PROCEEDINGS OF SCIENCE

AI-Assisted DevSecOps for Secure Software Delivery in Earth Observation Projects

Federico Fornari,[a,*] **Mohanad Albughdadi,**[a] **Marica Antonacci,**[a,b] **Vasileios Baousis,**[a] **Tolga Kaprol**[a] **and Claudio Pisa**[a]

[a] *European Centre for Medium-Range Weather Forecasts (ECMWF), Shinfield Road, Reading RG2 9AX, United Kingdom*

[b] *INFN Sezione di Bari, Via Orabona 4, 70125 Bari, Italy*

E-mail: federico.fornari@ecmwf.int, mohanad.albughdadi@ecmwf.int, marica.antonacci@ecmwf.int, vasileios.baousis@ecmwf.int, tolga.kaprol@ecmwf.int, claudio.pisa@ecmwf.int

As the Cloud Systems Team at ECMWF, we operate and secure the distributed cloud infrastructures — Kubernetes clusters hosted on OpenStack-based [1] private clouds, including the EWC (European Weather Cloud, co-managed by ECMWF and EUMETSAT) and partner sites — that underpin EU-funded Earth Observation platforms in the scope of this paper. In this environment, container images may accumulate CVEs silently between builds, manual remediation backlogs can grow faster than our team can clear, and static analysis alone cannot generally distinguish exploitable from theoretical vulnerabilities. We present the AI Software Security Agent, a fully automated, Large Language Model (LLM)-powered platform that integrates into existing CI/CD pipelines to perform four complementary security tasks without human intervention: (i) container image hardening through iterative scanning and LLM-driven patching, eliminating CRITICAL and HIGH CVEs; (ii) source code remediation of BLOCKER and CRITICAL issues via targeted LLM-generated fixes validated by the existing test suite; (iii) automated unit test enhancement using coverage-driven LLM prompts that increased test coverage by up to 39 percentage points; and (iv) automated penetration testing to classify each CVE by real-world exploitability. The system is orchestrated at the CI/CD level, and all LLM inference runs on-premise via Ollama on dedicated GPU nodes, ensuring that source code and vulnerability data never leave the organisation's perimeter. A YAML-driven configuration system provides multi-language support for .NET/C#, Python, Java, and Node.js which can be easily extended to other languages. We describe the architecture, the iterative fix-and-verify methodology, and results from deployment on four real-world Earth Observation applications.

International Symposium on Grids and Clouds (ISGC2026)
15–20, March, 2026
Academia Sinica, Taipei, Taiwan

*Speaker

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0) All rights for text and data mining, AI training, and similar technologies for commercial purposes, are reserved. ISSN 1824-8039 . Published by SISSA Medialab.

PoS(ISGC2026)002

1. Introduction

The pace of modern software delivery has outpaced the capacity of traditional security processes. As the Cloud Systems Team at ECMWF, we operate and secure the distributed cloud infrastructures that underpin multiple EU-funded Earth Observation (EO) platforms. These infrastructures run on OpenStack-based private clouds [1] — primarily the EWC and partner sites — with Kubernetes clusters managed through GitLab [2] CI/CD mechanisms and orchestrated by Dagger [3] and FluxCD [4]. In this fast-moving, multi-partner development environment, container images may accumulate CVEs silently between builds, manual remediation backlogs can grow faster than teams can clear, test coverage tends to stagnate, while static analysis alone cannot reliably distinguish exploitable vulnerabilities from merely theoretical ones.

Recent advances in Large Language Models (LLMs) have demonstrated strong performance on code understanding, generation and transformation tasks. At the same time, CI/CD tooling has matured to support reproducible, cacheable pipeline execution via frameworks such as Dagger. These technologies create an opportunity to close the loop between vulnerability detection and remediation entirely within the devops pipeline, with no human intervention required for routine fixes.

In this paper we present the *AI Software Security Agent*, a platform that embeds four LLM-powered security capabilities directly into a GitLab CI/CD pipeline:

1. *Container image hardening* — iterative scanning with Trivy [5] and LLM-driven Dockerfile patching;
2. *Source code remediation* — automatic fixing of SonarQube [6] BLOCKER and CRITICAL issues with test-suite validation;
3. *Test enhancement* — coverage-driven generation of additional unit tests;
4. *Automated penetration testing* — deployment to an ephemeral K3s [7] cluster and AI-guided exploit assessment via PentestGPT [8].

