AI APPSEC ACADEMY

FREE AI & APPSEC KNOWLEDGE HUB · AIAPPSECACADEMY.COM

FREE PRACTITIONER GUIDE · AI FOR APPSEC

Agentic AI Loop Engineering

Designing governed AI loops that find and fix security weaknesses in software, from executive view to step-by-step implementation



Executive View

AT A GLANCE

WHAT IT IS

Loop engineering designs the system that prompts, checks, remembers and stops AI agents. Instead of engineers prompting an assistant turn by turn, a governed loop discovers security work, directs an agent to fix it, verifies every change with deterministic evidence and escalates what it cannot prove.

WHAT IT DELIVERS

Six ready-to-adopt loops: third-party dependency patching, source code scan-and-patch for secure vibe coding (prompt to production), IaC and container hardening, secrets rotation, cryptographic inventory and post-quantum migration, and finding triage. Each ships as a reviewed pull request with evidence.

WHY NOW

AI coding agents now run unattended with scheduling, isolated worktrees, skills, tool connectors and sub-agents. Code is being produced faster than it can be reviewed, while vulnerability backlogs and supply-chain attacks keep growing. Remediation capacity, not detection, is the constraint.

HOW RISK IS CONTROLLED

Sandboxed runs, least-privilege tools, a diff guard against reward hacking, maker-checker review, finite budgets and humans as final authority. Autonomy is earned per change class (L0 to L4) and mapped to the OWASP Top 10 for Agentic Applications.

≥ 60%
target reduction in median time to remediate critical/high findings

≥ 80%
target acceptance rate for loop-generated pull requests at L1

0
tolerated unverified "done" claims; every merge carries evidence

90 days
from charter to three loops in production at L1

Recommended first step: run a 90-day pilot starting with the dependency patching loop at autonomy L1 (Section 10). Targets are pilot objectives, to be calibrated against a measured baseline.

Dr. Viswanath S Chirravuri
D.Eng · GSE #335 · SANS SEC545 and SEC546 Instructor
linkedin.com/in/vchirrav · aiappsecacademy.com

Version 1.0 · November 2026
AIAPPSECACADEMY.COM

Document control

Title	Agentic AI Loop Engineering: Autonomous Find-and-Fix Security Loops, from Executive View to Implementation
Author	Dr. Viswanath S Chirravuri · linkedin.com/in/vchirrav
Published by	AI AppSec Academy · aiappsecacademy.com
Version / date	1.0 · November 2026
Latest version	Download the current edition from the Resources section of aiappsecacademy.com
Distribution	Public. Free resource from AI AppSec Academy.
Intended audience	Security leaders, AppSec and DevSecOps engineers, platform teams, software engineers adopting AI coding agents, and beginners who need a complete path from concept to a running loop.

Version	Date	Change summary
1.0	November 2026	Initial release: concepts, reference model, six security use cases, step-by-step implementation guide, controls, metrics and roadmap.

Contents

1 Introduction 4

2 Foundations of Loop Engineering 5

2.2 Anatomy of a loop 5

2.4 Loop patterns 7

2.5 The three hard problems 8

3 The Secure Loop Reference Model 9

3.2 Autonomy levels 10

3.3 The verifier stack 11

3.4 Termination and escalation 11

4 Use Case Portfolio 13

5 Use Case Playbooks 15

5.1 Third-party dependency vulnerability find-and-patch 15

5.2 Source code security scan and patch (secure vibe coding) 17

5.3 Infrastructure-as-code and container hardening 19

5.4 Secrets leak detection and rotation 20

5.5 Cryptographic inventory and post-quantum migration 21

5.6 Security finding and alert triage 22

6 Implementation Guide: Building Your First Loop 24

Phase A: Design (Steps 1 to 3) 24

Phase B: Build (Steps 4 to 9) 26

Phase C: Operate (Steps 10 to 15) 31

7 Securing the Loop Itself 33

8 Measuring Success 34

9 Governance and Operating Model 36

10 Adoption Roadmap: First 90 Days 37

Appendices: checklist, tool matrix, glossary, references 38

About the Author 41

SECTION 1

Introduction

Security teams are not short of findings. They are short of the engineering hours needed to turn findings into safe, tested, merged fixes. Loop engineering changes the unit of work from "a person prompts an AI assistant" to "a governed system that discovers work, directs agents, verifies every result and stops when it should".

