International Journal of Computer Trends and Technology
ISSN: 2231–2803 / https://doi.org/10.14445/22312803/IJCTT-V74I9P105

Volume 74 Issue 9, 44-59, September 2026
© 2026 Seventh Sense Research Group®

Original Article

Same Code, Different Findings: An Empirical Study of Run-to-Run Instability in Rule-Based and LLM-Based Security Scanners

Naga Raju Doddi

Technical Architect at Dell Technologies, USA.

Corresponding Author : nagarajudoddi9@gmail.com

Received: 22 July 2026 Revised: 30 August 2026 Accepted: 13 September 2026 Published: 30 September 2026

Abstract *- Security scanners in CI/CD pipelines are believed to give the same answers for the same code. When a finding disappears, it is treated as fixed, and when a scan stage finds no blocking vulnerabilities, the build proceeds to production. This study tests that assumption. It ran 350 rule-based scans of 12 projects with Opengrep and CodeQL under normal, low-memory and busy-CPU conditions, and scanned 172 labelled cases five times each with four LLMs. For the rule-based scanners, resource budgets silently change the results: in one project, 32% of the findings appear and disappear across runs of the same commit ID. This study shows that when LLM-based scanners are added as a build step in CI/CD, for 3 out of the 4 models, between 28.2% and 91.1% of the findings do not occur across same runs; setting the temperature to 0 reduces but doesn't remove the variation, and new models are no longer accepting a temperature setting. Unstable LLM findings are more likely to be off-target noise, even on novel code. One scan recovers only 39–98% of the findings of five scans, and keeping only findings reported in all five runs raises the share of true positives at almost no loss of recall. Inconsistency means opposite things for the two types of scanners: a lost detection for the rule-based engines and noise for the LLMs. These results call for CI build gates that treat incomplete scans as failures, and for remediation tracking that does not treat a missing finding as a fix.*

Keywords *- Continuous integration, DevSecOps, Large language models, Nondeterminism, Static application security testing.*

1. Introduction

Currently, in CI/CD pipelines, many teams follow the same approach: they run CI/CD on default or protected branches, not on all branches, so SAST tools run on every commit or pull request to these branches. In the CI/CD stages, if there are no blocking findings, such as high-severity vulnerabilities, the build passes and proceeds to the next stages in the pipeline. When a finding does not appear again, it is treated as fixed. All of this assumes that the same code gives the same findings.

Recent scenarios show the problem with findings in practice. For example, the first scan may report 2 critical and 4 high vulnerabilities. When the security team asks the developers to fix them, and the scan runs again because of a new commit or pull request, the new result is shared with the developers and the old findings are gone, so the scan suggests that the previous vulnerabilities were fixed. [1,2] This is very confusing for developers: which result is the source of truth, and which findings should they work on? When they start fixing some vulnerabilities, new findings appear, and old ones vanish without any code fix. The scans are inconsistent, and developers waste time in an endless loop of fixing vulnerabilities. [3,4] LLM-based scanners are even worse, because they give different results on every run, and some tools use two or three models whose output varies widely. [5,6] This leaves developers confused about which findings to fix and which to ignore, since many of the results are inaccurate or unstable. Other researchers studied SAST tools with only one scan per tool and shared the results, but this does not show whether the results change when the scan is repeated. [7,8] Others looked at randomness in LLM code generation, but not at security scanners in CI, which is where teams see this problem the most and where nobody has checked it. [5,9] Some studies looked at either rule-based scanners or LLM scanners, but not both on the same code. [10–12] This study does both, and its analysis shows how often findings change, why they change, and whether the changing findings are real or not.

This study selected 12 open-source projects in 5 languages and pinned each one to a single commit, so the code never changed between the scans. Every project was then scanned many times with Opengrep and CodeQL under three

 This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)

conditions: a normal runner, a runner with little memory, and a runner with a busy CPU, the last two resembling a crowded CI machine. To check whether the findings were actually correct, the study also used code with known answers: 140 labelled cases from the OWASP Benchmark [13] and 32 new cases written for this study. Four LLMs scanned each of these cases five times: one local model, Qwen, and three Claude models. The rule-based tools scanned the same cases so that the two kinds of scanners could be compared directly. The study asks five questions:

- RQ1 (Prevalence). How often do findings change across repeated scans of the same code?
- RQ2 (Causes). What causes the changes?
- RQ3 (Meaning). Are changing findings real vulnerabilities that were lost, or noise?
- RQ4 (Visibility). Would a CI pipeline notice when findings go missing?
- RQ5 (Mitigation). How many repeated runs does it take to capture what an LLM scanner can find, and does keeping only findings that recur across runs improve the results?

The two kinds of scanner change for different reasons, and their changing findings mean opposite things. The rule-based engines gave the same results every time as long as they had enough resources. When they ran into timeouts, memory limits or a busy CPU, real findings disappeared without any warning while the scan still reported success. LLM scanners vary by nature, and the findings that came and went were mostly noise. A finding's stability is therefore useful information, but only once it is known which kind of scanner produced it.

To the author's knowledge, this is the first study to measure how stable findings are for both rule-based and LLM scanners on the same labelled code. It shows that for rule-based tools, the cause lies in the environment rather than the engine, and that the resulting losses cannot be seen in the tool's exit code. It also shows how many runs an LLM scanner needs and that voting across runs improves its results.

This paper makes the following contributions:

1. To the author's knowledge, the first measurement of how stable security findings are for both rule-based and LLM scanners, run on the same labelled cases.
2. Evidence that changes in rule-based results come from resource limits (such as timeouts, memory limits and a busy CPU) and that CI exit codes do not reveal them.
3. Evidence that unstable findings mean opposite things for the two kinds of scanner, lost real detections for rule-based tools and mostly noise for LLMs, along with the observation that the newest LLMs no longer let users set the temperature.
4. An analysis of how many repeated runs an LLM scanner needs and how much voting across runs helps, together with instability broken down by weakness category and the cost of running each scanner.
5. Practical guidance for teams using these tools in CI, and a replication package with all scripts, pinned commits and raw results.

The rest of the paper is organized as follows. Section 2 gives background on security scanning in CI and on the two kinds of scanner. Section 3 reviews related work and compares this study with the closest prior studies. Section 4 describes the study design, including the projects, labelled cases, scanners and runner conditions, and explains how instability and correctness were measured. Section 5 answers the five research questions and reports instability by weakness category and the cost of scanning. Section 6 discusses what the results mean for practitioners and how they relate to earlier findings. Section 7 covers threats to validity, and Section 8 concludes.

2. Background

2.1. Security Scanning in CI Pipelines

In DevSecOps, security scanners run automatically as part of the CI/CD pipeline. [14] A typical setup runs a SAST tool on every change and fails the build when it reports a finding above a chosen severity level. Teams also compare scans over time to track fixes: a finding that appears in one scan and is missing from the next is counted as resolved or fixed. This logic assumes that scanning the same code twice gives the same result. Vendors acknowledge that this is not always true. Rule updates can add or resolve issues even when the code has not changed, [1] and fast or incremental scan modes examine fewer code paths than a full scan. [2] Developers rarely see any of this because the pipeline shows only the final result in the build output.

2.2. Rule-Based SAST Engines

Rule-based engines find vulnerabilities by matching code patterns or by running data-flow queries over the code. Opengrep, [15] an open-source fork of Semgrep, applies a 5-second timeout to each rule on each file. If a file hits three timeouts, Opengrep skips the rest of that file, and it also enforces internal memory limits. CodeQL [16] first builds a database from the source code and then runs its queries against it, with memory capped by the --ram setting. These limits are reasonable safeguards against runaway scans. Their side effect is that the same code can produce different results on a slower or busier machine.

2.3. LLM-Based Scanners

A growing number of tools ask an LLM to review the code for vulnerabilities, either on its own or as a filter on SAST output. [12,17] By default, LLMs generate each answer by random sampling, so the same request can return a different answer each time. [5] A vendor study found that nearly half of the findings reported only by the LLM appeared in just one of five identical runs. [6] For a CI pipeline that treats every scan

as definitive, this variation matters as much as it does for rule-based tools, though for a different reason.

3. Related Work and Research Gap

3.1. Effectiveness of SAST Tools and Developer Trust

Security tools often disagree with one another. Different software composition analysis tools can report different vulnerabilities for the same code. [18] Developers also cite false positives and unclear results as a main reason they avoid static analysis [3,4], and they often suppress warnings they do not trust. [19] Knowing whether a vanished finding was actually fixed also matters, because fixes can be incomplete or can bring a vulnerability back. [20] Recent studies show how much single tools miss. Six C analysers each missed between 47% and 80% of 192 real vulnerabilities, and combining the tools still left 30% to 69% undetected. [7] Seven Java SAST tools found only 12.7% of real vulnerabilities, far below what they achieve on benchmarks. [8] All of these studies judge a tool by what it reports in a single scan. This study shows that the scan itself can change from one run to the next, and that the resulting losses leave no trace in the output.

