



CWE-bench: Refocusing Agentic Security from Exploitation to Defense

Gonzalo Gonzalez-Pumariiega Ajay Jagannath Adit Jain Guram Gogia
Soumyadeep Bakshi Nazneen Rajani

Cybersecurity evaluations of AI agents have focused largely on offense, yet a working exploit does not show that an agent can make software safe. An attacker needs to exploit only one vulnerability; a defender must find and fix all of them. We introduce CWE-bench, a private benchmark of 120 held-out repository audit-and-repair tasks spanning 73 CWE types, eight languages, and all OWASP Top 10 2025 categories. Agents are given a repository to audit and a description of its threat model, but are not told the weakness, where it occurs, or how many vulnerable sites there are. Each patch must block the exploit while preserving original functionality, as checked by a programmatic verifier. We validated the verifier against a panel of agentic judges, manually reviewing disagreements during development and refining the checks. The best agents reach 68% pass@1 and 81% pass@4. We further conduct analyses showing that (i) agents with the same pass@1 differ widely in retry gains and cost, and (ii) no single agent is best across languages or vulnerability categories, and difficulty varies far more by vulnerability category than by language. As the offensive capabilities of agents improve, CWE-bench provides a broad and rigorous measure of whether their defensive capabilities keep pace.

Correspondence: research@collinear.ai

Date: October 14, 2026

1 Introduction

Can an agent make a vulnerable codebase safe to use? Most cybersecurity evaluations of AI agents measure offense: solving capture-the-flag challenges, exploiting applications, or reproducing vulnerabilities (Zhang et al., 2025; Zhu et al., 2025; WANG et al., 2026). Defense has received far less attention, even though it is the harder problem. An attacker needs to exploit only one vulnerability; a defender must find and fix all of them without breaking the software. Consider a file server with a path-traversal flaw. An attacker needs just one handler that builds a file path from user input to read files outside the shared directory. A defender must find and fix every such handler, and cannot simply disable downloads, since that removes the service the server exists to provide.

A growing number of benchmarks evaluate defensive agents, but each simplifies the problem in a different way. Many build tasks directly on public Common Vulnerabilities and Exposures (CVEs), whose fixes may already be in training data, or on LLM-generated code bases that fall outside the distribution of real software (Wei et al., 2025; Luo et al., 2026a; Bui et al., 2026; Luo et al., 2026b). Others give the agent a vulnerability description, proof-of-concept input, or location hints, reducing discovery to repair of a known flaw (Wei et al., 2025; Shen et al., 2026; Bui et al., 2026). The rest cover only a narrow slice of weaknesses, such as memory-safety bugs in C/C++ (Shi et al., 2026; Shen et al., 2026; Lau et al., 2026). A real defender faces none of these simplifications: it must find unknown vulnerabilities across the full range of weaknesses in real code.

We introduce **CWE-bench**, a benchmark of 120 held-out repository auditing and repair tasks that addresses each of these gaps. Every task is built on a real open-source repository with injected vulnerabilities, and the evaluation set is kept private, so the injected vulnerabilities have no published fix. Agents receive only the repository and an audit instruction describing the system and its threat model; they are not told the weakness, where it occurs, or how many vulnerable sites there are. Tasks span 73 Common Weakness Enumeration (CWE) types, eight programming languages, and all ten OWASP Top 10:2025 categories. A patch passes only if it blocks the exploits while preserving the original functionality.

Benchmark	Tasks	CWEs	OWASP 2025	Lang.	Codebase	Vuln. source	Information given to agent
PatchEval (Wei et al., 2025)	230	39	7/10	3	Real	Public CVEs	Vulnerability description
CyberGym-E2E [†] (Shi et al., 2026)	920	NR	1/10	2	Real	Public OSS-Fuzz bugs	None
LiveCVEBench (Luo et al., 2026a)	190	74	9/10	14	Real	Public CVEs	Bug report
ZeroDayBench [†] (Lau et al., 2026)	22	15	4/10	6	Real	Injected	Varies (5 levels)
PatchBench (Shen et al., 2026)	213	16	1/10	2	Real	Injected	PoC and sanitizer report
Vul4Py (Bui et al., 2026)	100	60	7/10	1	Real	Public CVEs	CVE and file/line hints
SecProbe [†] (Luo et al., 2026b)	353	151	5/10	6	LLM-generated	Injected	Software specification
CWE-bench	120	73	10/10	8	Real	Injected	Threat model

Table 1 Comparison of recent vulnerability repair benchmarks. *OWASP 2025* counts the OWASP Top 10:2025 categories covered under each category’s official CWE mapping. *Codebase* indicates whether tasks are built on real open-source repositories or LLM-generated ones. *Injected* denotes vulnerabilities inserted into a codebase, such as real CVEs transplanted into new repositories. C/C++ and JavaScript/TypeScript count as separate languages. PatchEval counts its 230-task sandboxed subset. NR: not reported. [†]Some values (in italics) are derived by us: from source-CVE records (ZeroDayBench), sanitizer crash types (CyberGym-E2E), or the 20 most frequent CWEs reported (SecProbe, a lower bound).

To ensure the correctness of our verifiers, we compared their verdicts with those of a panel of three agentic judges and, during development, manually reviewed every rollout where the two disagreed. Table 1 compares CWE-bench with recent defensive benchmarks.

The best agents reach 68% pass@1 and 81% pass@4. Beyond these aggregate scores, agents with the same pass@1 differ widely in how much they gain from retries and in cost per attempt, and their ordering changes across languages. Difficulty varies far more across vulnerability categories than across languages, and no single agent leads in either.

The paper makes three contributions:

1. **CWE-bench**, a private benchmark of 120 auditing and repair tasks on real repositories, covering all ten OWASP Top 10:2025 categories with no diagnosis given to the agent.
2. **A validated grading procedure** that checks both exploit blocking and preserved functionality, confirmed against a panel of agentic judges.
3. **An evaluation of eleven frontier agents**, showing that equal pass@1 hides large differences in retry gains, task coverage, and cost, and that no single agent is best across languages or vulnerability categories.

