WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

AI code review & release-risk assistant – automated vulnerability detection and deployment readiness platform

Mrs. Manimegala Prasath[a], Vishwa B[a], Vijay K[a], Srihari K[a]

[a]Department of Computer Science and Engineering, Bachelor of Engineering, Sri Shakthi Institute of Engineering and Technology, Coimbatore-641062

A B S T R A C T :

The AI Code Review & Release-Risk Assistant is a web-based DevSecOps platform that inspects source code, finds security weaknesses and estimates how risky it is to release a software version. A developer submits a public GitHub repository or a ZIP archive. The system checks the input, places it in an isolated workspace, detects the programming languages and scans the code with an abstract-syntax-tree (AST) engine and a set of heuristic pattern detectors for Python, JavaScript/TypeScript, Java and C/C++. A universal scanner also looks for hard-coded credentials such as cloud access keys, personal access tokens and private keys. The findings can be explained by a Large Language Model (Google Gemini, OpenAI, or a local Ollama model) or by a built-in offline engine that needs no external service. A severity-weighted scoring model then converts all findings into a release-risk score from 0 to 100 and a clear verdict: GO FOR RELEASE, PROCEED WITH CAUTION or NO-GO. The platform is built with a FastAPI backend, uses temporary workspaces with several protections against unsafe archives, and includes a ground-truth benchmark. On its built-in benchmark of ten vulnerability scenarios across four languages, the system reached 100% precision and recall; because this benchmark is small, the figure shows that the detectors work as designed and is not a claim of general accuracy.

Keywords: Static Code Analysis, Vulnerability Detection, Release-Risk Scoring, DevSecOps, Large Language Models, Abstract Syntax Tree, Secret Detection, FastAPI

Introduction

Modern software is released faster than ever. Teams merge code many times a day, and every change can carry a security flaw, a leaked password or a hidden bug. Manual code review is slow and depends on the reviewer's experience and available time. Many weaknesses, such as SQL built from string concatenation or an access key pasted into a source file, are easy to miss during a busy review, yet they are exactly the problems that attackers look for.

Automated tools help, but most of them stop at a long list of warnings. A team still has to decide the question that matters in the end: is this version safe to release? Developers who are new to security also find a bare warning hard to act on, because it rarely explains why the code is dangerous or how to fix it.

To address this, we developed the AI Code Review & Release-Risk Assistant. It combines three ideas in one pipeline. First, it analyses the code with structure-aware static analysis, so it can reason about the program and not only about text. Second, it uses a Large Language Model, or an offline fallback, to explain each finding and suggest a fix. Third, it converts all findings into a single release-risk score and a decision (GO, CAUTION or NO-GO) together with a list of deployment blockers and a rollback plan. The result is a move from an unstructured list of warnings to a measurable and actionable release decision.

The main contributions of this work are: (i) a multi-language scanner covering Python, JavaScript/TypeScript, Java and C/C++ with a universal credential detector; (ii) a multi-provider AI review layer with a fully offline mode; (iii) a transparent, weighted release-risk model with blockers, blast-radius mapping and rollback guidance; and (iv) a secure repository-ingestion module with protection against path traversal and ZIP bombs.

Literature survey

Static application security testing (SAST) examines source code without running it and is one of the most widely used ways to find security defects early in development (Chess & West, 2007). Its strength is that it can inspect every code path. Its known weakness is that it may report false alarms and needs good rules and good presentation of results.

Bandit (PyCQA, 2024) is a well-known security linter for Python. It parses each file into an AST and runs checks for issues such as use of *eval* or unsafe subprocess calls. The proposed system also relies on the Python AST for Python files, but extends the idea to several languages and adds release-level scoring on top of the findings.

Semgrep (Semgrep, Inc., 2024) is a pattern-based analysis tool that supports many languages and lets teams write their own rules. Tools of this type are flexible and fast. They mainly report individual findings, and turning those findings into a release decision is usually left to the team. The AI Code Review & Release-Risk Assistant adds that missing step through its risk-scoring engine.

The OWASP Top Ten (OWASP Foundation, 2021) lists the most critical web application risks, including injection and insecure deserialization. The Common Weakness Enumeration (CWE; MITRE Corporation, 2024) catalogues software weakness types, including path traversal and uncontrolled resource consumption. The detectors in the AI Code Review & Release-Risk Assistant and its archive-handling defences are chosen with these two references in mind.