3.2. Actionable Warnings and Warning Tracking

A large body of work tries to predict which static-analysis warnings are worth acting on. Early approaches used features of the code, its history and its developers, and more recent ones use machine learning. [21–24] One study later showed that the accuracy reported in this area had been inflated by data leakage and by comparing a revision with a future one. [25] Other work matches warnings across versions of a project, so that triage decisions carry over and fixed issues can be identified. [26,27]

Both lines of work treat each scan as a reliable snapshot of the code. When a warning is missing from the later scan, it is assumed to have been fixed. The results of this study show that a warning can disappear even when the code is identical, which means some of the "fixed" issues these methods count may simply be findings the scanner lost.

3.3. Nondeterminism in Analysis Tools and Tests

The first large study of nondeterminism in static analysis tools examined Soot, WALA and FlowDroid and traced their varying results to how the tools are implemented. [28] This study differs in three ways. It looks at security scanners as they are used in CI, it shows that resource limits are the main cause of change, even when the engine itself is deterministic, and it checks whether unstable findings are correct.

A related line of work studies flaky tests, whose outcomes change even though the code does not. [29,30] Unstable security findings can be seen as flaky security tests. Closest in approach is a testing framework for configurable static analysers, which found 74 bugs where FlowDroid, Soot, WALA and DOOP produced results that contradicted their own configurations. [31] That work varies the configuration to expose bugs. This study keeps the configuration fixed and changes only the run and the resources of the machine it runs on.

3.4. LLMs for Vulnerability Detection

LLMs have been evaluated both as vulnerability detectors and as filters over SAST output. [12,17] A study of eight LLMs repeated every experiment ten times and found that all of them changed their answers across runs, often with reasoning that did not match their final answer. [10] That study reports how consistent the answers are for each test. This study goes down to individual findings and checks whether the unstable ones are real.

Benchmark results can also make models look better than they are. On a realistic dataset, a 7B code model fell from 68.26% F1 on an older benchmark to just 3.09% [11], which is why this study added 32 new cases that no model could have seen during training. LLMs are also used to triage static-analysis warnings, and one approach filtered false positives from 9,547 warnings with 81.13% precision and 94.64% recall. [32] Filters like this depend on the LLM giving the same verdict each time, so the stability of those verdicts matters here as well.

3.5. Nondeterminism of LLM Output

Other work has measured how much LLM output varies. For ChatGPT code generation on three benchmarks, repeated requests never produced identical test results for 75.76%, 51.00% and 47.56% of tasks, and setting the temperature to 0 did not make the output repeatable. [5] Hosted LLMs run at temperature 0, top-p 1 and a fixed seed still varied in accuracy by up to 15% across runs. [9] In security scanning, a vendor study found that nearly half of the findings reported only by the LLM appeared in just one of five identical runs. [6] These studies measure how much results vary. This study also checks what the varying findings are, real vulnerabilities or noise, and whether repeating runs can filter out the noise.

3.6. Research Gap and Positioning

Table 1 sets this study alongside seven closely related works and compares them on five points: which scanners were evaluated, whether the same run was repeated, whether the runner's resources were changed, whether ground-truth labels were available, and whether the authors looked into what unstable findings actually mean. Each earlier study addresses only one or two of these points.

Some evaluate tools based on a single scan; others rerun experiments on research frameworks or code-generation tasks but never check whether the shifting results are correct. By contrast, this study reruns identical scans under varying resource conditions on labelled code and examines what the unstable findings mean, and it does so for both rule-based and LLM-based scanners. No prior work combines all of these elements.

Table 1. Positioning of this study against closely related empirical work

Work	Scanners studied	Repeated identical runs	Resource conditions	Ground-truth labels	Meaning of unstable findings
SAST on C code [7]	Rule-based (C)	No	No	Yes	Not studied
SAST on Java code [8]	Rule-based (Java)	No	No	Yes	Not studied
Nondeterminism in static analysis [28]	Analysis frameworks	Yes	No	No	Not studied
Configurable analysis testing [31]	Analysis frameworks	No (configurations)	No	No	Not studied
LLM vulnerability reasoning [10]	LLMs	Yes (10 runs)	No	Yes	Not studied
Code LMs on realistic data [11]	Code language models	No	No	Yes	Not studied
ChatGPT code generation [5]	LLM (code generation)	Yes	No	Test suites	Not studied
This study	Rule-based and LLM	Yes (3–10 runs)	Yes (memory, CPU)	Yes	Yes (Fisher exact tests)

Rows are ordered by topic. "Repeated identical runs" means the same tool, version, configuration and input were run more than once, and the outputs were compared.

4. Study Design

Figure 1 summarises the design.

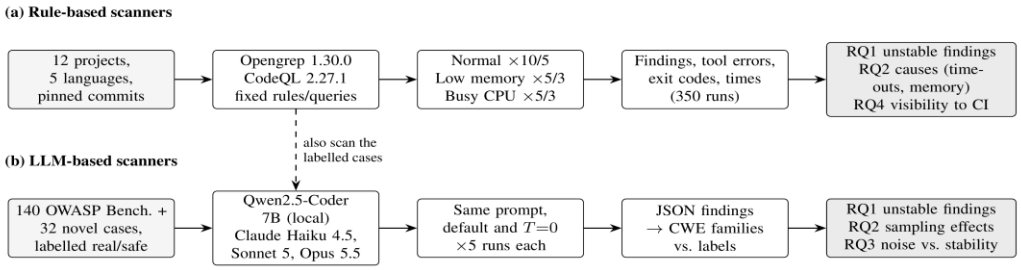


Fig. 1 Study design. (a) Rule-based scanners: 12 pinned projects are scanned repeatedly by Opengrep and CodeQL under three conditions (normal, low memory, busy CPU; runs per condition shown as Opengrep/CodeQL), and the findings, tool errors, exit codes and run times are compared across runs to answer RQ1, RQ2 and RQ4. (b) LLM-based scanners: 140 labelled OWASP Benchmark cases and 32 novel cases are sent five times with the same prompt to four LLMs, with default settings and, where accepted, temperature 0; the reported CWEs are grouped into families and compared with the labels to answer RQ1–RQ3 and RQ5. The rule-based engines also scan the labelled cases for comparison

Table 2. Study subjects, each pinned to one commit

Project	Lang.	Kind	Commit	Files	CodeQL
BenchmarkJava	Java	Benchmark	`20cbf3d`	5,725	✓
BenchmarkPython	Python	Benchmark	`f129148`	2,540	✓
WebGoat	Java	Vuln. app	`c037c95`	1,242	✓
juice-shop	JS/TS	Vuln. app	`1618a61`	1,300	✓
NodeGoat	JS	Vuln. app	`c5cb68a`	111	✓
DVWA	PHP	Vuln. app	`b496a5d`	252	–
pygoat	Python	Vuln. app	`19d17cc`	276	✓
VAmPI	Python	Vuln. app	`f16052d`	22	✓
flask	Python	Library	`d73fa1c`	236	✓
express	JS	Library	`9a34acf`	214	✓
parse-server	JS/TS	Framework	`0e1412e`	485	✓
gin	Go	Library	`3b08cd7`	130	–

Files: number of source files in the scanned languages. CodeQL: ✓ scanned with CodeQL; – not scanned (language outside the study's CodeQL extractors).

4.1. Subjects

Table 2 lists the 12 projects in the study. They cover 5 languages: Java, Python, JavaScript/TypeScript, PHP and Go. The projects fall into three groups. The first group is benchmarks with known answers, namely the OWASP Benchmark for Java and Python. [13] The second is deliberately vulnerable applications used in security training, such as OWASP Juice Shop, WebGoat and DVWA. The third is widely used libraries and frameworks, such as Flask, Express and Gin. Each project was pinned to a single commit, listed in Table 2, and never changed. Any difference between scans, therefore, comes from the scanning, not from the code.

4.2. Rule-Based Scanners and Conditions

The study used Opengrep 1.30.0 with the community rules for each language (opengrep-rules, commit f1d2b56) and CodeQL 2.27.1 with the security-extended query suite. Opengrep scanned all projects.

CodeQL scanned the projects in the languages it supports here: Java in build-less mode, Python and JavaScript/TypeScript. Tool versions, rules and settings stayed fixed for the whole study.