A key design constraint is *data sovereignty*: all LLM inference runs on-premise via Ollama [9], so source code and vulnerability reports never leave the organisation's network. The system supports .NET/C#, Python, Java, and Node.js through a YAML-driven configuration that requires no pipeline code changes to extend support to new languages.

The remainder of this paper is organised as follows. Section 2 reviews related work. Section 3 describes the overall architecture. Sections 4–7 detail the four capabilities. Section 8 presents the multi-language design. Section 9 reports initial results, and Section 10 concludes.

2. Related work

The use of LLMs for automated program repair has received considerable attention recently. Studies have shown that LLMs can generate correct patches for a significant fraction of real-world bugs [10], though most work targets single-file patches in isolation without integrating into a CI/CD verification loop.

PoS(ISGC2026)002

In container security, Trivy and similar scanners detect vulnerabilities but leave remediation to human operators. SonarQube provides static analysis for source code quality and security, yet fixing the reported issues remains a manual task. For penetration testing, PentestGPT combines LLM reasoning with real security tools. Our contribution integrates all three — vulnerability scanning, static analysis and penetration testing — into a single automated pipeline orchestrated by Dagger, a code-first CI/CD framework. To the best of our knowledge, no prior work combines Dagger-based orchestration with LLM-driven security remediation, test enhancement, and automated penetration testing in a single system.

3. Architecture

The AI Software Security Agent is built around three principles: *automation first* (every security action that can be automated is automated), *no external LLM/API dependencies* (all LLM inference runs on-premise), and *iterative improvement* (fixes are applied, verified and retried rather than accepted blindly).

3.1 Pipeline overview

The system operates as a GitLab CI/CD pipeline (Figure 1) composed of nine stages: (1) *prepare* — rebuild the PentestGPT image if changed; (2) *detect* — identify the programming language; (3) *analyze* — run SonarQube; (4) *build* — build and store the container image in MinIO [11]; (5) *scan* — scan for CVEs and iteratively harden using the LLM; (6) *enhance* — apply source code fixes and optionally generate unit tests; (7) *test* — deploy to ephemeral K3s and run PentestGPT; (8) *deploy* — push the hardened image to GitLab container registry; and (9) *cleanup* — remove temporary container images from the MinIO S3 staging registry. Every stage that produces output is exported as a downloadable GitLab artifact.

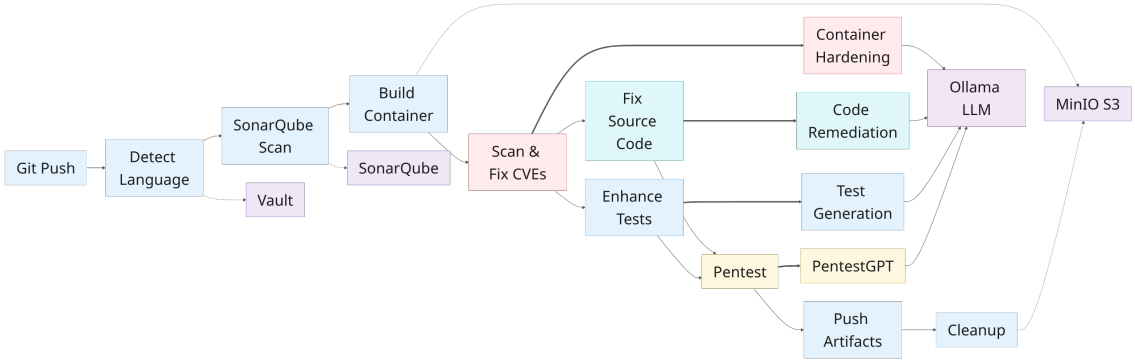

Figure 1: High-level architecture of the AI Software Security Agent. Pipeline stages (left to right) connect to colour-matched AI capabilities, all backed by on-premise Ollama LLM inference and supporting services.

3.2 Orchestration, LLM integration and secrets

All pipeline logic is implemented as Python functions using the Dagger SDK. Each function runs in a containerised environment, receives typed parameters, and returns files or directories. This provides reproducibility (identical execution on laptops and in CI), caching (content-addressable

PoS(ISGC2026)002

skipping of unchanged stages), and composability. The project uses two external Dagger modules: a K3s module for ephemeral clusters and a Trivy module for scanning.

