

Ensuring continuous security in CI/CD pipelines through LLM-assisted reduction of false positives in static analysis

Moayad Terro, Victoriia Korzhuk

Abstract — Static application security testing tools embedded in CI/CD pipelines generate considerable volumes of false positive alerts, which degrade developer trust and impede automated workflows. This paper describes an architecture where a large language model functions as a secondary triage component positioned downstream of a static analyzer within a CI/CD pipeline. Flagged code fragments along with contextual metadata are passed to the model, which classifies whether each reported finding reflects an actual vulnerability. Experiments on open-source Java and Python repositories show that this configuration reduces the false positive rate from approximately 61% to 17%, raising precision from 0.389 to 0.833 while retaining 91.3% of the true positives reported by the analyzers. The findings suggest that LLM-assisted filtering can be practically integrated into existing CI/CD workflows without substantial latency overhead.

Keywords: CI/CD, static analysis, large language model, vulnerability detection, DevSecOps, Semgrep, SonarQube

I INTRODUCTION

The shift toward continuous integration and continuous delivery has fundamentally changed how development teams manage code quality and security. In a typical CI/CD pipeline, every commit triggers an automated chain of steps that may include compilation, unit testing, dependency resolution, and security scanning. Static application security testing tools occupy a well-established position in this chain: they inspect source code or compiled artifacts without executing them, searching for patterns that correspond to known vulnerability classes such as SQL injection, cross-site scripting, insecure deserialization, and related issues [1,2,3,4]. Despite their utility, SAST tools carry a well-documented limitation. A substantial fraction of their alerts does not correspond to real exploitable flaws. This fraction, referred to as the false positive rate, commonly ranges from 40% to 70% depending on the tool, the codebase, and the rule configuration in use [1,5,6]. When developers encounter repeated warnings that turn out to be harmless, they gradually stop treating alerts as meaningful signals. This phenomenon is sometimes called alert fatigue, and its practical consequences are non-trivial: actual

vulnerabilities embedded among dozens of spurious warnings receive insufficient attention, and security review becomes a procedural formality rather than a substantive check [2,7,8].

The problem intensifies in high-velocity environments where pull requests are opened and merged multiple times per day. A pipeline that blocks merges pending security review must resolve alerts quickly; if triage requires manual inspection of every flagged location, the overhead becomes untenable. Teams respond by either relaxing blocking rules, suppressing entire rule categories, or deferring security review to later stages where remediation costs tend to be higher. None of these adaptations is satisfactory from a security standpoint, yet all of them are common in practice. Large language models have demonstrated unexpectedly strong capabilities in code comprehension tasks [9,10,11]. They can explain what a given function does, reason about data flow across multiple call sites, and assess whether a particular variable reaches a sensitive sink in a way that constitutes an actual vulnerability rather than a superficial pattern match. These capabilities suggest a natural fit for false positive triage: rather than relying solely on syntactic or semantic pattern matching, a pipeline could route flagged code fragments to a language model that evaluates each finding against a broader understanding of the surrounding code context [12,13,14,15,16]. The core questions this paper addresses are practical. Can an LLM-based filter be inserted into a CI/CD pipeline, with acceptable latency and cost, and meaningfully reduce the proportion of alerts that developers must manually examine? The study investigates this by constructing a triage architecture, running it on a labeled dataset of SAST findings from real open-source projects, and measuring the resulting changes in precision, recall, and pipeline latency. The rest of the paper is organized as follows. Section II describes the experimental setup, the dataset construction procedure, the triage architecture, and the evaluation metrics. Section III presents quantitative results. Section IV discusses failure modes, cost implications, and generalizations. Section V concludes.