All scans ran in a Linux virtual machine with 2 vCPUs and 7 GB of memory, which is similar to a standard hosted CI runner. Each project was scanned under three conditions:

- Normal: tool defaults, with 4 GB of memory for CodeQL. Opengrep ran 10 times and CodeQL 5 times.
- Low memory: a smaller memory budget, as on a small runner (Opengrep --max-memory 512 MB; CodeQL --ram 1024 MB). Opengrep ran 5 times and CodeQL 3 times.
- Busy: two CPU-heavy background processes occupied both cores during the scan, as on a shared or overloaded runner. Opengrep ran 5 times and CodeQL 3 times.

For every run, the result file (JSON or SARIF), the errors reported by the tool, the exit code and the wall-clock time were recorded. An earlier pilot phase repeated default Opengrep scans of four projects and happened to capture a change of host machine between runs. That result is reported separately.

4.3. LLM-Based Scanners

The study took 140 cases from the OWASP Benchmark for Python. [13] Using the benchmark's expected results file, up to five truly vulnerable cases and five safe look-alikes were selected for each of its 14 categories. The licence header was removed, and category names in routes, variable names and messages were replaced, so that the code itself gave no hint of the answer.

In addition, 32 new cases were written for this study, covering eight categories: SQL injection, command injection, path traversal, cross-site scripting, code injection, insecure deserialization, weak randomness and open redirect. Each category has two vulnerable versions and two safe look-alikes, and each safe version differs from a vulnerable one by a realistic fix, such as a parameterised query, an allow-list or yaml.safe_load. Because the cases were built this way, the correct answer for each one is known.

Every case was sent with the same prompt, which asked the model to act as a security scanner and return only JSON findings, each with a CWE number, line, title and confidence. Four models were tested. Qwen2.5-Coder 7B [33] ran locally through Ollama [34] with its default quantised weights on a laptop with 16 GB of memory.

The other three, Claude Haiku 4.5, Sonnet 5 and Opus 5.5, were accessed through the Anthropic API. [35] Haiku and Opus requests went through the Message Batches API. Sonnet used the standard Messages API because its batch had not started after two hours, but the requests were otherwise identical, with an output limit of 1,024 tokens.

Each model scanned each case five times with default settings. Where the model allowed it, five more scans were run at a temperature 0. Sonnet 5 and Opus 5.5 rejected the temperature setting as deprecated, so they were run with default settings only.

4.4. Measures

4.4.1. Terminology

A finding is a pair of a rule (or weakness type) and a code location reported by a scanner. Within a group of repeated scans, a finding is stable if it appears in every scan and unstable otherwise. A lost finding is one that is stable under normal conditions but missing from a scan run under a resource limit.

4.4.2. Finding Identity

For rule-based scanners, a finding is identified by its rule, file, and start and end positions. Because the code never changes, identical findings always have identical locations. For LLM scanners, a finding is identified by the case and the reported weakness. Weaknesses are grouped into CWE families for each category (for example, CWE-78 and CWE-77 for command injection), and line numbers are ignored because LLMs report them loosely.

Let F_i be the set of findings in run i of n runs, U the set of all distinct findings, and S the set of findings present in every run:

$$U = \bigcup_{i=1,\dots,n} F_i, \quad S = \bigcap_{i=1,\dots,n} F_i \tag{1}$$

The instability rate is the share of distinct findings that do not appear in every run:

$$IR = \frac{|U \setminus S|}{|U|} \times 100\% \tag{2}$$

Agreement between runs is measured as the average pairwise Jaccard similarity:

$$J = \frac{2}{n(n-1)} \sum_{i<j} \frac{|F_i \cap F_j|}{|F_i \cup F_j|} \quad (3)$$

For a limited condition c, the lost findings L_c are the baseline findings that a run misses, where the baseline B is the stable set of the normal runs:

$$L_c = |B \setminus F_c|, \; B = S_{normal} \quad (4)$$

For RQ5, let r_f be the number of runs in which finding f appears. The expected share of U found by k randomly chosen runs is given below, where $C(a, b)$ is the number of ways to choose b of a runs:

$$Cov(k) = \frac{1}{|U|} \sum_{f \in U} [1 - \frac{C(n - r_f, k)}{C(n,k)}] \quad (5)$$

A vote filter with threshold m keeps the findings that appear in at least m of n runs. Setting $m = 1$ gives the union, $m = \lceil n/2 \rceil$ a majority vote, and $m = n$ the stable set:

$$F^{(m)} = \{ f \in U : r_f \geq m \} \quad (6)$$

Rule-based engines on the labelled cases. To compare the two kinds of scanner directly, Opengrep and CodeQL (with their Python rules and queries) also scanned both sets of labelled cases, three times each. A case counts as detected if a finding in its file maps to the case's CWE family.

Correctness of LLM findings. A finding that names the case's category is a true positive (TP) if the case is vulnerable and a false positive (FP) if it is safe. A finding that names any other weakness is off-target. Because each case is built around a single weakness, off-target findings are treated as noise (see Section 7). Stable and unstable findings are compared using two-sided Fisher exact tests.

The quality of a set of LLM findings is summarised by the share of true positives among all reported findings (TP share), where OFF is the number of off-target findings:

$$TP\ share = \frac{TP}{TP + FP + OFF} \quad (7)$$

Visibility. For each run, it was recorded whether the exit code and error output would have signalled a problem to a CI pipeline.

5. Results

The study ran 350 rule-based scans in total (240 with Opengrep and 110 with CodeQL) and collected five repeated LLM answers for every case, model and setting. Table 3 summarises the rule-based scans and Table 4 the LLM scans.

Table 3. Rule-based scans of unchanged code

	Opengrep				CodeQL			
Project	**Findings**	**Unstable**	**Lost (mem)**	**Lost (busy)**	**Findings**	**Unstable**	**Lost (mem)**	**Lost (busy)**
BenchmarkJava	2376	0	0	0	870	0	Failed†	0
BenchmarkPython	4363	0	0	0	319	0	0	0
WebGoat	54	0	0	0	10	0	0	0
juice-shop	93–137	44	0	0	94	0	Failed†	0
NodeGoat	17	0	0	0	17	0	0	0
DVWA	75	0	0	0	–	–	–	–
pygoat	218	0	0	0	40	0	0	0
VAmPI	2	0	0	0	3	0	0	0
flask	29	0	0	0	8	0	0	0
express	50	0	0	0	5	0	0	0
parse-server	336	0	0–10	10	17	0	Failed†	0
gin	27	0	0	0	–	–	–	–

Findings: number of findings per run under normal conditions (range if it varied between runs). Unstable: findings not present in every normal run. Lost (mem) and Lost (busy): findings present in every normal run but missing from a low-memory or busy run (range over runs). Failed†: the scan ran out of memory, exited with code 99 and produced no results, so every baseline finding was lost. –: not scanned with CodeQL (the study used CodeQL's Java, Python and JavaScript/TypeScript extractors). Every other run exited with code 0.

Table 4. LLM scanner repeatability over five identical runs per case.

	OWASP Benchmark (140 cases)					Novel cases (32)	
Model	**Findings**	**Unstable %**	**Seen once %**	**Flips %**	**Acc. %**	**Unstable %**	**Acc. %**
Qwen2.5-Coder 7B (local)	481	91.1	53.8	25.0	49–55	71.4	50–59
Qwen2.5-Coder 7B (local) (T=0)	170	0.0	0.0	0.0	53	6.5	53–56
Claude Haiku 4.5	327	50.5	32.1	5.7	56–59	30.2	72
Claude Haiku 4.5 (T=0)	226	10.2	3.1	2.1	56–57	5.9	72
Claude Sonnet 5	131	28.2	14.5	10.7	72–76	20.8	84–91
Claude Opus 5.5	75	2.7	2.7	0.0	100	25.0	100

Findings: distinct findings over the five runs. Unstable %: share of distinct findings not reported in all five runs (Equation 2). Seen once %: share reported in exactly one run. Flips %: share of cases whose verdict on their own weakness category changed between runs. Acc. %: accuracy of each run against the labels, given as the minimum–maximum over the five runs; it is not a confidence interval, and a single value means that all five runs had the same accuracy. T=0: temperature 0.

