



# International Journal of Research Publication and Reviews

Journal Homepage: [www.ijrpr.com](http://www.ijrpr.com) ISSN 2582-7421

## ReleaseGuard AI: An Explainable Multi-Signal Release Risk Analyzer and Deployment Gate for CI/CD Pipelines

**Mrs. R. Jenifer <sup>a</sup>, Mahirunnisa S F <sup>b</sup>, Monika S<sup>b</sup>, Sanjanaa S <sup>b\*</sup>**

<sup>a</sup> Assistant Professor, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology (An Autonomous Institution), Coimbatore - 641 062, Tamil Nadu, India

<sup>b</sup> IV Year B.E. Computer Science and Engineering, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology (An Autonomous Institution), Coimbatore - 641 062, Tamil Nadu, India

### ABSTRACT

Continuous integration and continuous delivery (CI/CD) pipelines shorten release cycles, yet most pipelines still decide whether to ship using binary signals such as “did the tests pass?”. Such gates cannot express how risky a release is when the tests pass but the change touches authentication code, erodes test coverage or carries known vulnerabilities. This paper presents ReleaseGuard AI, an explainable release risk analyzer and deployment gate. The system fuses seven signals collected from Git, Pytest, Trivy and SonarQube (failed tests, coverage drop, CVE severity, code-quality debt, critical-module changes, change volume and deployment history) into a transparent weighted score between 0 and 100, applies hard overrides for failed tests and critical CVEs, and maps the score to ALLOW, WARN or BLOCK decisions that are enforced through CI exit codes. A second layer uses Google Gemini 2.5, with an offline heuristic fallback, to turn the flagged factors into remediation steps and a rollback plan without influencing the decision itself. Results are stored in SQLite, shown in a Streamlit dashboard and pushed to Slack or Teams. We evaluate the scoring engine on nine release scenarios with sensitivity and ablation analyses. The study shows graded, explainable decisions, and also exposes a calibration limit: without the overrides, a release can reach the 75-point block threshold only if every remaining signal saturates. We discuss this and other limitations and outline data-driven recalibration as future work.

Keywords: Release risk; DevOps; CI/CD quality gate; DevSecOps; explainable scoring; large language models; software quality

### 1. Introduction

Continuous integration and continuous delivery (CI/CD) allow teams to ship several times a day, but a higher release frequency also increases the number of opportunities for a bad change to reach production. The DevOps Research and Assessment programme treats the change failure rate and the time needed to restore service as the two stability measures that must be kept in check while delivery speeds up (Forsgren et al., 2018; Google Cloud DORA, 2024). A pipeline that only asks whether the build and the unit tests passed answers a narrow question. A release can pass every test and still be dangerous: it may rewrite payment or authentication code, lower coverage by twenty points, or ship a container image with a critical vulnerability. Release engineers then have to read Git diffs, test reports, SonarQube dashboards and Trivy scans separately and combine them by judgement, which is slow and subjective.

Research on change-level risk has mostly followed the just-in-time (JIT) defect-prediction line, which trains a classifier on historical commits to flag defect-inducing changes (Kamei et al., 2013; Zhao et al., 2023). Recent industrial work shows that carefully chosen diff-based features can support deployment decisions at scale (Kurup et al., 2026). These approaches, however, need labelled incident history, output a probability rather than a pipeline decision, and rarely explain their output in terms an engineer can act on. Teams that lack such history, such as student projects, start-ups and new services, are left with pass/fail gates.

This paper presents ReleaseGuard AI, an automated release gate that sits inside the CI/CD workflow. It computes a deployment risk score from 0 to 100 from seven DevOps signals, converts the score into an ALLOW, WARN or BLOCK decision, and explains the decision with an AI-generated remediation plan. The scoring layer is deliberately deterministic and rule based, so every score can be audited and reproduced; the AI layer only explains and never decides. The main contributions are:

- A transparent seven-signal weighted scoring model with hard overrides for failed tests and critical cves (Section 3);
- A two-mode explanation layer that uses Google Gemini 2.5 when an API key is available and a local heuristic synthesiser otherwise, so the gate keeps working offline (Section 3.4);
- An end-to-end implementation with a CLI gate, github Actions workflow, Docker and Kubernetes manifests, a sqlite audit log, webhook alerts and a Streamlit dashboard (Sections 3 and 4); and

- A scenario-based evaluation with baseline, sensitivity and ablation analyses, including an honest account of where the current calibration falls short (Sections 5 and 6).

The remainder of the paper is organised as follows. Section 2 reviews related work, Section 3 describes the design, Section 4 the implementation, Section 5 the evaluation, Section 6 the limitations, and Section 7 concludes.

