL retains its own backend requirements.

F.1 Independent Replay and Diagnostic Predicates

Before collecting trajectories, we instantiate and freeze each task's verifier $\mathcal{V}_i$ from its recovered revision pair and pinned environment $\mathcal{E}_i$. Given checker source q_i, the verifier ignores agent-produced binaries, rebuilds or compiles the candidate in a clean workspace, and scans the task-specific scope in c_i^- and c_i^+. CSA uses a loadable checker plugin; CodeQL uses submitted query source in its pinned analysis environment. Version-pinned scripts preserve each backend's context, while isolation prevents artifact leakage.

Replay produces diagnostic multisets $W_i^-(q_i)$ and $W_i^+(q_i)$, with $N_i^\pm(q_i) = |W_i^\pm(q_i)|$. Executable validity combines three binary indicators for build success, diagnostic contrast, and patch relevance:

$$\begin{aligned} \mathrm{Valid}_i(q_i) &= B_i(q_i)\, C_i(q_i)\, L_i(q_i), \\ B_i(q_i) &= \mathbf{1}[\mathrm{Build}_{\alpha_i}(q_i, \mathcal{E}_i)], \\ C_i(q_i) &= \mathcal{C}_{\alpha_i,i}(W_i^-, W_i^+), \\ L_i(q_i) &= \mathcal{L}_{\alpha_i,i}(W_i^-, W_i^+, \Delta_i). \end{aligned} \tag{3}$$

Here, $\mathcal{C}_{\alpha_i,i}$ is the frozen differential-detection predicate and $\mathcal{L}_{\alpha_i,i}$ is its target/patch-relevance predicate. CSA contrast requires $N_i^- > N_i^+$ and $N_i^+ < T_{\mathrm{res}}$, with $T_{\mathrm{res}} = 50$ fixed before collection. Reports are matched using the line-insensitive signature

$$\kappa_i(w) = (\mathrm{checker}(w), \mathrm{file}(w), \mathrm{func}(w), \mathrm{msg}(w)).$$

At least one disappearing report must lie in a patch-modified function. CodeQL applies its own frozen query, differential, and relevance rules; CSA's count and matching predicates are not substituted for them. Replay retains normalized locations, diff-hunk distances, residual diagnostics, and in- and out-of-patch reports as evidence.

Let O_i denote the completed objective gate, including required artifacts, independent replay, and the backend's scope and diagnostic-quality checks. CSA uses the operational G6 gate below; CodeQL uses its own frozen objective gate. In both cases, $O_i \Rightarrow \mathrm{Valid}_i(q_i) = 1$. Executable validity, process quality, semantic fidelity, and mainline false-positive control are non-compensating requirements.

F.2 CSA Deterministic Objective Gate

This gate invokes no LLM. The source and rebuilt plugin are `checker.cpp` and `checker.so`. The full frozen CSA contrast and localization predicates must hold in addition to the recorded gate evidence.

G0. The task work directory and interaction trace exist.

G1. The checker source exists and contains at least 50 bytes.

G2. Independent compilation produces the loadable analyzer artifact.

G3. `verify_result.json` exists and the vulnerable revision produces at least one diagnostic, $n_{\text{buggy}} > 0$.

G4. Vulnerable and fixed diagnostics satisfy the frozen CSA contrast rule; equal positive counts are rejected.

G5. Patch relevance is at least 0.5 and diagnostic sparsity is at least 0.3.

G6. All G0–G5 requirements pass.

Only G6 makes a run eligible for a nonzero CSA process score. Four additional objective diagnostics—Differential Detection (15 points), Diagnostic Sharpness (15), False-Positive and Scope Control (15), and basic CSA Implementation Quality (4)—are reported for diagnosis. These 49 points are excluded from S_p to avoid counting executability again as process quality.

F.3 CSA Process and Semantic Quality

The process stage examines the patch, vulnerable and fixed functions, `state.json`, scope statement, frozen source, synthesis summary, and validation evidence. Its raw score Q_i comprises Patch Semantic Understanding (12 points), Detection Plan Quality (12), Plan-to-Code Consistency (12), and Validation Quality (6), for $Q_{\max} = 42$. Repeated reads, redundant commands, excessive edits, and non-progressing build loops incur a waste penalty $p_i^{\text{waste}} \in [0, 5]$. The process floor is 80/100.

The semantic stage first infers the root cause, trigger and safe conditions, defect class, propagation requirements, and minimum analyzer mechanism from the patch and vulnerable functions. It then compares the frozen checker with those requirements. The ordered mechanism ladder is

$$\textit{syntax-only} \rightarrow \textit{call-match} \rightarrow \textit{path-sensitive} \rightarrow \textit{dataflow} \rightarrow \textit{cross-procedure}.$$

Mechanism adequacy is classified as `adequate`, `partial`, `inadequate`, or `overkill`; semantic correctness as `yes`, `partial`, or `no`. For ten-point correctness and mechanism-fit subscores $s_{c,i}, s_{m,i}$, the process and semantic scores are

$$S_{p,i} = \mathbf{1}[O_i]\,\frac{100\max(0, Q_i - p_i^{\text{waste}})}{Q_{\max}}, \qquad S_{s,i} = 0.6s_{c,i} + 0.4s_{m,i}. \tag{4}$$

The CSA semantic floor is 7/10. CSA additionally requires `mechanism_adapted=adequate`, `semantic_correct` in {`yes`,`partial`}, valid parsing, and no semantic-anchor risk. A patch-adjacent AST match may therefore fail semantic assessment despite satisfying a diagnostic contrast.

F.4 CodeQL Quality Gates

CodeQL follows the layered admission structure described above while retaining backend-specific executable, differential, relevance, scope, and hardcoding predicates. Its query is independently rebuilt and checked on the vulnerable and fixed revisions before scoring, and its refinement scan uses the shared mainline false-positive procedure with the CodeQL threshold of at most 25%. Queries that fail or incompletely satisfy these prerequisites are not scored on process or semantic quality.

The CodeQL process-quality score is a 100-point score with a process floor of 80. Its dimensions are vulnerability modeling (18 points), observable-evidence planning (14), mechanism selection (18), plan-to-code consistency (18), validation and recovery (14), efficiency and finalization (10), and generalization discipline (8). The semantic-quality score is a 10-point score in half-point increments with a floor of 8; it assesses the vulnerable sink, trigger and dataflow, safety barrier, CodeQL mechanism, and report-scope generalization. In addition to the numeric floor, automatic admission requires correct vulnerability and safety-barrier modeling, sound dataflow or control-flow support, acceptable scope generalization, an adequate mechanism, and no patch hardcoding or project-specific scope.

These CodeQL-specific conditions refine the common final requirements without changing the non-compensating relation between executable validity, false-positive control, process quality, and semantic quality.

F.5 Mainline False Positives and Final Verdict

The frozen candidate's mainline scan and at most ten judged warnings provide false-positive evidence. The sampled FP rate must be at most 25% for both CSA and CodeQL. For a nonempty, validly judged sample, FP rate is the fraction classified as false positives. Sample composition, warning-level decisions, and the backend's complete refinement outcome are stored. A backend-recognized clean scan follows its frozen empty-sample policy.

Let F_i denote normal generation completion; J_i the required valid judge decisions; R_i the mainline false-positive gate, including the backend's full refinement rules; M_i adequate mechanism fit; C_i^{sem} the required semantic-correctness classification; and H_i, A_i hardcoding and semantic-anchor risks. With $t_{\text{CSA}} = 7$ and $t_{\text{CodeQL}} = 8$, the final verdict is

$$\begin{aligned}\textsc{Pass}_i = F_i \wedge O_i \wedge J_i \wedge R_i \wedge [S_{p,i} \geq 80] \wedge [S_{s,i} \geq t_{\alpha_i}] \\ \wedge M_i \wedge C_i^{\text{sem}} \wedge \neg(H_i \vee A_i).\end{aligned} \tag{5}$$

The main evaluation uses Claude Opus 5 (Anthropic, 2026c) for staged false-positive, process, and semantic judgments and retains each stage's first valid decision. A valid low score is a failure, not a reason to ask again.