The system communicates with LLMs through an OpenAI-compatible API served by Ollama: a reasoning model (`gpt-oss:20b`) for analysis tasks and a coding-specialist model (`qwen3-coder:30b`) for patch generation, both at temperature 0.2. All credentials are stored in HashiCorp Vault [12] and retrieved at runtime via AppRole authentication; no credentials are stored in CI variables or code repositories.

4. Container image hardening

As new vulnerabilities are discovered continuously, a container image that is clean at deployment may accumulate newly disclosed CVEs over time. Traditional workflows discover these vulnerabilities only when a scanner runs in a staging or production environment — by which point it may already be too late.

4.1 Methodology

The agent runs Trivy against every built image, targeting CRITICAL and HIGH severity CVEs. When vulnerabilities are found, it enters an iterative fix loop (Figure 2): the full vulnerability report (JSON), the current Dockerfile, and a *blacklist* of previously failed fixes are sent to the LLM, which generates a patched Dockerfile employing strategies such as base image upgrade, explicit package pinning, non-root execution (`USER` directives), and removal of unnecessary tools. The patched Dockerfile is rebuilt using Kaniko [13] inside Dagger and re-scanned. If the count decreased, the new Dockerfile becomes the current best; otherwise, the failed fix is blacklisted. The loop repeats for up to a maximum number of iterations (5 by default). The blacklist mechanism prevents the LLM from repeating unsuccessful strategies.

Each pipeline run yields a `Dockerfile.hardened`, a human-readable `vulnerabilities.html`, and a machine-readable `vulnerabilities.json`. An environment file (`hardening.env`) exports `USE_HARDENED` and `HARDENED_TAG` so that subsequent stages use the hardened image.

5. Source code remediation

Static analysis tools such as SonarQube are excellent at finding problems but do not fix them. The result is a growing backlog of BLOCKER and CRITICAL issues that rarely reach the top of a development sprint.

5.1 Methodology

The agent queries the SonarQube API to retrieve all open issues of severity BLOCKER or CRITICAL. For each affected file, it constructs a prompt containing the source code and issue descriptions with line numbers, then asks the LLM to generate a corrected version. The fix is not accepted until the existing test suite passes, preventing regressions. If issues remain or tests fail, error output is fed back to the LLM for the next attempt, up to `MAX_ITERATIONS` (default: 3) per file, processing at most `MAX_FILES` (default: 10) per run. The agent handles security vulnerabilities (SQL injection, XSS, hardcoded credentials), bugs (null dereference, resource leaks),

