Vulnerability Localization Benchmark: Measuring Agentic Security Analysis at Repository Scale

Aman Priyanshu[*,1], Supriti Vijay[*,1], Kimia Majd[*,1], Xuhong He[1,3,†], Fraser Burch[1], Takahiro Matsumoto[1], Jianliang He[1,2,†], Baturay Saglam[1,2,†], Arthur Goldblatt[1], Zhuoran Yang[1,2,†], Amin Karbasi[1]

[1]Foundation AI–Cisco Systems Inc.
[2]Yale University
[3]Carnegie Mellon University

[†]Work done while at Foundation AI
[*]Equal Contribution. Corresponding authors: {ampriyan,suprivij,kimia}@cisco.com

Code & Data Website

Abstract

Language-model agents increasingly operate over complete software repositories, yet cybersecurity evaluations primarily measure whether they can detect, reproduce, or repair vulnerabilities rather than whether they can locate the relevant code. We study *vulnerability localization*: given a weakness class and an unfamiliar repository, identify the implementation files associated with that weakness. We introduce the Vulnerability Localization Benchmark (VLoc Bench), comprising 500 real world vulnerabilities from 290 repositories across six package ecosystems and 147 CWE categories. Each task pairs repository snapshots immediately before and after a security fix. On the vulnerable snapshot, an agent receives only the CWE description and read-only terminal access and must return the affected files; on the patched snapshot, it must determine that the recorded vulnerability is no longer present. We evaluate 27 language models and four static-analysis tools under a common agent interface. Repository-scale vulnerability localization remains difficult: the strongest system achieves 0.229 File F1, and 38.4% of tasks receive no correct localization from any evaluated model. We further find that stronger localization does not imply reliable behavior after remediation: systems that identify vulnerable files effectively can still report unsupported locations on patched repositories. These results establish vulnerability localization as a distinct repository-scale capability and provide a setting for studying both how security agents search for vulnerable code and when they should refrain from reporting it.

1. Introduction

Language model agents have become capable of software-engineering tasks that require reasoning over and modifying real codebases, and these capabilities increasingly extend to security analysis. A

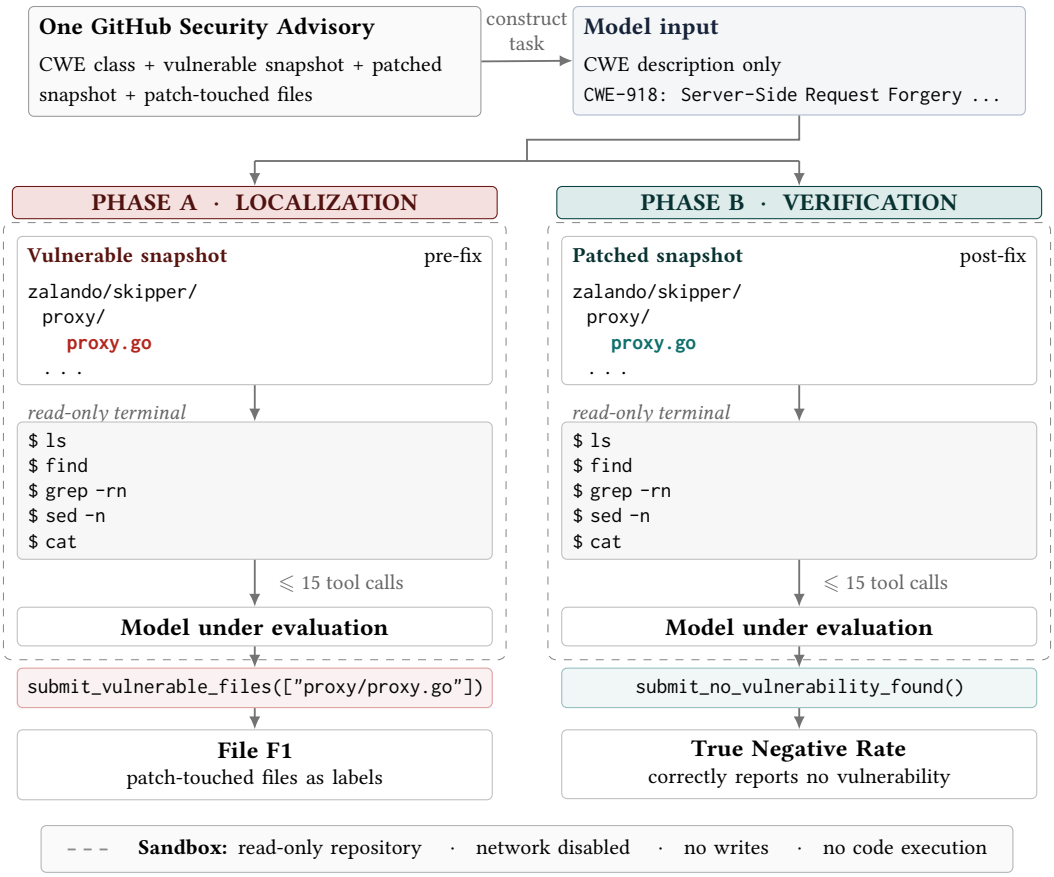

Figure 1 | **Task structure.** Each task is constructed from a GitHub Security Advisory by pairing its weakness class with vulnerable and patched snapshots of the same repository. The model receives only the generic CWE description and read-only repository access; advisory text, CVE identifiers, affected versions, and patch-derived file labels are withheld. Phase A asks the model to localize vulnerable files and is scored against files touched by the security patch. Phase B presents the patched snapshot and measures whether the model correctly reports no vulnerability. Both phases use the same system prompt, tools, and interaction budget; only the repository state changes.

growing body of benchmarks evaluates agents on real vulnerabilities by asking them to generate a patch, produce an input that triggers vulnerable behavior, or determine whether a given piece of code is vulnerable [3, 8, 42]. These tasks measure important security capabilities, but they do not necessarily measure *localization*: identifying where in a repository the vulnerable implementation lies. Detection and localization are distinct problems. Vulnerability detection asks whether code that has already been selected is vulnerable; vulnerability localization asks which code should have been selected in the first place. For a security engineer, this distinction is consequential: remediation begins by narrowing a repository to the files that require investigation, which determines what must be patched, what other components may be affected, and whether the same defect propagates to forks or vendored copies.

Current evaluations capture localization only indirectly. Existing security benchmarks largely fall into two settings: those that preselect the code to be analyzed, and those that evaluate a downstream security outcome. In the former, models judge functions or fragments that have already been retrieved, making the task one of vulnerability recognition rather than repository search [8]. In the latter, agents may search a full codebase, but success is defined by whether they repair or reproduce vulnerable behavior; the relevant location may therefore be supplied explicitly [3] or remain unobserved because

correctness is determined through execution [35]. In either case, localization is not the object of evaluation. This leaves a basic capability poorly characterized: given a weakness to investigate and an unfamiliar repository, can an agent identify the files in which that weakness is implemented?

We introduce the **Vulnerability Localization Benchmark (VLoc Bench)** to evaluate this capability directly. We construct VLoc Bench from public GitHub Security Advisories (GHSAs) [13], which link disclosed vulnerabilities to affected repositories and their security fixes. Each of its 500 tasks pairs a repository snapshot from immediately before the corresponding security fix with the snapshot after it, spanning 290 repositories across six packaging ecosystems. An agent receives the pre-fix repository together with the generic MITRE description of one weakness class [39], explores the codebase through a read-only terminal, and returns the implementation files it judges to contain the vulnerability. The weakness description specifies what type of vulnerability to search for, but provides no repository-specific evidence: the agent receives no advisory text, CVE identifier, fixing commit, file hint, or line range. We score the returned set against the implementation files modified by the security patch using file-level F_1. A second phase presents the corresponding patched repository under the same interface, where the recorded vulnerability has been removed and the correct response is to report no file. We then use this benchmark to study how localization varies with model scale, task-specific training, repository structure, and agent search behavior.

Our results show that repository-scale vulnerability localization remains difficult for current systems. The strongest configuration only reaches 0.229 File F_1. Difficulty also depends strongly on the repository being searched: localization performance falls substantially as repositories become larger and relevant code becomes more dispersed. At the same time, successful localization does not necessarily correlate with correctly recognizing when the vulnerability has been removed. Systems that perform well in Phase A can still report vulnerable files in already-patched repositories, revealing a tension between aggressively searching for vulnerabilities and avoiding unsupported reports. Evaluating localization alone can favor systems that report broadly, whereas evaluating only patched repositories can favor systems that rarely commit to a localization. VLoc Bench therefore evaluates both sides of the problem: whether an agent can identify security-relevant implementation when a vulnerability is present, and whether it can refrain from reporting that vulnerability after remediation.

2. Related Work

Vulnerability localization connects two lines of work that have developed largely in parallel. Software-engineering research studies how agents navigate repositories and identify code relevant to a reported issue, while cybersecurity benchmarks increasingly place agents in real codebases to detect, reproduce, or repair vulnerabilities. These literatures share repository-scale reasoning as a common challenge, but differ in what they make observable: software-engineering benchmarks increasingly score where an agent looks, whereas security benchmarks have primarily scored what an agent concludes or does after reasoning about vulnerable code. Table 1 summarizes the individual benchmarks; here, we focus on the broader themes that connect them.

Repository-Scale Localization in Software Engineering Localization is an established problem in software engineering: given a reported defect, identify the files, functions, or regions of a repository relevant to resolving it. Classical work formulated this as ranking source files against bug reports, with files modified by the eventual developer fix serving as ground truth; recent agentic work extends the same formulation to systems that actively search repositories and choose their own retrieval sets [5, 6, 19, 23, 44, 47, 48, 49]. This shift makes precision as important as recall: an agent must not only recover relevant files, but avoid filling its context with unrelated code [5, 23]. The task nevertheless begins from a known software issue. An issue report establishes that something is wrong and often

supplies repository-specific evidence about the failure; indeed, recent benchmarks explicitly filter reports that reveal the answer location because such information can make retrieval nearly solved before repository search begins [47]. Thus, software engineering provides a mature formulation of repository localization, but typically under the assumption that a concrete defect has already been reported.