A valid failed gate or low score yields a failed quality outcome; no new checker is requested. An unavailable judge or missing verification evidence cannot grant a pass and follows the bounded infrastructure-recovery rules, without discarding valid earlier decisions. Only a normally completed generation whose candidate satisfies every required gate receives final `passed`. Objective-only scoring cannot issue that final outcome.

G Representative CSA Checker Patterns

This appendix summarizes a selected set of implementation patterns from official LLVM 18 Clang Static Analyzer (CSA) checkers (LLVM Project, 2026). The catalog applies only to the CSA-backed C/C++ subset of CheckerBench; tasks in other language ecosystems use the corresponding analyzer-native rule APIs. We retain patterns that expose distinct synthesis challenges rather than reproducing complete checker implementations.

Table 16: Selected CSA checker patterns used as implementation references for C/C++ tasks, from stateless local checks to path-sensitive lifecycle analyses with persistent program state.

Pattern / official checker	**Callbacks / state**	**Synthesis challenge**
Constraint violation `DivZeroChecker`	`PreStmt` *State:* none	Split feasible states and report only a definitely zero denominator.
Undefined value `UndefResultChecker`	`PostStmt` *State:* none	Identify the undefined operand and retain useful diagnostic provenance.
Resource lifecycle `SimpleStreamChecker`	`PostCall`, `PreCall`, `DeadSymbols`, `PointerEscape` *State:* symbol-to-state map	Track opened and closed resources, double close, escape, and leaks.
Nullability propagation `NullabilityChecker`	Binding, calls, returns, locations, assumptions, and events *State:* two maps and an invariant flag	Propagate annotations and constraints across multiple semantic events.
Memory management `MallocChecker`	Calls, locations, dead symbols, escapes, and assumptions *State:* region-state map plus auxiliary maps and sets	Model allocation families, deallocation compatibility, leaks, double free, and use after free.

Stateless constraint checks. `DivZeroChecker` and `UndefResultChecker` illustrate the smallest useful checker structures. They register a single statement callback, query the symbolic value already maintained by CSA, and emit a report without introducing persistent state. The former

uses a dual feasibility assumption to separate zero and nonzero denominators; the latter inspects an undefined result after expression evaluation and attributes it to the relevant operand. These patterns test whether an agent can select the correct callback phase and translate a local defect predicate into analyzer constraints.

Finite-state resource tracking. `SimpleStreamChecker` represents the minimal lifecycle pattern. A program-state map associates each resource symbol with an opened or closed state. Post-call handling records acquisition, pre-call handling validates and updates release, dead-symbol handling reports leaks, and pointer-escape handling stops tracking resources whose ownership can no longer be established. This pattern requires the synthesized checker to coordinate several callbacks while preserving a compact state machine.

Multi-callback semantic propagation. `NullabilityChecker` and `MallocChecker` represent the complex end of the catalog. They combine call, statement, location, assumption, liveness, and escape events with multiple state traits. Their purpose here is not to provide templates for direct copying, but to expose the design decisions required for repository-level synthesis: choosing semantic events, defining stable state keys, suppressing infeasible or duplicate reports, and attaching path evidence to diagnostics.

Common reporting contract. Across the selected patterns, a checker obtains symbolic values from the current analysis state, creates a sink or non-fatal error node as appropriate, constructs a path-sensitive bug report, marks relevant symbols and source ranges, optionally installs a visitor for path notes, and finally emits the report. CHECKERBENCH evaluates the resulting behavior through independent rebuilding and vulnerable–fixed diagnostic contrast rather than by matching these reference implementations.

H ADDITIONAL EXPERIMENTAL RESULTS

H.1 KNIGHTER COMPARISON DETAILS

Tasks and comparison. The comparison in Section 5.3 uses the 61 Linux-kernel tasks from KNighter's original benchmark (Yang et al., 2025). Each task takes a bug-fixing commit as input and requires a CSA checker that distinguishes buggy code from its patched version. We evaluate our OpenCode-based method, KNighter, and our method without skills, all using DeepSeek-V4-Pro. Table 3 reports valid-checker counts and rates.

Time budget. All three configurations use the same 50-minute (3,000-second) active generation budget per task. The budget covers model responses, tool execution, and checker-repair iterations across the complete workflow, including KNighter's internal agent coordination. Recorded admission queues and maintenance pauses, as well as subsequent independent verification, are excluded under the timing convention in Appendix H.3. The submitted checker is frozen when generation terminates.

Accessible documentation. Our method and KNighter receive the same fixing patch, source-code context, repository revisions, and pinned CSA environment. Both can consult the same read-only Linux/CSA documentation, analyzer API references, and official checker examples described in Appendix B.3. These shared resources exclude held-out reference checkers and task-specific checker solutions.

Skill library. Both systems have access to the same reusable skill and strategy content for patch interpretation, checker design, build recovery, and validation. The content is exposed through each system's native interface. The *w/o skills* variant of our method removes these skill packages and strategy references while retaining the same task inputs, factual API documentation, tools, and execution budget.

Task prompts. The task-facing instructions standardize the synthesis objective, supplied patch and code context, available resources, required checker artifact, and validation requirements. System-specific role prompts and internal orchestration retain each harness's workflow, while the task information and reusable knowledge made available to our method and KNighter are held constant.

Validation criteria. All three configurations use the same independent verifier, scan scopes, and acceptance predicates. The verifier rebuilds each frozen checker from source in the pinned CSA

Table 17: Artifact pass and differences from Full. All rates use the full task denominator; CSA and CodeQL use 159 and 141 tasks. Intervals use 10,000 paired task bootstrap samples.

Configuration	All (%)	Δ pp [95% CI]	CSA (%)	CodeQL (%)
Full	48.67	Reference	44.03	53.90
w/o Skills	18.67	-30.00 $[-36.33, -23.67]$	15.72	21.99
w/o Execution Feedback	20.33	-28.33 $[-34.67, -22.00]$	17.61	23.40
w/o Harness	1.33	-47.33 $[-53.00, -41.33]$	0.00	2.84

environment and applies the same build-success, vulnerable–fixed diagnostic-contrast, and patch-relevance checks (Equation 3). Agent-reported success does not determine the verdict. Table 3 reports the number of valid checkers and the corresponding percentage over the same 61 tasks.

H.2 Ablation Outcomes and Coverage

Each configuration covers the same 300 tasks: 159 CSA and 141 CodeQL tasks. The three new variants have 900 frozen candidates in total; Full reuses the original 300 candidates. We independently evaluated the 21 historical artifacts whose assessment had stopped at the generation deadline. All 21 fail an artifact requirement, including three with no submitted source; Full therefore has 146/300 artifact passes and its original 121/300 full passes. No candidate was regenerated or edited during this completion.

Configurations and generation. Each new configuration uses one generation per task. *w/o Skills* jointly removes skill packages and strategy references while retaining task inputs, repository tools, and factual API documentation. *w/o Execution Feedback* retains skills but returns neutral receipts for build and analyzer calls; actual outputs remain outside the agent workspace. Both use ▣ OpenCode with a 3,000-second active budget. *w/o Harness* makes one model call without tools or skills, using the full patch and deterministically extracted code context. The new variants share the model route, sampling settings, and 16,384-token output limit per response; one-shot generation has no feedback or repair call.