II METHODOLOGY

The study used a corpus of 18 open-source repositories, nine written in Java and nine in Python, selected from GitHub according to three criteria. Each repository had to have at least one confirmed security

vulnerability documented in a public advisory during its history, taken as evidence of a codebase subject to genuine security scrutiny, had to exceed 10,000 lines of non-comment source code, and had to be actively maintained at the time of data collection. For each repository, the default branch as checked out on 20 May 2026 was used as the primary analysis target, supplemented by two or three earlier revisions to increase sample diversity. Static analysis was performed using two tools: Semgrep with its default security ruleset [17] and SonarQube Community Edition with the standard quality profile [18]. Both tools were run in their default configurations without custom rule additions or suppressions, so that the results would reflect a realistic out-of-the-box deployment rather than a tuned configuration. All alerts produced by either tool were collected into a unified finding set. Duplicate findings, defined as alerts from both tools pointing to the same line and vulnerability class in the same file, were merged into a single record attributed to both tools.

Each finding was manually labeled by two independent reviewers as either a true positive, meaning a genuine vulnerability or security flaw with exploitable consequence, or a false positive, meaning an alert with no meaningful security implication in the given codebase context. Reviewers had access to the full repository source, the relevant advisory documentation, and automated data flow traces generated by a separate program analysis tool used only for labeling purposes. Disagreements between reviewers arose in approximately 9% of cases and were resolved through structured discussion with reference to the program analysis traces.

The triage component was built around GPT-4.1 [19] accessed through the OpenAI API. For each finding produced by the static analyzer, the system assembled a prompt containing four elements. The first was a description of the reported vulnerability class along with the rule identifier that triggered the alert. The second was the flagged code fragment, typically spanning 20 to 40 lines centered on the reported location. The third was limited call graph context extracted from the repository: the signatures of functions that call the flagged location and the signatures of functions it calls, without their full bodies, to keep the prompt focused on the flagged location while still providing structural orientation. The fourth was an explicit instruction directing the model to reason about whether the reported pattern constitutes an actual vulnerability given the provided context, and to respond with a binary classification followed by a brief justification of no more than three sentences. The prompt was designed to constrain the model's output format while leaving the reasoning process unconstrained. Responses were parsed programmatically using a lightweight extraction routine that looked for the classification label in the first sentence of the response. Cases where the model returned an ambiguous or malformed response were treated as abstentions and excluded from precision and recall calculations. Of the 2,941 alerts raised by the analyzers, 94 (3.2%) were abstentions; the 2,847 findings reported

throughout this paper are the evaluation set that remains after their exclusion.

The integration into the pipeline was implemented as a Python microservice that subscribes to the artifact channel of a Jenkins-based CI system. After Semgrep and SonarQube complete their runs and publish their finding reports, the microservice fetches the relevant code fragments from the repository, constructs prompts, dispatches requests to the LLM API in parallel batches of up to 10 requests, collects responses, and writes a filtered finding report back to the pipeline artifact store. Downstream pipeline stages receive the filtered report as a prioritized review queue, while the complete unfiltered report is retained as the authoritative artifact for audit and escalation. Findings the model classifies as false positives are demoted within the queue rather than removed from the retained report, so the filter reduces reviewer workload without narrowing the set of findings available for inspection. The choice of batch size of 10 was determined empirically by measuring throughput at batch sizes of 5, 10, 15, and 20 and selecting the size that minimized average per-finding latency without triggering API rate limiting under the account tier used. At batch size 10, average per-finding processing time was 1.3 seconds; at batch size 20, the rate limiter was triggered intermittently, increasing average time to 2.1 seconds.

Performance was evaluated using precision, recall, and F1 score calculated against the manually labeled ground truth. Precision measures the fraction of findings passed through by the filter that are genuine vulnerabilities. Recall measures the fraction of genuine vulnerabilities that the filter does not suppress. F1 score is the harmonic mean of the two, giving equal weight to both. Beyond classification metrics, latency overhead was measured per pipeline run, where one run corresponds to the analysis of a single repository revision, as wall-clock time added to the pipeline by the triage service and averaged across three independent executions of each run on a dedicated CI agent with a fixed hardware specification.

III RESULTS

Across the full evaluation set of 2,847 findings that received a parseable label, the LLM-based filter achieved a precision of 0.833 and a recall of 0.913, producing an F1 score of 0.871. For comparison, the unfiltered output of the static analyzers had a precision of 0.389 and a recall of 1.0. The latter holds by construction rather than by measurement: the ground truth is defined over the alerts the analyzers produced, so the unfiltered configuration retains all 1,107 true positives in that set. Recall throughout this paper is therefore relative to the analyzer-flagged population, not to all vulnerabilities present in the code. The filter raised precision by approximately 44.4 percentage points while suppressing only 8.7% of true positives, a trade-off that appears favorable for most practical security workflows.