PoS(ISGC2026)002

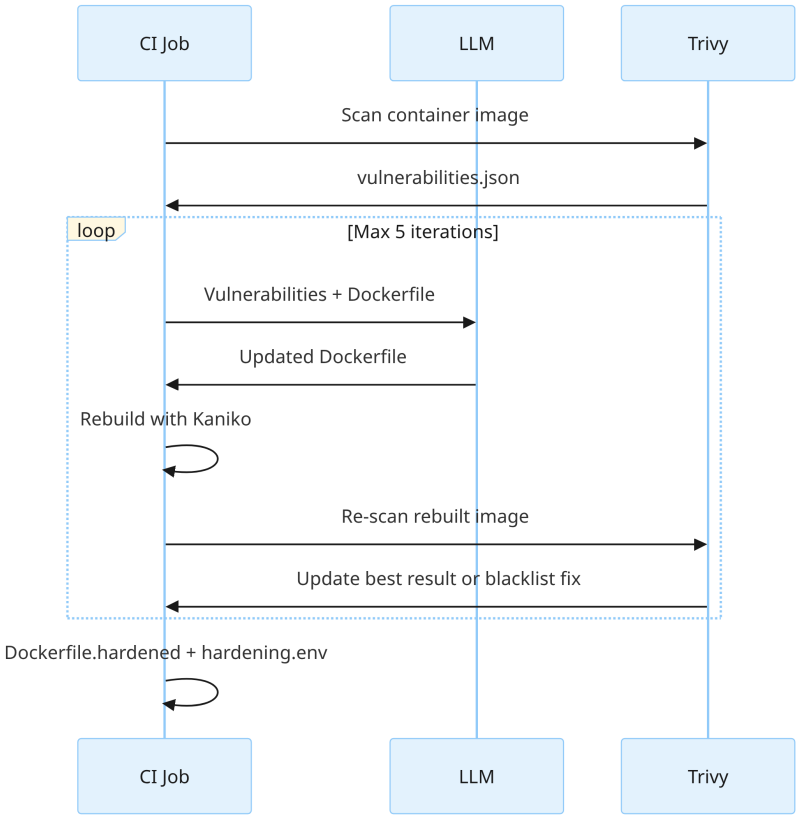

Figure 2: Iterative container hardening: scan, fix, rebuild, re-scan. Failed fixes are blacklisted to prevent repetition.

and critical code smells. Figure 3 illustrates the iterative process. Fixed files are exported to `generated_src_files/` as downloadable pipeline artifacts.

6. Test enhancement

As developers focus mostly on shipping new features, test coverage for existing code may lag behind, especially for edge cases and error paths.

6.1 Methodology

Once source code is fixed, the agent analyses the existing test suite and asks the LLM to generate additional test cases targeting coverage gaps, edge cases (null values, empty collections, boundary conditions), and error conditions (invalid inputs, exception paths). The process is iterative: (1) measure baseline coverage; (2) for each source file, send the source, tests and coverage data to the LLM; (3) each generated test is individually *build-validated* — passing tests are merged, failing tests are excluded with their errors stored as feedback; (4) coverage is recomputed and the process repeats. This stage is *opt-in*: it only runs when the commit message contains "enhance unit tests". Figure 4 shows the flow. The `LanguageAgnosticEnhancer` class generates framework-specific prompts for each supported language (Table 1).

PoS(ISGC2026)002

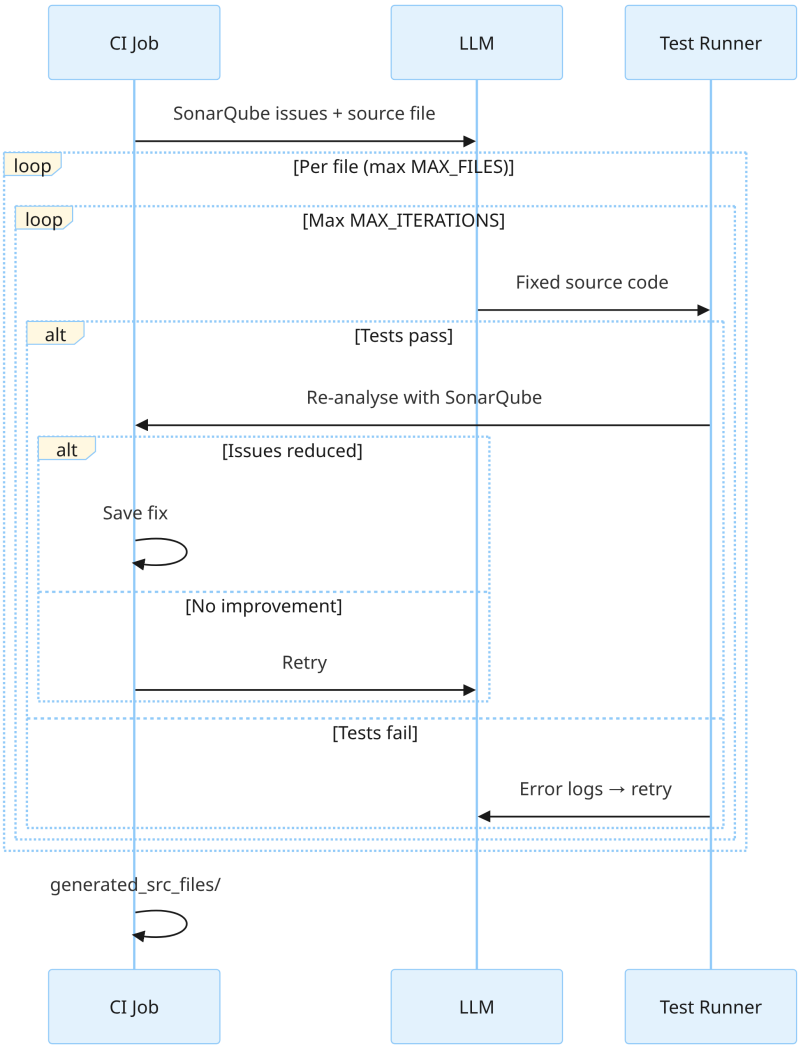

Figure 3: Source code remediation flow: SonarQube issues are sent to the LLM, fixes are validated by the test suite before acceptance.