1.1 Purpose

This guide gives security and engineering teams a single, practical reference for designing, building, securing and operating **agentic AI loops** that find and remediate security weaknesses in software. It explains the concepts in plain terms, defines a reference model with explicit guardrails, walks through six security use cases in detail, and then provides a step-by-step implementation path that a beginner can follow to stand up a working loop for any use case of their choice. The guiding principle throughout: **an agent may draft any change; only verified evidence may merge it**, and accountability for every merged change remains with a named human owner.

1.2 How to use this document

If you are...	Read	Outcome
An executive or programme sponsor	Executive View (page 1), Sections 4, 8 and 10	Investment case, risk posture, KPIs and a 90-day plan
A security architect	Sections 2, 3, 7 and 9	Reference model, autonomy policy, threat model and controls
An engineer building a loop	Section 2, one playbook in Section 5, then all of Section 6	A running, verified loop in a sandbox with evidence and telemetry
A beginner new to agents	Glossary (Appendix C) first, then Sections 2, 3.2 and 6 in order	Enough vocabulary and procedure to build a first L1 loop safely

Table 1. Reading paths by role.

1.3 Scope and assumptions

The guide is **vendor-neutral**. Every step names open-source defaults and treats commercial coding agents and agent frameworks as interchangeable components behind a common contract: the agent receives a prompt, edits files in an isolated working tree, and the loop, not the agent, decides whether the work is accepted. The examples use Git, a CI service with scheduled and event-driven jobs, and standard scanners that emit JSON or SARIF. They translate directly to GitLab CI, Jenkins, Azure DevOps or an internal pipeline.

In scope: application and supply-chain security loops that change code, configuration, dependencies or tickets. Out of scope: autonomous production operations such as live incident response actions, network blocking or account suspension, which require a separate safety case.

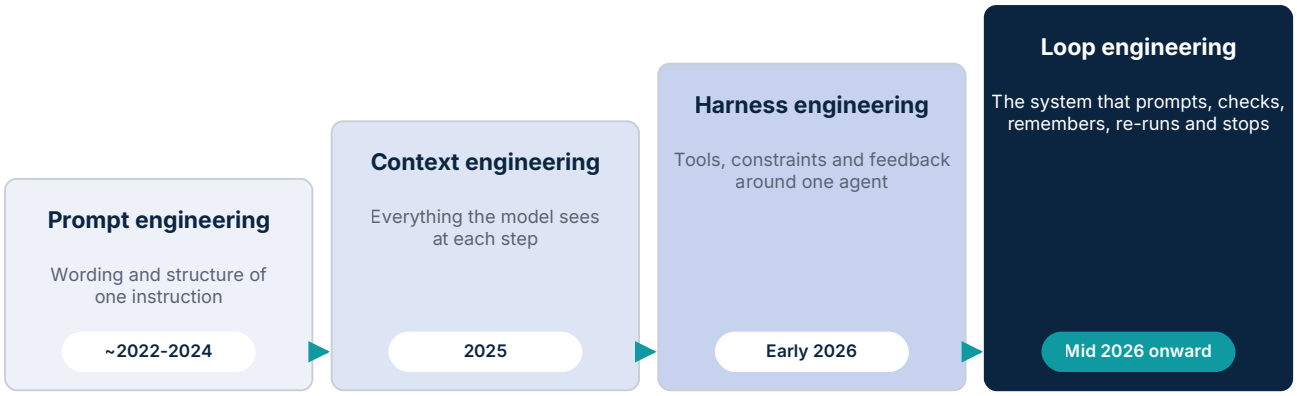
SECTION 2

Foundations of Loop Engineering

Loop engineering is the practice of designing the system that prompts, checks, remembers, re-runs and stops an AI agent, so that the agent converges on a verifiable goal with minimal human prompting and with humans placed where their judgement matters most.

2.1 From prompts to loops

The term entered mainstream engineering vocabulary in June 2026, when practitioners argued that developers should stop prompting coding agents turn by turn and instead design the loops that prompt them. It names the latest of four layers that have accumulated since 2022. Each layer still matters; the loop simply wraps the others and adds the missing pieces: *when* work starts, *how* it is judged, *what* is remembered between runs and *when* to stop.



Each layer wraps the one before it; the loop owns when work starts, how it is judged and when it stops.

Figure 1. The four layers of agent engineering. Dates indicate when each layer became a common practice, not when the ideas originated.