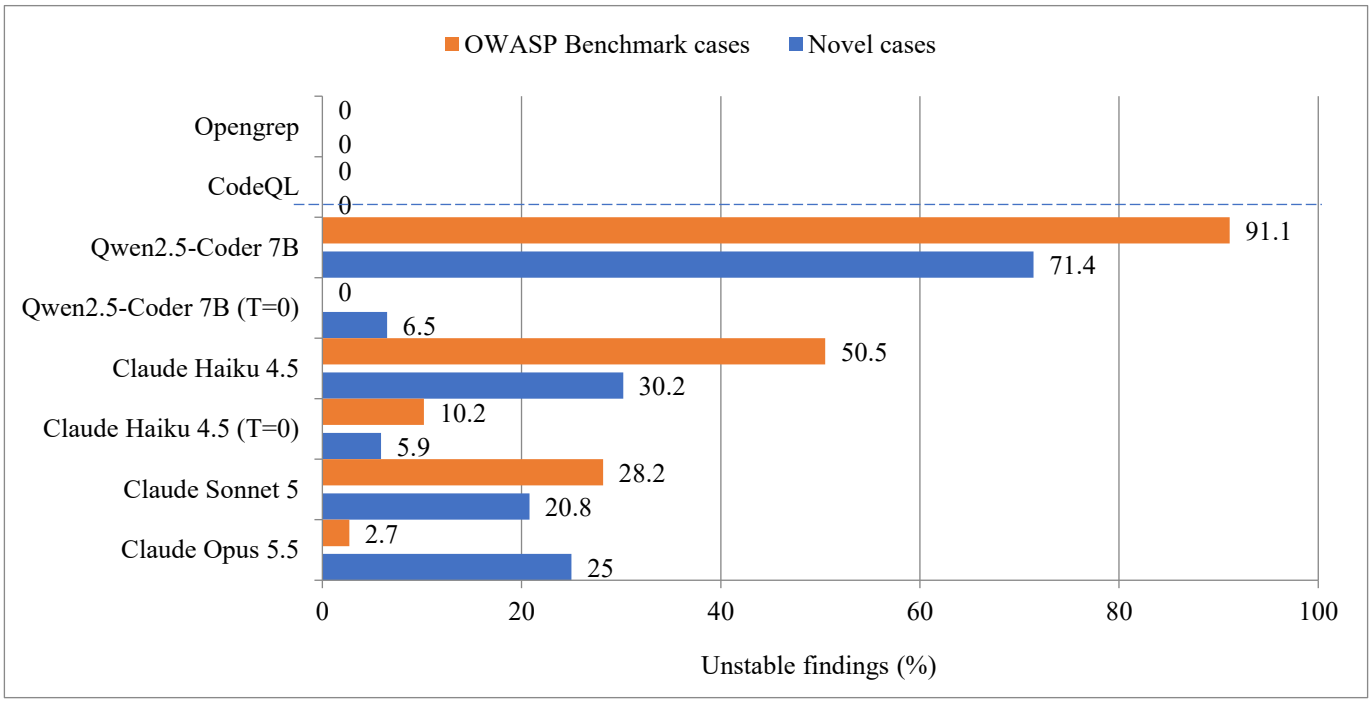


Fig. 2 Share of distinct findings that were not reported in every one of five identical runs (instability rate, Equation 2), for each scanner on the same labelled cases. Blue bars: 140 OWASP Benchmark cases; orange bars: 32 novel cases. Rule-based scanners (above the dashed line) returned identical findings in every run (0%); every LLM setting except Qwen at temperature 0 on the benchmark returned different findings across runs. T=0: temperature 0

5.1. RQ1: How Often Do Findings Change?

Under normal conditions, the rule-based scanners were almost perfectly repeatable: 21 of the 22 tool–project pairs gave exactly the same findings in every run. The exception was Opengrep on OWASP Juice Shop, which reported anywhere from 93 to 137 findings on the same commit.

Of these, 44 findings (32%) appeared in some runs and not in others. In the main study, 1 of the 10 normal runs gave the higher count. In the pilot, the same project gave 137 findings in six consecutive runs and 93 in the next four, after the scanning host changed. On one host, three back-to-back runs on an idle machine gave 137, 93 and 137. Normal conditions therefore, do not guarantee stable results, because whether a borderline file gets fully analysed can flip from one run to the next.

Limiting resources made the losses larger and more frequent (Table 3, "Lost" columns). With less memory, Opengrep lost up to 10 of the 336 findings in Parse Server, and on a busy machine, it lost 10 in every run. With less memory, CodeQL could not finish 3 of its 10 projects (Juice Shop, Parse Server and BenchmarkJava) and returned no results for them at all. For Juice Shop, both limited conditions always gave the lower count of 93, so the 44 extra findings never appeared.

The LLM scanners varied far more (Table 4, Figure 2). With default settings, 50.5% of Haiku 4.5's findings on the benchmark did not appear in all five runs. The rate was 28.2% for Sonnet 5, 2.7% for Opus 5.5 and 91.1% for Qwen 7B. The overall verdict also changed. For 10.7% of cases, Sonnet's answer to whether the case's vulnerability was present differed between runs, compared with 5.7% for Haiku and 25.0% for Qwen. Temperature 0 cut instability sharply, but did not reliably remove it. The local Qwen model became fully repeatable on the benchmark (from 91.1% to 0.0%) but not on the novel cases (from 71.4% to 6.5%), and Haiku still varied on both (10.2% and 5.9%).

Larger models were more stable on the benchmark, but this was partly due to familiarity with it (Section 5.8). On the 32 novel cases, instability for Opus rose from 2.7% to 25.0%, close to Sonnet (20.8%) and Haiku (30.2%).

RQ1: Rule-based engines were repeatable when they had enough resources, yet one project in twelve varied by 32% across identical runs, and limited resources removed findings or entire results in several projects. LLM scanners changed a large share of their findings across identical runs, and temperature 0 reduced this sharply but did not reliably remove it for either the local or the hosted model.

5.2. RQ2: What Causes the Changes?

Every unstable or lost rule-based finding traced back to a resource budget. In Juice Shop, all 44 unstable findings were in a single file, a bundled copy of three.js (frontend/src/assets/private/three.js). Opengrep drops a file after three rule timeouts. Runs that hit two timeouts on this file still analysed it and reported 137 findings, while runs that hit a third timeout skipped the rest of the file and reported 93. Whether that third timeout occurs depends on how fast the machine happens to be at that moment. Under CPU load, a different limit took effect: the parser ran into an internal memory limit, reported the file as a syntax error and skipped it entirely. In Parse Server, the 10 lost findings were all in one large test file (spec/ParseGraphQLServer.spec.js) and followed the same timeout pattern. CodeQL's failures under reduced memory were caused by Java heap exhaustion during query evaluation.

For the LLM scanners, the instability comes from random sampling. Temperature 0 removed most of it, but some remained both locally and on the hosted API, which is consistent with earlier findings that temperature 0 does not guarantee identical output. [5] The two newest models rejected the temperature setting altogether, so their users do not have this control. The study also observed an LLM version of the budget effect: in 4 of 700 Sonnet 5 requests, the model spent its entire output budget on internal reasoning and returned no answer, which a pipeline would read as "no findings".

RQ2: Rule-based instability came from the environment, as timeouts, memory limits and a busy CPU acted on large or unusual files. LLM instability came from sampling, and on the newest models, it could not be controlled through the temperature setting.

5.3. RQ3: What Do Unstable Findings Mean?

For the rule-based engines, the findings that disappeared were ones the engine reports every time it completes its analysis. Their absence says nothing about whether they are correct, only that the scan was incomplete. In Juice Shop, the lost findings included 17 prototype-pollution findings, a security-relevant class, in third-party code shipped with the application. On the labelled cases, both rule-based engines gave identical verdicts in every run, with an accuracy of 59% for both Opengrep and CodeQL on the benchmark cases, and 69% and 81%, respectively, on the novel cases.

For the LLMs, stability did carry information about correctness (Table 5). On the benchmark, every model and setting that produced unstable findings showed the same pattern: unstable findings were significantly more likely to be off-target than stable ones (all $p < 0.01$). On the novel cases, this held for Qwen 7B ($p = 0.001$), Haiku 4.5 and Opus 5.5 (both $p < 0.001$), while the remaining settings had too few unstable findings for a conclusive test. Among the findings that named the correct weakness, however, stability did not reliably separate vulnerable cases from safe ones. Stability is therefore a filter for noise, not a substitute for ground truth.

Table 5. Off-target (noise) share among stable and unstable LLM findings. p: two-sided Fisher exact test.