2 Related Work

Offensive evaluation. Offensive success is self-certifying: a captured flag, a triggered crash, or a working exploit is proof on its own. This has let offensive evaluation scale quickly, from capture-the-flag challenges (Zhang et al., 2025) to exploiting real web applications (Zhu et al., 2025), reproducing real vulnerabilities (WANG et al., 2026), achieving consequential attacks (Wang et al., 2026a), and grading intermediate exploitation steps (Lee and Brumley, 2026), with multi-stage penetration tests now separating tooling gaps from planning failures (Deng et al., 2026). Defensive success is not self-certifying. Blocking the known exploit is necessary but not sufficient, since a patch can stop one payload while leaving variants open, or disable the feature entirely. Offensive benchmarks therefore show whether agents can expose weaknesses, but not whether they can remove them.

Defensive auditing and repair. Defensive benchmarks work around the grading problem through shortcuts that limit what they measure. First, many build tasks on public CVEs and grade against the developer’s own fix and tests (Lee et al., 2025; Wei et al., 2025; Luo et al., 2026a; Bui et al., 2026). Those fixes are published and may appear in training data, so a correct patch can reflect recall rather than investigation. Generating the repository along with its tests avoids contamination (Luo et al., 2026b), but produces code outside the distribution of real software. Second, many give the agent a vulnerability description, proof-of-concept input, or location hints (Wei et al., 2025; Shen et al., 2026; Bui et al., 2026; Luo et al., 2026a). The agent then only repairs a known flaw and skips auditing, a large part of real defensive work. Third, many cover only a narrow range of weaknesses (Shi et al., 2026; Shen et al., 2026; Lau et al., 2026). Several are limited to memory-safety bugs in C/C++, where sanitizers give an automatic check, leaving access-control, authentication, and logic flaws untested; these reach only one to four of the ten OWASP Top 10:2025 categories (Table 1).

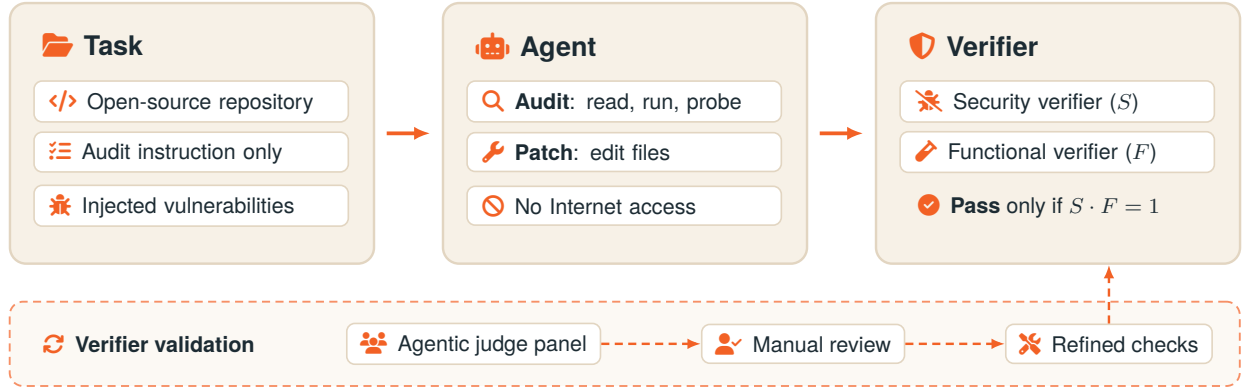


Figure 1 CWE-bench evaluation. Each task provides an open-source repository with injected vulnerabilities and an audit instruction that does not name them. The agent audits and patches the repository without Internet access. A patch passes only if both the security verifier (S) and the functional verifier (F) succeed. The verifiers are validated against a panel of agentic judges, with disagreements reviewed during development to refine the checks.

CWE-bench addresses all three: real repositories with injected vulnerabilities, no diagnosis given to the agent, and coverage of all ten OWASP Top 10:2025 categories.

Evaluation validity and integrity. Even with executable checks, defensive grading can go wrong in three ways. First, test-passing patches can still be incorrect (Qi et al., 2015). Prior work strengthens validation with fuzzing and differential checks, maintainer-added tests, or semantic checks (Byun et al., 2025; Yu et al., 2026; Shen et al., 2026), and finds errors in proof-of-concept grading itself (Lee et al., 2026). Second, LLM judges are not a reliable substitute: they can agree with one another while disagreeing with execution-based checks (Deininger and Slany, 2026). Third, agents can game the evaluation by retrieving published fixes or exploiting weaknesses in the evaluation harness (Kouremetis et al., 2026; Wang et al., 2026b; Zheng et al., 2026). CWE-bench combines these lessons: a programmatic verifier provides the score, a panel of agentic judges checks it, and disagreements found during development are manually reviewed, while agents can reach only their model API during evaluation.

3 Benchmark Design

3.1 Problem formulation

We formulate each task as a Markov decision process $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, R \rangle$. A state $s_t \in \mathcal{S}$ consists of the repository, the audit instruction, and the interaction history up to step t , where the instruction does not name the vulnerabilities or their number. Actions $a_t \in \mathcal{A}$ are the operations available through the agent’s harness, such as inspecting code, running commands, and editing files. The transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ applies an action’s effect on the repository and appends its output to the history; actions that attempt to access the Internet are blocked and leave the repository unchanged. The agent acts according to a policy $\pi(a_t | s_t)$. When the agent submits or reaches the one-hour time limit at step T , the reward $R : \mathcal{S} \rightarrow \{0, 1\}$ is computed on the final state, so a timed-out rollout is graded on whatever repairs it has made:

$$R(s_T) = S(s_T) \cdot F(s_T), \quad (1)$$

where $S(s_T) = 1$ if the exploits for the target vulnerabilities are blocked, and $F(s_T) = 1$ if the repository’s existing functionality tests still pass. A rollout passes only if both checks succeed.