The research lineage is older than the name. Reason-act-observe cycles (ReAct, 2022), self-reflection with episodic memory (Reflexion, 2023) and the evaluator-optimizer and orchestrator-worker patterns described in industry agent-design guidance (2024) are the building blocks. What changed in 2026 is that mainstream coding agents gained the primitives needed to run these patterns unattended: scheduled automations, isolated Git worktrees, reusable skill files, tool connectors over the Model Context Protocol (MCP), sub-agents and lifecycle hooks.

2.2 Anatomy of a loop

Every loop in this guide, whatever its use case, is built from the same six stages, a persistent state store, a set of guardrails that bound what the agent can touch, and a set of explicit exits.

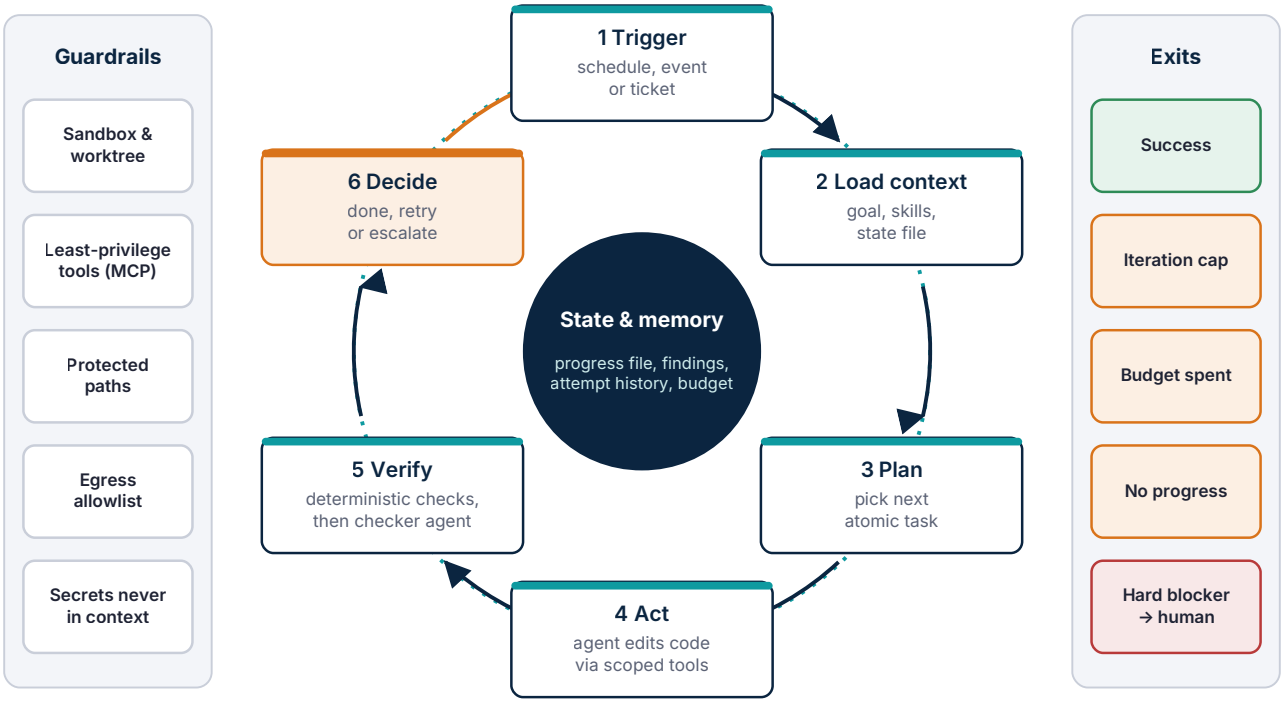


Figure 2. Anatomy of an agentic security loop: six stages around persistent state, bounded by guardrails (left) and terminated only through defined exits (right).