## 2. Related Work

### 2.1 Change-level risk prediction

JIT defect prediction estimates whether a commit is likely to introduce a defect using change metrics such as size, diffusion and developer experience (Kamei et al., 2013). A systematic survey of JIT defect prediction is given by Zhao et al. (2023). Kurup et al. (2026) apply diff-aware features to deployment risk at Prime Video and report that change-volume metrics such as lines added and deleted are noisy predictors compared with structural complexity. This agrees with our decision to give change volume the smallest weight (5%). Learned models are powerful but depend on labelled history and are hard to inspect, whereas ReleaseGuard AI trades predictive power for transparency and immediate usability.

### 2.2 AI in DevSecOps and scanner reliability

Fu et al. (2025) review 99 AI-driven security approaches across the DevSecOps workflow and identify open challenges for their adoption. For vulnerability inputs, Javed and Toor (2021) assess the coverage and accuracy of container scanners on 59 Docker images and report that even the best-performing tool missed 34% of vulnerabilities. Boles et al. (2024) collected 927 Docker images and found that, among the 865 with vulnerabilities, Trivy and Gripe disagreed on both the number of vulnerabilities and their identifiers. Scanner output is therefore evidence, not ground truth, which motivates combining it with other signals instead of gating on a single scanner.

### 2.3 Static-analysis signals

SonarQube reports bugs, vulnerabilities and code smells that are often treated as technical debt. Lenarduzzi et al. (2020a) studied 21 open-source projects and found that violations classified by SonarQube as bugs were generally not fault-prone, and Lenarduzzi et al. (2020b) report that SonarQube issues have a significant but small effect on faults and changes. We therefore include code-quality metrics as a moderate (10%) contribution rather than a dominant one.

### 2.4 Large language models in DevOps

Surveys of large language models (LLMs) in software engineering report gains in code generation, testing and DevOps automation, together with hallucination, security vulnerabilities and auditability concerns (Bhalla and Jodhka, 2025), and limited context handling and lack of explainability (Radenković et al., 2025). Gemini 2.5 is a family of models with advanced reasoning, long-context and agentic capabilities (Comanici et al., 2025). These limitations shaped our architecture: the LLM receives computed facts and writes advice, but the score and the pipeline decision come only from the deterministic layer. Table 1 summarises how the approach families differ.

**Table 1 - Qualitative comparison of release-gating approaches.**

| Approach                                 | Output                   | Needs labelled history | Natural-language explanation | Gates the pipeline |
|------------------------------------------|--------------------------|------------------------|------------------------------|--------------------|
| Pass/fail CI gate                        | Binary                   | No                     | No                           | Yes                |
| Per-tool gates (Sonar, Trivy exit codes) | Binary per tool          | No                     | No                           | Yes                |
| JIT defect prediction (ML)               | Probability              | Yes                    | Limited                      | Rarely             |
| ReleaseGuard AI                          | 0–100 score, 3 decisions | No                     | Yes (LLM or offline)         | Yes                |

## 3. System Design

### 3.1 Architecture

ReleaseGuard AI is organised as a four-stage pipeline (Fig. 1): signal collection, risk scoring (Layer 1), AI remediation synthesis (Layer 2), and enforcement with persistence. Release data enter as a validated payload, flow through the scoring engine, and leave as a score, a decision, a ranked list of risk factors and an action plan.