3.2 Task coverage and construction

The evaluation set contains 120 private held-out tasks spanning 73 CWE types, eight programming languages, and all ten OWASP Top 10:2025 categories. Figure 2 breaks the set down by language and by OWASP category; each category contains 10 to 14 tasks.

Each task starts from an open-source repository with a real CVE, into which we inject new vulnerabilities; these vulnerabilities have no published fix and are kept private, so they cannot appear in training data. For each task, we write

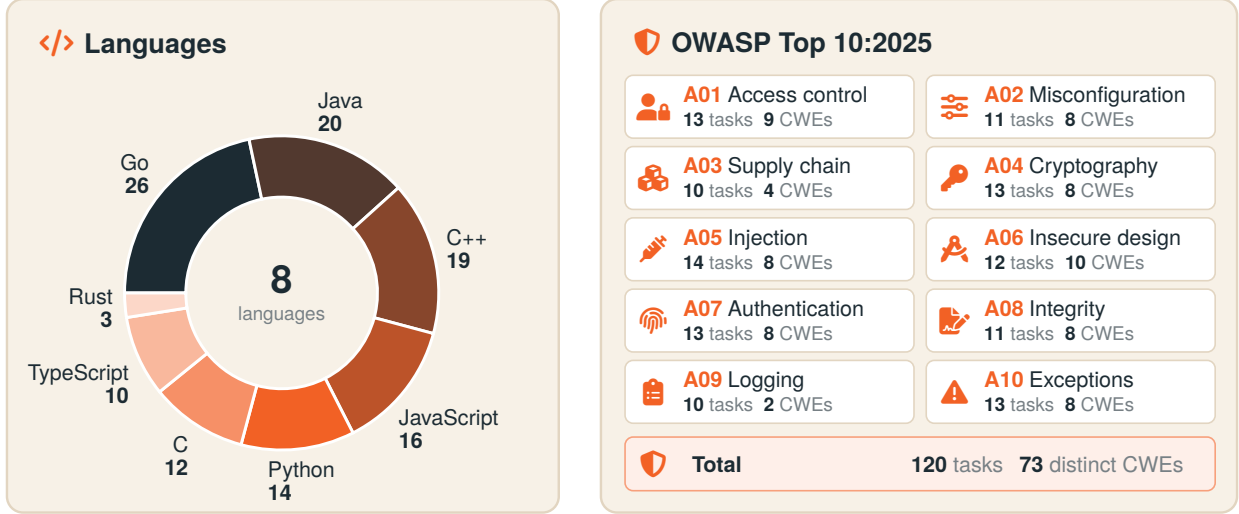


Figure 2 Composition of the 120-task evaluation set. **Left:** tasks per programming language. **Right:** tasks and distinct CWEs per OWASP Top 10:2025 category, with abbreviated category names.

exploits that trigger the injected vulnerabilities and functionality tests that exercise the repository’s intended behavior; these form the checks S and F in Equation 1. A task is accepted only if a reference patch passes both checks.

3.3 Verifier validation

We validate the programmatic verifier against a panel of three agentic judges (Appendix B). Each judge works in a separate copy of the patched repository, where it can inspect code, build the project, and run the verifier, and accepts a patch only if it satisfies five criteria: security, functionality, robustness to attack variants, grading integrity, and build soundness. The panel accepts by majority vote. During verifier development, we manually reviewed every rollout where the panel and verifier disagreed (Appendix D). These reviews led us to add probes that catch incomplete repairs, to replace checks that assumed the reference patch’s structure with checks of behavior, and to correct judge rubrics where the judges erred. Table 2 reports programmatic and panel scores separately.

4 Experiments

4.1 Experimental Setup

Agents. We evaluate eleven frontier models, each run in a coding-agent harness: Claude Code, Codex, Antigravity, or OpenCode (Table 2). Scores therefore reflect the model and its harness together. Each agent uses high reasoning effort and makes four independent attempts per task, with a one-hour limit per attempt.

Execution. Each attempt runs in an isolated container where the agent can reach only its model API. Attempts that end in a provider refusal count as failures.

Metrics. Each attempt is scored by the programmatic verifier (Equation 1), and we report the verdicts of the agentic judge panel (Section 3.3) as a second score. We report pass@1, the mean fraction of passing attempts per task, and pass@4, the fraction of tasks with at least one passing attempt (Chen et al., 2021), along with the mean API cost per rollout.

4.2 Main Results

Table 2 reports results for all eleven agents. Grok 4.7, Gemini 4 Argon, and GPT-6 Astra reach the highest pass@1 at 68%, with Claude Opus 5.5 at 67%, so pass@1 alone does not separate the leading agents. With four attempts, Grok 4.7 reaches the highest pass@4 at 81%. The same four agents lead under both programmatic and panel scores, and the two graders agree on 93.9% of individual attempts (Appendix C).

System	Harness	Programmatic		Judge panel		Mean cost (USD)
		Pass@1	Pass@4	Pass@1	Pass@4	
Grok 4.7	OpenCode	68%	81%	64%	82%	2.75
Gemini 4 Argon	Antigravity	68%	75%	62%	73%	6.63
GPT-6 Astra	Codex	68%	74%	66%	77%	2.85
Claude Opus 5.5	Claude Code	67%	79%	67%	81%	0.79
Claude Fable 5.1	Claude Code	58%	73%	60%	77%	3.47
Grok 4.6	OpenCode	57%	72%	55%	73%	1.09
DeepSeek V4.1 Flash	OpenCode	55%	74%	53%	75%	0.09
Muse Spark 1.3	OpenCode	55%	69%	51%	68%	3.95
Hy4 Preview	OpenCode	53%	75%	50%	73%	0.47
GPT-6 Sol	Codex	52%	68%	52%	72%	0.96
Inkling	OpenCode	37%	53%	33%	49%	3.00

Table 2 Programmatic and judge-panel pass@1 and pass@4, with mean API cost per rollout. Scores are rounded percentages.