Localization Is Implicit in Security Evaluation Security evaluation has traditionally exposed a different part of the analysis pipeline. Detection benchmarks ask whether code is vulnerable after a function, snippet, commit, or other candidate region has already been selected; this measures vulnerability recognition, not the preceding problem of finding that region in a repository [1, 8, 27, 43, 45]. Repair benchmarks similarly focus on whether vulnerable behavior can be removed and may therefore provide localization information explicitly, while reproduction and exploitation benchmarks can evaluate success through execution without consulting a source-code location at all [3, 22, 34, 42]. As security evaluation has moved toward complete repositories, however, the search required before these downstream actions has become increasingly visible. Performance changes sharply with the amount of vulnerability-specific information supplied—from a weakness class or report to crash traces and fixing patches—and end-to-end evaluations identify vulnerability discovery as a major bottleneck when such evidence is withheld [35, 42, 46]. Repository search is therefore already part of security-agent performance, even when it is not itself the scored output.

Vulnerability Localization as Repository Search Recent work has begun to bring these two perspectives together by scoring locations associated with real vulnerabilities [9, 18, 26, 50]. These evaluations make clear that "localizing a vulnerability" can refer to several related targets: vulnerable functions or statements, security-relevant data-flow paths, entry and critical-operation points, or statements that trigger vulnerable behavior. They also vary in how much of the search problem remains for the system: the analysis may begin from an entire repository, a reduced file or function, a vulnerability-specific advisory, a weakness specification, or the fixing commit itself [9, 18, 25, 26, 50]. These formulations address different stages of vulnerability analysis, but together they establish an important point: security reasoning and repository localization can be evaluated separately from the downstream act of producing an exploit or patch. This motivates treating vulnerability localization as a security-aware repository-search problem, where the system must connect an abstract weakness to the concrete implementation that realizes it.

Localization When the Vulnerability May Be Absent A second distinction emerges when localization is considered in a security setting. Conventional bug localization assumes that the issue described in the prompt exists, so failing to return a candidate location is simply a localization failure [6, 19, 47]. Security tools, by contrast, must also operate on code in which a suspected vulnerability is absent or has already been remediated. False positives have therefore long been treated as a practical concern in static analysis, and recent security evaluations increasingly use paired vulnerable and patched code or pre/post-patch execution to measure whether systems continue to report behavior after a fix [2, 8, 37, 42, 45, 46, 50]. Bringing this distinction to repository-scale localization changes the decision being evaluated: a system must determine not only *where* evidence for a weakness lies, but whether the repository supports reporting a location at all. Localization and restraint therefore become complementary aspects of the same security-analysis problem and the basis for our work in this paper.

Table 1 | Related work by scored output, what the prompt discloses, and whether localization and behavior on patched code are scored. Instance counts follow each paper's reported unit.

	Scored output	Inst.	Scope	Location disclosed	Localization	Patched code
Repair and completion						
Vul4Py [3]	Patch	100	Python	Files, line ranges	—	—
SecRepoBench [34]	Completion	318	C/C++	Region masked out	—	—
Triggering						
CyberGym [42]	Crashing input	1,507	C/C++	Approximate, in description	—	Pre/post criterion
BountyBench [46]	Exploit	40	25 codebases	Four conditions	—	Detect snapshots
Detection						
PrimeVul [8][a]	Label	6,968 fn.	C/C++	Fragment supplied	—	5,480 pairs
JitVul [45][b]	Label	879 fn.	C/C++	Commit supplied	—	879 pairs
RepoPairBench (DREA) [37]	Label	200	Python	Function, file path	—	False-positive rate
VulEval [43][c]	Label, ranked deps.	232,239 fn.	C/C++	Function supplied	Dependencies, Pre@k	—
SecVulEval [1][c]	Label, statements	25,440 fn.	C/C++	Function supplied	Statements, 300 judged	Fixed versions included
VulDetectBench [27]	Label; lines	1,000; 100	C/C++	One vuln., under 4K tokens	Lines, LoC recall	—
Localization, general software issues						
ContextBench [23]	Context set	1,136	8 languages	Issue report	Files, F1	—
CodeGrep [5]	File list	Undisclosed	Python	Issue report	Files, $F_{\beta=0.5}$	—
CoSIL [19]	Ranked files, functions	800	Python	Issue report	Files, Top-k	—
SWE-Explore [48]	Ranked regions	848	10 languages	Issue report	Files, recall	—
CORE-Bench [47]	Ranked chunks	5,061	11 languages	Query, leaks filtered	Chunks, NDCG@k	—
Loc-Bench (LocAgent) [6]	Ranked locations	560	Python	Issue report	Files, Acc@k	—
Localization, real vulnerabilities						
RustMizan [9]	Locations	173	Rust	None; 96 reduced	Functions, lines, F1	78, not split out
CWE-Bench-Java (IRIS) [26]	Dataflow paths	120	Java	Weakness class	Methods, one-sided	—
VulnGym [18]	Entry, operation, trace	408	23 repositories	Advisory text	Files, oracle only	—
AutoTrace [50]	Trigger statements	744	C/C++	Fixing commit	Statements, hit rate	771 pairs, SinkTrace
VLoc Bench[d]	Files	500	6 ecosystems	Weakness class	Files, F1	True negative rate

[a] PrimeVul's instance count refers to vulnerable functions rather than all functions in the corpus.
[b] JitVul's instance count refers to vulnerable functions; its matched pairs are drawn from PrimeVul, so the two rows are not independent corpora.
[c] VulEval and SecVulEval report counts over all functions in their respective corpora rather than only vulnerable functions.
[d] To our knowledge, no prior benchmark scores file-level vulnerability localization over a whole repository with the location withheld while also evaluating the corresponding patched repository.

3. Benchmark Design

VLoc Bench evaluates whether an agent can identify vulnerable files in a repository given only a weakness class. The design rests on one principle: since security matters in any codebase, the task stays repository-agnostic. Nothing specific to the repository enters it, so every file the agent names comes from its own search. Each task links a weakness class to two repository states and a set of vulnerable implementation files derived from the fixing patch. The remainder of this section describes how these states and labels are constructed, how agents interact with them, and how their predictions are scored. We use CWE descriptions as the task specification because they identify the class of security weakness to search for without exposing the repository-specific localization cues often present in advisories or vulnerability reports. More broadly, CWE provides a common taxonomy through which the same localization task can be defined across otherwise heterogeneous repositories, languages, and software ecosystems.

3.1. Benchmark Construction

Threat model. Vulnerability localization can target different points in the lifecycle of a weakness. Prior work has considered the *trigger site*, where vulnerable state manifests as an unsafe operation [24, 50], while patch-based localization instead identifies the implementation that must be modified to remediate the vulnerability. We adopt the latter, defense-oriented view: given that a vulnerability of a particular CWE may exist somewhere in a repository, the agent must identify the files a defender would need to investigate and remediate. Accordingly, VLoc Bench defines the localization target as the *patch site* and uses the implementation files modified by the eventual security fix as ground truth. Whereas many cybersecurity evaluations are organized around the capability being measured—detecting, triggering, exploiting, or repairing a vulnerability—our formulation also makes explicit who the localization signal is intended to serve, defenders. Beyond matching the needs of triage and remediation, this definition gives us a reproducible target that can be recovered from real-world vulnerability fixes, enabling the benchmark construction described next.

Sourcing. Each task begins with a GitHub Security Advisory (GHSA) [13], a record linking a weakness class (CWE) [39] to a repository and the commit that resolved it. The fixing commit provides the repository states and the patch used to derive the vulnerable-file labels.

Repository states. Each task comprises two snapshots of the same repository, one preceding and one following the security fix. The *pre-push* state is the commit immediately before the patch and contains the vulnerability. The *post-push* state is the patch commit itself and contains the fixed implementation. Both snapshots preserve the directory structure, configuration files, and source code. The paired states support localization before the fix and verification after it.

Ground-truth labels. We derive the vulnerable-file labels deterministically from the patch diff. The label set contains every implementation file modified by the patch, excluding files that are exclusively test files. We detect test and documentation files using path-based heuristics, including files under `test/`, `tests/`, `spec/`, `__tests__/`, `doc/`, and `docs/` directories or matching `*_test.*`, `*_spec.*`, `test_*.*`, `*.md`, and `*.rst` patterns. Thus, the released label sets contain only implementation source files changed by the security fix; they exclude tests, documentation, and configuration files. Using the implementation files changed by a fix as ground truth follows established repository-scale localization practice, including recent agent benchmarks [6, 23, 44, 49].

Quality control. We exclude tasks where: (1) the patch modifies only tests, documentation, CI configuration, or lock files; (2) the repository has been deleted or made private since the advisory; (3) the patch spans more than 50 files; or (4) the advisory lacks a CWE classification. The final benchmark contains 500 tasks from advisories disclosed between 2016 and 2026, including 21 disclosed in 2026.

Benchmark composition. The 500 tasks span 6 package ecosystems: Go (n=215, 43%), Maven (n=104, 21%), npm (n=88, 18%), pip (n=52, 10%), Rust (n=40, 8%), and Composer (n=1, <1%). They cover 147 unique CWE categories, with the most frequent being CWE-400 (Resource Exhaustion, n=53), CWE-20 (Improper Input Validation, n=45), CWE-200 (Information Exposure, n=27), CWE-22 (Path Traversal, n=27), and CWE-770 (Allocation without Limits, n=23). By CVSS severity: Critical (n=57, 11%), High (n=219, 44%), Medium (n=194, 39%), and Low (n=30, 6%). 78% of tasks carry assigned CVE identifiers. Repository sizes range from 12 KB to 840 MB (median 4.2 MB), and ground-truth file counts have a median of 3, with 28.4% of tasks containing a single ground-truth file and 14.8% containing ten or more.

3.2. Evaluation Protocol

Task inputs and outputs. Each task is evaluated in two phases using the paired repository snapshots. The model receives only the CWE category description (e.g., "CWE-79: Improper Neutralization of Input During Web Page Generation") and terminal access to the repository. We withhold advisory text, file hints, severity scores, affected version ranges, and CVE identifiers from the model's prompt. In **Phase A** (localization), the model receives the pre-push repository and submits the files that contain the vulnerability. In **Phase B** (verification), it receives the post-push repository and the same CWE description, then declares that no vulnerability is present.

Both phases use the same system prompt, tools, submission options, and command budget; only the repository state changes. In Phase A, the model must submit one or more vulnerable file paths to receive a nonzero File F1. Calling `submit_no_vulnerability_found` or exhausting the budget without submitting a prediction is incorrect. In Phase B, the correct outcome is that no vulnerable file path is submitted. Calling `submit_no_vulnerability_found`, exhausting the budget without a submission, or returning no prediction is scored as a true negative; submitting any file path is a false positive. We use this shared interface for all evaluations to ensure consistent comparisons across models and phases.