### ReleaseGuard AI - four-stage pipeline

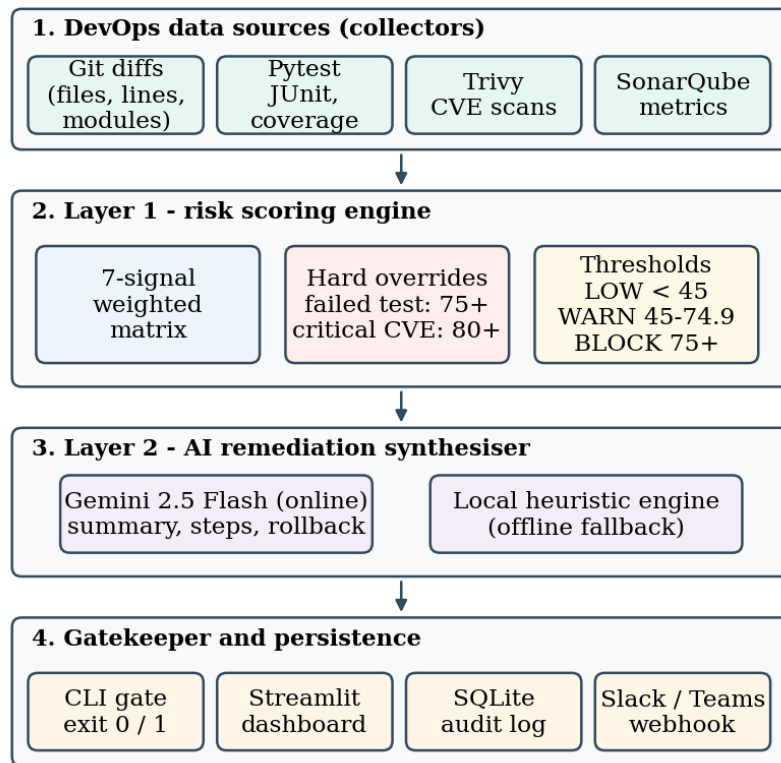


Fig. 1 - Architecture of ReleaseGuard AI.

### 3.2 Signal collection

Four collectors normalise raw tool output into a single Pydantic release record. The Git collector reads commit history and diffs through GitPython to obtain the number of modified files, lines added and deleted, commit count and the module paths touched. The test collector parses Pytest JUnit XML and coverage JSON to obtain total and failed tests, failed test names, current coverage and the drop against a stored baseline. The security collector reads the Trivy JSON report and counts Critical, High, Medium and Low CVEs, and also parses SonarQube output for bugs, vulnerabilities, code smells and duplicated-line density. Missing report files yield zero-valued summaries instead of exceptions, so the gate degrades gracefully when a scanner is not configured. Deployment history (recent failed deployments and failures of similar releases) is supplied in the payload.

### 3.3 Risk scoring model

Each signal  $i$  is converted to a subscore  $s_i$  between 0 and 100 and multiplied by a weight  $w_i$ . The weights sum to one, and the raw score is

$$R_{\text{raw}} = \sum_{i=1 \dots 7} w_i \cdot s_i \quad (1)$$

with the weights and subscores defined in Table 2. Here  $\Delta = \max(0, c_{\text{prev}} - c)$  is the coverage drop,  $c$  is current coverage,  $F$  is the number of failed tests,  $C$ ,  $H$  and  $M$  are the Critical, High and Medium CVE counts,  $B$ ,  $V$  and  $S$  are SonarQube bugs, vulnerabilities and smells,  $D$  is the duplicated-line density,  $m$  is the number of changed modules whose path matches one of eleven sensitive patterns (auth, authentication, security, payment, billing, checkout, finance, core, database, migrations, crypto),  $f$  is the number of modified files,  $l$  is the number of changed lines,  $r$  is the number of recent failed deployments and  $q$  is the number of failures of similar releases.

Table 2 - Signals, subscores and weights used by the scoring engine.

| Signal           | Source        | Subscore $s_i$ (capped at 100)                                                         | Weight $w_i$ |
|------------------|---------------|----------------------------------------------------------------------------------------|--------------|
| Failed tests     | Pytest        | $25 \cdot F$                                                                           | 0.25         |
| CVE severity     | Trivy         | $35 \cdot C + 15 \cdot H + 5 \cdot M$                                                  | 0.20         |
| Coverage drop    | Coverage JSON | $2.5 \cdot \Delta + 1.5 \cdot \max(0, 75 - c)$ ; applied when $\Delta > 0$ or $c < 70$ | 0.15         |
| Critical modules | Git           | $40 \cdot m$                                                                           | 0.15         |

| Signal             | Source        | Subscore $s_i$ (capped at 100)                                              | Weight $w_i$ |
|--------------------|---------------|-----------------------------------------------------------------------------|--------------|
| Code quality       | SonarQube     | $15 \cdot B + 20 \cdot V + 1.5 \cdot S + 2 \cdot D$ ; counted when above 15 | 0.10         |
| Deployment history | Release store | $30 \cdot r + 35 \cdot q$                                                   | 0.10         |
| Change volume      | Git           | $1.8 \cdot f + 0.03 \cdot l$ ; counted when above 20                        | 0.05         |

Two safety rules prevent a weighted sum from hiding a hard failure. If any test fails, the score is raised to at least 75.0, and if any critical CVE is present it is raised to at least 80.0:

$$R = \min(100, \max(R_{\text{raw}}, 75 \cdot [F > 0], 80 \cdot [C > 0])) \quad (2)$$

where  $[\cdot]$  is the Iverson bracket. The final score is mapped to a decision with fixed thresholds:  $R < 45$  gives LOW risk and ALLOW;  $45 \leq R < 75$  gives MEDIUM risk and WARN; and  $R \geq 75$  gives HIGH risk and BLOCK. The weights are 0.15, 0.25, 0.20, 0.10, 0.15, 0.05 and 0.10, which sum to 1.00. A direct consequence is that a release with no failed tests and no critical CVE can score at most 75.0, so it reaches the BLOCK threshold only if all six remaining signals saturate at once. Each signal that passes its reporting threshold is also emitted as a risk factor with a title, category, impact score, description and severity, which feeds the explanation layer and the dashboard.