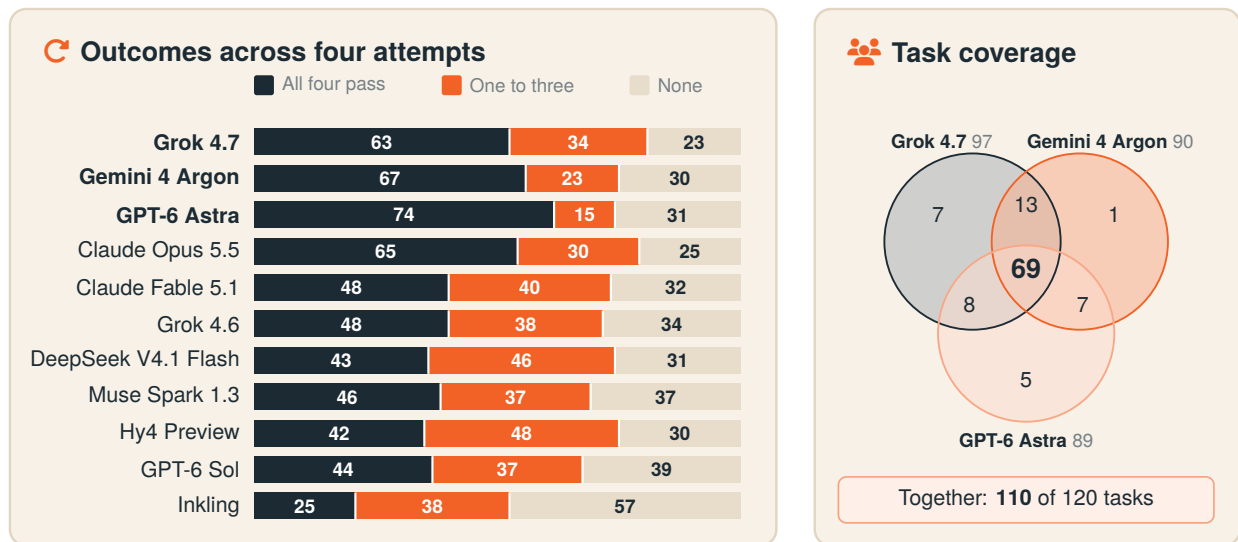


Figure 3 Outcomes across four attempts and task coverage. **Left:** for each agent, the number of tasks passed on all four attempts, on one to three, and on none; agents tied at 68% pass@1 are in bold. **Right:** tasks passed at least once by each of the three tied agents, and their overlap.

4.3 How do repeated attempts and different agents expand task coverage?

Grok 4.7, Gemini 4 Argon, and GPT-6 Astra share 68% pass@1, yet reach 81%, 75%, and 74% pass@4 (Table 2). The difference lies in how consistently each agent repairs the tasks it can solve (Figure 3, left). GPT-6 Astra passes all four attempts on 74 tasks and has mixed outcomes on only 15, so additional attempts add little. Grok 4.7 passes all four attempts on fewer tasks (63) but has mixed outcomes on 34, which retries convert into passes.

The three agents also solve different tasks (Figure 3, right). Grok 4.7, Gemini 4 Argon, and GPT-6 Astra pass at least one attempt on 97, 90, and 89 tasks, respectively, but only 69 are common to all three, and together they pass 110 of the 120. Grok 4.7 passes 20 tasks that GPT-6 Astra never does, while GPT-6 Astra passes 12 that Grok 4.7 never does.

4.4 How does pass rate scale with inference cost?

Grok 4.7, GPT-6 Astra, and Gemini 4 Argon reach the same 68% pass@1 at mean costs of \$2.75, \$2.85, and \$6.63 per rollout (Figure 4). Claude Opus 5.5 reaches 67% at \$0.79, and DeepSeek V4.1 Flash reaches 55% at \$0.09. These five agents form the Pareto frontier. Along it, the step from Claude Opus 5.5 to the 68% agents costs 3.5× more per rollout for one point of pass@1, and the full span from DeepSeek V4.1 Flash to Gemini 4 Argon costs 74× more for 1.2× the

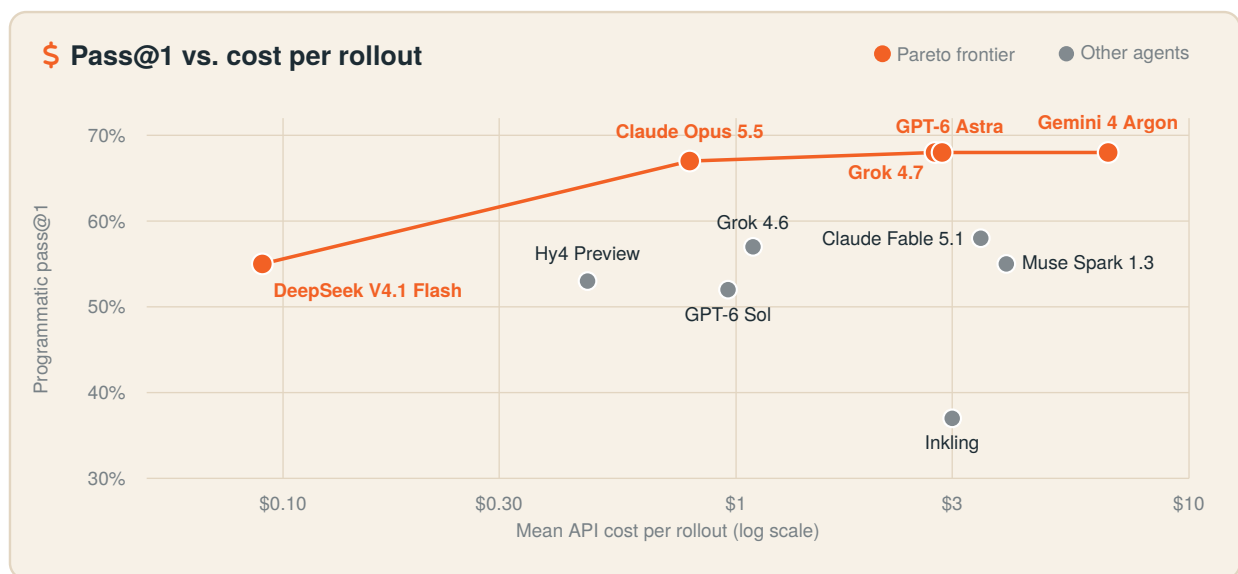


Figure 4 Programmatic pass@1 against mean API cost per rollout, using the values in Table 2. Orange points mark the Pareto frontier: agents for which no agent of equal or lower cost scores higher.

pass rate.