Component	What it does	Typical implementation
Trigger	Starts a run on a schedule, an event or a ticket. Makes the loop self-initiating.	CI schedule, webhook on new advisory, PR event, issue label
Goal and done-definition	States the objective in testable terms so the loop can tell success from activity.	`charter.yaml` with explicit pass criteria
Context and skills	Supplies project knowledge the agent would otherwise guess: conventions, build steps, known pitfalls.	`AGENTS.md`, `SKILL.md` files, retrieved code context
Agent and tools	Reasons, edits files and calls tools inside a sandbox with least privilege.	Headless coding agent or agent framework; MCP servers; CLI tools
Verifier	Judges each attempt with evidence. Deterministic checks first; model-based review second.	Build, tests, scanner re-run, diff guard, checker agent
State and memory	Persists progress outside the model so each run resumes rather than restarts.	`progress.md`, `attempts.jsonl`, issue tracker
Termination and escalation	Ends the loop on success, cap, budget, no-progress or a hard blocker, and routes to a human.	Controller logic plus an escalation queue
Telemetry	Records every decision, cost and outcome for audit and improvement.	OpenTelemetry traces, cost metrics, KPI dashboard

Table 2. Loop components and their implementations.

2.3 Loops are stacked

Production systems run several loops nested inside each other. The innermost loop is the agent's own reason-act cycle. Around it, a verification loop grades each attempt. Around that, an event loop decides when runs happen. The outermost improvement loop studies traces from past runs and improves the harness itself: skills, prompts, verifiers and budgets. Beginners should build loops 1 to 3 first and add loop 4 only after several weeks of stable operation.

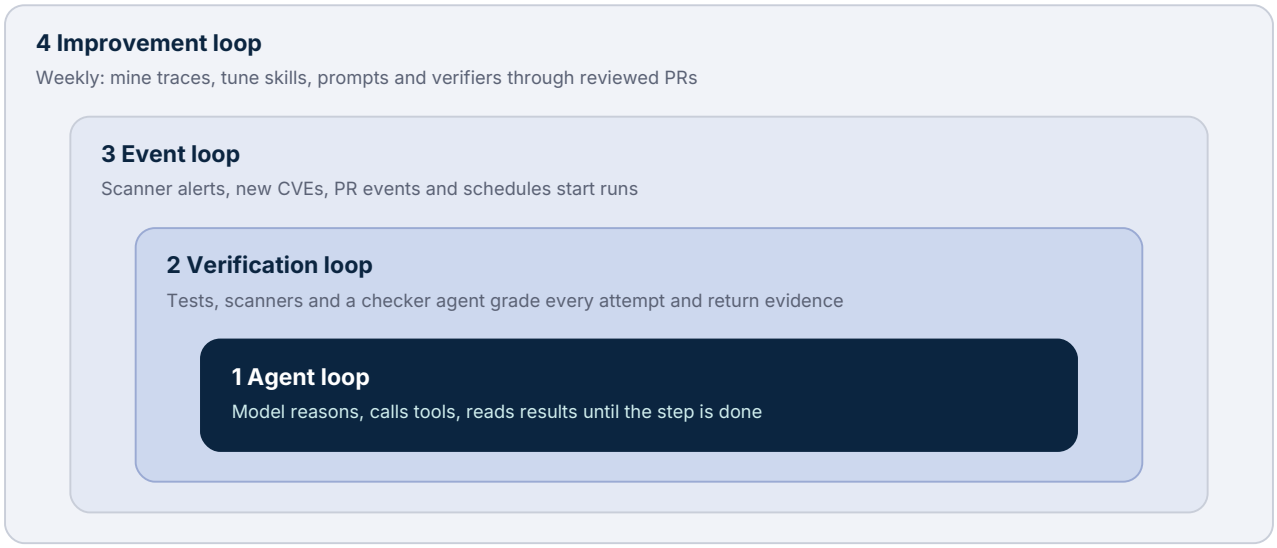


Figure 3. Stacked loops. Each outer loop monitors, corrects or improves the loops inside it.

2.4 Loop patterns

Pattern	Best for	Main risk	Mitigation
Retry	Short, atomic tasks with a fast verifier (one dependency bump)	Repeating the same broken approach	Failure-signature detection; feed evidence, not just "try again"
Plan-execute-verify	Multi-step changes (refactor to remove an injection sink)	Committing to a bad plan	Verify the plan itself; re-plan after two failed steps
Maker-checker	Any change that will be merged	Checker shares the maker's blind spots	Fresh context, different prompt, ideally a different model
Explore-narrow	Unfamiliar code, root-cause analysis, triage	High context and token cost	Sub-agents for exploration; summarise before narrowing
Human-gated	High-impact or irreversible actions (crypto, history rewrite)	Gates so frequent that nothing is saved	Gate only at decision points; batch approvals with evidence

Table 3. Common loop patterns. Security loops in this guide combine plan-execute-verify with maker-checker by default.