Agent interface. All models use the same system prompt and agent interface. The terminal exposes read-only commands for inspecting the repository, including file listing, text search, file viewing, and metadata queries. Write operations, network access, and process execution are prohibited. The model terminates a task by calling `submit_vulnerable_files` with a ranked list of paths or `submit_no_vulnerability_found`. It has a budget of 15 terminal commands and 20 generation turns; three consecutive turns without a tool call trigger forced termination. These constraints are identical across all evaluated systems. We use temperature 0.3 with a 16,384-token completion limit; runs are stochastic, and we report the mean across three independent runs.

Sandbox. Each task runs in a fresh Docker container (Ubuntu 24.04) with 2 CPU cores, 4 GB RAM, network disabled, and a 10-second per-command timeout. The container is destroyed after each task, ensuring no information leakage between evaluations. The repository is mounted read-only at `/repo/`. Together, these restrictions isolate each evaluation and prevent external access, repository modification, and cross-task information leakage.

3.3. Scoring

VLoc Bench scores the model's final submission under the same interface in both phases. The model searches a repository snapshot and then either submits vulnerable file paths or declares that no vulnerability is present. The score depends on the repository state: Phase A rewards overlap with patch-derived files, while Phase B rewards a correct clean-repository judgment. The phases therefore require different metrics. Phase A is a set-retrieval task: its ground-truth set may contain several implementation files, and the score must penalize both omissions and unrelated predictions. Phase B has an empty ground-truth set by construction, so it requires a binary measure of whether the model declares the patched repository clean. We use File F1 for Phase A and TNR for Phase B, and report abstain rate separately.

File F1 (Phase A). We score localization with File F1, the harmonic mean of file-level precision and recall:

$$\text{Precision} = \frac{|\text{submitted} \cap \text{ground\_truth}|}{|\text{submitted}|}, \quad \text{Recall} = \frac{|\text{submitted} \cap \text{ground\_truth}|}{|\text{ground\_truth}|} \tag{1}$$

$$\text{File F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2}$$

If the model submits no files or calls `submit_no_vulnerability_found` on a vulnerable repository, File F1 is 0. The submitted paths are treated as an unordered set, so path order does not affect the score. This follows recent repository-localization benchmarks that evaluate final file predictions with file-level precision–recall measures [5, 23, 38].

Metric integrity. Prior work aligns evaluation with the observable outcome of the task. Patch-generation benchmarks such as SWE-bench assess whether a submitted change satisfies the repository's behavioral checks, while work on agentic cybersecurity capabilities commonly uses execution-based criteria to test whether a generated proof of concept reproduces the target behavior across pre- and post-patch builds [20, 42]. Other agent benchmarks verify state-changing actions against task-specific oracle annotations. These criteria are appropriate when correctness is expressed through program behavior or environment state. VLoc Bench instead asks the model to submit a final set of repository paths, so Phase A is evaluated against patch-derived file labels with File F1. The precision–recall formulation distinguishes incomplete localization from overinclusive predictions, which an exact-match or recall-only score would not. The same principle applies to Phase B: its binary clean-repository output is scored with TNR. Across both phases, the score is determined solely by the final submission, independent of tests, execution traces, and output formatting.

True Negative Rate (Phase B). We score verification with the true negative rate (TNR), the fraction of Phase B tasks where the model correctly calls `submit_no_vulnerability_found`:

$$\text{TNR} = \frac{|\text{tasks correctly declared clean}|}{|\text{total Phase B tasks}|} \tag{3}$$

Submitting any file path on a patched repository is a false positive and does not count as a true negative.

Abstain Rate. We also report the fraction of Phase A tasks where the model fails to submit a localization prediction, either by calling `submit_no_vulnerability_found` on a vulnerable repository or by exhausting its turn budget. An explicit clean-code judgment or an unfinished run counts as an abstention. Submitting incorrect files does not: it is a localization attempt and receives File F1 of 0 if none of the submitted paths is correct.

Macro-aggregation. For each run, we compute File F1 separately for each task and report the macro-average over the 500 tasks. TNR and abstain rate are computed as proportions over their respective phase tasks, which is equivalent to averaging binary task-level outcomes. We then report the mean of each run-level metric across three independent runs. File F1 is therefore a macro-averaged task-level metric rather than a pooled file-level statistic.

Together, File F1 and TNR measure the two required decisions: which files to report in the pre-push state and whether to report no vulnerability in the post-push state. Existing benchmarks such as CyberGym and ExploitGym assess agentic cybersecurity capabilities that can directly support offensive workflows, including vulnerability reproduction and exploitation [41, 42]. VLoc Bench instead targets a more neutral, defensive capability: locating the implementation files associated with a weakness in source code. Agents use a read-only interface and cannot modify or execute the repository, keeping the evaluation focused on analysis rather than operational exploitation. We view exploitation and vulnerability localization as distinct capabilities, and design the benchmark construction to preserve that distinction.

4. Experiments

4.1. Models Evaluated

We evaluate 27 models and four static-analysis tools. The model suite spans frontier closed-source systems, open-weight models across the [350M, 753B] parameter range, and models specialized for code search or vulnerability localization. This coverage allows us to compare general-purpose capability with parameter scale, domain-specific training, and non-agentic baselines. The evaluated systems are:

- **Frontier (closed-source):** GPT-5.5 with default and xhigh reasoning effort [30], GPT-5, GPT-5 Mini, and GPT-5 Nano [36]; Gemini 3 Pro [16], Gemini 2.5 Flash [10], and Gemini 3.1 Flash-Lite [15].
- **Open-weight large ($\geqslant$20B):** GLM-5.2 (753B) [14], MiniMax-M2.7 (229B) [4], Qwen3.5-122B, Qwen3.5-35B-A3B, and Qwen3.5-27B [31], GPT-OSS-120B and GPT-OSS-20B [29], Llama-3.3-70B [28], and Gemma-4-31B [11].
- **Open-weight small (<20B) and specialized:** CodeScout-14B [38], Qwen3.5-9B [31], Gemma-4-E4B (4B) and Gemma-4-E2B (2B) [11], Antares-3B, Antares-1B, and Antares-350M [40], and Granite-4.0-Micro (3B), Granite-4.0-1B, and Granite-4.0-350M [17].
- **Static analysis tools (non-model baselines):** Semgrep and Semgrep-CWE [32], CodeQL [12], and Horusec [51].

We keep the task interface fixed across model evaluations. Open-weight models are served with vLLM [21] on H100 GPUs, while closed-source models use their respective API endpoints. Detailed serving and generation settings are provided in Appendix D. Static-analysis tools run directly on each repository with their default rulesets; Semgrep-CWE uses CWE-specific rules matched to each task's classification. The system prompt and tool schemas are included in the released code repository. All reported scores are means over three independent evaluation runs.

4.2. Main Results

Table 2 summarizes Phase A performance across all evaluated systems. We compare model families and static-analysis baselines before examining how localization quality varies with parameter scale and domain-specific training; Phase B verification is reported separately below.

Table 2 | Phase A performance on VLoc Bench (File F1, Precision, Recall). Systems are grouped by category and sorted by File F1 within each group. The five highest-scoring model systems achieve at least 0.18 File F1, while all remaining systems score at most 0.16.

Model	Params	Open	File F1	Precision	Recall
Frontier (Closed-Source)					
GPT-5.5 (xhigh)	—	×	0.229	0.310	0.221
GPT-5.5 (default)	—	×	0.221	0.305	0.211
Gemini 3 Pro	—	×	0.152	0.190	0.153
Gemini 2.5 Flash	—	×	0.102	0.132	0.098
GPT-5 Mini	—	×	0.098	0.115	0.096
Gemini 3.1 Flash Lite	—	×	0.095	0.131	0.090
GPT-5	—	×	0.048	0.062	0.048
GPT-5 Nano	—	×	0.024	0.038	0.021
Open-Weight ($\geqslant$20B)					
GLM-5.2	753B	✓	0.186	0.226	0.186
Gemma-4-31B	31B	✓	0.101	0.131	0.097
Qwen3.5-27B	27B	✓	0.091	0.116	0.088
Qwen3.5-122B	122B	✓	0.091	0.124	0.083
Qwen3.5-35B-A3B	35B	✓	0.085	0.115	0.081
GPT-OSS-20B	20B	✓	0.070	0.095	0.065
GPT-OSS-120B	120B	✓	0.069	0.095	0.062
MiniMax-M2.7	229B	✓	0.054	0.078	0.050
Llama-3.3-70B	70B	✓	0.012	0.016	0.014
Open-Weight (<20B)					
CodeScout-14B	14B	✓	0.044	0.065	0.039
Qwen3.5-9B	9B	✓	0.043	0.058	0.039
Gemma-4-E2B	2B	✓	0.039	0.045	0.042
Gemma-4-E4B	4B	✓	0.034	0.039	0.034
Granite-4.0-350M	350M	✓	0.001	0.001	0.001
Granite-4.0-Micro	3B	✓	0.000	0.000	0.000
Granite-4.0-1B	1B	✓	0.000	0.000	0.000
Specialized Models					
Antares-3B	3B	✓	0.223	0.298	0.219
Antares-1B	1B	✓	0.209	0.268	0.221
Antares-350M	350M	✓	0.135	0.144	0.176
Static Analysis Tools					
Semgrep	N/A	✓	0.086	0.091	0.155
Semgrep-CWE	N/A	✓	0.052	0.057	0.071
CodeQL	N/A	✓	0.023	0.025	0.030
Horusec	N/A	✓	0.020	0.021	0.038

The leaderboard shows substantial variation across systems, with no single category uniformly dominating. Because the table brings together general-purpose models, domain-specialized systems, and static-analysis baselines, the overall ordering should be read as a comparison of different approaches to repository search rather than as a single capability ranking. We therefore examine how model scale, training, and system design relate to localization performance in the analyses that follow.

Parameter count does not provide a reliable ordering among general-purpose models. Across open-weight and closed-source families, larger systems often underperform smaller counterparts, and the aggregate relationship between scale and File F1 is only moderate. The GPT family makes this especially visible: GPT-5.5 substantially outperforms GPT-5, while GPT-5 Mini also exceeds GPT-5 despite its smaller size. GPT-5.5's release explicitly considered cyber-use safety, suggesting that differences in model behavior and safety alignment across generations contribute to the ordering beyond parameter count alone. GPT-5's system card reports greater conservatism on dual-use cybersecurity tasks, including refusals in agentic security evaluations, while GPT-5 Mini performs better on Cyber Range evaluations [36].