### 3.4 AI remediation synthesiser

After scoring, the explanation layer builds a prompt containing the release identifier, score, level, decision, modified files, coverage and previous coverage, failed tests, CVE counts, code-quality counts, changed modules and the list of risk factors. It asks for a response in a fixed format of SUMMARY, ACTIONABLE STEPS and ROLLBACK PLAN, which is parsed into a recommendation object. When the GEMINI\_API\_KEY environment variable is set, the request goes to Gemini 2.5 Flash through the google-genai SDK (Comanici et al., 2025). If the key is absent, the SDK is not installed, or the call fails for any reason, a local heuristic synthesiser generates the recommendation from the same facts: it proposes fixing the failing tests, patching critical CVEs, restoring coverage to the previous baseline, resolving SonarQube bugs and requesting a second review when authentication or payment modules changed, and it suggests a Kubernetes rollback. The recommendation is flagged as LLM-generated or offline, and it never changes the score or decision.

### 3.5 Gatekeeper, persistence and alerting

The command-line interface loads a release payload, runs the scoring and explanation engines, prints a 60-character-wide ASCII report suitable for CI logs, and stores the full report in a SQLite database. For WARN and BLOCK decisions it posts an alert to a configured Slack or Teams webhook. The process exits with code 1 for BLOCK and 0 otherwise, which makes a failed gate stop downstream deployment steps in GitHub Actions or GitLab CI. The block threshold, the webhook address and a JSON output path are command-line options. The supplied GitHub Actions workflow checks out the code, installs dependencies, runs Pytest with JUnit and coverage output, executes the gate and proceeds to the Kubernetes deployment step only if the gate succeeds.

### 3.6 Dashboard and deployment

A Streamlit dashboard has four tabs. The Live Risk Analyzer shows the release tag, score, level and decision, a Plotly gauge with the 45 and 75 thresholds, a six-axis radar chart of signal impacts, the ranked risk factors, the AI action plan and the raw ASCII report. The Signal Simulator lets an engineer change coverage, failed tests, CVE counts, files modified and changed modules with sliders and see the score and decision update immediately. The Release Analytics tab reads the SQLite store to show the number of analysed releases, the average risk score, the number of blocked releases, the deployment success rate and a risk trend line. The Integration Setup tab provides a ready-to-use workflow template and CLI reference. Screenshots of the tabs are shown in Fig. 2. The application is packaged with a multi-stage Dockerfile, a Docker Compose file and Kubernetes Deployment and Service manifests.