Table 1 presents a performance breakdown by language and tool combination. Because duplicate findings

were merged and attributed to both tools, the per-tool rows are not disjoint and do not sum to the overall row. No significance testing was performed on the differences between configurations, so the comparisons that follow describe observed values rather than established effects. Java repositories showed higher precision gains than Python ones in this sample. One possible explanation is the more structured type of system in Java and the correspondingly more reliable call graph extraction available for statically typed code, which would give the model more precise context about variable types and method dispatch. Because this study includes no ablation isolating the contribution of call graph context, the explanation remains a hypothesis. The difference between Semgrep and SonarQube was modest overall, though Semgrep findings showed slightly lower initial precision in both languages, and the LLM filter achieved correspondingly slightly higher precision gains on Semgrep output. This is consistent with the expectation that more pattern-driven tools generate more findings whose validity depends on context rather than structure [1,6,15].

Table 1. Precision and recall by language and tool combination

Configuration	Unfiltered Precision	Filtered Precision	Recall (Filtered)	F1 (Filtered)
Java, Semgrep	0.341	0.856	0.921	0.887
Java, SonarQube	0.398	0.848	0.908	0.877
Python, Semgrep	0.362	0.805	0.901	0.850
Python, SonarQube	0.421	0.822	0.922	0.869
Overall	0.389	0.833	0.913	0.871

Latency measurements showed that the triage service added an average of 47 seconds to total pipeline duration across all analyzed revisions, each of which constitutes one pipeline run carrying approximately 40 findings on average. The range was 31 to 84 seconds, corresponding to runs of roughly 26 to 71 findings, and varied with the number of findings the static analyzers produced for a given revision. In no case did the added latency push total pipeline duration beyond the threshold that triggers timeout-based failures in the Jenkins configuration used. For runs with fewer than 30 findings, the overhead was consistently below 35 seconds.

Table 2 shows the distribution of finding counts before and after filtering, broken down by vulnerability class. Injection-related findings and path traversal findings benefited most from filtering, with raw false positive rates above 71% reduced to 18.0% and 16.1% respectively. The model appeared particularly effective at identifying cases where user input was validated or sanitized before reaching the flagged sink, even when the sanitization occurred through indirect call chains. Cryptographic misuse findings showed the smallest improvement in false positive rate. This is consistent with the observation that cryptographic issues are often directly apparent from code

structure, leaving less room for context-sensitive reasoning to add value [15,20].

Table 2. Finding counts and false positive rates before and after filtering by vulnerability class

Vulnerability Class	Raw Findings	Raw FP Rate	Filtered Findings	Post-filter FP Rate
SQL Command Injection	481	73.2%	139	18.0%
Path Traversal	374	71.4%	112	16.1%
Cross-site Scripting	312	58.3%	143	14.0%
Insecure Deserialization	218	54.1%	107	12.1%
Cryptographic Misuse	287	41.1%	201	22.4%
Other / Miscellaneous	1,175	59.8%	512	16.0%
Total	2,847	61.1%	1,214	16.7%

The overall finding count dropped from 2,847 to 1,214 after filtering, a reduction of 57.4%. Of the 1,633 findings suppressed by the filter, manual review confirmed that 1,537 (94.1%) were genuine false positives and 96 (5.9%) were true positives incorrectly suppressed. All aggregate values reported in this section are derived from the single confusion matrix given in Table 3; the corresponding integer counts of true positives, false positives, and false negatives are stated there explicitly. The 96 suppressed true positives constitute the false negatives of the filtered configuration, so precision rises from $1,107/2,847 = 0.389$ before filtering to $1,011/1,214 = 0.833$ after filtering, and recall is $1,011/1,107 = 0.913$.