Model	Stable	Unstable	p
OWASP Benchmark			
Qwen 7B	47% (20/43)	92% (403/438)	<0.001
Qwen 7B (T=0)	78% (132/170)	0% (0/0)	1.000
Haiku 4.5	37% (60/162)	95% (157/165)	<0.001
Haiku 4.5 (T=0)	47% (96/203)	87% (20/23)	<0.001
Sonnet 5	10% (9/94)	59% (22/37)	<0.001
Opus 5.5	4% (3/73)	100% (2/2)	0.004
Novel cases			
Qwen 7B	31% (5/16)	80% (32/40)	0.001
Qwen 7B (T=0)	55% (16/29)	50% (1/2)	1.000
Haiku 4.5	17% (5/30)	100% (13/13)	<0.001
Haiku 4.5 (T=0)	22% (7/32)	100% (2/2)	0.064
Sonnet 5	5% (1/19)	40% (2/5)	0.099
Opus 5.5	24% (5/21)	100% (7/7)	<0.001

Stable: reported in all five runs; unstable: reported in one to four runs. Values are the off-target share with counts in brackets (off-target findings / all findings in the group). p: two-sided Fisher exact test of the difference between the stable and unstable groups.

RQ3: The two kinds of instability mean opposite things. A rule-based finding that disappears is a lost detection caused by an incomplete scan, while an LLM finding that does not recur is, most of the time, noise.

5.4. RQ4: Would a Pipeline Notice?

Opengrep exited with code 0 in all 240 runs, including every run that lost findings. The only sign of skipped work was the "errors" list in its JSON output, which recorded timeouts, partial parses and syntax errors. A gate that checks only the findings or the exit code ignores this list. CodeQL failed visibly: all 9 runs that could not finish exited with a non-zero code and produced no results file. Even so, a pipeline that treats a missing results file as "no findings" would still pass. LLM scanners have no exit code for an unanswered request, so an empty answer looks like a clean result unless the pipeline checks for it.

RQ4: Silent loss was the norm for Opengrep and for LLM answers that ran out of output budget. Only CodeQL's failures were visible, and only to pipelines that check exit codes.

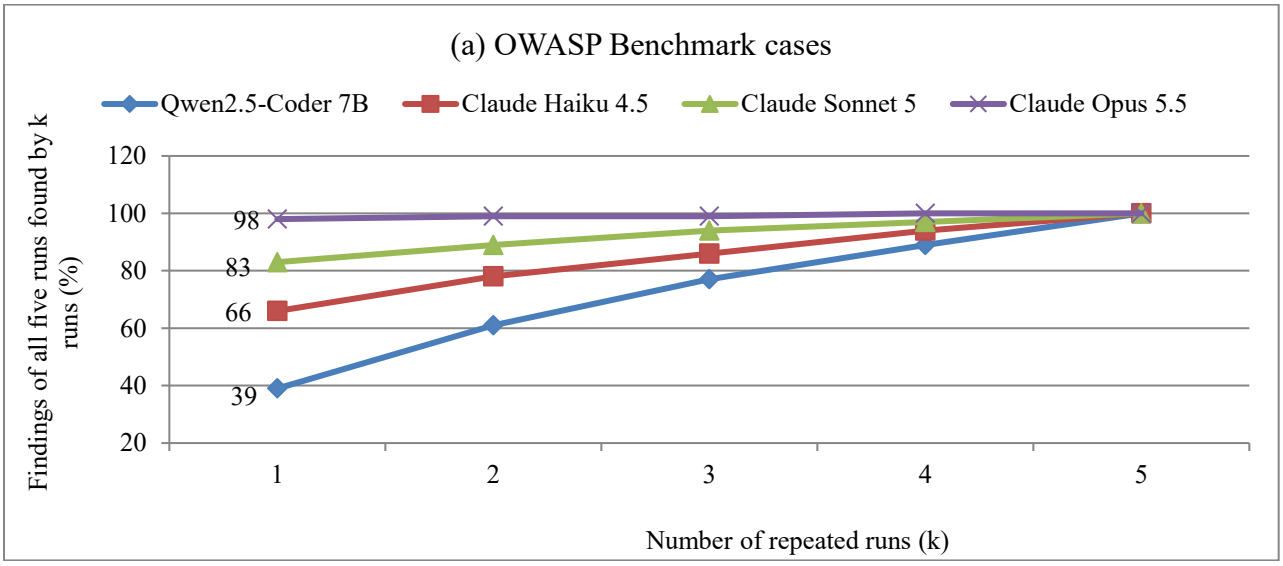
5.5. RQ5: How Many Runs Are Needed, and Does Voting Help?

For most models, a single scan misses a large share of what an LLM scanner can find. Figure 3 shows the share of findings recovered as more runs are added, relative to the combined output of all five runs. On the benchmark cases, one run recovers 39% of the findings for Qwen, 66% for Haiku, 83% for Sonnet and 98% for Opus. Three runs raise these figures to 76%, 86%, 94% and 99%. The novel cases follow the same pattern, with a single run recovering 53%, 80%, 90% and 83%, respectively. For Qwen and Haiku, the curves are still rising at the fifth run, so even five runs do not reach the full set of findings. Rule-based tools are not immune to this effect either. When Opengrep scanned Juice Shop ten times, 44 of its 137 distinct findings appeared in only one run, and a single run captured on average just 71% of the total.

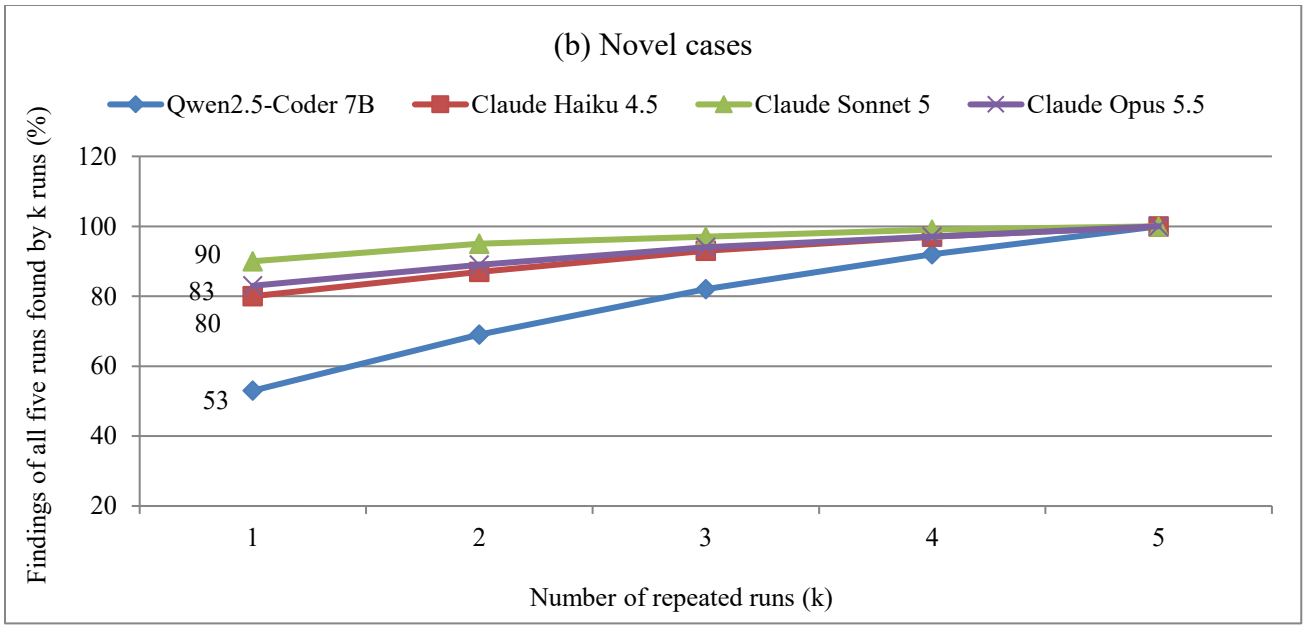
Keeping only the findings that recur across runs makes the output noticeably more precise. Table 6 and Figure 4 report the share of true positives among the reported findings (TP share, Equation 7) when a finding is retained only if it appears in all five runs; Table 6 also gives recall. On the benchmark, the TP share rises from 11% to 28% for Qwen, from 27% to 35% for Haiku and from 57% to 66% for Sonnet, while Opus stays near 96%. For the Claude models, this comes at almost no loss of recall: Haiku moves from 83% to 81%, Sonnet stays at 89%, and Opus remains at 100%. Qwen is the exception, with recall falling from 29% to 17%, since many of its correct detections appear in only some of the runs. On the novel cases, the same filter lifts the TP share from 74% to 84% for Sonnet, from 69% to 76% for Opus and from 47% to 53% for Haiku, and all three still detect every vulnerable case.

Not every way of combining runs is helpful. Taking the union of all runs, that is, keeping anything reported by any run, lowers the TP share because it collects the noise from every run; Haiku, for example, drops from 27% to 18%. A majority vote, which keeps findings reported in at least three of five runs, gives a modest gain at almost no cost for the Claude models. At temperature 0, the runs already agree closely, so voting changes very little: Haiku's TP share moves only from 27% to 29%.