4.5 How does performance vary across languages and vulnerability categories?

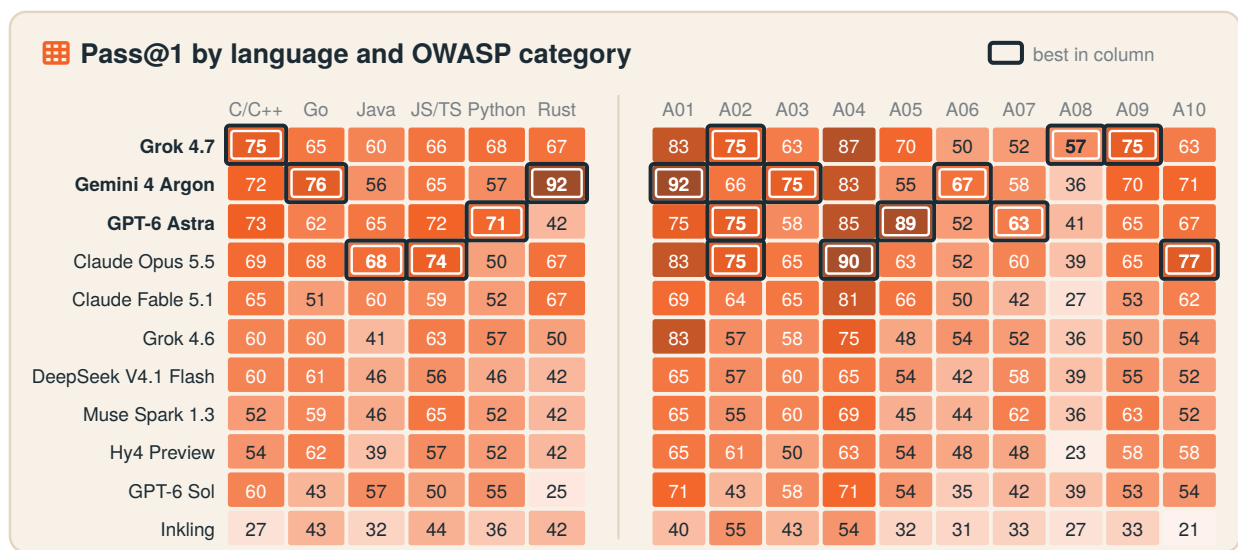


Figure 5 Programmatic pass@1 by language (left) and OWASP Top 10:2025 category (right). C/C++ and JS/TS are grouped. Outlined cells mark the best score in each column.

The leading agents trade places across both languages and vulnerability categories. Among the three agents tied at 68% pass@1, Grok 4.7 leads on C/C++ (75%), Gemini 4 Argon on Go (76%), and GPT-6 Astra on JS/TS (72%) and Python (71%) (Figure 5, left). Across OWASP categories, Integrity failures (A08) are the hardest: the best agent reaches 57%. Cryptography (A04) is the easiest, with Claude Opus 5.5 reaching 90% (Figure 5, right).

5 Discussion

CWE-bench asks agents to find and repair vulnerabilities in real repositories without being told the weakness or where it occurs, across 73 CWEs and all ten OWASP Top 10:2025 categories. Across eleven agents, the leaders tie on pass@1 but differ in retries, coverage, and cost, and vulnerability category matters more than language. We draw three takeaways from these results.

Automated vulnerability repair is becoming practical. The leading agents audit and repair most tasks end to end, without being told the weakness or where it occurs, and reach 81% pass@4 with retries. This performance is no longer limited to the most expensive models: Claude Opus 5.5 comes within a point of the leaders at a fraction of their cost (Section 4.4), which makes repair cheap enough to run routinely rather than only in periodic audits. Agents also solve different tasks: together they cover 110 of the 120 tasks, against 97 for the best single agent (Section 4.3). Systems that combine several agents, rather than relying on one, are a natural direction for automated defense.

Narrow benchmarks give a misleading picture of security capability. Performance varies far more across vulnerability categories than across languages, and the leading agent changes from one category to the next (Section 4.5). A benchmark that covers a few weakness classes can therefore rank agents differently than one that covers many, and a single aggregate score hides where each agent is weak. CWE-bench covers 73 CWEs across all ten OWASP Top 10:2025 categories, but this is a small fraction of the more than 900 weaknesses catalogued in CWE. Judging how ready agents are for defensive work requires a broader map of vulnerability classes, with results reported per class rather than as one number.

Verifiers must be hard but fair. A benchmark is only as good as what its verifier accepts. A verifier that is too lenient rewards incomplete patches, such as one that blocks the known exploit but leaves variants open, and overstates how ready an agent is to secure code. A verifier that is too strict penalizes correct patches that remove the vulnerability and preserve behavior but differ from the expected fix. Both errors mislead anyone choosing an agent, and the same holds for any team that uses automated checks to accept agent-written patches. CWE-bench grades every attempt twice, with a programmatic verifier and an independent panel of agentic judges. The two agree on 93.9% of attempts ($\kappa = 0.88$; Appendix C), and the same four agents lead under both (Section 4.2).

6 Conclusion

Security evaluations of AI agents have largely measured how well agents attack software. As agents are deployed to audit and maintain code, the more consequential question is whether they make software safer. CWE-bench measures this directly: agents must find and repair unlabeled vulnerabilities in real repositories, and a patch counts only if it blocks the exploit and preserves the program’s behavior. Our results show that agents differ in how they respond to retries, which tasks they cover, what they cost, and which vulnerability classes they handle, differences that a single pass rate hides. As the offensive capabilities of agents improve, their defensive capabilities must keep pace, and the only way to know whether they do is to measure defense rigorously across the full breadth of vulnerabilities that real software contains.