More generally, general software-engineering capability is not a reliable proxy for cybersecurity performance: a system may underperform because safety alignment constrains work in a dual-use setting, or because security-specific reasoning is out of distribution relative to its software-engineering and coding training.

Static-analysis tools remain reliable, widely available open-source baselines: they outperform several language models, although their fixed rules provide limited coverage when the relevant implementation files are not known in advance. Their closest resource-efficient LLM counterparts, models below 10B parameters that can run on consumer hardware, do not achieve comparable localization performance without task-specific training. Thus, interactive search alone is insufficient at this scale: static-analysis tools remain competitive and are displaced only by models fine-tuned for vulnerability localization.

4.3. False Positive Verification Results

Table 3 summarizes Phase B results for all 27 evaluated models. The ranking differs substantially from Phase A, showing that successful localization does not by itself imply reliable verification. This contrast is visible even within the frontier systems: GPT-5.5 (xhigh), which leads Phase A localization, is comparatively prone to reporting files in patched repositories, whereas GPT-5 Nano shows the opposite profile, with weak localization but strong restraint. The Granite base models make the abstention caveat clear: high TNR can result from declining to engage with the task rather than from recognizing that no vulnerability is present. TNR therefore captures a distinct behavioral requirement, calibrated restraint when the repository is clean. Taken together, the two phases expose the difference between pursuing evidence of a weakness and withholding an unsupported claim, making their joint evaluation necessary for distinguishing useful security analysis from indiscriminate alerting.

These results show that current models capable of vulnerability localization are not uniformly calibrated against false positives. Because alert fatigue is a persistent concern in cybersecurity operations, improving localization without preserving the ability to recognize clean code is insufficient for practical use. As we build more capable agentic systems for cybersecurity, false-positive control should remain a first-class requirement for genuinely usable defensive assistance.

4.4. Scaling Analysis

To summarize the association between model size and localization performance, we fit an ordinary least squares (OLS) regression, which estimates the linear trend that minimizes the squared differences between observed and predicted File F1 values. The fit uses the 13 general-purpose open-weight models

Table 3 | Phase B verification results for all 27 evaluated models, reported as true negative rate (TNR). TNR is the fraction of patched repositories correctly identified as clean; lower values indicate more false-positive vulnerability reports.

Model	**Params**	**TNR**
Frontier (Closed-Source)		
GPT-5 Nano	—	0.868
GPT-5	—	0.743
GPT-5 Mini	—	0.702
Gemini 3.1 Flash Lite	—	0.632
Gemini 2.5 Flash	—	0.392
Gemini 3 Pro	—	0.329
GPT-5.5 (xhigh)	—	0.279
GPT-5.5 (default)	—	0.192
Open-Weight ($\geqslant$20B)		
Qwen3.5-122B	122B	0.750
Qwen3.5-27B	27B	0.748
Llama-3.3-70B	70B	0.745
Qwen3.5-35B-A3B	35B	0.740
GPT-OSS-120B	120B	0.720
GPT-OSS-20B	20B	0.719
Gemma-4-31B	31B	0.682
GLM-5.2	753B	0.582
MiniMax-M2.7	229B	0.321
Open-Weight (<20B) and Specialized		
Granite-4.0-1B	1B	1.000
Granite-4.0-350M	350M	0.976
Granite-4.0-Micro	3B	0.863
Gemma-4-E4B	4B	0.845
Qwen3.5-9B	9B	0.814
Gemma-4-E2B	2B	0.755
CodeScout-14B	14B	0.563
Specialized Models		
Antares-3B	3B	0.034
Antares-1B	1B	0.008
Antares-350M	350M	0.012
Static Analysis Tools		
Semgrep	N/A	0.912
Semgrep-CWE	N/A	0.996
CodeQL	N/A	0.988
Horusec	N/A	0.980

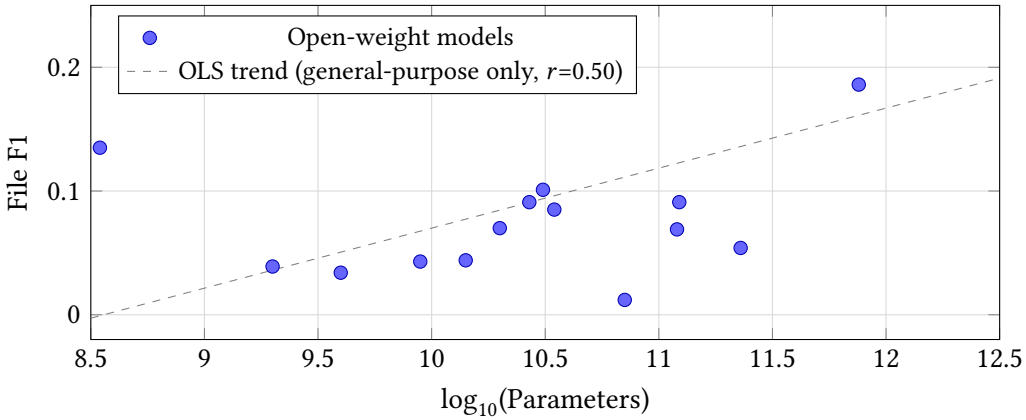

Figure 2 | File F1 as a function of $\log_{10}$(parameter count) for open-weight models. The ordinary least squares (OLS) trend is fitted only to the 13 general-purpose models with available parameter counts; domain-specialized models are shown separately. Mixture-of-experts models are positioned by total parameter count rather than active parameter count.

with available parameter counts. Figure 2 examines the relationship between File F1 and parameter count for open-weight systems. The fitted relationship is positive but accompanied by substantial scatter: model size provides some signal, yet does not determine localization quality. Models with comparable sizes can perform quite differently, and smaller systems can exceed much larger ones, yielding a non-monotonic scaling pattern. This pattern also appears in software-engineering localization. CodeScout [38] reports that smaller models trained directly on file-level localization rewards can outperform much larger base models, while SWE-Bench Pro shows similar reversals across general-purpose systems [7]. These comparisons motivate asking whether the same scaling behavior holds when localization is conditioned on a security weakness.

Software-engineering localization and cybersecurity localization are related but distinct capabilities. VLoc Bench requires an agent to connect a CWE's security semantics to concrete implementation patterns in an unfamiliar repository and then identify the files implicated by that reasoning. General coding ability may support repository navigation, but it does not guarantee the security reasoning needed to recognize and localize the relevant weakness, particularly under dual-use alignment. The benchmark therefore measures security-aware localization rather than general-purpose scaling alone.

Evaluation cost. Evaluation cost varies substantially across systems. A full 500-task Phase A sweep costs between $0.60 for Antares-350M (approximately 11 minutes on a single H100) and $141 for GPT-5.5 xhigh via the OpenAI API (approximately 5 hours), a 170× range. GLM-5.2 via OpenRouter provides an intermediate reference at $12.50 and approximately 50 minutes. Local open-weight models up to 31B parameters, served with 16 parallel workers on a single H100, complete evaluation in under one hour, making continuous benchmarking practical within a standard CI pipeline stage. Per-system cost and runtime details are provided in Appendix D.

5. Analysis

Aggregate performance leaves the sources of difficulty unresolved. We therefore examine which repository properties make localization difficult, whether model characteristics explain performance beyond the repository itself, which search behaviors accompany successful localization, and how unsuccessful trials differ in their decisions and tool use. Together, these analyses connect the properties of the

searched codebase to the behavior of the searching agent, providing a basis for interpreting File F1 beyond the leaderboard.

5.1. Repository Difficulty Regression

To identify which properties of a repository make a task intrinsically difficult, we fit a Lasso regression using features available before any model is evaluated. Lasso is a regularized linear regression that adds an ℓ_1 penalty to coefficient magnitudes, shrinking weak associations toward zero and yielding an interpretable set of predictors. The fitted regression predicts per-task File F1 from repository structure and task metadata, rather than from agent actions or outputs. This isolates benchmark-intrinsic difficulty from the behavior of the evaluated systems.

For each of the 500 tasks, we extract 42 features from the vulnerable repository snapshot and its associated metadata. Thirty-four repository-structural features summarize the source tree, including file counts and lines-of-code (LOC) distributions, directory topology, project signals such as CI, Docker, tests, and READMEs, and compressed and uncompressed size. Eight additional features describe vulnerability metadata, including CVSS score, the number of ground-truth files, associated CWEs and CVEs, disclosure year, and CWE super-categories covering input validation, memory, and resource management. We add 10 one-hot indicators for ecosystem membership and severity, giving 52 variables in total. The number of ground-truth files is included as a task-complexity covariate: it indicates how many files must be found, not which files constitute the answer.

The regression uses approximately 40,000 model–task–run observations (27 models × up to 500 tasks × 3 runs), with per-task File F1 as the outcome. We fit LassoCV with 5-fold cross-validation over 100 log-spaced alpha values and standardize all features to zero mean and unit variance before fitting. The R^2 values reported in Table 4 are the in-sample fit and the corresponding five-fold cross-validated fit of this repo-only regression.

Feature	Coef.	Direction
`top5_loc_share`	+0.082	easier
`repo_size_bucket`	+0.038	easier
`log10_zip_size_kb`	−0.037	harder
`total_source_loc`	+0.025	easier
`max_file_loc`	−0.023	harder
`pct_files_depth_1`	+0.018	easier
`eco_go`	−0.017	harder
`has_ci`	−0.016	harder
`depth_std`	+0.016	easier
`pct_large_files`	−0.015	harder

Table 4 | Top 10 repo-level Lasso coefficients (52 features, R^2=0.188, CV=0.164). Positive coefficients indicate features that make localization easier; negative indicate harder.

The repo-only regression explains R^2=0.188 of the observed File F1 variation in-sample and R^2=0.164 under five-fold cross-validation, with 48 of 52 features retaining nonzero weight. Table 4 reports the strongest associations. Code concentration is the clearest signal: repositories whose lines are concentrated in a small number of large files are easier to localize, consistent with search strategies reaching relevant code in fewer commands. In contrast, larger compressed repositories, deeper directory structures, and Go projects are associated with lower File F1. The ground-truth file count carries negligible weight (−0.006), suggesting that the number of files to find is less informative than the

structure in which those files are embedded.

The positive coefficient for `repo_size_bucket` and negative coefficient for `log10_zip_size_kb` are not interpreted as opposing size effects: their signs reflect multicollinearity between a continuous size measure and its bucketed counterpart. After controlling for continuous size, the bucket variable captures a residual nonlinear pattern in which mid-sized repositories tend to be easier than both extremes. Severity and CWE category contribute negligible total weight (sum $|\beta| = 0.010$). The similar in-sample and cross-validated fits suggest limited overfitting. Bootstrap resampling over 100 iterations provides a second check on coefficient stability. The leading predictor is nonzero in every resample and has a 95% confidence interval of [+0.072, +0.091].