Endpoints and comparability. Artifact pass applies the compilation, differential, semantic, anti-hardcoding, and false-positive gates without requiring Process or normal agent termination. Frozen artifacts are evaluated even when generation reaches the time limit. The new runs retain their frozen ✳ Claude Opus 5 judgments for Process, Semantic, and task gates; a separate ✳ Opus 5 audit of all saved alerts supplies the reported $\mathrm{FP}_{\mathrm{pass}}$ without changing the original verdicts. Historical Full differs from the new runs in execution date, client context limit, and workspace/tool isolation, so the comparisons do not isolate causal effects of individual components. The one-shot result also reflects its strict output protocol: 237/300 requests yield invalid or incomplete structured submissions, and another 53 fail compilation.

The corresponding original Pass@1 differences are -32.00 $[-37.67, -26.33]$, -27.00 $[-33.33, -21.00]$, and -40.00 $[-45.67, -34.33]$ percentage points. These intervals describe variation over paired tasks for one generation per task; they do not cover model resampling, shared-CVE dependence, or differences between historical and new execution conditions.

Process/Semantic valid sample counts are 239/239 for Full, 119/122 for w/o Skills, 100/100 for w/o Execution Feedback, and 10/10 for w/o Harness. These conditional means are not averages over all 300 tasks. Three unavailable Process judgments are excluded instead of using heuristic fallback scores. Missing evidence is never treated as a pass. One failed candidate has seven false positives among ten sampled alerts, exceeding its 25% threshold.

✳ Opus 5 independently adjudicates every saved alert of the 25, 40, and 1 passed new checkers (16, 64, and 0 alerts, respectively). All 80 labels are resolved after source-evidence completion; the Full FP figure reuses the 83-alert ✳ Opus 5 audit from the main experiment. These post hoc labels define $\mathrm{FP}_{\mathrm{pass}}$, not the original sampled FP gates. The false-positive counts are 5/83 for Full, 1/16 for w/o Skills, and 16/64 for w/o Execution Feedback. One-shot has no saved alerts, so its N/A entry does not denote a zero false-positive proportion. No-alert checkers add nothing to the alert denominator. The metric is a false-discovery proportion over saved alerts, not a repository-wide false-positive rate.

Table 18: Interaction cost on tasks passed by both artifacts. Each pair uses its own common task subset; these means should not be compared across rows as if the subsets were identical.

Variant	Tasks	Full turns	Variant turns	Full tools	Variant tools
w/o Skills	42	49.29	64.02	63.21	89.24
w/o Execution Feedback	43	45.21	63.19	60.63	84.09
w/o Harness	3	22.00	1.00	39.33	0.00

Because the ablation variants have substantially lower pass rates and therefore fewer successful trajectories, Table 4 averages turns and tool calls over all 300 tasks, including failed trajectories. This differs from Table 2, which reports interaction metrics over successful trajectories only. Table 18 additionally restricts each comparison to tasks whose artifacts pass in both Full and the corresponding variant.

H.3 Time-Budget Sensitivity

Tables 19–21 provide the complete time-budget results for the analysis in Section 5.5. The tables cover all seven models and three harnesses at 10, 20, 30, 40, and 50 minutes of active generation time, using the completed-within-budget policy below.

Timing and eligibility. We retrospectively apply the cutoffs to existing trajectories, retaining each task's current evaluated attempt. Active generation time includes model responses and tool execution, but excludes recorded API and compute admission queues, maintenance pauses, and subsequent external verification and judging. Only final artifacts completed within a cutoff contribute their existing evaluation outcomes; unfinished runs count as unsuccessful for Pass@1, Build, and Diff. SR, with all 300 tasks retained in each denominator. Intermediate artifacts are not evaluated, and agents are not rerun with shorter deadlines. Realized execution time and trajectory length can differ from the budget because agents may terminate early, batch tool operations, or spend time waiting for responses and recovering from build failures.

Conditional metrics and model differences. Process and Semantic average available valid scores from eligible completed runs; Avg. Turns and Avg. Tool Calls average eligible passed runs. $\mathrm{FP}_{\mathrm{pass}}$ pools recorded alerts from those passed runs, using the same adjudicated labels as the main table. These statistics describe changing subsets of completed runs, not within-trajectory quality improvements. Averaged across harnesses, the 40-to-50-minute Pass@1 gains are largest for Qwen3.8-Max (3.89 percentage points) and HY4-Preview (3.67), compared with 0.22 for GPT-5.6 and 0.67 for Claude Opus 4.6. These differences describe when the retained trajectories finish successfully.

Table 19: Time-budget sensitivity on CheckerBench: OpenCode.

Model	Budget (min)	Pass@1 (%) ↑	Build (%) ↑	Diff. SR (%) ↑	$\mathrm{FP}_{\mathrm{pass}}$ (%) ↓	Process ↑	Semantic ↑	Avg. Turns ↓	Avg. Tool Calls ↓
GPT-5.6	10	10.67	29.33	29.00	25.00	80.35	7.57	23.59	54.16
	20	20.67	66.00	63.33	14.29	79.29	7.60	26.52	56.58
	30	25.00	88.00	79.33	9.09	78.45	7.47	27.80	57.75
	40	27.33	97.00	86.67	9.09	78.01	7.41	28.59	58.45
	50	27.67	98.67	88.33	9.09	78.01	7.41	28.55	58.54
Claude Opus 4.6	10	6.00	19.00	18.00	50.00	77.34	7.74	24.11	35.61
	20	15.00	57.33	55.67	25.00	76.91	7.21	30.13	44.22
	30	20.00	81.00	79.00	18.52	76.56	7.19	33.03	47.55
	40	22.33	92.67	90.00	20.00	76.35	7.20	35.61	50.45
	50	22.67	96.33	93.67	20.00	76.29	7.18	35.61	50.45
Qwen3.8-Max	10	6.33	7.67	7.67	—	92.15	9.22	24.11	31.68
	20	23.00	33.33	33.00	14.29	91.72	8.69	31.19	41.36
	30	31.00	50.33	50.00	25.53	91.16	8.50	33.75	44.53
	40	37.00	63.00	62.67	25.35	90.58	8.49	35.27	46.50
	50	39.33	69.33	69.00	24.66	90.15	8.46	36.31	47.84

Continued on next page

Table 19 continued

Model	Budget (min)	Pass@1 (%) ↑	Build (%) ↑	Diff. SR (%) ↑	FP_{pass} (%) ↓	Process ↑	Semantic ↑	Avg. Turns ↓	Avg. Tool Calls ↓
DeepSeek-V4-Pro	10	13.67	22.67	21.67	25.00	89.98	8.40	34.15	47.85
	20	26.33	55.67	54.33	13.04	88.05	8.18	39.52	52.86
	30	35.00	75.33	73.67	16.00	86.34	8.19	44.59	58.28
	40	39.33	87.67	85.67	6.58	85.32	8.16	49.41	63.56
	50	40.33	92.67	90.67	6.02	84.89	8.12	50.96	65.04
GLM-5.2	10	3.67	4.33	4.33	0.00	91.61	9.58	24.82	43.73
	20	18.33	34.33	33.00	25.53	88.84	8.20	33.87	53.18
	30	27.67	59.67	58.33	18.75	87.41	8.15	38.49	58.06
	40	32.00	73.00	71.00	20.00	86.90	8.12	40.28	59.76
	50	33.00	77.33	75.33	20.00	86.73	8.11	41.23	61.19
Kimi-K3	10	8.33	9.67	9.67	0.00	92.15	9.34	22.32	33.28
	20	23.00	40.33	40.00	11.11	90.99	8.53	26.55	37.20
	30	31.67	62.00	60.33	10.64	89.67	8.36	29.18	40.14
	40	37.67	76.00	74.33	7.69	88.90	8.34	31.46	42.47
	50	40.00	83.33	81.67	9.46	88.48	8.30	32.58	43.60
HY4-Preview	10	5.67	7.67	7.00	50.00	90.07	8.76	24.76	31.88
	20	20.00	34.33	33.67	25.00	89.70	8.40	32.85	41.00
	30	29.67	55.67	55.00	19.80	88.20	8.32	38.33	47.00
	40	33.33	65.33	64.67	20.00	87.54	8.31	40.18	49.08
	50	35.67	72.67	71.33	19.83	86.90	8.32	41.25	50.17