References

- Andy K Zhang, Neil Perry, Riya Dulepet, Joey Ji, Celeste Menders, Justin Lin, Eliot Jones, Gashon Hussein, Samantha Liu, Donovan Jasper, Pura Peetathawatchai, Ari Glenn, Vikram Sivashankar, Daniel Zamoshchin, Leo Glikbarg, Derek Askaryar, Haoxiang Yang, Aolin Zhang, Rishi Alluri, Nathan Tran, Rinnara Sangpisit, Kenny Oseleononmen, Dan Boneh, Daniel Ho, and Percy Liang. Cybench: A framework for evaluating cybersecurity capabilities and risks of language models. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu, editors, *International Conference on Learning Representations*, volume 2025, pages 25094–25243, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/3e9412a9c1d93810ef3ef7825115016b-Paper-Conference.pdf.
- Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, Jet Geronimo, Avi Dhir, Sudhit Rao, Kaicheng Yu, Twm Stone, and Daniel Kang. CVE-bench: A benchmark for AI agents’ ability to exploit real-world web application vulnerabilities. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu, editors, *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 79850–79867. PMLR, 13–19 Jul 2025. URL <https://proceedings.mlr.press/v267/zhu25i.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 C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust, editors, *International Conference on Learning Representations*, volume 2026, pages 123341–123386, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/c876a2935a90aff21874e14c07dc3e33-Paper-Conference.pdf.
- Zichao Wei, Jun Zeng, Ming Wen, Zeliang Yu, Kai Cheng, Yiding Zhu, Jingyi Guo, Shiqi Zhou, Le Yin, Xiaodong Su, and Zhechao Ma. Patcheval: A new benchmark for evaluating llms on patching real-world vulnerabilities, 2025. URL <https://arxiv.org/abs/2511.11019>.
- Xianzhen Luo, Jingyuan Zhang, Shiqi Zhou, Jinyang Huang, Chuan Xiao, Qingfu Zhu, Zhiyuan Ma, Yue Xing, Yang Yue, Wencong Zeng, and Wanxiang Che. CVE-factory: Scaling expert-level agentic tasks for code security vulnerability. In Tong Zhang, Miroslav Dudik, Martin Jaggi, Alekh Agarwal, Sharon Li, Dale Schuurmans, Jerry Zhu, Felix Berkenkamp, Hanze Dong, and Alberto Bietti, editors, *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pages 83137–83168. PMLR, 06–11 Jul 2026a. URL <https://proceedings.mlr.press/v306/luo26aj.html>.
- Tan Bui, Ting Zhang, Ferdian Thung, Yunpeng Xiong, Penghao Jiang, Xin Zhou, and David Lo. Vul4py: Benchmarking automated vulnerability repair in python with paired exploit and functional oracles, 2026. URL <https://arxiv.org/abs/2608.00692>.
- Xiaonan Luo, Yue Huang, Kehan Guo, Ping He, Chuan Zou, Chujie Gao, Lichi Li, Yuchen Ma, Zhangchen Xu, Zichen Chen, Yufei Han, and Xiangliang Zhang. Secprobe: Adaptive evaluation of coding agents on cybersecurity vulnerabilities, 2026b. URL <https://arxiv.org/abs/2609.33763>.
- Chihao Shen, Jiacheng Li, Aastha Mahajan, Jeffery Siyuan Tian, Yonghwi Kwon, and Yizheng Chen. Patchbench: Evaluating ai agents for vulnerability patching, 2026. URL <https://arxiv.org/abs/2609.04075>.
- Tianneng Shi, Robin Rheem, Dongwei Jiang, Mona Wang, Francisco De La Riega, Zhun Wang, Jingzhi Jiang, Alexander Cheung, Sean Tai, Jonah Cha, Jianhong Tu, Gabriel Han, Chenguang Wang, Jingxuan He, Wenbo Guo, and Dawn Song. CyberGym-E2E: Scalable real-world benchmark for AI agents’ end-to-end cybersecurity capabilities. In Tong Zhang, Miroslav Dudik, Martin Jaggi, Alekh Agarwal, Sharon Li, Dale Schuurmans, Jerry Zhu, Felix Berkenkamp, Hanze Dong, and Alberto Bietti, editors, *Proceedings of the 43rd International Conference on Machine Learning*, volume 306 of *Proceedings of Machine Learning Research*, pages 112094–112108. PMLR, 06–11 Jul 2026. URL <https://proceedings.mlr.press/v306/shi26r.html>.
- Nancy Lau, Louis Sloot, Jyoutir Raj, Giuseppe Marco Boscardin, Evan Harris, Dylan Bowman, Mario Brajkovski, Jaideep Chawla, and Dan Zhao. Zerodaybench: Evaluating llm agents on unseen zero-day vulnerabilities for cyberdefense, 2026. URL <https://arxiv.org/abs/2603.02297>.
- Zhun Wang, Nico Schiller, Hongwei Li, Srijiith Sesha Narayana, Milad Nasr, Nicholas Carlini, Xiangyu Qi, Eric Wallace, Elie Bursztein, Luca Invernizzi, Kurt Thomas, Yan Shoshitaishvili, Wenbo Guo, Jingxuan He, Thorsten Holz, and Dawn Song. Exploitgym: Can ai agents turn security vulnerabilities into real attacks?, 2026a. URL <https://arxiv.org/abs/2605.11086>.
- Seunghyun Lee and David Brumley. Exploitbench: A capability ladder benchmark for llm cybersecurity agents, 2026. URL <https://arxiv.org/abs/2605.14153>.
- Gelei Deng, Yi Liu, Yuekang Li, Ruozhao Yang, Xiaofei Xie, Jie Zhang, Han Qiu, and Tianwei Zhang. What makes a good llm agent for real-world penetration testing?, 2026. URL <https://arxiv.org/abs/2602.17622>.
- Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and LINGMING ZHANG. Sec-bench: Automated benchmarking of llm agents on real-world software security tasks. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen, editors, *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pages 116342–116378. Curran Associates, Inc., 2025. doi: 10.52202/085713-3878. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/a9168f1c54e5147027f1e8cf83e1a775-Paper-Conference.pdf.
- Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA ’15*, page 24–36. ACM, 2015. doi: 10.1145/2771783.2771791. URL <http://dx.doi.org/10.1145/2771783.2771791>.
- TJ Byun, Cornelius Aschermann, Kai Yuan Thng, Wu Zhou, Yupeng Yang, Lauren Deason, and Joshua Saxe. Introducing AutoPatch-Bench: A benchmark for AI-powered security fixes. Engineering at Meta, April 2025. URL <https://engineering.fb.com/2025/04/29/ai-research/autopatchbench-benchmark-ai-powered-security-fixes/>. Technical release, April 29, 2025. Accessed September 30, 2026.
- Zheng Yu, Wenxuan Shi, Xinqian Sun, Zheyun Feng, Meng Xu, and Xinyu Xing. Patch validation in automated vulnerability repair, 2026. URL <https://arxiv.org/abs/2603.06858>.