2.5 The three hard problems

Context. Long loops overflow the model's context window or degrade as stale output accumulates ("context rot"). Treat context as a budget: start each iteration from a compact state file, summarise previous attempts, prune tool output and isolate exploration in sub-agents. A fresh context per iteration is often better than one long conversation.

Termination. A loop that cannot stop is a cost and safety incident. Use layered exits: a verifier that confirms the goal, an iteration cap, a token, cost and wall-clock budget, no-progress detection and an explicit path to a human.

Verification. Prefer checks that cannot be argued with: compilers, test suites, type checkers, scanners and policy engines. Model-based judges are flexible but gameable. The classic failure is *objective misspecification*: the agent deletes or skips a failing test, adds a suppression comment or weakens a scanner rule so that the pipeline turns green. Section 3.3 shows how to design this out.

FAILURE MODES TO EXPECT

Context overflow; no-progress loops that repeat one failing action; reward hacking (green checks without a real fix); hallucinated success (claiming completion without running the verifier); and cost blow-up. Every control in Sections 3, 6 and 7 maps to at least one of these.

SECTION 3

The Secure Loop Reference Model

The reference model separates what decides (the control plane) from what acts (the execution plane), routes all model traffic through an enterprise gateway, and makes humans the final authority on merge and release.

3.1 Reference architecture

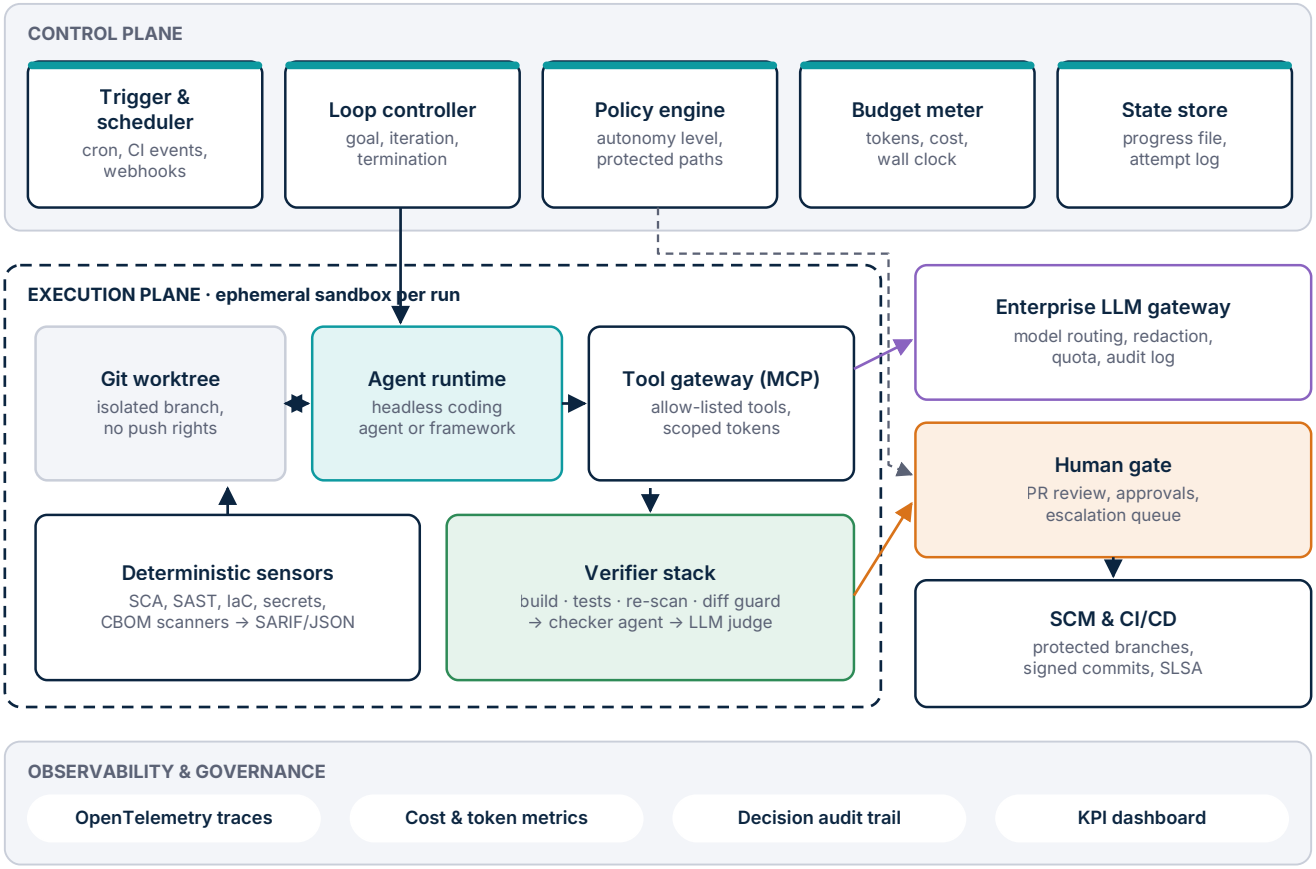


Figure 4. Secure Loop reference architecture. The agent runs in a disposable sandbox with no push rights; only the controller can open a pull request, and only after the verifier stack passes.

Control plane. The trigger and scheduler start runs. The loop controller owns the goal, iteration count and termination decisions; it is ordinary, reviewable code, not a model. The policy engine enforces the autonomy level and protected paths. The budget meter stops runs that exceed tokens, cost or time. The state store holds progress and attempt history.

Execution plane. Each run receives an ephemeral sandbox (container, VM or microVM) with a fresh Git worktree, the agent runtime, a tool gateway that exposes only allow-listed tools with scoped, short-lived credentials, the deterministic sensors (scanners) and the verifier stack.

Boundary services. All model calls traverse the enterprise LLM gateway for routing, redaction, quota and audit. Source control enforces protected branches, required reviews and signed commits. Build provenance (for example SLSA attestations) links each released artefact back to the loop run that produced its change.

3.2 Autonomy levels

Autonomy is a policy decision, set per loop and per change class, and earned with evidence. A loop starts at L1 and is promoted only when its metrics justify it (Section 8).

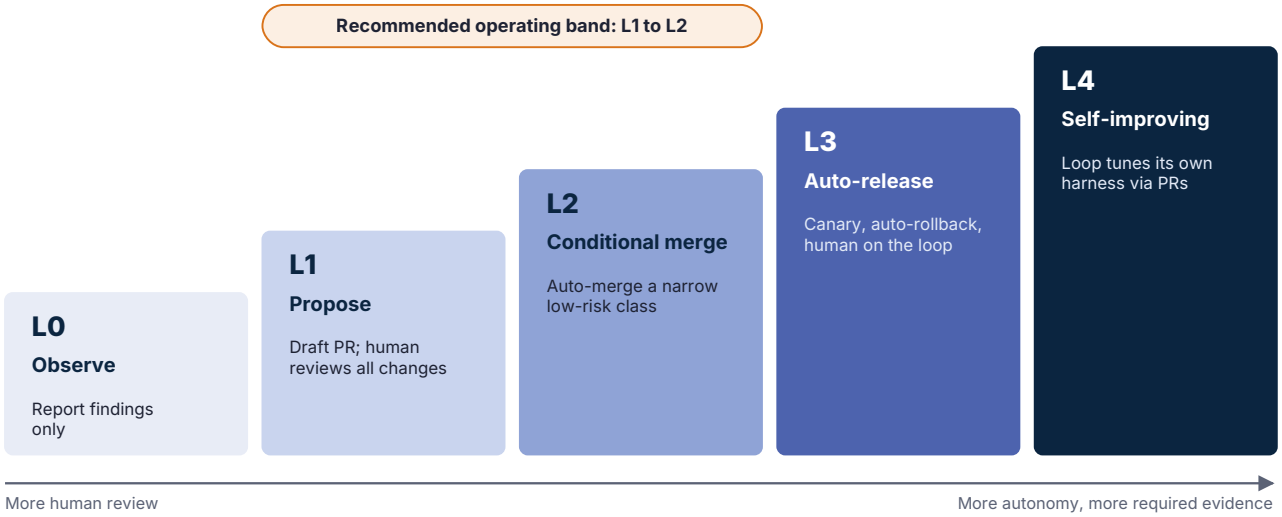


Figure 5. Autonomy ladder. Most security loops operate at L1; narrow, well-verified change classes may run at L2.