7. Automated penetration testing

Knowing that a CVE exists in a container image is not the same as knowing whether it is exploitable in the specific deployment environment. A vulnerability rated CRITICAL in the National Vulnerability Database (NVD) may be effectively harmless behind a reverse proxy — or it may be trivially exploitable. Static analysis cannot tell the difference.

7.1 Ephemeral K3s cluster

The agent deploys the application to an ephemeral K3s cluster managed by a Dagger module (Figure 5), created fresh for each pipeline run. It includes a K3s server, CoreDNS with external resolvers, an in-cluster registry, the application pod deployed via Helm, and a PentestGPT pod. Unique per-run cache volumes prevent K3s state corruption while preserving fast cold starts.

PoS(ISGC2026)002

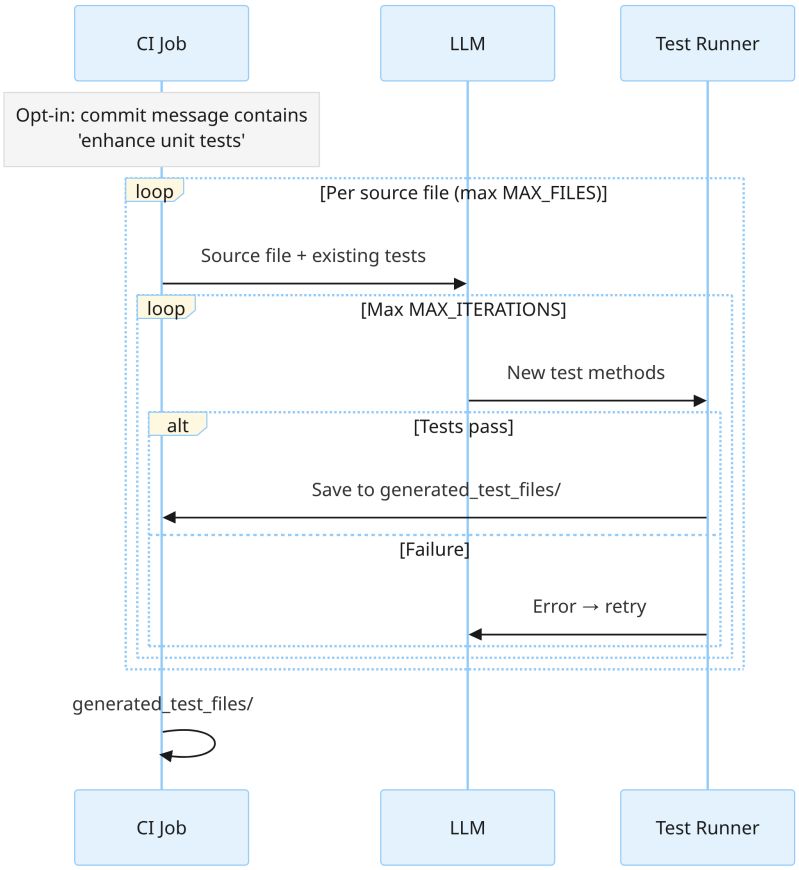

Figure 4: Test enhancement flow: existing tests are analysed, the LLM generates new test methods, and only validated tests are retained.

Table 1: Supported test frameworks per language.