- Hwiwon Lee, Jiawei Liu, Dongjun Kim, Wubing Xia, Ziqi Zhang, Chunqiu Steven Xia, and Lingming Zhang. Sec-bench pro: Can language models solve long-horizon software security tasks?, 2026. URL <https://arxiv.org/abs/2605.26548>.
- Patrick Deininger and Wolfgang Slany. Fixing, breaking, or faking it? an execution-calibrated evaluation of llm vulnerability patching in javascript, python, go, and java, and the limits of llm-as-judge. *Journal of Cybersecurity and Privacy*, 6(5):153, 2026. ISSN 2624-800X. doi: 10.3390/jcp6050153. URL <http://dx.doi.org/10.3390/jcp6050153>.
- Michael Kouremetis, Ads Dawson, Raja Sekhar Rao Dheekonda, and Brian Greunke. Every model cheats: Prompt-level mitigation of cheating on offensive cyber tasks, 2026. URL <https://arxiv.org/abs/2607.21763>.
- Hao Wang, Hanchen Li, Qiuyang Mang, Alvin Cheung, Koushik Sen, and Dawn Song. Do androids dream of breaking the game? systematically auditing ai agent benchmarks with benchjack, 2026b. URL <https://arxiv.org/abs/2605.12673>.
- Shenghan Zheng, Zonglin Di, Yimin Liu, Kyoung Whan Choe, Jiankai Sun, Heguang Lin, Penghao Jiang, Yifeng He, Xiao Cheng, Jicheng Wang, Wenbo Chen, Alex Yates, Yinzhe Zhao, Bingran You, Yuan Gao, Ayush Munot, Shubham Gaur, Zhe Ye, Hao Wang, Xiangyi Li, Dawn Song, and Christophe Hauser. Benchshield: Formal model-backed instrumentation for reward integrity in llm-agent evaluation infrastructure, 2026. URL <https://arxiv.org/abs/2609.11028>.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebggen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL <https://arxiv.org/abs/2107.03374>.
- Pedro Rodriguez, Joe Barrow, Alexander Hoyle, John P. Lalor, Robin Jia, and Jordan Boyd-Graber. Evaluation examples are not equally informative: How should that change NLP leaderboards? In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4486–4503, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.346. URL <https://aclanthology.org/2021.acl-long.346/>.

A Evaluation Details

Each agent pairs a model with the harness listed in Table 2, runs at high reasoning effort, and makes four independent attempts per task with a one-hour limit per attempt. An attempt that reaches the time limit is graded on the repository state at that point. Each attempt runs in an isolated container whose only network access is the model provider’s API. Some providers flag defensive security requests: in Codex, a flagged request ends the attempt, which counts as a failure, while in Claude Code the request is completed by a fallback model (Claude Opus 5 or Claude Opus 4.8) in the same session. We credit these attempts to the original model, since this is the behavior a user of that model receives. Cost is the API cost of an attempt as reported by the harness, or, where the harness does not report it, computed from token usage and the provider’s published per-token prices.

B Agentic Judge Panel

The panel comprises GPT-5.4, Claude Sonnet 5, and Gemini 3.8 Flash. Each judge receives the task instruction, a description of the injected vulnerability and of what a correct fix must do, and the reference patch. Each judge then works independently in its own copy of the patched repository with shell access: it can inspect the code, build the project, run its tests, and run the programmatic verifier, whose result is available as evidence. A judge assesses each criterion in Table 3 separately and accepts the patch only if all five hold. The panel accepts a patch when a majority of judges accept it.

Criterion	Requirement
Security	The repair closes the target vulnerability.
Functionality	Ordinary, legitimate behavior remains available.
Generalization	The repair blocks attack variants beyond the particular verifier probe.
Integrity	The submission repairs the software rather than manipulating the grading mechanism.
Soundness	The patched codebase builds and runs in the task environment.

Table 3 Judge-panel criteria. A judge accepts a repair only if all five criteria are satisfied.

C Agreement between Programmatic and Panel Verdicts

We compare the programmatic and panel verdicts on each of the 5,280 attempts in Table 2. These come from the final evaluation, run after the verifier revisions described in Appendix D. The two agree on 93.9% of attempts (Cohen’s $\kappa = 0.88$; Table 4), and per-agent agreement ranges from 91.0% to 95.4%. Of the 320 disagreements, 200 are patches that the programmatic verifier accepts and the panel rejects.

The two graders check different things by design. The programmatic verifier runs fixed exploit probes and functionality tests. Each judge also assesses whether a patch blocks variants of the attack beyond those probes (Table 3), a criterion that fixed probes cannot fully encode. This is consistent with most disagreements being patches the verifier accepts and the panel rejects.

	Panel pass	Panel fail
Programmatic pass	2,860	200
Programmatic fail	120	2,100

Table 4 Attempt-level agreement between the programmatic verifier and the agentic judge panel.