Large Language Models trained on code have shown that they can read, explain and generate programs (Chen et al., 2021). Providers such as Google offer such models through an API (Google, 2024), and tools such as Ollama (2024) make it possible to run open models locally. Cloud models give strong explanations but require sending code to a third party, while local models keep the code private. For this reason the AI Code Review & Release-Risk Assistant supports both, and also includes a built-in offline engine for cases where no model is available.

Research on software delivery performance shows that frequent, reliable releases depend on automated checks and safe deployment practices (Forsgren et al., 2018). Ideas such as gradual (canary) rollout and fast rollback reduce the damage of a bad release. The AI Code Review & Release-Risk Assistant follows this thinking by attaching a rollout recommendation to each verdict.

Code-quality metrics are also useful signals of risk. Cyclomatic complexity (McCabe, 1976) counts the independent paths through a program, and Halstead measures (Halstead, 1977) estimate size and effort from operators and operands. Complex code is harder to test and more likely to hide defects, so the AI Code Review & Release-Risk Assistant reports both metrics beside the security findings.

The Python *ast* module (Python Software Foundation, 2024) gives a structured view of source code, which allows the scanner to inspect function calls and arguments directly instead of matching raw text. The platform is served through FastAPI (Ramírez, 2024), a Python framework that provides typed request handling and automatic interactive API documentation.

Finally, a detection system must be measured. Precision, recall and the F1 score are standard measures of how many reported findings are correct and how many real problems are found (Powers, 2011). The AI Code Review & Release-Risk Assistant includes an automated harness that calculates these values, and its behaviour is checked with the pytest framework (pytest-dev, 2024).

System architecture and design

The AI Code Review & Release-Risk Assistant follows a layered client–server architecture, shown in Fig. 1. The interface layer is a browser dashboard used by a developer or a CI/CD pipeline. The backend is a FastAPI service that receives requests, manages projects and runs the analysis pipeline. Each stage hands a structured result to the next, so the stages can be changed or tested one at a time.

The five layers are as follows.

- **Presentation and governance:** an interactive audit dashboard where the developer or pipeline starts a scan and reads the verdict.
- **Ingestion:** the FastAPI gateway and a Git and workspace service that obtain the code and keep it in an isolated temporary folder.
- **Analysis and parsing:** a unified multi-language scanner made up of language engines, a universal secrets detector and a metrics module.
- **AI review and cognitive enrichment:** an AI review engine that turns raw findings into explanations, root causes and before/after code differences.
- **Release-risk assessment:** a risk matrix, blast-radius mapping, a 0–100 score, a deployment-blocker checklist, a rollback strategy and the final verdict.

The design is modular. The ingestion module, the language scanners, the AI review layer and the risk engine work independently and exchange data in a normalised form. A new language scanner or a new AI provider can therefore be added without changing the rest of the system.