Repository structure is therefore the strongest measured source of task difficulty, although it does not explain all of the variation. The hardest cases are shaped less by the number of labeled files than by how widely relevant code is distributed through the repository, motivating the combined analysis of repository and model factors below.

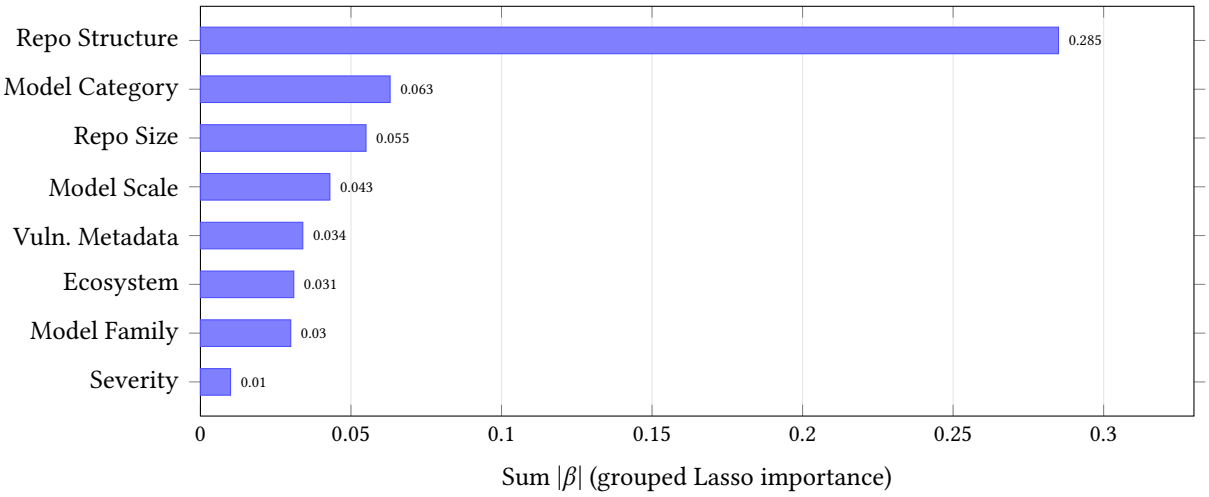

Figure 3 | Grouped Lasso importance from the combined regression (67 features, R^2=0.241, ~40k samples). Repository structure carries 4.5× more summed coefficient weight than model category.

5.2. Combined Regression: Structure Dominates

To test whether model identity explains performance beyond repository structure, we combine 52 repository-level and 15 model-level features in a single Lasso regression. The combined regression uses 67 predictors, achieves R^2=0.241 in-sample and R^2=0.192 under cross-validation, and retains 57 nonzero coefficients. Figure 3 compares feature groups by their summed absolute coefficient magnitudes. Repository structure carries 4.5× more coefficient weight than model category (0.285 vs. 0.063).

To test whether this result depends on the Lasso penalty or on how feature-group importance is measured, we repeat the comparison using two alternatives. Elastic Net combines the Lasso's coefficient-shrinking penalty with an ℓ_2 penalty; with an l_1-ratio of 0.5, the two penalties contribute equally. This model gives a 5.7$imes$ ratio between repository structure and model category. Group-level permutation importance instead measures the reduction in R^2 when the values of one feature group are shuffled while the remaining features are left unchanged; this gives a 3.2$imes$ ratio.

The ratios are also affected by how the features vary in the dataset. Repository features vary across 500 tasks, whereas model features repeat across only 27 systems, so the two groups do not contribute comparable amounts of independent variation. Permutation importance reduces the dependence on

coefficient magnitude while preserving the same qualitative ordering. Taken together, the Lasso, Elastic Net, and permutation analyses consistently associate repository structure with more predictive information than model category, but these ratios describe predictive association rather than a causal contribution.

Model identity in isolation. We next ask how much task performance can be predicted from model descriptors without using repository features. We fit a model-only Lasso with 15 predictors covering parameter count, model category, and model family. The regression explains R^2=0.053 of the observed variation in-sample and R^2=0.007 under cross-validation, as reported in Table 5. Thus, model descriptors explain little of the task-level variation. This result is compatible with the moderate model-aggregate correlation (r=0.50, Section 4.4), which ranks 13 model-level means. The model-only regression asks a different question: whether those descriptors predict which particular tasks a system will solve. The aggregate correlation describes average ordering, whereas the cross-validated regression tests per-task explanatory power. Its near-zero cross-validated R^2 indicates that model descriptors shift average performance but have limited ability to predict task-specific outcomes. These coefficients should be read as associations, not causal effects. In this regression, the specialized-training indicator has the largest positive coefficient, followed by parameter count (Table 5).

Feature	Coef.	Interpretation
`cat_specialized`	+0.053	domain training helps
`log10_params`	+0.043	scale helps
`fam_llama`	−0.015	underperforms at scale
`fam_gemma`	+0.006	slight edge
`cat_open-small`	−0.005	small generalists struggle
`cat_granite`	−0.005	untrained baselines fail

Table 5 | Model-level Lasso coefficients (15 features, R^2=0.053, CV=0.007). Coefficients are descriptive effect sizes; the near-zero CV indicates model identity has negligible entry-level predictive power.

Taken together, these results support VLoc Bench's measurement premise: cybersecurity localization reflects both the complexity of the repository being searched and the model's cybersecurity understanding and capability. Repository structure accounts for more variation than model identity, while model identity still shifts average performance but provides little information about which individual tasks a system will solve. The benchmark's repositories therefore define the evaluation landscape, while models serve as comparative probes of cybersecurity capability within it. Holding this task distribution fixed while interchanging agent harnesses also makes VLoc Bench a system-level evaluation setting, supporting comparisons along both model-capability and system-design axes.

Repository and model summaries provide a useful first-order account of localization difficulty, but 76% of the observed variation remains unexplained. This residual likely reflects task-specific properties that aggregate features cannot capture, including code idioms, naming conventions, framework-specific import patterns, and whether the vulnerable logic is reachable through textual search. We next examine these structural effects more directly by decomposing performance across repository conditions.

5.3. Difficulty Decomposition

The regression identifies repository structure as a broad source of difficulty; we now examine how that difficulty is distributed across concrete repository conditions. We consider repository size, ecosystem, and the subset of tasks that no evaluated system solves.

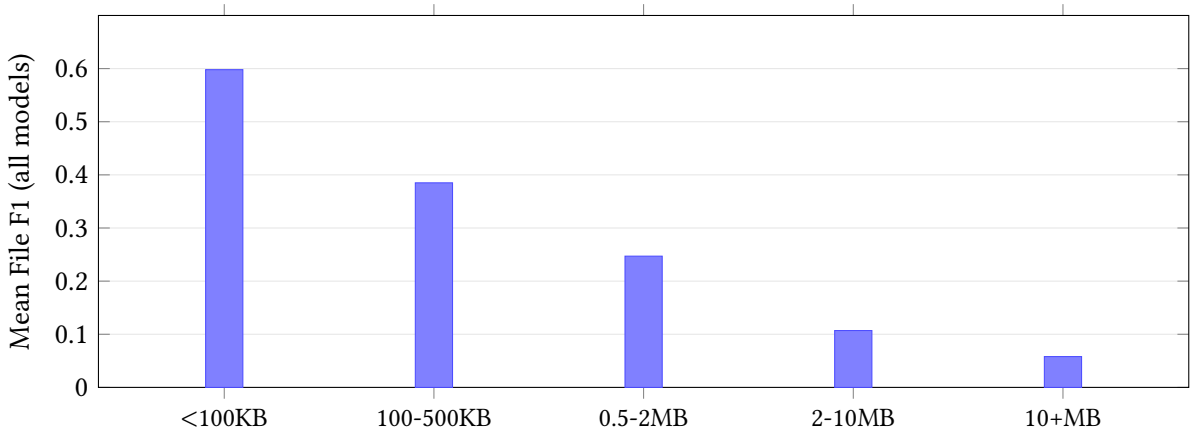

Figure 4 | Mean File F1 averaged across all 27 models, decomposed by repository size. Performance decays sharply: the smallest repositories (<100 KB, n=20) yield 10× higher scores than the largest (10+ MB, n=223). The majority of tasks (63%) fall in the two hardest buckets ($\geqslant$2 MB).

Repository size. Repository size provides the clearest broad separation in difficulty. Mean File F1 falls from 0.598 for repositories smaller than 100 KB to 0.058 for repositories larger than 10 MB, a roughly 10× difference. The ordering is consistent across models: each model performs better on smaller repositories, while the relative ordering of models is mostly preserved across size buckets. This pattern is consistent with a larger search space reducing the signal available to search-based agents and making it harder for reasoning-based agents to retain the relevant code. It also parallels SWE-Bench Pro's finding that localization performance decreases as the number of ground-truth files grows [7]. Repository scale therefore provides a strong, model-independent indicator of localization difficulty.

Ecosystem. The size gradient does not, however, account for all of the observed differences. The ecosystem breakdown shows a clear ordering: Maven tasks are hardest on average, pip and npm tasks are easiest, and Go lies between these groups. Go is especially consequential for the benchmark because it accounts for 43% of tasks, making its lower average performance a major component of the overall difficulty distribution. We interpret this variation as a consequence of ecosystem-specific coding conventions and project organization, which shape how relevant implementation is distributed across files and directories while remaining intertwined with overall repository size.

Unsolved tasks. The interaction between repository scale and ecosystem is clearest among the tasks that remain unsolved. In 38.4% of tasks, every model receives File F1 = 0; large Go and Maven repositories account for 47% and 31% of this subset, respectively. Additional models or larger parameter counts do not remove this subset under the evaluated settings. The result is a persistent hard core of repository conditions that current localization systems do not reliably handle.

Taken together, these results show that difficulty is shaped by both the scale of the search space and the way code is organized within it. Repository size supplies the strongest general gradient, while the unsolved subset shows that current systems do not yet cover the full range of repository conditions. We next examine the search behavior behind these outcomes, asking whether successful localization is associated with deeper exploration, more targeted command use, or more reliable interaction with the tool interface.