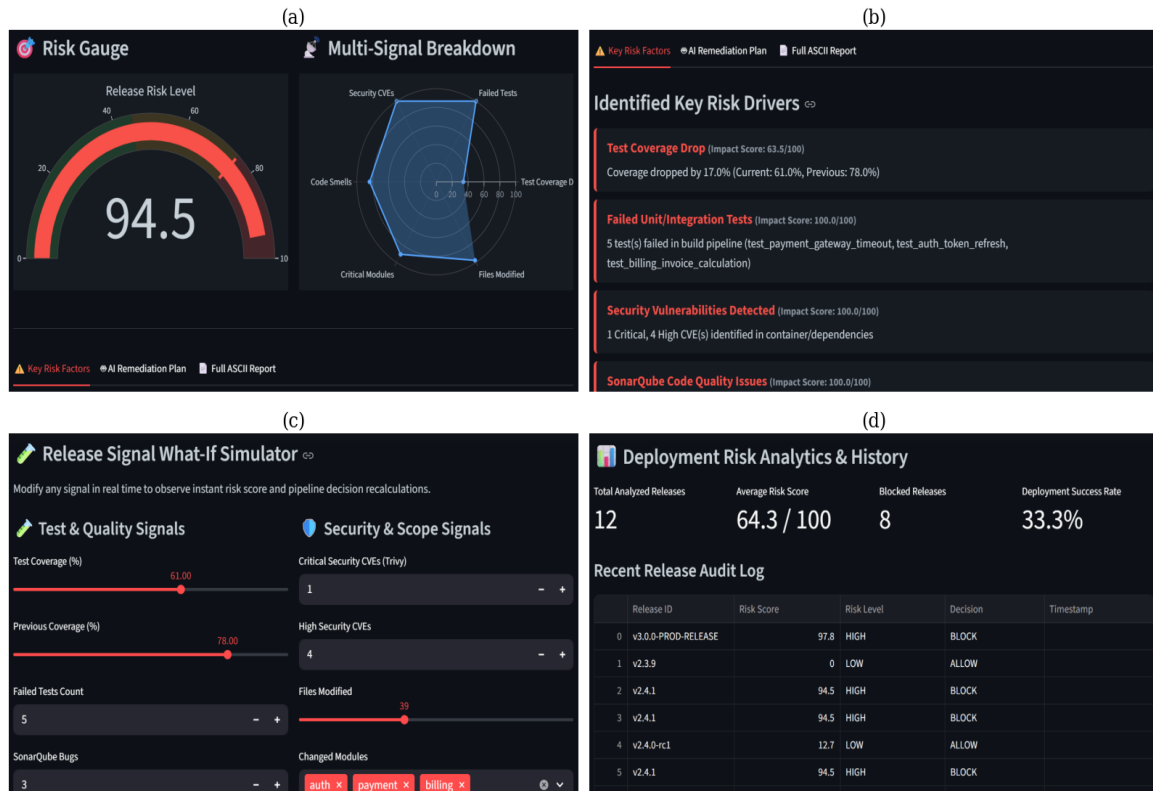


Fig. 2 - Streamlit dashboard: (a) Live Risk Analyzer with risk gauge and signal radar for the high-risk sample (94.5, BLOCK); (b) ranked key risk factors; (c) what-if signal simulator; (d) release analytics and audit log, populated by repeated runs of the sample payloads.

#### 4. Implementation and System Specification

ReleaseGuard AI is written in Python and comprises about 1,800 lines of code across the application package, the CLI and the test suite. Pydantic models validate the release payload, the risk factors and the report, which prevents malformed input from reaching the scoring engine. Table 3 lists the technology stack. Development and demonstration were carried out on a laptop or desktop running Windows 11; the containerised build runs on any Docker-capable host.

Table 3 - Technology stack.

| Layer                   | Technology                                                  | Role                                                              |
|-------------------------|-------------------------------------------------------------|-------------------------------------------------------------------|
| Language and validation | Python 3.10+, Pydantic v2                                   | Scoring logic, collectors, CLI, typed models                      |
| Dashboard               | Streamlit, Plotly, Pandas                                   | Live analyzer, simulator, analytics, charts                       |
| AI engine               | Google GenAI SDK (Gemini 2.5 Flash); local heuristic engine | Explanations, remediation steps, rollback plan                    |
| Data collection         | GitPython, Requests; Trivy, SonarQube, Pytest outputs       | Git diffs, CVE counts, quality metrics, test and coverage reports |
| Storage and alerts      | SQLite3; Slack/Teams webhooks                               | Release audit log, team metrics, notifications                    |
| Delivery                | Docker, Docker Compose, Kubernetes, GitHub Actions          | Packaging, deployment, CI quality gate                            |
| Testing                 | Pytest                                                      | 11 unit tests                                                     |

The test suite contains 11 test functions in three files. They cover the scoring thresholds for low and high-risk releases, the mapping of scores to decisions, the exact width of every line of the ASCII report, the offline explanation fallback, the handling of missing scanner files and the database operations.

## 5. Experimental Evaluation

### 5.1 Setup

No production incident data were available, so this evaluation does not measure predictive accuracy and makes no claim about reducing real change failures. Instead it examines how the scoring engine behaves on a controlled set of nine release scenarios. Scenarios S1, S2 and S7 are the low, medium and high-risk sample payloads shipped with the project. The other six were designed by the authors to probe specific behaviours: a critical CVE with green tests (S3), a sensitive-module change with a coverage loss (S4), a large refactor of non-sensitive code (S5), a single failing test (S6), an unstable deployment history in core modules (S8) and compounded moderate risk (S9). Each payload was scored by the project's risk calculator, which is deterministic, so the results can be reproduced by running the CLI on the same inputs. The dashboard reports the same values for the shipped samples: 94.5 for the high-risk release and 12.7 for the medium-risk release (Fig. 2). Two baselines are compared: B1, a test-only gate that blocks when any test fails, and B2, a gate that blocks on any failed test or critical CVE.