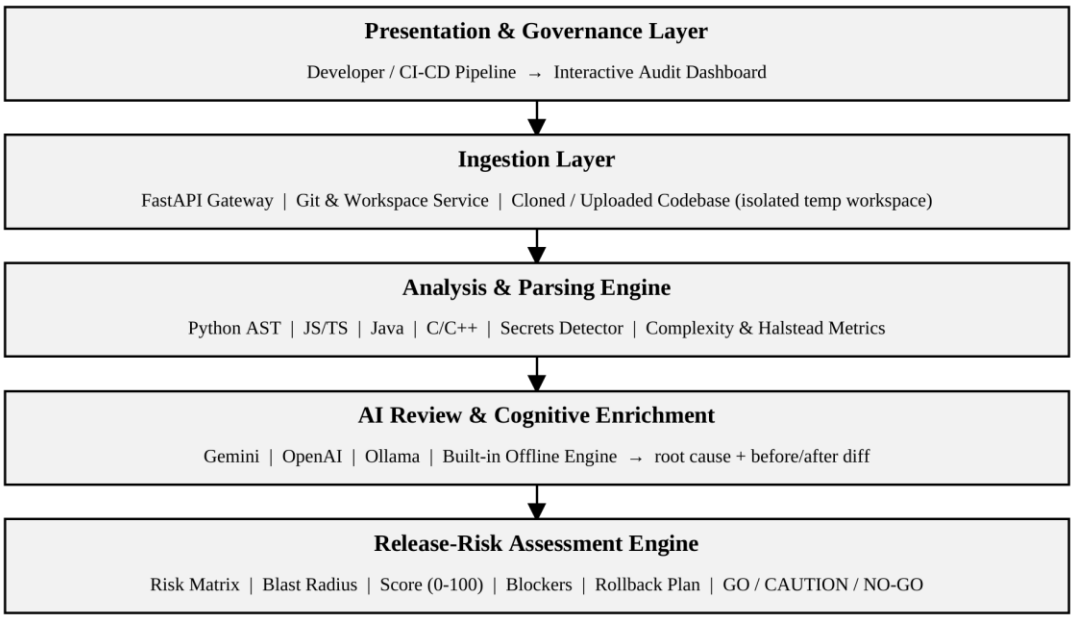


Fig. 1 - Layered architecture and component flow of the AI Code Review & Release-Risk Assistant.

Implementation

1.1. Repository ingestion module

Module 1 securely brings source code into the platform in two ways: a public GitHub repository URL (with automatic branch discovery and an archive fallback) or an uploaded ZIP archive. Fig. 2 shows the flow. The code is never executed. It is only read, measured and analysed.

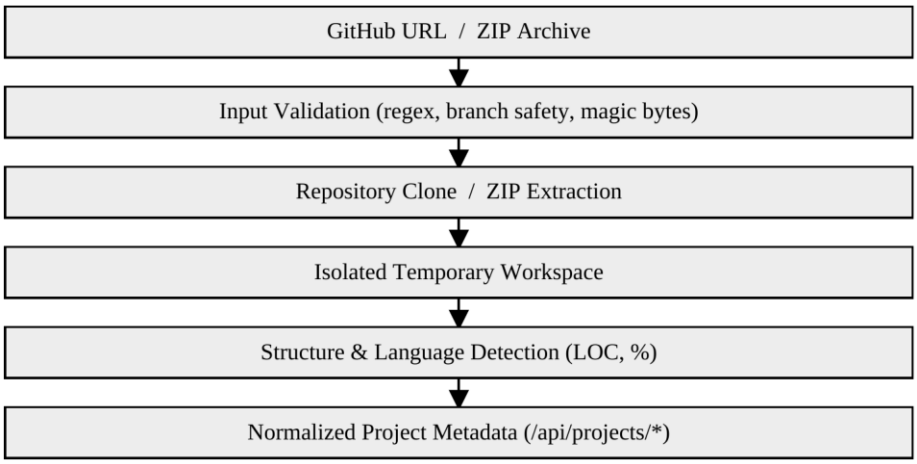


Fig. 2 - Repository ingestion pipeline (Module 1).

The module exposes a small REST interface, summarised in Table 1. A project is first ingested and receives an identifier, which is then used to list its files, view the detected languages, run the analysis or delete the project and its workspace.

Table 1 - REST API of the ingestion module

Method	Endpoint	Purpose
POST	/api/projects/github	Ingest a public repo and branch
POST	/api/projects/upload	Upload a ZIP (max 50 MB)
GET	/api/projects	List ingested projects
GET	/api/projects/{id}	Get project metadata
GET	/api/projects/{id}/files	List files with LOC and language
GET	/api/projects/{id}/languages	Language breakdown
DELETE	/api/projects/{id}	Remove project and workspace
POST	/api/projects/{id}/analyze	Run review and risk analysis

Security defences. Because the platform accepts files from users, the ingestion module treats every input as untrusted. It checks each ZIP entry path against the workspace root to stop path traversal ("zip-slip") and rejects absolute paths and drive letters. It limits the upload to 50 MB (returning HTTP 413 above that), the uncompressed size to 250 MB and the number of files to 10,000 as a defence against ZIP bombs. It verifies file signatures (magic bytes), ignores folders such as *.git*, *node_modules*, *venv* and *dist*, and skips binary files. Branch and URL values are sanitised to prevent injection into the git command line (for example, options such as *--upload-pack*).

1.2. Multi-language analysis engine

The unified scanner dispatches each file to a language-specific engine. Table 2 lists what each engine looks for.