Table 3. Aggregate confusion matrix before and after filtering

Alert Set	True Positives	False Positives	Total
Before filtering	1,107	1,740	2,847
Retained after filtering	1,011	203	1,214
Suppressed by filter	96	1,537	1,633

IV DISCUSSION

The results indicate that LLM-assisted triage addresses the false positive problem at a scale that would be practically meaningful in real development environments. Reducing the volume of alerts requiring human attention from roughly 2,800 to roughly 1,200, while retaining more than 91% of the true positives the analyzers reported, changes the character of the security review task considerably. A reviewer working through 1,214 findings at a 17% false positive rate faces a different workload than one confronting the unfiltered set, and the probability of a real vulnerability being overlooked in a shorter, higher-precision list is correspondingly lower [2,7,8]. Several failure modes deserve examination. The 96 true positives that the filter suppressed were concentrated in cases where the vulnerable pattern depended on runtime behavior not apparent from the static code fragment. In

several instances, a variable passed to a sensitive sink was conditionally sanitized through a helper function whose body was not included in the prompt context, and the model inferred from the function name that sanitization was performed, which turned out to be incorrect. This suggests a clear path for improvement: enriching the prompt with deeper call resolution, including function bodies at one or two levels of indirection, could recover a portion of these suppressed findings. The practical constraint is prompt length, which at deeper resolution levels may require summarization strategies or a model with a larger context window. The residual false positive rate of 16.7% still leaves room for further reduction, though the improvement relative to the 61.1% baseline is already substantial. In a real deployment, 17 to 33 false alerts per 100 to 200 total findings represents a workload that most security teams can absorb, particularly if dismissed findings are automatically recorded to avoid repeat review of identical patterns in subsequent pipeline runs. A suppression registry of this kind would not only reduce reviewer burden over time but would also provide a feedback signal that could be used to fine-tune the prompt or to update the static analyzer's rule configuration [5,21,22]. The cost and dependency implications of routing every finding through an external API are worth examining. At pricing levels current at the time of writing, the average inference cost per pipeline run in the experimental setup was approximately \$0.18, which is negligible compared to other pipeline infrastructure costs. The dependency on an external API introduces an availability risk: if the API is unreachable, the triage service cannot function. This risk could be mitigated by running a locally hosted open-weight model as a fallback, accepting some performance degradation. The deployed configuration additionally fails open: when triage is unavailable, the unfiltered analyzer report is forwarded unchanged and the run is marked as untriaged, as specified in A.5 [15,23,24]. The choice of GPT-4.1 as the backbone model was driven by its well-documented code reasoning capabilities at the time the experiments were designed. Models specifically fine-tuned on vulnerability datasets, such as those trained on the National Vulnerability Database or on curated pairs of vulnerable and patched code, might achieve higher recall on rare or structurally unusual vulnerability classes that GPT-4.1 handles less reliably [9,12]. The architecture described in Section II is model agnostic in that substituting a different model requires only changing the API endpoint and adjusting the prompt format to match the new model's conventions.

One consideration that the experimental setup does not fully capture is the effect of prompt engineering choices on model performance. The prompt format used here, with a fixed four-element structure and a constrained output format, was developed without a formally partitioned validation set. During the prompt-design stage, model outputs on representative examples were inspected manually, and these checks were used qualitatively to refine wording and output format rather than to tune inference parameters or to select the results reported here.

The prompt was then fixed and applied uniformly to every evaluation finding, as recorded in A.4. The absence of a separate held-out validation set remains a methodological limitation of this study, and a more systematic optimization using techniques such as chain-of-thought prompting or few-shot exemplars might yield different results. Few-shot prompting, where the prompt includes examples of confirmed true positives and false positives from the same codebase, could improve performance on repositories with distinctive coding conventions that the model's general training may not have captured well [25,26]. From a broader methodological standpoint, the reliance on GPT-4.1 as an evaluator in a security-critical context raises a question about adversarial robustness [9,27,28]. A sophisticated attacker who knows that LLM triage is in the pipeline might craft code that looks innocuous to the model while remaining exploitable. This attack surface is real but qualitatively different from the problem this work addresses, which is reducing false positives in ordinary development workflows. Defense-in-depth principles apply: LLM triage should be understood as one layer in a broader security process, not as a standalone gate.