Budgets threshold existing trajectories by active generation time, excluding recorded resource-admission queues and maintenance pauses. External verification and judging are outside the generation clock. Only final artifacts completed by the cutoff contribute; unfinished runs contribute no success to Pass@1, Build, or Diff. SR, whose denominator remains 300. Process and Semantic average available valid scores among eligible completed runs. Turns and Tool Calls average only eligible passed runs. FP_{pass} is the pooled alert-level FP fraction for those passed runs, using the same adjudicated labels as the main table. A dash denotes no eligible score or a zero alert denominator. Per-cell sample sizes accompany the CSV. These are retrospective completion curves, not separately rerun deadline experiments.

Table 20: Time-budget sensitivity on CHECKERBENCH: Claude Code.

Model	Budget (min)	Pass@1 (%) ↑	Build (%) ↑	Diff. SR (%) ↑	FP_{pass} (%) ↓	Process ↑	Semantic ↑	Avg. Turns ↓	Avg. Tool Calls ↓
GPT-5.6	10	9.00	21.33	20.33	20.00	81.78	8.24	28.56	52.30
	20	18.33	56.00	50.67	4.76	80.96	7.79	30.91	53.82
	30	23.33	76.33	67.00	7.41	79.35	7.54	34.06	58.39
	40	26.67	89.67	77.33	6.67	78.95	7.56	35.38	60.71
	50	27.00	93.00	79.00	6.67	78.85	7.57	35.59	60.93
Claude Opus 4.6	10	8.33	17.67	17.67	0.00	82.15	7.92	17.64	30.40
	20	16.33	48.00	47.00	25.00	82.77	7.59	21.67	34.14
	30	21.00	66.33	65.33	22.22	83.01	7.38	25.19	37.92
	40	23.67	77.33	75.67	22.22	82.90	7.36	25.99	38.51
	50	24.00	83.00	81.00	22.22	82.51	7.38	26.39	38.85
Qwen3.8-Max	10	7.00	8.67	8.67	0.00	92.41	9.12	24.00	29.33
	20	24.00	39.33	39.00	26.67	90.96	8.50	28.25	34.31
	30	28.33	51.33	51.00	22.41	90.33	8.41	30.34	36.75
	40	32.33	61.00	60.67	20.00	89.86	8.35	32.30	39.41
	50	35.67	67.33	67.00	19.12	89.58	8.38	33.98	41.55
DeepSeek-V4-Pro	10	14.00	22.67	22.33	0.00	89.43	8.56	35.07	48.45
	20	24.00	49.33	49.00	4.76	87.26	8.29	39.26	52.99
	30	32.00	73.67	72.67	9.09	85.86	8.26	42.14	56.77
	40	37.67	86.00	84.33	7.58	85.61	8.29	45.53	60.65
	50	39.33	91.33	89.67	7.35	85.25	8.27	46.97	62.31
GLM-5.2	10	1.00	1.33	1.33	—	95.75	9.38	21.67	40.33
	20	13.67	23.00	22.33	26.67	93.37	8.89	30.85	42.12
	30	25.00	47.00	45.33	28.00	91.55	8.64	34.05	45.99
	40	31.00	62.67	61.00	18.42	90.55	8.48	36.60	47.87
	50	37.00	76.33	74.67	21.82	89.67	8.39	38.70	49.92
Kimi-K3	10	11.00	13.33	13.33	0.00	92.51	9.33	24.88	31.21
	20	23.67	47.67	47.00	46.00	90.11	8.29	27.35	33.54
	30	32.67	66.33	65.33	29.25	88.49	8.27	30.45	36.72
	40	38.33	78.33	77.00	28.57	88.04	8.29	31.77	38.14
	50	40.67	83.67	82.33	24.87	87.79	8.34	32.08	38.84

Continued on next page

Table 20 continued

Model	Budget (min)	Pass@1 (%) ↑	Build (%) ↑	Diff. SR (%) ↑	FP_{pass} (%) ↓	Process ↑	Semantic ↑	Avg. Turns ↓	Avg. Tool Calls ↓
HY4-Preview	10	2.00	2.33	2.33	0.00	91.99	9.14	24.00	32.33
	20	8.67	13.33	13.00	0.00	93.03	8.66	33.73	42.62
	30	16.67	29.67	29.00	0.00	91.86	8.63	42.48	53.24
	40	20.67	41.00	39.33	21.88	91.42	8.44	46.47	57.81
	50	24.67	49.33	47.33	20.00	90.83	8.35	50.05	61.66

Budgets threshold existing trajectories by active generation time, excluding recorded resource-admission queues and maintenance pauses. External verification and judging are outside the generation clock. Only final artifacts completed by the cutoff contribute; unfinished runs contribute no success to Pass@1, Build, or Diff. SR, whose denominator remains 300. Process and Semantic average available valid scores among eligible completed runs. Turns and Tool Calls average only eligible passed runs. FP_{pass} is the pooled alert-level FP fraction for those passed runs, using the same adjudicated labels as the main table. A dash denotes no eligible score or a zero alert denominator. Per-cell sample sizes accompany the CSV. These are retrospective completion curves, not separately rerun deadline experiments.

Table 21: Time-budget sensitivity on CHECKERBENCH: Hermes Agent.

Model	Budget (min)	Pass@1 (%) ↑	Build (%) ↑	Diff. SR (%) ↑	FP_{pass} (%) ↓	Process ↑	Semantic ↑	Avg. Turns ↓	Avg. Tool Calls ↓
Hermes Agent									
GPT-5.6	10	7.33	31.67	18.00	—	82.39	7.89	23.41	42.77
	20	13.33	65.33	37.33	0.00	79.46	7.61	24.45	44.25
	30	16.00	86.00	46.00	22.22	78.52	7.53	24.65	44.69
	40	16.33	94.00	49.33	22.22	78.13	7.48	24.53	44.27
	50	16.33	98.33	50.00	22.22	77.81	7.47	24.53	44.27
Claude Opus 4.6	10	6.33	18.00	17.33	0.00	81.43	7.81	20.26	29.47
	20	13.67	50.33	48.67	21.05	78.87	7.35	23.20	34.37
	30	18.33	73.67	70.67	34.88	78.83	7.22	28.89	39.98
	40	21.00	86.00	82.33	18.18	78.78	7.16	32.71	46.10
	50	22.33	97.33	93.00	18.18	78.30	7.13	34.19	48.34
Qwen3.8-Max	10	3.00	3.67	3.67	—	93.73	9.05	20.56	28.67
	20	11.33	15.00	14.67	59.09	93.15	9.07	30.65	40.59
	30	17.67	25.33	25.00	50.00	92.03	8.76	34.62	45.17
	40	22.00	34.33	34.00	45.71	91.35	8.62	37.09	48.09
	50	28.00	44.00	42.33	20.35	90.92	8.53	40.82	52.43
DeepSeek-V4-Pro	10	10.00	15.00	15.00	16.67	90.90	8.78	28.37	49.07
	20	24.00	49.33	48.00	34.78	88.11	8.34	33.83	55.78
	30	30.33	70.67	69.00	25.81	86.73	8.09	36.73	59.03
	40	34.00	85.00	82.33	41.46	85.93	8.07	38.04	60.66
	50	36.00	94.00	90.67	21.46	85.29	8.03	39.47	62.28
GLM-5.2	10	4.67	6.00	6.00	20.00	90.69	8.59	24.57	42.64
	20	14.00	28.00	27.33	12.50	87.49	8.27	33.81	53.69
	30	21.00	53.00	51.67	6.67	85.54	7.99	38.56	58.79
	40	24.67	70.00	68.67	4.35	83.98	7.90	41.09	61.15
	50	25.67	79.67	78.00	4.00	82.87	7.81	42.19	61.94
Kimi-K3	10	8.67	14.67	14.33	63.64	90.14	8.38	22.73	30.19
	20	19.33	41.00	39.33	27.55	87.42	8.30	26.84	35.90
	30	25.67	67.67	63.00	25.47	84.78	8.18	31.23	40.74
	40	27.00	78.67	72.67	25.47	83.07	8.06	31.69	41.28
	50	27.33	87.00	79.67	24.47	82.62	8.08	32.65	42.24
HY4-Preview	10	1.33	1.33	1.33	—	94.00	9.00	24.50	35.25
	20	8.00	12.00	12.00	0.00	92.19	8.91	31.71	43.00
	30	13.67	25.33	25.00	11.11	89.98	8.69	38.17	49.12
	40	19.67	38.33	37.00	50.00	90.34	8.53	45.27	56.54
	50	24.33	57.67	52.67	24.00	88.42	8.44	50.33	62.51