Table 2 - Detection coverage by language engine

Language	Example issues detected
Python	Bare *except*, mutable default arguments, arbitrary code execution (*eval*/*exec*), command injection (*subprocess* with *shell=True*), insecure *pickle* deserialization, raw SQL query formatting
JavaScript / TypeScript	Prototype pollution (*__proto__*), unhandled Promise rejections, DOM XSS via *innerHTML*, loose equality bugs, cookies without *Secure* / *HttpOnly*
Java	SQL injection via *Statement.executeQuery()*, insecure *ObjectInputStream* deserialization, weak *java.util.Random*, fatal *Throwable* catch blocks
C / C++	Unbounded buffer calls (*strcpy*, *strcat*, *sprintf*), format-string flaws, missing *free* after *malloc* (memory leaks)

All languages	High-entropy and pattern-based secrets: AWS keys (*AKIA…*), GitHub tokens (*ghp_…*), private keys, hard-coded JWTs; complexity and Halstead metrics

For Python, the scanner walks the AST, so it can recognise a dangerous call and its arguments even when the code is formatted differently. For the other languages it uses carefully designed heuristic patterns. These are quick to run but cannot understand the full meaning of the program, which is discussed under limitations in Section 7.

1.3. AI-powered code review engine

After the static scan, the AI review engine adds human-readable detail to each finding: a description of the weakness, a root-cause explanation and a before-and-after code suggestion. The engine is multi-provider. It can call Google Gemini (*gemini-1.5-flash* or *gemini-2.0-flash*), OpenAI (*gpt-4o* or *gpt-4o-mini*), or a local Ollama model (such as *deepseek-coder* or *codellama*). When no provider is configured, a built-in Cognitive AI engine runs fully offline and produces structured developer comments, root-cause explanations and replacement diffs, so the platform works without internet access and without sending code to any third party.

1.4. Release-risk scoring and governance engine

The risk engine converts the list of findings into one number. Each finding has a severity (Critical, High, Medium or Low) with a weight of 25, 15, 6 and 2 points. If n_C, n_H, n_M and n_L are the counts of findings at each level, the base release-risk score R is

$$R = \min(100, 25n_C + 15n_H + 6n_M + 2n_L) \quad (1)$$

The score is limited to 100. It is mapped to a verdict as shown in Table 3. The mapping is deliberately simple so that a team can see exactly why a release was approved or stopped.

Table 3 - Release-risk score and decision

Score	Verdict	Action
0 – 30	GO FOR RELEASE	Low risk, standards met; eligible for automatic CI/CD deployment
31 – 64	PROCEED WITH CAUTION	Moderate risk; manual QA sign-off and a 5% canary rollout
65 – 100	NO-GO / BLOCKED	Critical vulnerabilities, credential leaks or high defect density; pipeline halted

Beyond the score, the engine lists *deployment blockers*, which are fatal issues (for example, a leaked credential) that stop a release automatically. It also maps the *architectural blast radius* by showing which parts of the system (data layer, authentication, public APIs or core services) the affected files belong to, and it writes a context-aware *rollback strategy* that tells the team how to recover if the release goes wrong.

Core features and workflow

A typical session proceeds as follows. The developer opens the dashboard at the local address of the service and submits a GitHub URL or ZIP file. The ingestion module validates and stores the code, and the interface shows the detected languages with their share of the code. When the developer starts the analysis, the scanner runs on every supported file, the AI engine enriches the findings, and the risk engine produces the score, the blockers, the blast radius and the rollback plan. The dashboard then shows the verdict together with the findings, each with its explanation and suggested fix.

Interactive API documentation is generated automatically by FastAPI, which also lets other tools, such as a CI/CD pipeline, call the same endpoints without using the browser. This makes it possible to place the AI Code Review & Release-Risk Assistant as a release gate: the pipeline submits the repository, reads the verdict and continues only on GO, or after a manual sign-off on CAUTION.

The main features can be summarised as multi-language automated analysis, universal credential scanning, quantified 0–100 risk scoring, automatic deployment decisions, blast-radius mapping, rollback guidance, multi-provider AI review with an offline mode, and a built-in benchmark harness.

Implementation and preliminary testing

The project has been implemented as a working full-stack application using Python 3.10 or later (tested on Python 3.12 and 3.13). Installation needs only the dependency list and a single command to start the server. Three levels of checks are provided: a verification script for the ingestion module, a comprehensive pytest suite, and a pipeline core check that runs the whole flow.