Language	Frameworks	Coverage format
.NET/C#	xUnit, NUnit, MSTest	OpenCover
Python	pytest, pytest-mock	Cobertura
Java	JUnit 5, Mockito	JaCoCo
Node.js	Jest, Mocha	Cobertura / LCOV

7.2 PentestGPT workflow

PentestGPT combines LLM reasoning with real penetration testing tools. Within the ephemeral cluster it targets the deployed application through a structured workflow: (1) *discovery* of the application's API surface; (2) *authentication* against the application using test credentials provisioned at deployment; (3) *exploit research* per CVE against NVD advisories; (4) *active testing* with crafted requests against the application endpoints; and (5) *assessment*, classifying each CVE as Class A (exploitable), Class B (present but not exploitable), or Class C (not applicable). Because PentestGPT is built on Claude Code, it expects an API-compatible LLM endpoint. Claude Code Router [14]

PoS(ISGC2026)002

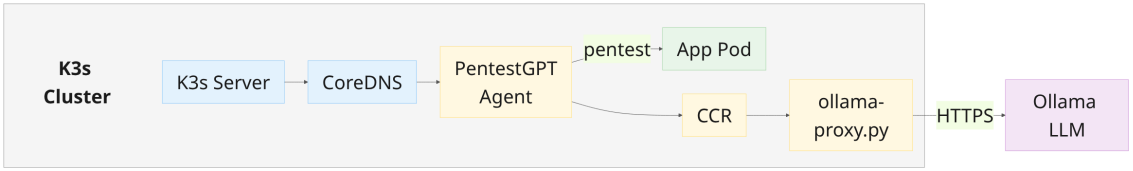

Figure 5: Ephemeral K3s cluster architecture for penetration testing. The PentestGPT pod accesses the application pod and communicates with the Ollama LLM via a proxy chain.

(CCR), an open-source API routing proxy, intercepts Claude API calls inside the PentestGPT pod and redirects them to the local Ollama instance. However, Ollama is served over HTTPS while CCR can only emit plain HTTP requests to local endpoints; a lightweight `ollama-proxy.py` sidecar bridges HTTP to HTTPS with DNS-hardened resolution, keeping all inference on-premise (Figure 5). The bootstrap script includes retry logic (up to three attempts) and fallback report generation. After the test, the cluster is torn down and a `pentest-report.md` artifact is exported.

8. Multi-language support

The agent is designed to be language-agnostic. All language-specific behaviour — which base image to use, how to run tests, which SonarQube scanner to invoke — is defined in a single `config/languages.yaml` file. Adding support for a new language primarily requires a YAML block for build, scan and deploy toolchains; however, the test enhancement stage may require language-specific prompt adjustments for new unit-test frameworks. The pipeline supports two modes (Figure 6): (1) *explicit configuration*, where the `TARGET_LANGUAGE` CI variable overrides auto-detection — useful for multi-language repositories; and (2) *auto-detection*, where the agent inspects the `apps/` directory for language markers (`.csproj/.sln` for .NET, `pom.xml/build.gradle` for Java, `package.json` for Node.js, `requirements.txt/*.py` for Python) and selects the appropriate toolchain.

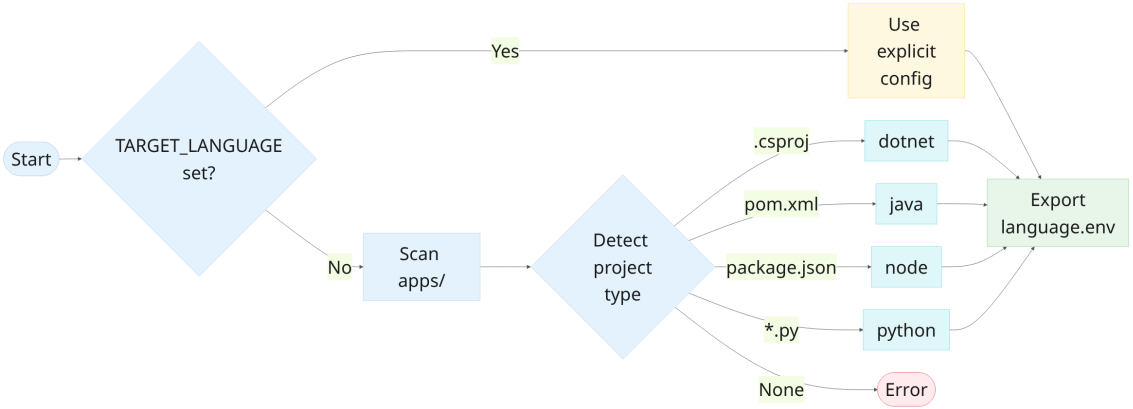

Figure 6: Language detection flow: explicit configuration takes priority over automatic detection from repository file markers.