Budgets threshold existing trajectories by active generation time, excluding recorded resource-admission queues and maintenance pauses. External verification and judging are outside the generation clock. Only final artifacts completed by the cutoff contribute; unfinished runs contribute no success to Pass@1, Build, or Diff. SR, whose denominator remains 300. Process and Semantic average available valid scores among eligible completed runs. Turns and Tool Calls average only eligible passed runs. FP_{pass} is the pooled alert-level FP fraction for those passed runs, using the same adjudicated labels as the main table. A dash denotes no eligible score or a zero alert denominator. Per-cell sample sizes accompany the CSV. These are retrospective completion curves, not separately rerun deadline experiments.

H.4 LANGUAGE, REPOSITORY, AND CWE PERFORMANCE

These breakdowns average the same three independent repeats as Table 2 and Figure 6. Each model–harness pair has the same 300 tasks in each repeat. The main heatmaps and the repository weighting figure average the three harness-specific estimates equally; the tables below retain each harness. The abbreviated model names in the CWE tables follow the full names in Table 2. All rates are percentages and are descriptive; small task subsets do not establish reliable model rankings. The downloadable CSVs preserve full precision.

Languages and analyzers. C/C++ uses CSA (159 tasks); Python (51), Go (30), Java (30), and JavaScript (30) use CodeQL. Language and analyzer effects cannot be separated.

Table 22: Language-specific Pass@1 (%) by harness. All tasks in each language remain in the denominator.

Model	C/C++	Python	Go	Java	JavaScript
		OpenCode			
GPT-5.6	10.69	58.82	36.67	46.67	36.67
Claude Opus 4.6	14.47	41.18	26.67	26.67	26.67
Qwen3.8-Max	16.98	76.47	46.67	56.67	70.00
DeepSeek-V4-Pro	28.93	58.82	53.33	43.33	53.33
GLM-5.2	16.98	60.78	56.67	46.67	33.33
Kimi-K3	28.93	68.63	36.67	53.33	40.00
HY4-Preview	15.72	56.86	60.00	53.33	63.33
		Claude Code			
GPT-5.6	15.09	52.94	16.67	50.00	33.33
Claude Opus 4.6	16.98	50.98	30.00	16.67	16.67
Qwen3.8-Max	14.47	54.90	60.00	73.33	53.33
DeepSeek-V4-Pro	29.56	58.82	46.67	43.33	46.67
GLM-5.2	24.53	70.59	43.33	46.67	30.00
Kimi-K3	27.67	60.78	50.00	56.67	50.00
HY4-Preview	6.29	52.94	40.00	46.67	36.67
		Hermes Agent			
GPT-5.6	6.29	37.25	23.33	20.00	23.33
Claude Opus 4.6	17.61	41.18	16.67	16.67	26.67
Qwen3.8-Max	4.40	64.71	46.67	53.33	46.67
DeepSeek-V4-Pro	22.64	56.86	60.00	43.33	40.00
GLM-5.2	10.06	54.90	43.33	36.67	30.00
Kimi-K3	13.21	45.10	36.67	43.33	46.67
HY4-Preview	4.40	45.10	43.33	50.00	50.00

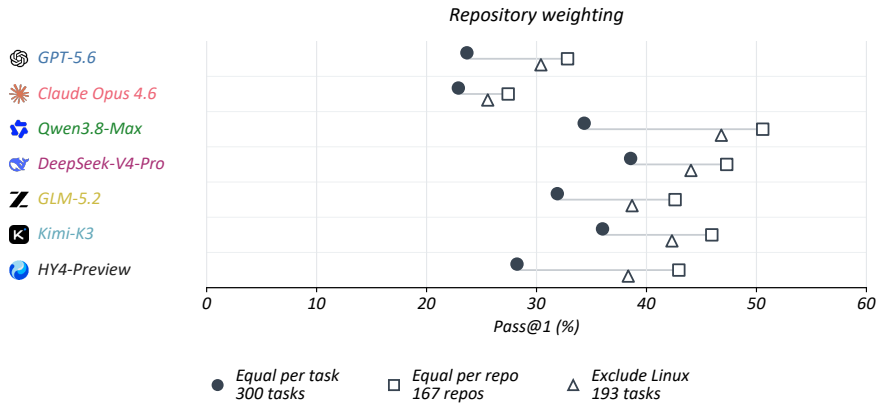

Figure 10: Pass@1 under three repository weightings. Each point averages the three harness-specific rates. Task equal covers 300 tasks; Repo equal averages 167 repositories; No Linux covers 193 tasks.

Repository composition. Linux contributes 107 tasks; 151 of the 167 repositories have only one. Repository-equal weighting estimates a different target from task-equal weighting. Excluding Linux also changes language, analyzer, and CWE composition; neither comparison isolates a causal repository effect. Individual repository counts and outcomes are retained in the companion CSV. Figure 10 compares all seven models. Qwen3.8-Max increases from 34.33% task-weighted Pass@1 to 50.57% repository-weighted Pass@1, overtaking DeepSeek-V4-Pro (38.56% and 47.29%, respectively). Excluding Linux gives 46.80% and 44.04% on the remaining 193 tasks. Thus the overall ordering depends on repository composition; these reweightings also change the influence of language, analyzer, and CWE strata.

Table 23: Repository sensitivity by harness (%). Task equal uses 300 tasks; Repo equal averages 167 repository rates; No Linux uses 193 tasks in 166 repositories.