A built-in ground-truth benchmark evaluates detection accuracy and speed. It contains ten labelled vulnerability scenarios spread over four programming languages. Each scenario states which weakness is present, so the output of the scanner can be compared with the known answer. Precision is the share of reported findings that are correct, recall is the share of real weaknesses that are found, and F1 is their harmonic mean (Powers, 2011):

$$\text{Precision} = TP / (TP + FP), \quad \text{Recall} = TP / (TP + FN) \tag{2}$$

$$F_1 = 2 \times (\text{Precision} \times \text{Recall}) / (\text{Precision} + \text{Recall}) \tag{3}$$

Here *TP* is the number of correct detections, *FP* the number of false alarms and *FN* the number of missed weaknesses. The results are shown in Table 4.

Table 4 - Built-in benchmark results

Metric	Result	Target
Precision	100.0%	> 85.0%
Recall	100.0%	> 85.0%
F1 score	1.00	> 0.85
Avg. detection latency	≈ 0.20 ms per file	
Analysis throughput	> 15,000 LOC/s	> 1,000 LOC/s

All five measures meet or exceed their targets. The accuracy figures should be read with care. The benchmark has only ten scenarios and is not independent of the detector design, so the scores confirm that each detector behaves as intended but do not prove the same accuracy on large, unfamiliar code bases. The speed figures show that analysis is fast enough for use inside a CI/CD pipeline, where the target is a few seconds, not minutes.

Limitations

- The benchmark is small (ten scenarios, four languages). A larger and independently labelled data set, with real open-source projects, is needed to measure false positives properly.
- Only Python is analysed through a true AST in the described design. The other languages use heuristic patterns, which may miss flaws that depend on data flow across files.
- The scoring weights (25, 15, 6, 2) are fixed rules based on severity. They do not yet consider how reachable or exploitable a flaw is in practice.
- Only public GitHub repositories and ZIP archives are supported. Private repositories and other hosting services need authentication support.
- AI explanations from a language model can occasionally be wrong, so suggested fixes should be reviewed by a developer before they are applied.

Conclusion and future work

The AI Code Review & Release-Risk Assistant shows that code review and release decision-making can be combined in one automated pipeline. By uniting multi-language static analysis, credential detection, AI-based explanation and a transparent risk score, it turns a long list of warnings into a clear verdict (GO, CAUTION or NO-GO) with reasons, blockers and a rollback plan. The secure ingestion module and the offline AI mode make it practical for teams that cannot share code with external services, and the built-in benchmark makes its behaviour measurable.

Future work includes: extending the benchmark to a large labelled set drawn from real projects; replacing heuristic scanners with full AST or data-flow analysis for JavaScript, Java and C/C++; learning the scoring weights from historical incident data; adding support for private repositories and pull-request integration so that every merge is scored automatically; using a database in place of temporary storage for scan history and trend dashboards; and adding more languages such as Go and Rust.

Acknowledgements

The development of the AI Code Review & Release-Risk Assistant was made possible by open-source technologies. The authors thank the communities behind Python, FastAPI, pytest, Git, Ollama and the providers of the Gemini and OpenAI APIs.

REFERENCES :

- Chen, M., Tworek, J., Jun, H., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Chess, B., & West, J. (2007). *Secure programming with static analysis*. Boston, MA: Addison-Wesley.
- Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The science of lean software and DevOps*. Portland, OR: IT Revolution Press.
- Google. (2024). *Gemini API documentation*. https://ai.google.dev/
- Halstead, M. H. (1977). *Elements of software science*. New York: Elsevier North-Holland.
- McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, SE-2*(4), 308–320.
- MITRE Corporation. (2024). *Common Weakness Enumeration (CWE)*. https://cwe.mitre.org/
- Ollama. (2024). *Ollama: Run large language models locally*. https://ollama.com/
- OWASP Foundation. (2021). *OWASP Top Ten*. https://owasp.org/www-project-top-ten/
- Powers, D. M. W. (2011). Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation. *Journal of Machine Learning Technologies, 2*(1), 37–63.

- PyCQA. (2024). *Bandit: Security linter for Python code*. https://bandit.readthedocs.io/
- pytest-dev. (2024). *pytest documentation*. https://docs.pytest.org/
- Python Software Foundation. (2024). *ast — Abstract syntax trees*. Python 3 documentation. https://docs.python.org/3/library/ast.html
- Ramírez, S. (2024). *FastAPI*. FastAPI documentation. https://fastapi.tiangolo.com/
- Semgrep, Inc. (2024). *Semgrep documentation*. https://semgrep.dev/docs/