V CONCLUSION

This paper described and evaluated an approach to reducing false positive alerts produced by static analysis tools in CI/CD pipelines by inserting a large language model as a secondary triage layer. The experimental setup involved 2,847 manually labeled findings from 18 open-source repositories, drawn from 2,941 analyzer alerts after excluding 94 abstentions, analyzed using Semgrep and SonarQube. The LLM-based filter raised precision from 0.389 to 0.833 while maintaining a recall of 0.913, with a pipeline latency overhead averaging 47 seconds per run. The total alert volume presented to downstream reviewers dropped by 57.4%. The approach requires no modification to the static analysis tools themselves and integrates into existing pipeline architectures through a lightweight microservice layer. The primary limitation concerns findings whose validity depends on function bodies not included in the prompt context, where the model's reasoning about variable provenance is unreliable and produces a small but non-negligible number of suppressed true positives. Addressing this limitation through deeper code context extraction is the most direct direction for future work. The practical implication is that LLM-assisted triage appears ready for controlled deployment in teams that currently experience significant alert fatigue from their static analysis tooling, provided it is deployed as a prioritization layer over a retained unfiltered report rather than as a gate on which findings reach review. The approach does not eliminate human review, but it narrows the review surface to findings that have passed a context-sensitive plausibility check, which changes the nature of the reviewer's task from triaging a large noisy list to examining a shorter set of higher-confidence candidates.