5.4. Behavioral Analysis

Aggregate scores describe whether a system succeeds, but not how it searches the repository. We therefore examine three observable properties of the recorded traces: exploration depth, command allocation, and tool reliability. These analyses operate on model-level behavior and identify associations with performance.

Exploration depth and performance. We compute two summaries for each model from its Phase A traces: the mean number of terminal commands issued per task-run and the mean File F1 over those same task-runs. A Pearson correlation across the 27 paired model-level summaries gives r=0.72 (p<0.001). Models that use more of their available command budget therefore tend to achieve higher mean File F1, with weaker systems more likely to terminate after limited exploration and stronger systems more likely to continue searching. This relationship should not be read as evidence that additional commands alone improve every task: among the strongest systems, command counts are already near the budget limit, so their remaining differences reflect the quality of the search rather than its duration. The pattern suggests that VLoc Bench rewards agentic, multi-turn, goal-directed tool use: each additional inspection can add substantial code to the context, making long-horizon search and context management central to effective localization.

Command strategy diversity. Agents also differ in how they spend their commands. We assign every terminal operation in the Phase A traces to one of three categories: search commands (e.g., `grep`, `rg`, and `find`), file-reading commands (e.g., `cat`, `head`, and `sed`), or exploratory commands (e.g., `ls` and `tree`). For each model, we divide the number of commands in each category by its total number of commands, producing a three-part command-use profile, and compare that profile with the model's mean File F1. Higher-performing systems generally devote more of their interaction to targeted search and reading, while lower-performing systems spend more of their budget exploring the directory structure without a focused follow-up. Effective behavior is not tied to one fixed allocation: strong systems combine search and reading in different proportions, but they use exploration to narrow the search space rather than as an end in itself. This pattern is consistent with CodeScout's observation that trained agents converge toward targeted ripgrep-based search across different starting tool policies [38].

Tool reliability. The value of a command strategy also depends on whether the selected operations execute successfully. From each model's Phase A traces, we compute a tool-error rate by dividing invalid commands, timeouts, and permission errors by the model's total issued commands, then compare this rate with its mean File F1. Across the observed range, models with error rates in the low single digits tend to achieve higher localization scores, whereas rates above 15% coincide with lower scores. Failed commands consume interaction budget without yielding repository evidence, which helps explain this association. Reliable tool use therefore appears necessary for effective search, but it is not sufficient: a model can execute commands cleanly and still fail to connect the CWE description to the relevant implementation. The behavioral evidence points to productive interaction, rather than command volume alone, as the more useful distinction.

Taken together, these traces show that higher localization performance is associated with sustained exploration, targeted search and reading, and reliable tool use. Command volume alone is not sufficient.

5.5. Failure Mode Taxonomy

To complement File F1, we classify every non-perfect Phase A trial (File F1 < 1.0) by the behavior recorded in its evaluation trace. Prior work such as SWE-Bench Pro uses model-based trajectory

interpretation to assign semantic failure categories [7]; VLoc instead uses command counts, precision, recall, and submission status. This summarizes how a trial ended without requiring an additional model to interpret the trajectory. The same trace produces the same label, which makes the classification reproducible across the ~40,000 trials. Its limitation is that trace metadata does not identify semantic intent: it can distinguish early termination from an incorrect submission, but not whether an incorrect search reflected a misunderstood CWE or an unproductive choice of directory.

At Level 1, a failed trial is either *Abstained*, meaning that the model submitted no files, or *Submitted*, meaning that it submitted at least one file path but did not achieve perfect File F1. Level 2 then separates these groups by observable behavior. Abstentions are *Premature termination* when the model issues at most three terminal commands, *Exhausted budget* when it issues at least 13 of the 15 allowed commands without submitting, and *Inconclusive search* otherwise. A forced termination after three consecutive turns without a tool call follows the same command-count rule.

Submitted failures are divided by overlap with the ground-truth files and by submission breadth. *Partial recall* denotes a submission with recall > 0 and precision $\geqslant 0.5$, while *Overly broad* denotes recall > 0 and precision < 0.5. Among submissions with no correct files, *Wrong files* denotes submissions that are not overly broad, and *Overly broad, zero recall* denotes submissions containing more than twice the ground-truth file count. These conditions make the Level 2 leaves mutually exclusive. Every non-perfect Phase A trial therefore receives exactly one deterministic label.

Aggregate failure distribution. To characterize how localization attempts fail, we aggregate the deterministic labels across all non-perfect Phase A trials. Table 6 summarizes these outcomes across the 27 models and three runs per task, first separating trials that abstain from those that submit files and then dividing each group into more specific behavioral categories. Abstention accounts for the larger share of failures (59.1%), while submitted failures account for the remainder (40.9%). The two most common Level 2 categories are *Exhausted budget* (32.3%) and *Wrong files* (27.4%), covering trials that search without reaching a submission and trials that submit without identifying a correct file, respectively.

Level 1	**Level 2**	**%**
Abstained (59.1%)	Exhausted budget	32.3
	Premature termination	16.9
	Inconclusive search	9.9
Submitted (40.9%)	Wrong files	27.4
	Partial recall	8.5
	Overly broad	2.6
	Overly broad (zero recall)	2.4

Table 6 | Aggregate failure mode distribution across all 27 models (~38,500 non-perfect trials, 3 runs per model–task pair). Exhausted budget and Wrong files together account for 59.7% of failures.

Premature termination and partial recall represent different failure behaviors. Premature termination accounts for 16.9% of failures and is associated with systems that stop after limited exploration, sometimes following malformed tool calls or difficulty parsing the initial repository structure. It is concentrated among smaller systems and uncommon among frontier systems. Partial recall accounts for 8.5% of failures and is more common among stronger systems: these trials identify at least one relevant file but miss others, showing that multi-file localization remains difficult even when the search reaches the correct part of the repository.

		Abstained (%)			Submitted (%)			
Model	**F1%**	**Prem.**	**Exh.**	**Incon.**	**Wrong**	**Part.**	**Broad**	**Br.(0)**
GPT-5.5	10.0	0.0	15.6	0.2	46.4	23.2	2.8	1.8
Antares-3B	9.8	0.0	2.4	0.0	57.2	22.6	4.4	3.6
Antares-1B	9.6	0.4	0.2	0.0	52.2	15.8	12.2	9.6
GLM-5.2	8.6	0.0	28.6	0.8	38.4	16.2	5.0	2.4
Gemini 3 Pro	5.3	0.0	26.9	0.2	41.6	15.2	6.7	4.2
Qwen3.5-122B	5.4	0.0	66.2	4.0	16.2	7.4	0.6	0.2
MiniMax-M2.7	2.4	0.2	28.0	0.2	59.5	6.0	0.6	3.0
Llama-3.3-70B	0.6	75.3	0.0	0.8	19.8	1.2	0.4	1.8
CodeScout-14B	2.7	5.1	1.8	52.0	34.3	3.9	0.0	0.2
Antares-350M	3.8	0.8	0.0	0.2	53.6	7.4	17.0	17.2

Table 7 | Failure mode profiles for 10 representative models (percentages of all 500 trials, including successes). F1% = perfect-score rate. Prem. = premature termination, Exh. = exhausted budget, Incon. = inconclusive search, Wrong = wrong files entirely, Part. = partial recall, Broad = overly broad with some recall, Br.(0) = overly broad with zero recall. Each model exhibits a distinct dominant failure mode.

Per-model failure profiles. Table 7 shows that similar aggregate scores can arise from different failure distributions. Three recurring profiles are visible. *Commit-heavy* systems rarely abstain but often submit incorrect files. *Cautious exhausters* use most of their command budget without submitting, indicating that they search extensively but do not reach a decision within the available horizon. *Premature terminators* stop after limited interaction, leaving little evidence that they engaged with the repository. These profiles summarize observable behavior rather than fixed model classes, and a single system may exhibit more than one across tasks.

Taken together, the failure profiles show that File F1 can conceal materially different system behaviors. A model may stop after limited exploration, exhaust its budget without submitting, submit files with no overlap, or identify only part of a multi-file ground truth. These outcomes point to different limitations in tool use, search strategy, and submission decisions, but they are collapsed into the same aggregate score. The deterministic taxonomy preserves these distinctions across all trials, allowing future systems to be evaluated by whether they reduce premature termination, non-convergent search, incorrect submissions, and incomplete recall.

6. Conclusion

We introduce VLoc Bench to measure agentic vulnerability localization over complete software repositories, a capability that existing code and security benchmarks often leave implicit. Its paired repository states and agentic interface evaluate two complementary decisions: identifying the implementation files associated with a weakness when it is present, and recognizing that the repository is clean after the fix. File F1 and TNR make both outcomes measurable under the same task conditions.

The results show that repository-scale localization remains far from solved. The strongest system achieves 0.229 File F1, and 38.4% of tasks receive no correct localization from any evaluated model. Beyond this aggregate difficulty, the analyses identify how performance varies across repositories and systems. Repository structure explains more variation than model descriptors, while parameter count alone provides only a moderate guide to performance. The behavioral and failure analyses further show

that localization depends on sustained, targeted, and reliable tool use, and that strong localization does not guarantee restraint on patched code. These findings separate security-aware repository analysis from general software-engineering performance and from a model's willingness to submit an answer.

Most existing agentic cybersecurity benchmarks evaluate capabilities through CTF-style tasks, vulnerability reproduction, or exploit generation [33, 41, 42]. These benchmarks measure important cybersecurity capabilities, but their success criteria center on triggering or exploiting a weakness rather than locating and triaging the affected implementation. VLoc Bench complements this literature by evaluating a more neutral defensive capability: source-code localization under a CWE-only prompt and read-only repository access, with Phase B testing whether the agent avoids false positives after remediation. This positioning isolates a distinct stage of security work without requiring agents to modify, execute, or exploit the target code. We hope this work helps address this evaluation gap and supports both the agentic AI evaluation and cybersecurity communities.

7. Limitations and Future Work

Limitations. VLoc Bench is scoped to the vulnerability associated with each advisory, rather than to the security of the repository as a whole. In Phase A, patch-touched implementation files provide a reproducible localization target, but may not capture every file relevant to the vulnerability. In Phase B, the patched snapshot establishes that the recorded vulnerability has been remediated, not that no other vulnerability exists. This limits evaluation of general-purpose static analyzers such as Semgrep, CodeQL, and Horusec: a finding outside the benchmark target may be valid but cannot be adjudicated without an additional oracle. Phase A is sensitive to both incomplete and overly broad localization through File F1, while Phase B provides only a binary target-vulnerability judgment. The benchmark remains reproducible through fixed repository snapshots, deterministic scoring, and repeated runs, although Phase B also conflates explicit abstention with failures such as turn exhaustion, API errors, or refusals when no submission is produced.