Model	Task equal	Repo equal	No Linux
OpenCode			
GPT-5.6	27.67	38.90	35.75
Claude Opus 4.6	22.67	28.64	26.42
Qwen3.8-Max	39.33	55.84	51.81
DeepSeek-V4-Pro	40.33	49.92	47.15
GLM-5.2	33.00	44.23	39.90
Kimi-K3	40.00	49.13	45.60
HY4-Preview	35.67	50.56	45.60
Claude Code			
GPT-5.6	27.00	35.52	33.16
Claude Opus 4.6	24.00	29.06	26.42
Qwen3.8-Max	35.67	50.94	47.15
DeepSeek-V4-Pro	39.33	47.29	44.04
GLM-5.2	37.00	45.73	41.97
Kimi-K3	40.67	50.63	46.63
HY4-Preview	24.67	39.26	34.72
Hermes Agent			
GPT-5.6	16.33	24.04	22.28
Claude Opus 4.6	22.33	25.16	24.35
Qwen3.8-Max	28.00	44.93	41.45
DeepSeek-V4-Pro	36.00	44.67	40.93
GLM-5.2	25.67	37.84	34.20
Kimi-K3	27.33	38.06	34.72
HY4-Preview	24.33	39.01	34.72

Complete CWE breakdown. The following tables include all 85 explicit CWE labels and the 14 tasks without an explicit label (Other). A task contributes once to each distinct CWE label it carries; 14 tasks have two labels. Consequently the row counts sum to 314, not 300. Other does not pool known low-frequency CWEs. The main figure selects the eight explicit labels with at least ten tasks using counts alone, before examining performance. This display threshold is not a claim of statistical precision. CWE groups also differ in language, analyzer, and repository composition.

Table 24: Complete CWE-specific Pass@1 (%) for OpenCode. n counts distinct tasks, not harnesses or repeats.

CWE	n	GPT	Claude	Qwen	DeepSeek	GLM	Kimi	HY4
CWE-787	26	7.69	15.38	23.08	30.77	15.38	26.92	3.85
CWE-125	21	19.05	4.76	28.57	23.81	19.05	33.33	42.86
CWE-22	15	53.33	26.67	53.33	60.00	53.33	53.33	26.67
CWE-476	13	15.38	15.38	15.38	30.77	15.38	23.08	7.69
CWE-79	12	16.67	16.67	66.67	41.67	41.67	33.33	75.00
CWE-362	12	8.33	8.33	8.33	16.67	8.33	33.33	16.67
CWE-416	12	25.00	8.33	8.33	50.00	25.00	25.00	25.00
CWE-400	10	40.00	20.00	40.00	50.00	50.00	70.00	60.00
CWE-908	9	11.11	22.22	0.00	33.33	11.11	44.44	22.22
CWE-667	8	12.50	12.50	12.50	0.00	37.50	12.50	12.50
CWE-190	7	0.00	14.29	0.00	42.86	14.29	0.00	42.86
CWE-415	7	14.29	14.29	28.57	28.57	28.57	28.57	0.00
CWE-20	6	33.33	50.00	33.33	33.33	16.67	50.00	66.67
CWE-611	6	83.33	33.33	66.67	33.33	50.00	66.67	50.00
CWE-617	6	16.67	0.00	16.67	33.33	16.67	16.67	0.00
CWE-119	5	0.00	0.00	0.00	0.00	0.00	20.00	0.00
CWE-191	5	20.00	40.00	0.00	20.00	20.00	60.00	20.00
CWE-200	5	40.00	0.00	60.00	60.00	40.00	60.00	60.00
CWE-377	5	100.00	100.00	60.00	100.00	100.00	100.00	60.00
CWE-401	5	0.00	80.00	20.00	60.00	60.00	40.00	60.00
CWE-601	5	40.00	0.00	40.00	20.00	20.00	60.00	60.00
CWE-770	5	40.00	20.00	60.00	0.00	20.00	20.00	40.00
CWE-78	4	25.00	0.00	75.00	50.00	25.00	50.00	25.00
CWE-129	4	0.00	0.00	25.00	25.00	25.00	0.00	25.00
CWE-674	4	25.00	0.00	50.00	25.00	0.00	0.00	25.00
CWE-94	3	100.00	66.67	66.67	100.00	66.67	100.00	100.00
CWE-189	3	0.00	33.33	33.33	100.00	100.00	0.00	33.33
CWE-287	3	33.33	0.00	66.67	66.67	0.00	100.00	66.67
CWE-326	3	100.00	66.67	66.67	100.00	33.33	66.67	66.67
CWE-1333	3	66.67	100.00	66.67	66.67	66.67	66.67	33.33
CWE-23	2	50.00	0.00	100.00	50.00	50.00	50.00	100.00
CWE-59	2	50.00	100.00	100.00	100.00	100.00	50.00	50.00
CWE-74	2	50.00	50.00	100.00	50.00	50.00	100.00	50.00
CWE-77	2	50.00	50.00	100.00	100.00	100.00	50.00	100.00
CWE-284	2	50.00	50.00	50.00	50.00	50.00	50.00	0.00
CWE-338	2	50.00	0.00	50.00	50.00	100.00	50.00	100.00
CWE-352	2	50.00	50.00	100.00	50.00	0.00	100.00	100.00
CWE-502	2	100.00	100.00	100.00	100.00	50.00	100.00	100.00
CWE-532	2	50.00	0.00	100.00	0.00	50.00	50.00	50.00
CWE-672	2	0.00	50.00	50.00	100.00	50.00	100.00	50.00
CWE-772	2	0.00	0.00	100.00	50.00	0.00	50.00	0.00
CWE-835	2	0.00	0.00	0.00	0.00	0.00	50.00	0.00
CWE-863	2	50.00	0.00	100.00	50.00	100.00	50.00	50.00
CWE-29	1	100.00	0.00	100.00	100.00	0.00	0.00	0.00
CWE-88	1	0.00	0.00	100.00	100.00	100.00	100.00	0.00
CWE-91	1	100.00	0.00	100.00	100.00	100.00	100.00	100.00
CWE-120	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-122	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-131	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-193	1	0.00	0.00	100.00	0.00	0.00	0.00	0.00
CWE-201	1	0.00	0.00	0.00	0.00	100.00	0.00	100.00
CWE-203	1	0.00	0.00	100.00	0.00	0.00	0.00	0.00
CWE-209	1	100.00	100.00	100.00	100.00	100.00	100.00	100.00
CWE-250	1	0.00	0.00	100.00	0.00	100.00	0.00	0.00
CWE-256	1	0.00	100.00	100.00	0.00	100.00	100.00	0.00
CWE-264	1	100.00	0.00	0.00	100.00	0.00	100.00	100.00
CWE-266	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-295	1	100.00	100.00	100.00	100.00	100.00	100.00	100.00
CWE-312	1	0.00	100.00	100.00	0.00	100.00	100.00	0.00
CWE-327	1	100.00	100.00	0.00	100.00	100.00	100.00	100.00
CWE-331	1	0.00	0.00	100.00	0.00	0.00	0.00	0.00

Continued on next page

Table 24 (continued)

CWE	n	GPT	Claude	Qwen	DeepSeek	GLM	Kimi	HY4
CWE-347	1	100.00	0.00	100.00	100.00	100.00	0.00	100.00
CWE-364	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-366	1	0.00	100.00	0.00	100.00	100.00	0.00	0.00
CWE-369	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-399	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-407	1	0.00	100.00	100.00	100.00	100.00	0.00	0.00
CWE-459	1	0.00	0.00	0.00	0.00	0.00	100.00	0.00
CWE-460	1	0.00	0.00	100.00	0.00	0.00	100.00	0.00
CWE-522	1	100.00	0.00	100.00	0.00	100.00	0.00	100.00
CWE-538	1	0.00	0.00	100.00	0.00	100.00	0.00	100.00
CWE-552	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-613	1	0.00	100.00	100.00	0.00	0.00	0.00	100.00
CWE-639	1	100.00	0.00	100.00	0.00	0.00	0.00	0.00
CWE-668	1	100.00	100.00	100.00	100.00	100.00	100.00	100.00
CWE-670	1	0.00	0.00	100.00	0.00	100.00	0.00	0.00
CWE-680	1	0.00	100.00	0.00	0.00	0.00	0.00	0.00
CWE-692	1	0.00	0.00	0.00	100.00	0.00	0.00	0.00
CWE-789	1	0.00	100.00	100.00	100.00	100.00	0.00	100.00
CWE-826	1	100.00	0.00	0.00	100.00	0.00	100.00	100.00
CWE-843	1	0.00	100.00	0.00	100.00	100.00	100.00	0.00
CWE-862	1	100.00	0.00	100.00	0.00	100.00	100.00	0.00
CWE-918	1	0.00	100.00	100.00	100.00	0.00	0.00	100.00
CWE-1188	1	0.00	0.00	0.00	100.00	0.00	0.00	100.00
CWE-1336	1	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Other	14	28.57	21.43	42.86	50.00	42.86	57.14	57.14

Table 25: Complete CWE-specific Pass@1 (%) for Claude Code. n counts distinct tasks, not harnesses or repeats.