VI REFERENCES

- [1] Aloraini B. et al. An empirical study of security warnings from static application security testing tools // *Journal of Systems and Software*. – 2019. – Vol. 158. – Art. 110427.
- [2] Sadowski C. et al. Lessons from building static analysis tools at Google // *Communications of the ACM*. – 2018. – Vol. 61, no. 4. – pp. 58–66.
- [3] Wadhams Z., Reinhold A. M., Izurieta C. Automating static code analysis through CI/CD pipeline integration // *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)*. – IEEE, 2024. – pp. 119–125.
- [4] Chomatek L. et al. Decoding CI/CD Practices in Open-Source Projects with LLM Insights // *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. – ACM, 2025. – pp. 1638–1644.
- [5] Heckman S., Williams L. A systematic literature review of actionable alert identification techniques for automated static code analysis // *Information and Software Technology*. – 2011. – Vol. 53, no. 4. – pp. 363–387.
- [6] Muske T., Serebrenik A. Survey of approaches for handling static analysis alarms // *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. – IEEE, 2016.
- [7] Johnson B. et al. Why don't software developers use static analysis tools to find bugs? // *2013 35th International Conference on Software Engineering (ICSE)*. – IEEE, 2013. – pp. 672–681.
- [8] Christakis M., Bird C. What developers want and need from program analysis: an empirical study // *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. – ACM, 2016. – pp. 332–343.
- [9] Ullah S. 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)*. – IEEE, 2024. – pp. 862–880.
- [10] Salem D. O. et al. AI-Driven Continuous Integration: Automating Code Review and Deployment with LLMs // *2025 10th International Conference on Fog and Mobile Edge Computing (FMEC)*. – IEEE, 2025. – pp. 268–274.
- [11] Baqar M., Naqvi S., Khanda R. AI-Augmented CI/CD Pipelines: From Code Commit to Production with Autonomous Decisions // *arXiv preprint arXiv:2508.11867*. – 2025. – URL: <https://arxiv.org/abs/2508.11867> (accessed 20.05.2026).
- [12] Li Z., Dutta S., Naik M. IRIS: LLM-assisted static analysis for detecting security vulnerabilities // *arXiv preprint arXiv:2405.17238*. – 2024. – URL: <https://arxiv.org/abs/2405.17238> (accessed 20.05.2026).
- [13] Far S. M. T., Feyzi F., Esmi N. Enhancing Static Analysis through LLM-Based Semantic Validation // *2025 11th International Conference on Signal Processing and Intelligent Systems (ICSPIS)*. – IEEE, 2025. – pp. 119–123.
- [14] Agrawal V. et al. LLM-Driven SAST-Genius: A Hybrid Static Analysis Framework for Comprehensive and Actionable Security // *arXiv preprint arXiv:2509.15433*. – 2025. – URL: <https://arxiv.org/abs/2509.15433> (accessed 20.05.2026).
- [15] Ameen M. R. et al. QASecClaw: A Multi-Agent LLM Approach for False Positive Reduction in Static Application Security Testing // *arXiv preprint arXiv:2605.01885*. – 2026. – URL: <https://arxiv.org/abs/2605.01885> (accessed 20.05.2026).
- [16] Patil A. A., Patil L. A., Rambola R. Toward Hybrid LLM-Assisted Vulnerability Detection for Continuous Integration: A Bias-Free Framework with a Context-Rich Dataset // *2025 IEEE DELCON-International Conference on Recent Smart Technologies in Engineering for Sustainable Development*. – IEEE, 2025. – pp. 1–4.
- [17] Semgrep OSS [Electronic resource]. – URL: <https://semgrep.dev/> (accessed 20.05.2026).
- [18] SonarQube Community Build [Electronic resource]. – URL: <https://www.sonarsource.com/products/sonarqube/> (accessed 20.05.2026).
- [19] OpenAI. Introducing GPT-4.1 [Electronic resource]. – 2025. – URL: <https://openai.com/index/gpt-4-1/> (accessed 20.05.2026).
- [20] Xia Y. et al. Beyond Static Pattern Matching? Rethinking Automatic Cryptographic API Misuse Detection in the Era of LLMs // *arXiv preprint arXiv:2407.16576*. – 2024. – URL: <https://arxiv.org/abs/2407.16576> (accessed 20.05.2026).
- [21] Yang X. et al. Learning to recognize actionable static code warnings // *Empirical Software Engineering*. – 2021. – Vol. 26. – Art. 56.
- [22] Mittal A., Venkatesan V. Leveraging Generative AI for Proactive Security and Automated Remediation in Cloud-Native CI/CD Pipelines // *International Conference on Software Engineering and Data Engineering*. – Springer, 2025. – pp. 18–39.
- [23] Mittal A., Venkatesan V. Practical Integration of Large Language Models into Enterprise CI/CD Pipelines for Security Policy Validation: An Industry-Focused Evaluation // *2025 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. – IEEE, 2025. – pp. 197–203.
- [24] Kisielewicz M., Kotzbach P., Kędziora M. Enhancing DevSecOps Through Large Language Model Integration: A Pipeline-Centric Approach // *International Conference on Computational Collective Intelligence*. – Springer, 2025. – pp. 124–137.
- [25] Brown T. B. et al. Language models are few-shot learners // *Advances in Neural Information Processing Systems*. – 2020. – Vol. 33. – pp. 1877–1901.
- [26] Wei J. et al. Chain-of-thought prompting elicits reasoning in large language models // *Advances in Neural Information Processing Systems*. – 2022. – Vol. 35. – pp. 24824–24837.
- [27] Kiashemshaki K., Torkamani M. J., Mahmoudi N. Secure coding for web applications: Frameworks, challenges, and the role of LLMs // *arXiv preprint arXiv:2507.22223*. – 2025. – URL: <https://arxiv.org/abs/2507.22223> (accessed 20.05.2026).
- [28] Bhardwaj A. K. Enhancing DevSecOps with Generative AI for Secure Code Review and Threat Modeling // *2026 6th International Conference on Image Processing and Capsule Networks (ICIPCN)*. – IEEE, 2026. – pp. 912–920.

APPENDIX A. EXPERIMENTAL CONFIGURATION

This appendix records the configuration of the experimental pipeline described in Section II. The analysis runs were carried out in May 2026.

A.1 Corpus and analyzed revisions

The corpus comprised 18 public repositories, 9 in Java and 9 in Python. Each repository was analyzed at the head of its default branch as checked out on 20 May 2026, together with two or three earlier revisions of the same repository, giving approximately four analyzed revisions per repository and 71 analyzed revisions in total. The analyzed code corresponds to each project's mainline development state on the stated date rather than to any tagged release. Table A.1 lists the repositories and the public advisories that motivated their inclusion.