### 5.2 Scenario results

**Table 4 - Scenario results. Raw is the weighted sum before overrides (shown only when an override applied); B1 and B2 are the baseline gates.**

| ID | Scenario                      | Key inputs                                                              | Raw  | Score | Decision | B1    | B2    |
|----|-------------------------------|-------------------------------------------------------------------------|------|-------|----------|-------|-------|
| S1 | Routine fix (sample low)      | 3 files; coverage 92.0→92.5                                             | –    | 0.0   | ALLOW    | ALLOW | ALLOW |
| S2 | Minor feature (sample medium) | 14 files; cov. 76.5→74.0; 1 High CVE; 1 bug                             | –    | 12.7  | ALLOW    | ALLOW | ALLOW |
| S3 | Green tests, critical CVE     | 6 files; auth; 1 Critical + 1 High CVE                                  | 16.0 | 80.0  | BLOCK    | ALLOW | BLOCK |
| S4 | Sensitive-module change       | 28 files; auth, payment; cov. 80→62; 2 High CVE; 2 bugs                 | –    | 43.9  | ALLOW    | ALLOW | ALLOW |
| S5 | Large non-sensitive refactor  | 60 files, 5,000 lines; cov. 89→84                                       | –    | 14.2  | ALLOW    | ALLOW | ALLOW |
| S6 | Single failing test           | 1 failed test; otherwise clean                                          | 6.2  | 75.0  | BLOCK    | BLOCK | BLOCK |
| S7 | Sample high-risk release      | 5 failed tests; 1 Critical CVE; auth, payment, billing                  | –    | 94.5  | BLOCK    | BLOCK | BLOCK |
| S8 | Unstable history, core change | core, database; 2 recent failed deploys; 1 similar failure              | –    | 32.3  | ALLOW    | ALLOW | ALLOW |
| S9 | Compounded moderate risk      | 45 files; 4 sensitive modules; cov. 85→70; 3 High CVE; unstable history | –    | 55.2  | WARN     | ALLOW | ALLOW |

Fig. 3 shows how each score is composed. Three scenarios are blocked (S3, S6 and S7). The test-only gate B1 blocks only S6 and S7 and lets S3 through, although S3 carries a critical vulnerability; in S3 and S6 the weighted sum alone is only 16.0 and 6.2, so it is the hard overrides that produce the BLOCK. Gate B2 reaches the same blocking decisions as ReleaseGuard AI on this scenario set. The additional behaviour of ReleaseGuard AI is therefore the graded middle band, illustrated by S9 (55.2, WARN), together with the ranked factors and the remediation plan, rather than a different block set. Two results deserve attention. S4 modifies authentication and payment code, loses 18 coverage points and carries two High CVEs, yet scores 43.9, just 1.1 points below the WARN threshold. And the shipped medium-risk sample (S2) scores 12.7 and is therefore classified LOW, and the shipped high-risk sample (S7) scores 94.5.

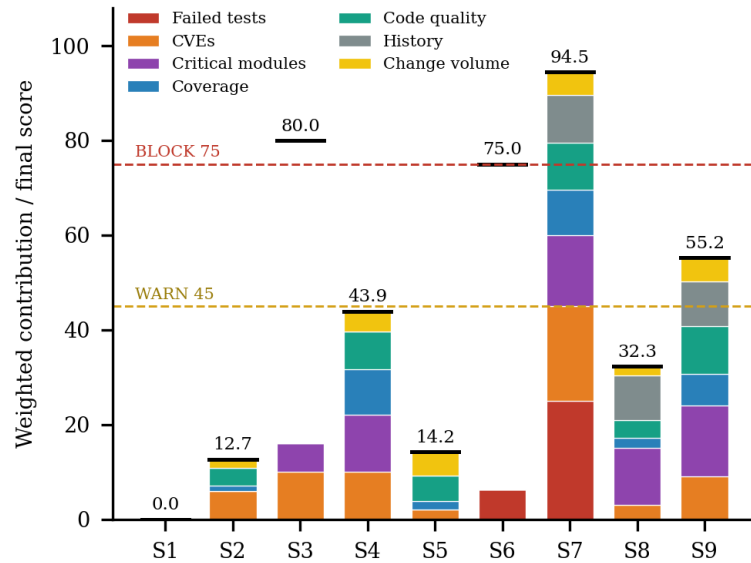


Fig. 3 - Weighted contribution of each signal per scenario. Black bars mark the final score after overrides.