All language-specific settings — SDK image, build command, test runner, coverage format, SonarQube scanner — are specified per language in YAML. Currently supported languages are

PoS(ISGC2026)002

.NET/C#, Python, Java, and Node.js. Adding Go, Rust, or any other language primarily requires appending a YAML block, although test enhancement prompts may need language-specific tuning for new frameworks.

9. Results and discussion

We evaluated the AI Software Security Agent on four real-world applications in .NET/C#, Java, Node.js, and Python, all used within EU-funded Earth Observation projects. Table 2 summarises the complete results for every language across all four capabilities.

Table 2: Complete results across all four languages and capabilities.

Language	Container hardening	Source code remediation	Test enhancement	Penetration testing
.NET	2→1 CVE[†]	9/13 files	43.4→48.6% (+5.2 pp)	Class B[‡]
Java	11→0 CVEs	no issues found	38.0→77.3% (+39.3 pp)	skipped[*]
Node.js	4→0 CVEs	6/7 files (+1 partial)	24.5→29.6% (+5.1 pp)	skipped[*]
Python	9→0 CVEs	4/4 files (100%)	26.7→30.6% (+3.9 pp)	skipped[*]

[†]Unresolved: EOL .NET 6 runtime; no clean base image available. [‡]CVE-2025-24070 assessed as not externally testable. [*]No remaining CRITICAL/HIGH CVEs after hardening; pentest skipped.

9.1 Container image hardening

Three of four applications were fully hardened. Python achieved the best result: all 9 CRITICAL/HIGH CVEs were eliminated in a single iteration by switching from the `python:3.12` base image to `python:3.11-alpine3.20`. Java had the highest initial count (11 CVEs: 5 OS-level including `gnutls`, `libpng`, and `zlib`, plus 6 `log4j` 1.2.17); after two failed builds and automatic blacklisting, the LLM succeeded in iterations 3 and 4, reducing 11 CVEs to zero. Node.js required four iterations to reduce 4 CVEs to zero; the LLM proposed Alpine variant tags, one of which was non-existent and was automatically blacklisted before a successful build was produced. The .NET 6 application started with 2 CRITICAL/HIGH CVEs and was reduced to 1, but the remaining CVE could not be eliminated because all available base images for the end-of-life .NET 6 runtime carry unfixable vulnerabilities.

9.2 Source code remediation

Python achieved 100% success: all 6 CRITICAL SonarQube issues across 4 files were resolved, including duplicated string literals, excessive cognitive complexity (55 → 15), and disabled SSL/TLS certificate validation. Each fix passed the test suite and a SonarQube rescan confirming zero remaining issues. Node.js fixed 25 issues across 7 files (6 completely, 1 partially), including deeply nested function refactoring and Redux state mutations. .NET fixed 9 of 13 affected files (25 total issues); the remaining 4 files failed due to compilation errors or SonarQube scan timeouts. Java had no BLOCKER or CRITICAL issues.

PoS(ISGC2026)002

9.3 Test enhancement

Java showed the largest coverage gain: 38.0% → 77.3% (+39.3 pp) over 2 iterations, with 2 of 3 source files successfully enhanced. .NET improved from 43.4% to 48.6% (+5.2 pp) over 4 iterations, enhancing 37 of 56 files with existing tests. Node.js gained 5.1 pp (24.5% → 29.6%) on a React application with 60 source files. Python improved from 26.7% to 30.6% (+3.9 pp). Across all languages, the implemented mechanism ensured that output always came from the iteration with the highest measured coverage.

9.4 Penetration testing

PentestGPT ran on the .NET application, the only one with remaining CVEs after the hardening stage. It assessed CVE-2025-24070 (ASP.NET privilege escalation via authentication through `SignInManager.RefreshSignInAsync`) and classified it as *Class B* — present but not externally testable, because the vulnerable method is internal to the authentication flow and no public endpoint accepts arbitrary credentials. The agent completed in under 4 minutes using 11 LLM calls routed through the proxy chain. For Node.js, Python, and Java, the hardening stage eliminated all CRITICAL/HIGH CVEs, so the pentest correctly determined that no vulnerabilities remained to test. This illustrates two complementary outcomes: automated triage confidently deprioritised a HIGH-severity CVE without manual analysis, while successful hardening rendered pentesting unnecessary for the other three applications.