CWE	n	GPT	Claude	Qwen	DeepSeek	GLM	Kimi	HY4
CWE-787	26	11.54	23.08	26.92	30.77	30.77	23.08	3.85
CWE-125	21	14.29	9.52	33.33	28.57	28.57	42.86	4.76
CWE-22	15	46.67	33.33	60.00	40.00	53.33	60.00	40.00
CWE-476	13	30.77	15.38	7.69	46.15	15.38	15.38	0.00
CWE-79	12	25.00	8.33	58.33	41.67	33.33	33.33	33.33
CWE-362	12	16.67	8.33	25.00	25.00	25.00	25.00	16.67
CWE-416	12	16.67	25.00	25.00	25.00	25.00	25.00	0.00
CWE-400	10	40.00	50.00	40.00	40.00	50.00	40.00	50.00
CWE-908	9	22.22	33.33	11.11	33.33	33.33	33.33	0.00
CWE-667	8	12.50	12.50	0.00	37.50	37.50	25.00	12.50
CWE-190	7	0.00	0.00	14.29	14.29	14.29	0.00	0.00
CWE-415	7	14.29	14.29	28.57	14.29	28.57	14.29	14.29
CWE-20	6	50.00	16.67	66.67	16.67	16.67	33.33	50.00
CWE-611	6	66.67	50.00	100.00	50.00	83.33	83.33	83.33
CWE-617	6	33.33	0.00	33.33	16.67	16.67	16.67	0.00
CWE-119	5	20.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-191	5	40.00	0.00	0.00	40.00	20.00	60.00	20.00
CWE-200	5	20.00	20.00	40.00	0.00	40.00	40.00	40.00
CWE-377	5	100.00	100.00	60.00	100.00	80.00	100.00	80.00
CWE-401	5	20.00	80.00	20.00	60.00	40.00	40.00	20.00
CWE-601	5	40.00	40.00	40.00	60.00	40.00	40.00	60.00
CWE-770	5	0.00	60.00	20.00	20.00	20.00	20.00	20.00
CWE-78	4	0.00	25.00	50.00	75.00	50.00	75.00	25.00
CWE-129	4	0.00	0.00	25.00	50.00	50.00	50.00	25.00
CWE-674	4	25.00	0.00	25.00	25.00	0.00	75.00	25.00
CWE-94	3	100.00	66.67	100.00	100.00	66.67	66.67	66.67
CWE-189	3	33.33	66.67	66.67	0.00	66.67	0.00	33.33
CWE-287	3	33.33	0.00	33.33	33.33	33.33	33.33	66.67
CWE-326	3	33.33	33.33	66.67	100.00	66.67	66.67	0.00
CWE-1333	3	33.33	33.33	33.33	33.33	66.67	33.33	66.67
CWE-23	2	50.00	50.00	50.00	50.00	0.00	50.00	0.00
CWE-59	2	50.00	100.00	50.00	100.00	50.00	100.00	0.00
CWE-74	2	50.00	0.00	100.00	50.00	50.00	50.00	50.00
CWE-77	2	100.00	50.00	100.00	50.00	50.00	100.00	100.00
CWE-284	2	50.00	50.00	50.00	50.00	0.00	50.00	0.00
CWE-338	2	0.00	0.00	0.00	50.00	0.00	50.00	50.00
CWE-352	2	50.00	0.00	0.00	100.00	50.00	100.00	0.00
CWE-502	2	50.00	100.00	50.00	100.00	100.00	50.00	100.00
CWE-532	2	50.00	0.00	50.00	50.00	50.00	50.00	50.00
CWE-672	2	0.00	50.00	50.00	0.00	50.00	0.00	0.00

Continued on next page

Table 25 (continued)

CWE	n	GPT	Claude	Qwen	DeepSeek	GLM	Kimi	HY4
CWE-772	2	50.00	0.00	50.00	0.00	0.00	100.00	0.00
CWE-835	2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-863	2	50.00	0.00	100.00	50.00	100.00	50.00	100.00
CWE-29	1	0.00	0.00	0.00	100.00	100.00	100.00	100.00
CWE-88	1	0.00	0.00	0.00	100.00	0.00	100.00	0.00
CWE-91	1	100.00	0.00	100.00	100.00	100.00	100.00	100.00
CWE-120	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-122	1	0.00	0.00	0.00	0.00	100.00	0.00	0.00
CWE-131	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-193	1	0.00	0.00	100.00	100.00	100.00	0.00	0.00
CWE-201	1	0.00	100.00	0.00	0.00	0.00	0.00	0.00
CWE-203	1	0.00	0.00	0.00	0.00	0.00	0.00	100.00
CWE-209	1	100.00	0.00	100.00	0.00	100.00	100.00	100.00
CWE-250	1	0.00	0.00	100.00	100.00	100.00	100.00	0.00
CWE-256	1	0.00	0.00	100.00	0.00	0.00	0.00	0.00
CWE-264	1	100.00	0.00	0.00	100.00	100.00	100.00	100.00
CWE-266	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-295	1	100.00	100.00	100.00	100.00	100.00	100.00	100.00
CWE-312	1	0.00	0.00	100.00	0.00	0.00	0.00	0.00
CWE-327	1	0.00	100.00	0.00	0.00	100.00	100.00	0.00
CWE-331	1	0.00	0.00	100.00	0.00	0.00	0.00	0.00
CWE-347	1	0.00	0.00	0.00	0.00	100.00	100.00	100.00
CWE-364	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-366	1	100.00	0.00	0.00	0.00	100.00	0.00	100.00
CWE-369	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-399	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-407	1	100.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-459	1	0.00	0.00	0.00	0.00	100.00	100.00	0.00
CWE-460	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-522	1	0.00	100.00	100.00	100.00	0.00	100.00	0.00
CWE-538	1	0.00	0.00	100.00	0.00	100.00	0.00	100.00
CWE-552	1	0.00	0.00	0.00	0.00	100.00	100.00	0.00
CWE-613	1	0.00	0.00	100.00	100.00	0.00	100.00	100.00
CWE-639	1	0.00	0.00	0.00	100.00	0.00	100.00	0.00
CWE-668	1	100.00	100.00	100.00	100.00	100.00	100.00	100.00
CWE-670	1	0.00	0.00	0.00	100.00	100.00	0.00	0.00
CWE-680	1	0.00	0.00	0.00	0.00	100.00	100.00	0.00
CWE-692	1	0.00	0.00	100.00	0.00	0.00	0.00	0.00
CWE-789	1	0.00	0.00	100.00	100.00	100.00	100.00	100.00
CWE-826	1	0.00	100.00	0.00	0.00	0.00	100.00	0.00
CWE-843	1	0.00	0.00	0.00	100.00	0.00	0.00	0.00
CWE-862	1	100.00	100.00	100.00	100.00	0.00	100.00	0.00
CWE-918	1	0.00	0.00	100.00	100.00	0.00	100.00	100.00
CWE-1188	1	100.00	0.00	100.00	100.00	0.00	0.00	0.00
CWE-1336	1	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Other	14	21.43	35.71	35.71	71.43	50.00	50.00	21.43

Table 26: Complete CWE-specific Pass@1 (%) for Hermes Agent. n counts distinct tasks, not harnesses or repeats.