### 5.3 Sensitivity analysis

We varied one input at a time from a clean baseline (Fig. 4). A single failing test moves the score from 0.0 to 75.0, and additional failures leave it at 75.0, so the override rather than the weighted sum determines the outcome. One critical CVE gives 80.0 whatever else is present, while High CVEs alone rise slowly (3.0 for one, 9.0 for three) and saturate at 20.0 from seven High CVEs, which can never reach WARN on their own. A coverage drop from 90% to 60% yields 14.6 points and saturates at 15.0 at 50%. Each sensitive module adds 6.0 points up to a cap of 15.0 at three modules, and change volume saturates at 5.0. The maximum contribution of each signal equals its weight times 100, which makes the influence of every signal explicit and bounded.

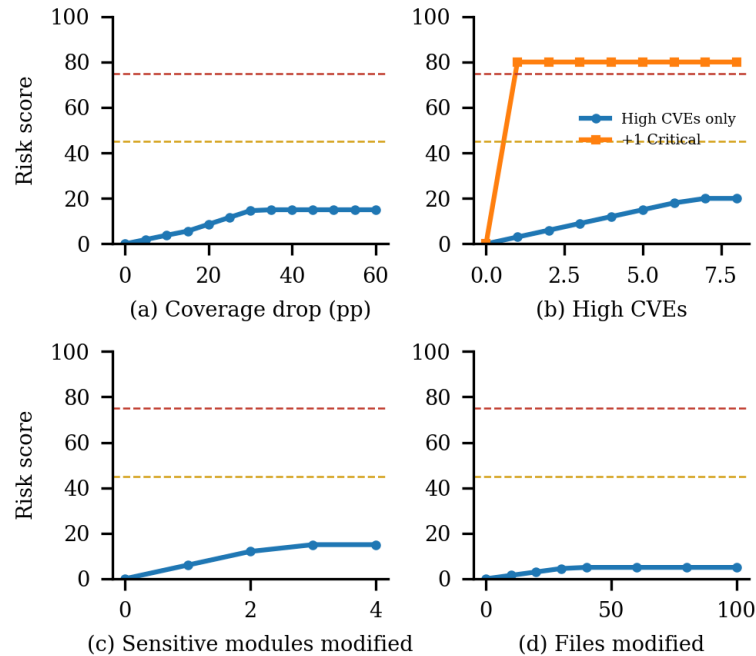


Fig. 4 - One-at-a-time sensitivity: (a) coverage drop, (b) High CVEs with and without one Critical CVE, (c) sensitive modules, (d) files modified. Dashed lines mark 45 and 75.

### 5.4 Ablation

To see which signals carry each decision, we set one signal to its neutral value and recorded the change in score for the three scenarios without failed tests or critical CVEs that fall in the LOW or WARN bands (Table 5). The sensitive-module signal has the largest effect in all three cases (-12.0, -12.0

and -15.0). For S9, removing any single signal other than the sensitive-module signal leaves the release in the WARN band (lowest result 45.2), whereas removing that signal drops it to 40.2 and changes the decision to ALLOW. Removing the history signal changes S8 by -9.5, showing that deployment history matters when other signals are moderate. Finally, if the two override conditions are cleared in S7 (failed tests set to 0 and critical CVEs set to 0), its weighted score falls from 94.5 to 69.5, which is only WARN, so the overrides are what keep that scenario blocked once the remaining signals are considered alone.

**Table 5 - Change in score when one signal is neutralised (points).**

| Signal removed     | S4 (43.9) | S8 (32.3) | S9 (55.2) |
|--------------------|-----------|-----------|-----------|
| Coverage drop      | -9.6      | -2.3      | -6.7      |
| CVE severity       | -10.0     | -3.0      | -9.0      |
| Code quality       | -7.9      | -3.6      | -10.0     |
| Critical modules   | -12.0     | -12.0     | -15.0     |
| Change volume      | -4.3      | -1.9      | -5.0      |
| Deployment history | 0.0       | -9.5      | -9.4      |

### 5.5 Test suite and run time

The 11 unit tests exercise the scoring thresholds, report formatting, fallback explanation and storage. Scoring one payload takes about 0.04 ms on average over 10,000 evaluations on a single core of our test container. This excludes file input and output, database writes and the optional LLM request, which dominate the end-to-end time of the gate.

## 6. Discussion and Limitations

Calibration. The weighted sum is linear and each subscore is capped, so no single signal other than the override signals can move the total far. The scenario results show the consequence: S4, which a human reviewer would probably flag, stays in the ALLOW band, and the WARN band is reached only when many signals are elevated together (S9). The 45 and 75 thresholds, the weights and the subscore coefficients are expert-chosen heuristics that have not been fitted to incident data. Non-linear aggregation (for example, interaction terms between sensitive modules and coverage loss) and threshold tuning against historical outcomes are the clearest next steps.