Table A.1. Analyzed repositories and the public advisories motivating their inclusion

Language	Repository	Example public advisory
Java	Apache Tomcat	CVE-2020-1938 (Ghostcat); additional GHSAs
Java	Apache Struts	CVE-2017-5638
Java	Apache Kafka	Multiple CVEs and GHSAs
Java	Jenkins	Numerous CVEs and GitHub advisories
Java	Elasticsearch	Multiple CVEs
Java	Spring Framework	Multiple CVEs (e.g., CVE-2022-22965, Spring4Shell)
Java	Apache Dubbo	Multiple CVEs
Java	Apache Shiro	CVE-2020-1957 and others
Java	Netty	Multiple GitHub security advisories
Python	Django	Multiple CVEs (e.g., CVE-2021-44420)
Python	Flask	Multiple security advisories
Python	Apache Airflow	Multiple CVEs
Python	Salt	CVE-2020-11651, CVE-2020-11652
Python	Home Assistant	Multiple GitHub advisories
Python	OpenStack Nova	Multiple CVEs
Python	Mercurial	Multiple security advisories
Python	Ansible	Multiple CVEs
Python	Zulip	Multiple GitHub security advisories

A.2 Static analyzers

Semgrep v1.163.0 was run with the p/default rule set, and SonarQube Community Edition v26.5.0.122743 with the sonar way quality profile for both Java and Python. Both tools were executed in their shipped configurations, with no custom rules, no disabled rules, and no inline or project-level suppressions. The rule sets were those distributed as of May 2026.

A.3 Language model and inference settings

Triage requests were issued to the OpenAI Chat Completions API using GPT-4.1, model identifier gpt-4.1-2025-04-14. The model's context window is approximately

one million tokens; assembled prompts were far shorter, so no prompt required truncation. Inference used temperature 0.2, top_p 0.1, and a maximum completion length of 4,096 tokens, with 42 as the random seed. The seed parameter provides best-effort rather than guaranteed reproducibility on the OpenAI API and run-to-run variation in classification outcomes was not measured. Requests were dispatched in parallel batches of up to 10, with 4 retries under exponential backoff. Responses that did not yield a parseable label after retries were recorded as abstentions together with malformed responses.

A.4 Prompt template

The template below was instantiated once per finding and held fixed across the evaluation set; braced fields denote values substituted at request time.

System message:

You are a security code reviewer. You assess whether a static analysis finding describes a real, exploitable vulnerability in the code as given. Answer with a single label on the first line, then a justification of at most three sentences.

User message:

```
Reported vulnerability class:
{vuln_class}\nAnalyzer rule identifier:
{rule_id}\n\nFlagged code fragment ({file_path},
lines {start}-
{end}):\n{code_fragment}\n\nCallers of the
flagged location (signatures
only):\n{caller_signatures}\n\nCallees of the
flagged location (signatures
only):\n{callee_signatures}\n\nDetermine whether
the reported pattern constitutes an actual
vulnerability given the context provided.
Respond with exactly TRUE_POSITIVE or
FALSE_POSITIVE on the first line, followed by a
justification of no more than three sentences.
```

The label was taken from the first line of the response by exact string match; responses in which neither label or both labels appeared were recorded as abstentions and excluded from the precision and recall figures.

A.5 Operational policy

The triage service is deployed as a prioritization layer rather than as a gate on which findings reach review. The complete unfiltered analyzer report is retained for every run as the authoritative artifact and remains available for audit and escalation; the filtered report determines only the order and initial scope of human review. Findings for which no parseable label is obtained after the retry budget given in A.3 are recorded as abstentions and routed to human review unfiltered, so an abstention never removes a finding from the review queue. If the API is unreachable or a request exceeds its timeout, the service fails open: the run is annotated as untriaged and the unfiltered report is forwarded to downstream stages unchanged. Triage unavailability therefore degrades prioritization without weakening the security posture of the pipeline.