RQ5: A single LLM scan recovers only part of what several scans find together, and merging all runs just adds each run's noise. Keeping only findings that recur across runs raises the share of correct findings, with little or no loss of real vulnerabilities for the Claude models.



(a)



(b)

Fig. 3 Share of all findings from five runs that are recovered by k randomly chosen runs (Equation 5), for the four LLMs with default settings. (a) OWASP Benchmark cases; (b) novel cases. The value at k=1 (labelled) is the share a single scan would report; a curve that is still rising at k=5 indicates that more runs would reveal further findings.

Table 6. Effect of repeat-and-vote filters on LLM findings (default settings).

		OWASP Benchmark (140 cases)			Novel cases (32)		
Model	**Filter**	**Findings**	**TP share %**	**Recall %**	**Findings**	**TP share %**	**Recall %**
Qwen2.5-Coder 7B	Single run	189.4	10.7	28.9	29.6	25.0	46.2
	Union of 5 runs	481	7.1	48.6	56	17.9	62.5
	Majority of 5 runs	125	13.6	24.3	23	26.1	37.5
	All 5 runs	43	27.9	17.1	16	37.5	37.5
Claude Haiku 4.5	Single run	216.6	26.9	83.1	34.2	46.8	100.0
	Union of 5 runs	327	18.3	85.7	43	37.2	100.0
	Majority of 5 runs	195	29.7	82.9	32	50.0	100.0
	All 5 runs	162	35.2	81.4	30	53.3	100.0
Claude Sonnet 5	Single run	108.6	57.3	88.9	21.6	74.1	100.0
	Union of 5 runs	131	48.1	90.0	24	66.7	100.0
	Majority of 5 runs	105	59.0	88.6	21	76.2	100.0
	All 5 runs	94	66.0	88.6	19	84.2	100.0
Claude Opus 5.5	Single run	73.4	95.4	100.0	23.2	69.0	100.0
	Union of 5 runs	75	93.3	100.0	28	57.1	100.0
	Majority of 5 runs	73	95.9	100.0	22	72.7	100.0
	All 5 runs	73	95.9	100.0	21	76.2	100.0

Single run: expected value for one run, computed by weighting each finding by the share of runs that reported it. Union: findings reported in at least one of five runs (m=1 in Equation 6). Majority: at least three of five (m=3). All: every run (m=5). TP share: true positives among reported findings (Equation 7). Recall: vulnerable cases whose own weakness was reported, out of 70 (benchmark) or 16 (novel) vulnerable cases.

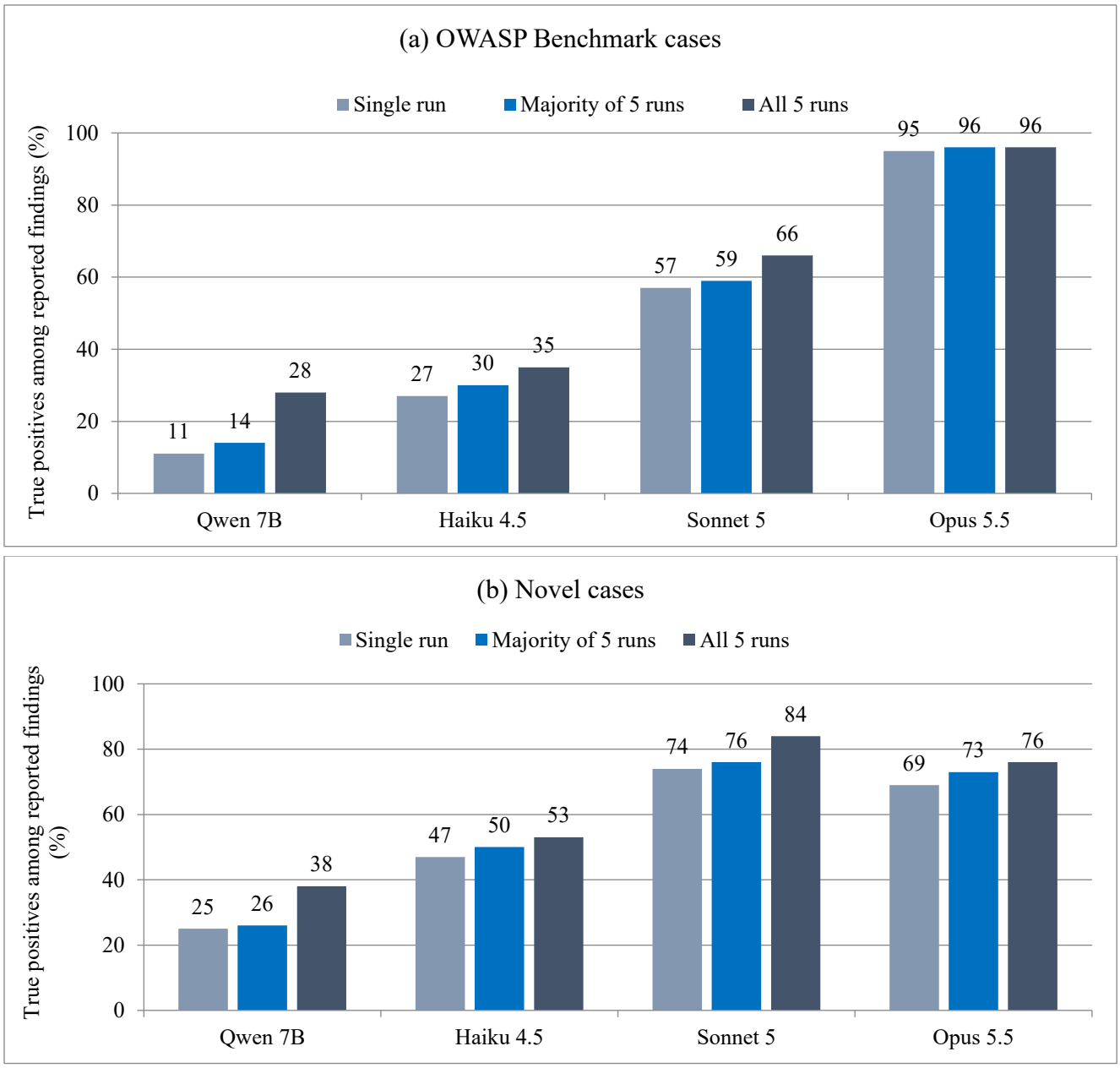


Fig. 4 True positives among reported findings (TP share, Equation 7) for a single run and for two repeat-and-vote filters over five runs: keeping findings reported in a majority (at least three) of the runs, or in all five runs. (a) OWASP Benchmark cases; (b) novel cases. Default settings. Recall for each filter is given in Table 6

5.6. Instability by Weakness Category

Table 7 shows how unstable the LLM scanners' findings were for each vulnerability category in the OWASP Benchmark. Instability varies widely between categories. Injection flaws were the most stable: only 24% of command injection findings and 35% of SQL injection findings were unstable. At the other extreme, instability reached 81% for trust boundary violations, 80% for weak hashing, and 78% for both insecure cookies and weak randomness. One likely explanation, though the data cannot confirm it, lies in what each category asks of the model. An injection flaw has a concrete path from user input to a dangerous function, a pattern the model can follow and report in the same way every time. Weak hashing, cookie settings and randomness, by contrast, call for a judgement about whether a given configuration is actually risky enough to report, and a model may make that call differently from one run to the next. Opus stood out: it produced no unstable findings in 13 of the 14 categories, and its only unstable findings were in command injection (29%). The "All models" column should be read with care, since Qwen reported far more findings than the other models and therefore dominates the combined figures.

Table 7. Unstable LLM findings (%) by weakness category of the OWASP Benchmark case (default settings).

Weakness category	Qwen 7B	Haiku 4.5	Sonnet 5	Opus 5.5	All models	Findings
Trust boundary violation	95.7	71.0	87.5	0.0	81.1	90
Weak hashing	100.0	76.7	27.3	0.0	79.9	134
Insecure cookie	100.0	68.4	50.0	0.0	78.2	101
Weak randomness	100.0	52.4	60.0	0.0	77.9	77
XML external entity	100.0	23.1	40.0	0.0	69.6	69
Path traversal	100.0	50.0	10.0	0.0	67.6	74
Insecure deserialization	97.2	66.7	10.0	0.0	67.5	83
Cross-site scripting	96.4	40.0	16.7	0.0	63.0	54
Open redirect	85.2	47.4	16.7	0.0	57.9	57
LDAP injection	82.9	0.0	7.7	0.0	41.1	73
XPath injection	80.0	0.0	26.7	0.0	40.6	69
Code injection	83.3	16.7	10.0	0.0	39.1	46
SQL injection	41.2	43.8	25.0	0.0	34.8	46
Command injection	16.7	23.1	33.3	28.6	24.4	41

Values are the share of distinct findings on cases of that category that were not reported in all five runs (Equation 2). All models: pooled over the four models; Findings: number of distinct findings pooled over the four models. Rows are sorted by the pooled value.