Future work. We plan to extend VLoc Bench by introducing levels of task context, from the current CWE-only setting to increasingly detailed vulnerability descriptions, to measure how additional security information changes localization performance. We also plan to strengthen Phase B by validating additional model findings through manual review, proofs of concept, or other behavioral oracles, allowing genuine discoveries to be separated from false positives. More broadly, vulnerability localization is only one stage of a security analyst's workflow. Future benchmarks should evaluate these intermediate capabilities intrinsically, alongside end-to-end extrinsic evaluations, so that progress can be measured not only by whether a system ultimately detects or repairs a vulnerability, but by which parts of the security-analysis pipeline it can reliably assist with.

8. Ethics and Responsible Use

VLoc Bench is constructed entirely from public GitHub Security Advisories describing vulnerabilities that are already disclosed and already patched; we introduce no new vulnerability information and release no exploits. The benchmark evaluates *localization*—identifying which files instantiate a known weakness class—not exploitation or weaponization, and all agent interaction is confined to read-only terminal access within an isolated, network-disabled sandbox. Our intent is to advance defensive tooling: measuring and improving the ability of AI systems to help maintainers and security teams find vulnerable code faster. We note the dual-use nature of any vulnerability-analysis research and emphasize that automated localization should augment, not replace, expert human review; as our results show, automated localization at repository scale remains an open problem, and outputs from any current

system should be treated as leads for human review rather than conclusions. Because every entry corresponds to a fix that is already public, the benchmark does not expand the attack surface of the included projects beyond what is already documented in the public advisory record.

References

[1] Md Basim Uddin Ahmed, Nima Shiri Harzevili, Jiho Shin, Hung Viet Pham, and Song Wang. Secvuleval: Benchmarking llms for real-world c/c++ vulnerability detection, 2025. URL https://arxiv.org/abs/2505.19828.

[2] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010. doi: 10.1145/1646353.1646374.

[3] 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.

[4] Aili Chen et al. The minimax-m2 series: Mini activations unleashing max real-world intelligence, 2026. URL https://arxiv.org/abs/2605.26494. We evaluate the MiniMax-M2.7 point release; this report describes the M2 series.

[5] Wuya Chen, Yihao Yang, Yang Cao, and Yue Lin. Codegrep: An rl-trained retrieval agent for llm coding agents, 2026. URL https://arxiv.org/abs/2608.05886.

[6] Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8697–8727, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.426. URL https://aclanthology.org/2025.acl-long.426/.

[7] 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.

[8] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1729–1741. IEEE Computer Society, 2025. doi: 10.1109/ICSE55347.2025.00038.

[9] Tarek Elsayed, Shiping Yang, Eunsong Koh, Sanika Goyal, Vincent Huang, Paul Ngo, Nathan Young, Mohammad Omidvar Tehrani, Alvyn Kang, Arnell Kang, Zeyu Chen, Angélica Moreira, Xuan Feng, Angel X. Chang, Nick Sumner, and Steven Y. Ko. Rustmizan: A compilable, contamination-aware benchmarking framework for rust vulnerabilities, 2026. URL https://arxiv.org/abs/2607.04729.

[10] Gemini Team, Google. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodal, long context, and next generation agentic capabilities, 2025. URL https://arxiv.org/abs/2507.06261.

[11] Gemma Team. Gemma 4 technical report, 2026. URL https://arxiv.org/abs/2607.02770.

[12] GitHub. Codeql: Semantic code analysis. https://codeql.github.com, 2025. Accessed: 2026-07-01.

[13] GitHub. GitHub advisory database. https://github.com/advisories, 2026. Accessed: 2026-09-03.

[14] GLM-5 Team. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763. We evaluate the GLM-5.2 point release; this report describes the GLM-5 family.

[15] Google DeepMind. Gemini 3.1 flash-lite model card. https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-1-Flash-Lite-Model-Card.pdf, May 2026. Accessed: 2026-09-03.

[16] Google DeepMind. Gemini 3 pro model card. https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Pro-Model-Card.pdf, May 2026. Accessed: 2026-09-03.

[17] Granite Team, IBM. Granite 4.0 language models. https://huggingface.co/collections/ibm-granite/granite-40-language-models-6811a18b820ef362d9e5a82c, October 2025. Apache License 2.0. Model documentation at https://www.ibm.com/granite/docs/. Accessed: 2026-09-03.

[18] Kexing Ji, Jiachen Liu, Enze Hu, Cuiyun Gao, Keke Lian, Yongheng Liu, Lei Zhang, Tian Dong, Hao Chen, and Wang Bin. Vulngym: Benchmarking coding agents for repository-level vulnerability detection, 2026. URL https://arxiv.org/abs/2608.02001.

[19] Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. Issue localization via llm-driven iterative code graph searching, 2025. URL https://arxiv.org/abs/2503.22424. Accepted at ASE 2025.

[20] 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?, 2024. URL https://arxiv.org/abs/2310.06770.

[21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP '23)*, pages 611–626. Association for Computing Machinery, 2023. doi: 10.1145/3600006.3613165.

[22] Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. Sec-bench: Automated benchmarking of llm agents on real-world software security tasks, 2025. URL https://arxiv.org/abs/2506.11791. NeurIPS 2025.

[23] Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T. Barr, Federica Sarro, Zhaoyang Chu, and He Ye. Contextbench: A benchmark for context retrieval in coding agents, 2026. URL https://arxiv.org/abs/2602.05892.

[24] Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. On the effectiveness of function-level vulnerability detectors for inter-procedural vulnerabilities, 2024. URL https://arxiv.org/abs/2401.09767.

[25] Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. On the effectiveness of function-level vulnerability detectors for inter-procedural vulnerabilities. In *46th International Conference on Software Engineering (ICSE)*, pages 157:1–157:12. ACM, 2024. doi: 10.1145/3597503.3639218.

[26] Ziyang Li, Saikat Dutta, and Mayur Naik. Iris: Llm-assisted static analysis for detecting security vulnerabilities. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=9LdJDU7E91.

[27] Yu Liu, Lang Gao, Mingxin Yang, Yu Xie, Ping Chen, Xiaojin Zhang, and Wei Chen. Vuldetectbench: Evaluating the deep capability of vulnerability detection with large language models, 2024. URL https://arxiv.org/abs/2406.07595.

[28] Llama Team, AI @ Meta. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783. We evaluate Llama-3.3-70B-Instruct, released as a model-card update to this family.

[29] OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

[30] OpenAI. GPT-5.5. https://developers.openai.com/api/docs/models/gpt-5.5, 2026. Model snapshot gpt-5.5-2026-04-23; reasoning effort settings none, low, medium (default), high, and xhigh; knowledge cutoff 2025-12-01. Accessed: 2026-09-03.

[31] Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388. We evaluate the Qwen3.5 generation, for which no separate base technical report has been released.

[32] Semgrep, Inc. Semgrep. https://semgrep.dev, 2025. Accessed: 2026-07-01.

[33] Minghao Shao, Sofija Jancheska, Meet Udeshi, Brendan Dolan-Gavitt, Haoran Xi, Kimberly Milner, Boyuan Chen, Max Yin, Siddharth Garg, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Muhammad Shafique. Nyu ctf bench: A scalable open-source benchmark dataset for evaluating llms in offensive security, 2025. URL https://arxiv.org/abs/2406.05590.

[34] Chihao Shen, Connor Dilgren, Purva Chiniya, Luke Griffith, Yu Ding, and Yizheng Chen. Secrepobench: Benchmarking code agents for secure code completion in real-world repositories, 2026. URL https://arxiv.org/abs/2504.21205.

[35] 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, 2026. URL https://arxiv.org/abs/2606.04460. ICML 2026.

[36] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card, 2026. URL https://arxiv.org/abs/2601.03267.

[37] Mingyang Sun and Guozhu Meng. Drea: Decoupled reasoning and exploration agents for repository-level vulnerability detection, 2026. URL https://arxiv.org/abs/2607.13439. Internetware 2026.

[38] Lintang Sutawika, Aditya Bharat Soni, Bharath Sriraam R R, Apurva Gandhi, Taha Yassine, Sanidhya Vijayvargiya, Yuchen Li, Xuhui Zhou, Yilin Zhang, Leander Melroy Maben, and Graham Neubig. Codescout: An effective recipe for reinforcement learning of code search agents, 2026. URL https://arxiv.org/abs/2603.17829.

[39] The MITRE Corporation. CWE: Common weakness enumeration. https://cwe.mitre.org, 2026. Version 4.20, released 2026-04-30. Sponsored by CISA and managed by HSSEDI. Accessed: 2026-09-03.

[40] Supriti Vijay, Aman Priyanshu, Didier Chapoteau, Arthur Goldblatt, Jianliang He, Kimia Majd, Fraser Burch, Baturay Saglam, Takahiro Matsumoto, Zhuoran Yang, and Amin Karbasi. Antares: Foundation models for agentic vulnerability localization, 2026. URL https://arxiv.org/abs/2608.02407.

[41] 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?, 2026. URL https://arxiv.org/abs/2605.11086.

[42] Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. Cybergym: Evaluating ai agents' real-world cybersecurity capabilities at scale, 2026. URL https://arxiv.org/abs/2506.02548.

[43] 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.

[44] Xin Ye, Razvan C. Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 689–699. Association for Computing Machinery, 2014. doi: 10.1145/2635868.2635874.

[45] 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.

[46] Andy K. Zhang, Joey Ji, Celeste Menders, Riya Dulepet, Thomas Qin, et al. Bountybench: Dollar impact of ai agent attackers and defenders on real-world cybersecurity systems, 2025. URL https://arxiv.org/abs/2505.15216. NeurIPS 2025 Track on Datasets and Benchmarks. Senior authors: Dawn Song, Dan Boneh, Daniel E. Ho, Percy Liang.

[47] Fuwei Zhang, Yanzhao Zhang, Mingxin Li, Dingkun Long, Lexiang Hu, Pengjun Xie, Zhao Zhang, and Fuzhen Zhuang. Core-bench: A comprehensive benchmark for code retrieval in the era of agentic coding, 2026. URL https://arxiv.org/abs/2606.11864.

[48] Shaoqiu Zhang, Yuhang Wang, Jialiang Liang, Yuling Shi, Wenhao Zeng, Maoquan Wang, Shilin He, Ningyuan Xu, Siyu Ye, Kai Cai, and Xiaodong Gu. Swe-explore: Benchmarking how coding agents explore repositories, 2026. URL https://arxiv.org/abs/2606.07297.