CWE	n	GPT	Claude	Qwen	DeepSeek	GLM	Kimi	HY4
CWE-787	26	3.85	15.38	19.23	34.62	11.54	11.54	0.00
CWE-125	21	14.29	9.52	23.81	14.29	4.76	14.29	14.29
CWE-22	15	26.67	20.00	46.67	46.67	40.00	40.00	33.33
CWE-476	13	0.00	7.69	0.00	15.38	7.69	30.77	7.69
CWE-79	12	25.00	8.33	75.00	33.33	33.33	25.00	58.33
CWE-362	12	8.33	0.00	25.00	8.33	25.00	8.33	0.00
CWE-416	12	16.67	16.67	0.00	25.00	25.00	16.67	0.00
CWE-400	10	40.00	50.00	30.00	20.00	30.00	30.00	30.00
CWE-908	9	0.00	44.44	0.00	33.33	11.11	22.22	11.11
CWE-667	8	0.00	0.00	0.00	12.50	12.50	0.00	0.00
CWE-190	7	0.00	14.29	0.00	14.29	0.00	0.00	14.29
CWE-415	7	14.29	28.57	0.00	28.57	0.00	14.29	0.00
CWE-20	6	16.67	33.33	33.33	33.33	50.00	16.67	33.33
CWE-611	6	16.67	33.33	50.00	66.67	50.00	66.67	66.67
CWE-617	6	0.00	0.00	0.00	0.00	0.00	16.67	16.67
CWE-119	5	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-191	5	0.00	80.00	0.00	60.00	20.00	40.00	20.00
CWE-200	5	20.00	20.00	60.00	60.00	20.00	20.00	20.00
CWE-377	5	100.00	80.00	100.00	100.00	100.00	100.00	80.00

Continued on next page

Table 26 (continued)

CWE	n	GPT	Claude	Qwen	DeepSeek	GLM	Kimi	HY4
CWE-401	5	0.00	60.00	0.00	80.00	20.00	20.00	20.00
CWE-601	5	20.00	20.00	20.00	20.00	20.00	0.00	40.00
CWE-770	5	0.00	20.00	20.00	40.00	0.00	20.00	20.00
CWE-78	4	25.00	25.00	75.00	50.00	75.00	25.00	75.00
CWE-129	4	0.00	0.00	0.00	25.00	25.00	0.00	0.00
CWE-674	4	0.00	25.00	25.00	50.00	50.00	0.00	25.00
CWE-94	3	100.00	66.67	33.33	100.00	66.67	33.33	66.67
CWE-189	3	33.33	0.00	0.00	100.00	33.33	0.00	33.33
CWE-287	3	0.00	33.33	33.33	66.67	33.33	33.33	66.67
CWE-326	3	33.33	100.00	66.67	66.67	66.67	100.00	33.33
CWE-1333	3	33.33	66.67	66.67	66.67	100.00	66.67	33.33
CWE-23	2	50.00	50.00	50.00	100.00	50.00	0.00	0.00
CWE-59	2	50.00	100.00	100.00	100.00	100.00	100.00	50.00
CWE-74	2	50.00	0.00	100.00	50.00	50.00	50.00	100.00
CWE-77	2	100.00	50.00	100.00	100.00	50.00	100.00	100.00
CWE-284	2	50.00	50.00	0.00	50.00	50.00	50.00	0.00
CWE-338	2	0.00	0.00	50.00	50.00	50.00	50.00	50.00
CWE-352	2	0.00	50.00	0.00	0.00	50.00	100.00	50.00
CWE-502	2	100.00	50.00	100.00	100.00	100.00	100.00	50.00
CWE-532	2	0.00	50.00	50.00	50.00	100.00	50.00	100.00
CWE-672	2	0.00	0.00	0.00	50.00	50.00	0.00	0.00
CWE-772	2	0.00	0.00	50.00	0.00	0.00	50.00	0.00
CWE-835	2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-863	2	0.00	0.00	100.00	50.00	50.00	50.00	50.00
CWE-29	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-88	1	0.00	0.00	100.00	0.00	100.00	0.00	100.00
CWE-91	1	100.00	100.00	100.00	100.00	100.00	0.00	0.00
CWE-120	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-122	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-131	1	100.00	100.00	0.00	100.00	0.00	0.00	0.00
CWE-193	1	0.00	100.00	0.00	0.00	0.00	0.00	0.00
CWE-201	1	0.00	0.00	0.00	0.00	100.00	0.00	0.00
CWE-203	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-209	1	0.00	0.00	100.00	100.00	0.00	100.00	100.00
CWE-250	1	0.00	0.00	100.00	100.00	0.00	0.00	0.00
CWE-256	1	0.00	0.00	0.00	0.00	100.00	0.00	0.00
CWE-264	1	0.00	0.00	0.00	100.00	0.00	100.00	0.00
CWE-266	1	0.00	0.00	0.00	0.00	0.00	100.00	0.00
CWE-295	1	100.00	100.00	100.00	100.00	100.00	100.00	100.00
CWE-312	1	0.00	0.00	0.00	0.00	100.00	0.00	0.00
CWE-327	1	100.00	0.00	0.00	100.00	0.00	0.00	100.00
CWE-331	1	0.00	0.00	100.00	100.00	0.00	0.00	0.00
CWE-347	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-364	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-366	1	0.00	0.00	100.00	0.00	0.00	100.00	0.00
CWE-369	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-399	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-407	1	0.00	0.00	0.00	100.00	0.00	100.00	100.00
CWE-459	1	0.00	0.00	0.00	100.00	0.00	0.00	0.00
CWE-460	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-522	1	100.00	0.00	100.00	0.00	0.00	100.00	0.00
CWE-538	1	0.00	0.00	100.00	100.00	0.00	100.00	100.00
CWE-552	1	0.00	0.00	0.00	0.00	0.00	100.00	100.00
CWE-613	1	0.00	0.00	0.00	100.00	100.00	0.00	0.00
CWE-639	1	0.00	0.00	0.00	0.00	0.00	100.00	100.00
CWE-668	1	100.00	100.00	100.00	100.00	100.00	100.00	100.00
CWE-670	1	0.00	0.00	0.00	0.00	0.00	0.00	100.00
CWE-680	1	0.00	0.00	100.00	100.00	100.00	0.00	0.00
CWE-692	1	0.00	0.00	0.00	0.00	0.00	0.00	100.00
CWE-789	1	100.00	0.00	100.00	100.00	0.00	100.00	0.00
CWE-826	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CWE-843	1	0.00	0.00	0.00	100.00	0.00	0.00	0.00
CWE-862	1	0.00	0.00	0.00	0.00	0.00	0.00	100.00
CWE-918	1	0.00	0.00	0.00	0.00	0.00	100.00	0.00
CWE-1188	1	0.00	0.00	0.00	0.00	0.00	100.00	0.00
CWE-1336	1	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Other	14	14.29	28.57	35.71	35.71	28.57	21.43	28.57

Interaction turns. Figure 6(c) compares all 21 combinations. Pass@1 uses all 300 tasks; Avg. Turns uses passed trajectories, including subagents. Different task sets do not establish a causal effect of more turns. Passed-trajectory counts appear in the companion summary.