5.7. Cost of Scanning and of Repetition

Table 8 shows what each scanner costs to run. The rule-based tools are cheap. On average, a project took about 41 seconds with Opengrep and 48 seconds with CodeQL, and the CodeQL figure covers only the analysis step, not building the database. Under CPU load, these times grew to 1.69 times as long for Opengrep and 1.64 times as long for CodeQL on a typical project. That slowdown is what pushed large files past Opengrep's timeout, and it explains the findings that disappeared. The local model, Qwen, needed 16.4 seconds per case on a laptop. The hosted Claude models used roughly 570 to 700 input tokens and 140 to 300 output tokens per case, and for Sonnet 5 and Opus 5.5, between 93 and 131 of those output tokens went to internal reasoning. These costs point to different fixes for the two kinds of scanner. For rule-based tools, scanning again is not worth it: reading the tool's error log costs nothing and reveals exactly which files were skipped. For LLM scanners, every extra run costs as much as the first one. Three runs with a majority vote are a reasonable compromise, and five runs that keep only the findings reported every time give the most correct findings if the budget allows.

Table 8. Cost of one scan and effect of CPU contention.

Scanner	Unit	Typical cost per unit	Effect of a busy runner
Opengrep 1.30.0	project scan	41 s mean (32 s median)	1.69× slower (median)
CodeQL 2.27.1	query run on a prebuilt database	48 s mean (37 s median)	1.64× slower (median)
Qwen2.5-Coder 7B (local)	case, one run	16.4 s mean (11.0 s median) on a laptop	–
Claude Haiku 4.5	case, one run	566 input, 298 output tokens	–
Claude Sonnet 5	case, one run	696 input, 142 output tokens (93 reasoning)	–
Claude Opus 5.5	case, one run	697 input, 179 output tokens (131 reasoning)	–

Times are wall-clock seconds on the 2-vCPU study machine under normal conditions; CodeQL times exclude database creation. Busy-runner effect: median over projects of mean busy time divided by mean normal time. Token counts are means per request over all benchmark and novel cases; output tokens include reasoning tokens. Repeating an LLM scan n times multiplies its cost by n.

5.8. Robustness: Novel Cases

Because the OWASP Benchmark is public, the study checked whether the models recognised it. In 2 of 700 answers, Opus 5.5 named the OWASP Benchmark explicitly, for example, reasoning that a helper "in OWASP BenchmarkPython-style code" returns a constant. Therefore, 32 new cases were written across eight categories, each with two vulnerable versions and two safe look-alikes that differ by a realistic fix, such as a parameterised query, an allow-list or yaml.safe_load. On these cases, the largest model remained accurate, but its instability rose to 25.0%, and every one of its unstable findings was off-target. The main conclusions, that LLM findings vary across identical runs and that unstable LLM findings are mostly noise, therefore also hold on code written specifically for this study.

6. Discussion

6.1. Two Kinds of Instability

"The scanner gave a different answer" can mean two quite different things, and treating them the same way leads to wrong decisions. The rule-based engines were repeatable. Their results changed only when a limit ran out, such as a timeout, a memory cap or the CPU on a busy runner, and the engine skipped part of its work. The findings lost this way were ones the engine reports reliably whenever it has enough resources. For LLM scanners, variation comes from the sampling itself, and the findings that come and go are mostly off-target. Stability, therefore works as a sign of trust for LLM findings. For a rule-based tool, a missing finding does not mean the original was a false alarm. It means the scan was incomplete.

6.2. Comparison with Prior Findings

The results of this study agree with the earlier work on LLM variability and go a step further. An earlier study [10] showed that LLMs change their answers from one run to the next. This study sees the same thing at the level of individual security findings, where between 2.7% and 91.1% of findings on the benchmark were unstable. What earlier work did not show is what those unstable findings are worth. In this study's data, they were mostly noise, and voting across runs removed much of it and improved the results. Other studies [5,9] reported that setting the temperature to 0 does not make a model fully repeatable, and this study confirms it as well: Haiku at temperature 0 still left 10.2% and 5.9% of its findings unstable, and the local Qwen model left 6.5% on the novel cases. This study also found that the newest models no longer let the user set a temperature at all, so that option is gone for them. Finally, another study [11] found that benchmarks make models look better than they really are. Stability follows the same pattern: Opus was only 2.7% unstable on the public benchmark but 25.0% unstable on cases it had not seen before.

For rule-based tools, the story is different. An earlier study [28] traced nondeterminism in research analysis tools to the way those tools are implemented. The rule-based engines tested here were repeatable when they had enough resources. Their results changed only because of the environment, through timeouts, memory limits and a busy CPU, a cause not reported before. Earlier evaluations [7,8] showed that SAST tools miss many vulnerabilities even in a single scan. The findings that disappeared in this study add to those misses, and they are harder to catch because nothing in the output shows that they are missing. The two kinds of scanner differ for a simple reason. A rule-based engine either scans a file completely or skips it when it hits a limit, so its results change only when a limit is reached. An LLM samples every answer, so some variation is always there. This is why voting across runs helps LLM scanners but does little for rule-based tools, where the fix is to find and remove the resource limit.

6.3. Implications for Practitioners

6.3.1. Do not assume a finding is fixed just because it is gone

If no related code change explains why a finding disappeared, it should not be marked as fixed. Remediation tracking [26,27] should require a matching code change or confirm the result with a second scan.

6.3.2. Treat scanner errors as gate failures

In every run where findings were lost, the scanner still exited successfully, and only its error list or log showed the skipped work. Pipelines should fail, or at least warn, when a scan reports timeouts, memory-limit events or skipped files.

6.3.3. Size runners for the scan and set explicit budgets

Losses depend on how fast and how busy the machine is. Teams can use dedicated or larger runners, raise timeouts and memory limits for large generated or vendored files, or exclude those files on purpose so that skipping them is a decision rather than an accident.

6.3.4. Repeat LLM scans and trust agreement

Findings reported in every run were far more likely to be correct. With a majority vote across runs, the Claude models kept recall within 0.3 points of a single run. Keeping only findings reported in all five runs raised the TP share for the Claude models by up to 9 points on the benchmark and 10 points on the novel cases. Merging all runs made the results worse. Temperature 0 reduces variation but does not remove it, and the newest models no longer accept a temperature setting, so repeating the scan is the one control that is always available.

6.4. Implications for Tool Builders and Researchers

Scanners should report their coverage (which files and rules were skipped, and why) in a machine-readable field that CI can act on. Researchers who label warnings as false positives because they later disappear [25] should first check that the disappearance was not caused by an incomplete scan. A finding's history across runs is useful for triage, but only when read together with the type of scanner and whether the scan was complete. This points to confidence-based triage that

combines how persistent a finding is, whether the scan was complete, and whether there is evidence of a fix, which is left for future work.

7. Threats to Validity

7.1. Construct Validity

Rule-based findings are identified by their exact rule and location. This matching is exact in this setup because the code never changed, but comparing findings across versions would require location-robust matching. [26,27] For LLM findings, the reported CWEs are grouped into families for each category, so a correct finding labelled with an unusual CWE is counted as off-target. Off-target findings are also treated as noise, although a few may be real issues outside the labelled vulnerability. This would affect the conclusions only if such extra issues were more unstable than on-target findings, which is unlikely.

7.2. Internal validity

The low-memory and busy-CPU conditions were created using the tools' own memory settings and artificial CPU load rather than container limits, which were not available in the study environment. These conditions trigger the same mechanisms, timeouts and memory limits, but the exact rates will differ on other hardware. Only one scan ran at a time, and any run that overlapped with an unintended second process was discarded and repeated. All models received the same neutral prompt, so differences between them reflect the models rather than prompt tuning.

7.3. Benchmark Familiarity

The OWASP Benchmark is public, so larger models may have seen it during training. There is direct evidence of this: 2 of Opus's 700 answers mentioned the OWASP Benchmark by name. Such familiarity could inflate both accuracy and stability, which is why 32 new cases were written for this study (Section 4.3).