[49] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *34th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE, 2012. doi: 10.1109/ICSE.2012.6227210.

[50] Arastoo Zibaeirad, Marco Vieira, and Thomas Zimmermann. Autotrace: From patches to triggers via agentic interprocedural exploration, 2026. URL https://arxiv.org/abs/2607.12058.

[51] ZupIT. Horusec: Static application security testing. https://github.com/ZupIT/horusec, 2025. Apache License 2.0. Accessed: 2026-09-03.

A. Example Task

To make the task interface and scoring concrete, we show one representative Phase A task together with selected steps from a successful and an unsuccessful run. As in the main evaluation, the models receive only the CWE description and read-only access to the pre-push repository.

Task T0aMD9EF. The repository is `yahoo/serialize-javascript` (134 KB, npm ecosystem). The model receives:

> *CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') — The product does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users.*

The patch-derived label is `index.js`, the sole implementation file containing the affected serialization logic.

Successful run (GPT-5.5, xhigh). The full run contains 15 terminal commands over 17 turns. The representative steps below show the progression from repository inspection to submission:

1. lists the repository files with `find . -type f | sort`;
2. reads `index.js` in numbered segments using `nl` and `sed`;
3. searches for serialization and escaping logic with `rg`; and
4. submits `[index.js]`, obtaining **File F1 = 1.0**.

Unsuccessful run (Llama-3.3-70B). The agent inspects the directory structure and issues several broad searches, but does not read `index.js` in sufficient depth to identify the affected logic. It exhausts the terminal budget without submitting a file list, resulting in **File F1 = 0.0**. This run illustrates an abstention after extended but inconclusive exploration.

The two runs differ in how they allocate the same interface and command budget. Even on a small repository with a single labeled implementation file, success depends on connecting the CWE description to the relevant code and committing the correct file list.

B. Dataset Composition

VLoc Bench contains 500 tasks drawn from the natural composition of the GitHub Security Advisory database rather than from a deliberately balanced sampling scheme. We summarize the task distribution along two dimensions used in the main analysis: package ecosystem and CVSS severity. Ecosystem coverage is uneven: Go is the largest group (43%), followed by Maven (21%) and npm (18%), while Composer contributes one task (Figure 5). The severity distribution is concentrated in the High and Medium bands, which together account for 83% of tasks (Figure 6). These distributions define the empirical support for the per-ecosystem and severity analyses in Section 5.

C. Data and Code Availability

The complete benchmark and the code needed to reproduce its evaluation are publicly available at `https://github.com/cisco-foundation-ai/vulnerability-localization-benchmark`. The

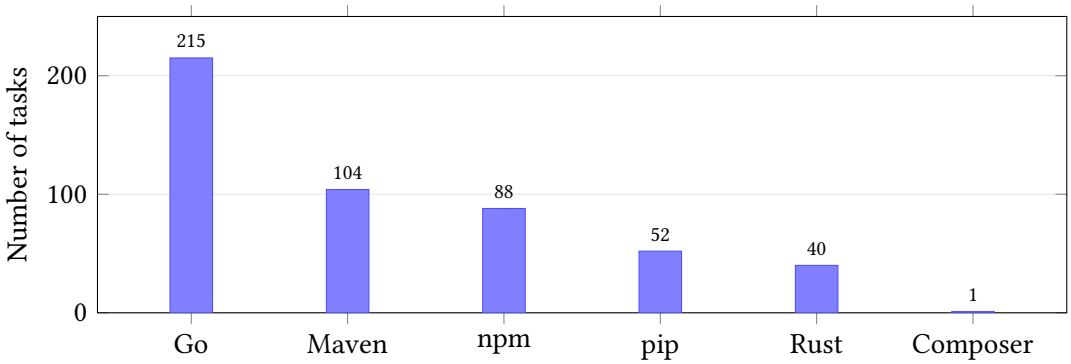

Figure 5 | Distribution of VLoc Bench tasks across six package ecosystems. Go is the largest group (43%), while Composer contributes one task.

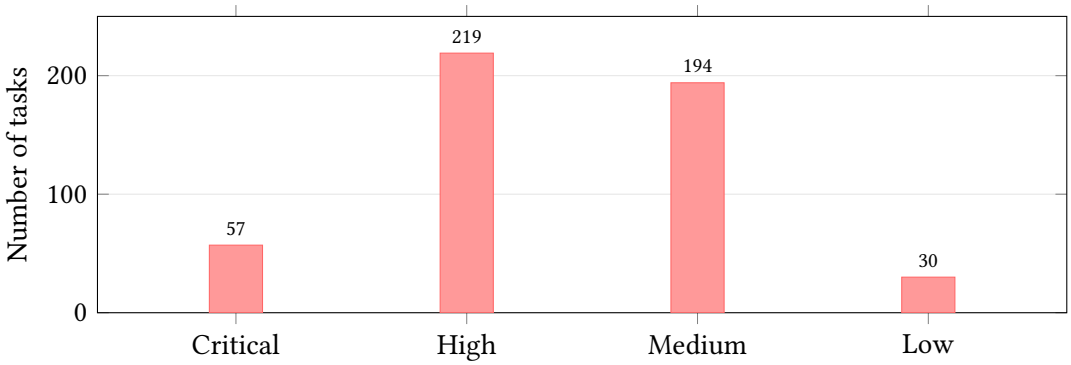

Figure 6 | Distribution of VLoc Bench tasks by CVSS severity. High- and Medium-severity tasks together account for 83% of the benchmark.

release includes the manifest, snapshot retrieval and verification tooling, evaluation harness, scoring implementation, and analysis scripts:

- **Data manifest** (`data/manifest.csv`): 500 tasks with repository metadata, commit SHAs for both vulnerable and patched states, CWE labels, ecosystem tags, and content-based MD5 checksums for reproducible verification.
- **Download and verification tooling**: scripts that retrieve repository snapshots as zip archives from GitHub at the exact commit SHAs recorded in the manifest, strip the archive prefix, resolve symlinks, and verify each task against its deterministic content MD5, computed over file paths and contents in lexicographic order.
- **Evaluation harness**: the full sandboxed evaluation pipeline including Docker image specification, agent protocol implementation, scoring code, and configuration files.
- **Scoring and analysis scripts**: code to compute File F1, TNR, abstain rates, and to reproduce the statistical analyses reported in Section 5.

All 500 tasks are downloadable directly from public GitHub repositories at the recorded commit SHAs. The release does not require gated access or authentication.

D. Evaluation Cost and Runtime

Model coverage. Anthropic models (Claude Opus, Sonnet, and Haiku) are not included because their evaluation cost exceeds the available budget. A full 500-task sweep under the standardized harness would exceed $600 at Claude Opus 4.8 pricing, while an unconstrained native-agentic configuration using Claude Code with subagent spawning costs $1,658 for the same sweep. Future versions of VLoc Bench will incorporate these models as budget permits.

Serving configuration. Open-weight models are served with vLLM (v0.19.1) on H100 GPUs using bfloat16 precision and a maximum sequence length of 32768. Native tool calling is enabled for Qwen3.5 and Gemma-4, and Antares uses a custom Granite chat template. Locally served models use temperature 0.3, a maximum of 16384 tokens per turn, and frequency penalty 0.3.

Closed-source models use their respective provider APIs with default temperature and penalty settings; GPT-5.5 xhigh additionally uses `reasoning_effort=xhigh`. Because provider defaults are not fully documented, the system prompt, tool definitions, agent loop, and command budget are the controlled variables across model evaluations. Static-analysis tools run directly on each repository with their default rulesets, while Semgrep-CWE uses CWE-specific rules matched to each task's classification.

Table 8 | Wall-clock runtime and estimated cost for a full 500-task Phase A evaluation. Local models are served via vLLM on a single H100 GPU with 16 parallel workers, with cost estimated from commodity H100 rental rates ($2–4/hour). API-model costs use provider pricing at the time of evaluation (June 2026).

System	Runtime	Total Cost	Cost/Task
Antares-3B (local, H100)	~15 min	$0.82	$0.002
Antares-1B (local, H100)	~13 min	$0.71	$0.001
Antares-350M (local, H100)	~11 min	$0.60	$0.001
GLM-5.2 (OpenRouter API)	~50 min	$12.50	$0.025
GPT-5.5 xhigh (OpenAI API)	~5 hrs	$141.00	$0.282

E. Lasso Regression Feature List

Table 9 enumerates the 67 features used in the combined Lasso regression described in Section 5.2. The 52 repository-level features are extracted by static analysis of each vulnerable repository snapshot, while the 15 model-level features encode system identity and remain constant across tasks for a given system. All features are standardized to zero mean and unit variance before fitting.

The feature blocks preserve the distinction between task properties and system properties used throughout the analysis. Repository features vary across tasks, whereas model features repeat across the tasks evaluated for each system; this structure supports the repository-only, model-only, and combined specifications reported in Section 5.

Group	Features	Count
Repo Structure	source_file_count, total_source_loc, avg_file_loc, median_file_loc, max_file_loc, top5_loc_share, pct_large_files, directory_depth, avg_directory_depth, depth_p90, depth_std, n_dirs_depth_3plus, pct_files_depth_1, n_directories, file_ext_entropy, max_directory_fanout, test_to_source_file_ratio, pct_test_loc, n_languages, comment_line_ratio, has_readme, has_tests_dir, has_ci, has_docker, n_config_files, n_dependencies, n_build_scripts, n_dotfiles	28
Repo Size	log10_repo_size_kb, log10_zip_size_kb, log10_merge_size_kb, compression_ratio, size_delta_pct, repo_size_bucket	6
Vuln. Metadata	cvss_score, n_gt_files, n_cwes, n_cves, year, cwe_input_validation, cwe_memory, cwe_resource	8
Ecosystem	eco_go, eco_npm, eco_pip, eco_maven, eco_rust, eco_composer	6
Severity	sev_critical, sev_high, sev_medium, sev_low	4
Model Scale	log10_params	1
Model Category	cat_frontier, cat_open-large, cat_open-small, cat_specialized, cat_granite	5
Model Family	fam_gpt, fam_gemini, fam_qwen, fam_gemma, fam_gpt-oss, fam_antares, fam_granite, fam_llama, fam_other	9
Total		**67**

Table 9 | Complete feature inventory for the combined Lasso regression. Repository-level features are extracted once per task; model-level features are constant across tasks for a given system.