Input quality. Scanner output differs between tools in vulnerability counts and identifiers (Boles et al., 2024), SonarQube bugs are weak fault predictors (Lenarduzzi et al., 2020a) and change volume is a noisy signal (Kurup et al., 2026). The weights follow this evidence qualitatively, but we did not test it on real data. Teams should treat the score as decision support and keep a human in the loop for WARN decisions.

LLM layer. Hallucination and limited explainability are known risks of LLMs (Radenković et al., 2025). We reduce them by passing computed facts in a structured prompt, parsing a fixed output format and keeping the LLM outside the decision path. We did not evaluate the quality of the generated advice, and the offline mode produces simpler, rule-based text.

Validity. The nine scenarios were written by the authors and three of them are the project's own samples, so the evaluation shows behaviour and internal consistency, not accuracy or effectiveness. The claims that the gate would lower the change failure rate or the time to restore service are design goals that remain to be tested in a pilot deployment.

## 7. Conclusion and Future Work

ReleaseGuard AI turns a binary CI/CD gate into a graded and explainable release decision. It combines seven DevOps signals into an auditable 0 to 100 score, protects against hard failures with two overrides, explains the result with an LLM or an offline fallback, and enforces the decision through exit codes, alerts, a dashboard and an audit log. The scenario study confirms the intended behaviour, including the block on a critical CVE that a test-only gate misses, and also shows that the current linear calibration rarely produces WARN decisions. Future work will (i) calibrate weights and thresholds on real release outcomes, (ii) use Prometheus metrics to retrain the scoring model continuously, (iii) extend integration to GitLab CI, Jenkins and ArgoCD, (iv) add automated remediation, and (v) run a pilot study that measures change failure rate and time to restore.

### Acknowledgements

The authors thank the Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, Coimbatore, for the facilities and support provided for this work.

---

## References

- Bhalla, J. S., & Jodhka, M. K. (2025). Leveraging large language models in the software development lifecycle: Opportunities and challenges. *International Journal of Advanced Computer Science and Applications*, 16(11), 43–51.
- Boles, B., O'Donoghue, E., Manzi Muneza, R., Perkins, G., Izurieta, C., & Reinhold, A. M. (2024). Deciphering discrepancies: A comparative analysis of Docker image security. *Proceedings of the 24th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, 254–259. <https://doi.org/10.1109/SCAM63643.2024.00034>
- Comanici, G., et al. (Gemini Team, Google). (2025). Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv:2507.06261*.
- Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The science of lean software and DevOps*. IT Revolution Press.
- Fu, M., Pasuksmit, J., & Tantithamthavorn, C. (2025). AI for DevSecOps: A landscape and future opportunities. *ACM Transactions on Software Engineering and Methodology*, 34, 1–61. <https://doi.org/10.1145/3712190>
- Google Cloud DORA. (2024). *Accelerate State of DevOps Report 2024*. Google Cloud.
- Javed, O., & Toor, S. (2021). Understanding the quality of container security vulnerability detection tools. *arXiv:2101.03844*.
- Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., & Ubayashi, N. (2013). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6), 757–773.
- Kurup, M., Suh, H., Vaidyanathan, S., Vyas, P., Madabhushi, S., & Silyutin, Y. (2026). Deployment risk assessment using diff-aware features: A case study at Prime Video. *ASE 2026 Industry Showcase*. *arXiv:2607.06766*.
- Lenarduzzi, V., Lomio, F., Huttunen, H., & Taibi, D. (2020a). Are SonarQube rules inducing bugs? *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, London, Ontario, Canada.
- Lenarduzzi, V., Saarimäki, N., & Taibi, D. (2020b). Some SonarQube issues have a significant but small effect on faults and changes: A large-scale empirical study. *Journal of Systems and Software*, 170, 110750. <https://doi.org/10.1016/j.jss.2020.110750>
- Pytest Development Team. (n.d.). *pytest documentation*. <https://docs.pytest.org/>
- Radenković, B., Prokhorov, S., Radenković, M., & Labus, A. (2025). Application of large language models in software development: Review of the current state and development perspectives. *2025 6th International Workshop on Engineering Technologies and Computer Science (EnT)*, 1–7. <https://doi.org/10.1109/EnT68818.2025.11245717>
- SonarSource. (n.d.). *SonarQube documentation*. <https://docs.sonarqube.org/>
- Streamlit Inc. (n.d.). *Streamlit documentation*. <https://docs.streamlit.io/>
- Zhao, Y., Damevski, K., & Chen, H. (2023). A systematic survey of just-in-time software defect prediction. *ACM Computing Surveys*, 55(10), Article 201.