7.4. External Validity

The study covers two rule-based engines, four LLMs (one local model and three from a single vendor) and 12 projects in five languages. Commercial SAST platforms introduce further sources of change, such as server-side rule updates and incremental scanning [1,2], which the fixed-version setup of this study excludes on purpose. The rates reported here are therefore a lower bound for such platforms. Models from other vendors may behave differently, and hosted models change over time. To support repetition, all prompts and cases are released.

7.5. Conclusion Validity

Some groups are small, especially the stable on-target LLM findings. The analysis, therefore, uses exact tests, shows counts next to percentages, and makes no claims where a test is not significant. After a Bonferroni correction across all tests, every benchmark result reported as significant remains below 0.05. The RQ5 coverage and voting results are computed from the same five runs rather than a separate set, so additional runs would make those estimates more precise. The per-category values in Table 7 rest on 41 to 134 findings per category and should be read as indicative.

7.6. Author-Written Cases

The 32 new cases were written by the author, which could bias how difficult they are. To limit this, the cases follow common vulnerable and fixed patterns, each safe case differs from its vulnerable counterpart by a single realistic change, and all cases and labels are released for inspection.

8. Conclusion

Security scanning in CI assumes that scanning the same code gives the same findings. This study shows that this holds for rule-based engines only while they stay within their resource limits, and that it does not hold for LLM scanners at all. When a rule-based scanner hits a timeout or memory limit, it quietly drops real findings and still reports success. LLM scanners produce findings that come and go from run to run, and most of these are noise. A single LLM scan captures only part of what repeated scans find. Merging all runs adds noise, while keeping only the findings that recur raises the share that are correct. Instability also depends on the kind of weakness: it is highest for configuration and cryptography issues and lowest for injection flaws. For CI pipelines, three practices follow from these results: treat an incomplete scan as a failure, never mark a finding as fixed only because it disappeared, and use agreement across repeated LLM runs as a signal of confidence.

Several directions remain open. Commercial SAST platforms and software composition analysis tools should be studied next, since their results also shift with rule and vulnerability-database updates. The measurements should also be repeated on larger and more varied projects and on hosted CI runners, where resource pressure is harder to control. Finally, triage models could combine a finding's history across runs, the completeness of each scan and evidence of a fix into a single confidence score to support release decisions.

Conflicts of Interest

The author declares that there is no conflict of interest regarding the publication of this paper.

Funding Statement

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors. All computing and API costs were paid by the author.

Acknowledgments

The author used Claude (Anthropic) for language editing and to help write the data-analysis scripts. All figures were produced by these scripts directly from the measured data; no

figure or image was created or altered by generative AI. The author designed the study, verified all reported results against the raw data, and takes full responsibility for the content of this article. The Claude models evaluated in this study were accessed through the public API as a paying customer.

Data Availability

All scan scripts, pinned commit identifiers, tool versions, LLM prompts, the 32 novel test cases with labels, raw scanner and LLM outputs, and analysis scripts are available in the replication package [DOI to be added on publication].

References

[1] Snyk, Troubleshooting Snyk Reports. [Online]. Available: https://docs.snyk.io/scan-fix-and-prevent/prevent/analytics/reports-tab/troubleshooting-snyk-reports

[2] Checkmarx, SAST Scanner. [Online]. Available: https://docs.checkmarx.com/en/34965-324470-sast-scanner.html

[3] Brittany Johnson et al., "Why don't Software Developers use Static Analysis Tools to find Bugs?," *2013 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA, 2013. [CrossRef] [Google Scholar] [Publisher Link]

[4] Maria Christakis, and Christian Bird, "What Developers Want and Need from Program Analysis: An Empirical Study," *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Singapore, pp. 332-343, 2016. [CrossRef] [Google Scholar] [Publisher Link]

[5] Shuyin Ouyang et al., "An Empirical Study of the Non-determinism of ChatGPT in Code Generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1-28, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[6] Liran Tal, Snyk VulnBench JS 1.0: Can LLMs Find the same Bug Twice?, Snyk, 2026. [Online]. Available: https://snyk.io/blog/snyk-vulnbench-js-1-0-llm-security-review-repeatability/

[7] Stephen Lipp, Sebastian Banescu, and Alexander Pretschner, "An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection," *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Virtual, South Korea, pp. 544-555, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[8] Kaixuan Li et al., "Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java," *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, San Francisco, CA, USA, pp. 921-933, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[9] Berk Atıl et al., "Non-determinism of "Deterministic" LLM System Settings in Hosted Environments," *Proceedings of the 5th Workshop on Evaluation and Comparison of NLP Systems*, Mumbai, India, pp. 135-148, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[10] Saad Ullah et al., "LLMs cannot Reliably Identify and Reason about Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks," *2024 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, pp. 862-880, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[11] Yangruibo Ding et al., "Vulnerability Detection with Code Language Models: How Far are we?," *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, Ottawa, ON, Canada, pp. 1729-1741, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[12] Avishree Khare et al., "Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities," *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Napoli, Italy, pp. 103-114, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[13] OWASP, OWASP Benchmark. [Online]. Available: https://owasp.org/projects/benchmark

[14] Roshan N. Rajapakse et al., "Challenges and Solutions when Adopting DevSecOps: A Systematic Review," *Information and Software Technology*, vol. 141, pp. 1-26, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[15] Opengrep. [Online]. Available: https://github.com/opengrep/opengrep

[16] GitHub, Codeql-cli-binaries. [Online]. Available: https://github.com/github/codeql-cli-binaries

[17] Ziyang Li, Saikat Dutta, and Mayur Naik, "IRIS: LLM-assisted Static Analysis for Detecting Security Vulnerabilities," *International Conference on Learning Representations (ICLR)*, vol. 2025, pp. 1-24, 2025. [Google Scholar] [Publisher Link]

[18] Nasif Imtiaz, Seaver Thorn, and Laurie Williams, "A Comparative Study of Vulnerability Reporting by Software Composition Analysis Tools," *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1-11, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[19] Huimin Hu et al., "An Empirical Study of Suppressed Static Analysis Warnings," *Proceedings of the ACM Software Engineering (FSE)*, pp. 290-311, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[20] Larissa Braz et al., "An Exploratory Study on Regression Vulnerabilities," *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 12-22, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[21] Sarah Heckman, and Laurie Williams, "A Systematic Literature Review of Actionable Alert Identification Techniques for Automated Static Code Analysis," *Information and Software Technology*, vol. 53, no. 4, pp. 363-387, 2011. [CrossRef] [Google Scholar] [Publisher Link]

[22] Zhaoqiang Guo et al., "Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities," *IEEE Transactions on Software Engineering*, vol. 49, no. 12, pp. 5154-5188, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[23] Xiuting Ge et al., "Machine Learning for Actionable Warning Identification: A Comprehensive Survey," *ACM Computing Surveys*, vol. 57, no. 2, pp. 1-35, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[24] Anant Kharkar et al., "Learning to Reduce False Positives in Analytic Bug Detectors," *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pp. 1307-1316, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[25] Hong Jin Kang, Khai Loong Aw, and David Lo, "Detecting False Alarms from Automatic Static Analysis Tools: How far are we?," *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pp. 698-709, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[26] Jaime Spacco, David Hovemeyer, and William Pugh, "Tracking Defect Warnings Across Versions," *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR)*, Shanghai, China, pp. 133-136, 2006. [CrossRef] [Google Scholar] [Publisher Link]

[27] Pavel Avgustinov et al., "Tracking Static Analysis Violations over Time to Capture Developer Characteristics," *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, Florence, Italy, pp. 437-447, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[28] Miao Miao et al., "An Extensive Empirical Study of Nondeterministic Behavior in Static Analysis Tools," *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, Ottawa, Canada, pp. 1064-1076, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[29] Qingzhou Luo et al., "An Empirical Analysis of Flaky Tests," *Proceedings of the 22nd ACM SIGSOFT International Symposium Foundations of Software Engineering (FSE)*, pp. 643-653, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[30] Owain Parry et al., "A Survey of Flaky Tests," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 1, pp. 1-74, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[31] Austin Mordahl et al., "ECSTATIC: An Extensible Framework for Testing and Debugging Configurable Static Analysis," *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, pp. 550-562, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[32] Cheng Wen et al., "Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far are we?," *ACM Transactions on Knowledge Discovery from Data*, vol. 18, no. 7, pp. 1-34, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[33] Binyuan Hui et al., "Qwen2.5-Coder Technical Report," *arXiv preprint*, pp. 1-32, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[34] Ollama. [Online]. Available: https://github.com/ollama/ollama

[35] Claude Platform Docs, Models Overview. [Online]. Available: https://platform.claude.com/docs/en/models/overview