9.5 Strengths and limitations

The iterative fix-and-verify methodology proved essential: without blacklisting, the LLM repeats failing strategies. Test-suite gating prevented regressions in all observed runs. Results vary by language: Python achieved 100% fix success while .NET resolved 9 of 13 files, and Java showed a +39.3 pp coverage gain thanks to a small, well-structured codebase. Container hardening is limited when no viable base image exists for end-of-life runtimes, as observed with .NET 6. Running Ollama on-premise on dedicated GPU nodes (NVIDIA A100) eliminates per-token API costs and ensures that sensitive data never leaves the organisation's network.

10. Conclusions and future work

We have presented the AI Software Security Agent, an LLM-powered platform that automates four security-critical tasks within a GitLab CI/CD pipeline: container image hardening, source code remediation, test enhancement and penetration testing. Results on four real-world Earth Observation applications show that container hardening eliminated all CRITICAL/HIGH CVEs in three of four languages — Java (11→0), Python (9→0), and Node.js (4→0) — source code remediation resolved 100% of SonarQube issues in Python and 86% in Node.js, test coverage improved by up to 39.3 percentage points (Java), and PentestGPT correctly triaged a HIGH-severity CVE as non-exploitable in under 4 minutes. Container hardening remains challenging for end-of-life runtimes. These results demonstrate both the potential and the current limitations of LLM-driven security automation.

Several directions for future work are planned:

PoS(ISGC2026)002

- *Extend language coverage* — add Go, Rust, and other languages via the YAML configuration, with only minor prompt adjustments expected for new test frameworks;
- *Policy-driven hardening* — auto-enforce CIS Benchmarks and OWASP Top 10 as mandatory quality gates in the pipeline;
- *Runtime threat detection* — extend from static analysis to live log monitoring and anomaly detection;
- *Feedback loop to developers* — automatically open GitLab issues with LLM-generated explanations for each finding, closing the loop between automated remediation and developer awareness.

Acknowledgements

This work is carried out within the Cloud Systems Team at the ECMWF Computing Department. The authors acknowledge the European Union's Horizon Europe programme through the EO4EU project (Grant Agreement No. 101059548) and the CLIMRES project (Grant Agreement No. 101147777) for providing the applications' code used in the evaluation, and the ISGC 2026 organisers for the opportunity to present this work.

References

[1] OpenInfra Foundation, *OpenStack — Open Source Cloud Computing Infrastructure*, https://www.openstack.org/, accessed March 2026.

[2] GitLab, *GitLab Official Website*, Available: https://about.gitlab.com/, accessed March 2026.

[3] Dagger, *Dagger — CI/CD as Code That Runs Anywhere*, https://dagger.io/, accessed March 2026.

[4] CNCF, *Flux — GitOps for Kubernetes*, https://fluxcd.io/, accessed March 2026.

[5] Aqua Security, *Trivy — Comprehensive Security Scanner*, https://trivy.dev/, accessed March 2026.

[6] SonarSource, *SonarQube — Code Quality and Code Security*, https://www.sonarsource.com/products/sonarqube/, accessed March 2026.

[7] Rancher, *K3s — Lightweight Kubernetes*, https://k3s.io/, accessed March 2026.

[8] G. Deng et al., *PentestGPT: An LLM-empowered Automatic Penetration Testing Tool*, arXiv:2308.06782, 2024. https://arxiv.org/abs/2308.06782

[9] Ollama, *Ollama — Run Large Language Models Locally*, https://ollama.com/, accessed March 2026.

PoS(ISGC2026)002

[10] N. Jiang et al., *Impact of Code Language Models on Automated Program Repair*, in Proc. ICSE 2023, IEEE, 2023. https://doi.org/10.1109/ICSE48619.2023.00125

[11] MinIO, *MinIO — High Performance Object Storage*, https://min.io/, accessed March 2026.

[12] HashiCorp, *Vault — Manage Secrets and Protect Sensitive Data*, https://www.vaultproject.io/, accessed March 2026.

[13] Google, *kaniko — Build Container Images in Kubernetes*, https://github.com/GoogleContainerTools/kaniko, accessed March 2026.

[14] musistudio, *Claude Code Router — Route Claude Code Requests to Different Models*, https://github.com/musistudio/claude-code-router, accessed March 2026.

PoS(ISGC2026)002