D Human Review

During verifier development, we manually reviewed every attempt on which the programmatic verifier and the panel disagreed. The programmatic verifier encodes our definition of a correct repair, so each disagreement was resolved in its favor unless review showed an error in the verifier. When judges showed that the verifier was too strict, failing valid

repairs, we replaced the affected checks with checks of behavior. When judges showed that it accepted incomplete repairs, we added probes to catch them. When a judge rejected a patch over an issue that does not affect security or functionality, we kept the verifier’s verdict.

E Task Coverage

Table 5 lists the CWEs in the evaluation set under their OWASP Top 10:2025 category.

	Category	Tasks	CWEs
A01	Broken Access Control	13	22, 285, 441, 497, 601, 639, 862, 863, 918
A02	Security Misconfiguration	11	16, 310, 489, 611, 614, 776, 942, 1004
A03	Software Supply Chain Failures	10	1104, 1329, 1357, 1395
A04	Cryptographic Failures	13	208, 319, 326, 327, 330, 338, 347, 916
A05	Injection	14	20, 79, 89, 93, 94, 116, 470, 1336
A06	Insecure Design	12	73, 269, 362, 434, 444, 522, 693, 770, 913, 1284
A07	Authentication Failures	13	287, 288, 290, 295, 297, 306, 307, 613
A08	Software or Data Integrity Failures	11	345, 354, 494, 502, 565, 829, 915, 1321
A09	Security Logging and Alerting Failures	10	117, 532
A10	Mishandling of Exceptional Conditions	13	209, 252, 369, 390, 476, 674, 703, 755
	Total	120	73

Table 5 CWEs in the evaluation set by OWASP Top 10:2025 category.

F Item Response Theory Analysis

As another view of the results, we fit a Rasch item response theory (IRT) model (Rodriguez et al., 2021) to the programmatic outcomes in Table 2. The model treats the passing attempts c_{mj} of agent m on task j out of n_{mj} as

$$c_{mj} \sim \text{Binomial}(n_{mj}, p_{mj}), \quad p_{mj} = \frac{1}{1 + \exp\{-(\theta_m - b_j)\}}, \quad (2)$$

where θ_m is the ability of agent m and b_j the difficulty of task j . Abilities are centered at zero, and intervals come from 300 bootstrap refits.

Because every agent attempted every task four times, the fitted abilities follow the pass@1 ordering; what the model adds is a placement of tasks and agents on the same scale. Figure 6 shows the fitted success curves for three tiers of agents. Grok 4.7, Gemini 4 Argon, GPT-6 Astra, and Claude Opus 5.5 have overlapping intervals, above a middle group of six agents and Inkling. On the task of median difficulty, the predicted chance of success is 78%, 58%, and 28% for the three tiers. Thirty tasks have an estimated difficulty above the strongest agent’s ability (25 to 34 across bootstrap refits), meaning even the strongest agent is predicted to solve them less than half the time.

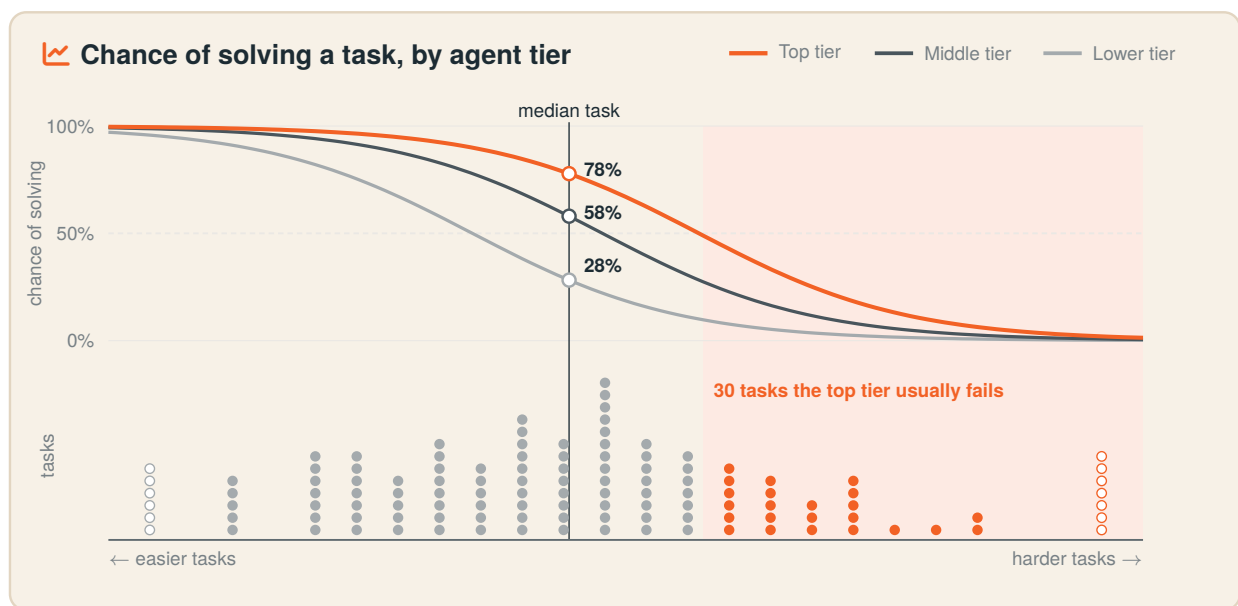


Figure 6 Predicted chance of solving a task as a function of task difficulty, from a Rasch model fit to the programmatic outcomes in Table 2. Each curve uses the mean ability of one tier: top (Grok 4.7, Gemini 4 Argon, GPT-6 Astra, Claude Opus 5.5), middle (Claude Fable 5.1, Grok 4.6, DeepSeek V4.1 Flash, Muse Spark 1.3, Hy4 Preview, GPT-6 Sol), and lower (Inkling). The vertical line marks the task of median difficulty. Below, each circle is one task placed by its estimated difficulty; hollow circles are tasks that every agent always or never solved. The shaded region contains the 30 tasks the strongest agent is predicted to solve less than half the time.