Level	Loop may...	Entry criteria	Example
L0	Detect, enrich and report. No code changes.	None	CBOM discovery report; triage recommendations
L1	Open a pull request with evidence. Humans review every change.	Charter, sandbox, deterministic verifier, diff guard	All six use cases at pilot start
L2	Auto-merge a narrowly defined change class; humans audit a sample.	≥ 50 L1 changes in the class, ≥ 95% acceptance, zero regressions escaped, sponsor sign-off	Patch-level dependency bumps with full test pass
L3	Release through progressive delivery with automatic rollback.	L2 for one quarter; canary and rollback proven	Base-image refresh for stateless services
L4	Propose changes to its own skills, prompts and verifiers.	Mature telemetry; changes reviewed as code	Improvement loop tuning a skill after repeated failures

Table 4. Autonomy levels and promotion criteria. Thresholds are recommended defaults; programme owners may tighten them.

RECOMMENDATION

Changes to cryptography, authentication, authorisation, secrets history and anything under `infra/prod` remain at L1 permanently, regardless of loop maturity.

3.3 The verifier stack

The verifier is the heart of the loop. It is built bottom-up: cheap, deterministic, hard-to-game checks first; independent agent review second; rubric-based model judgement third; and a human as the final authority.



Figure 6. Verifier stack. A change must clear every lower tier before a higher tier is consulted.

1. **Diff guard.** Rejects any change to protected paths, deletion of tests, or newly added suppression markers (`@Disabled`, `it.skip`, `nosemgrep`, `# nosec`). This single control removes the most common reward-hacking routes.
2. **Build and tests.** The full suite, not a subset chosen by the agent. Coverage must not decrease.
3. **Re-scan.** The same scanner that produced the finding must report it closed, and no new findings of equal or higher severity may appear.
4. **Policy.** Licence, provenance, version-range and change-size policies (for example, no major-version upgrades at L1/L2).
5. **Checker agent.** A separate agent with fresh context and read-only tools reviews the diff against a rubric and must return an explicit verdict with evidence.
6. **Human review.** CODEOWNERS review with the evidence bundle attached to the pull request.

3.4 Termination and escalation

After every attempt the controller evaluates the exits in a fixed order. The loop never decides to "keep going" without new evidence to act on.

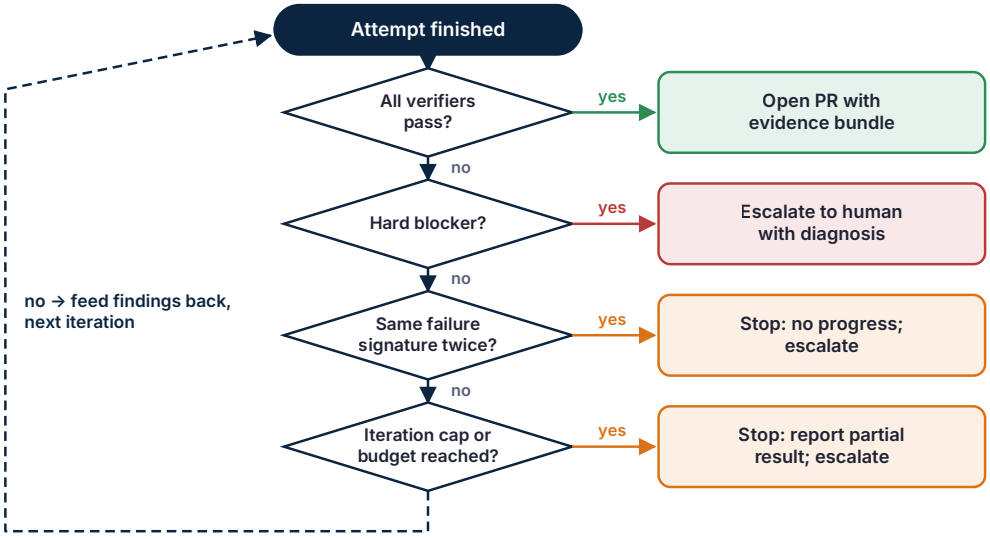


Figure 7. Termination logic evaluated after every attempt. Escalations carry a diagnosis bundle: findings, attempts, diffs and verifier output.

3.5 Design principles

Principle	In practice
Least agency	Give the agent the minimum tools, scopes and network reach the task needs. No push rights, no production credentials, no unrestricted shell on shared hosts.
Evidence over assertion	An agent's claim that "tests pass" is ignored; the controller runs the tests itself and records the output.
Deterministic first	Use a model only for what deterministic tools cannot do: choosing a fix, adapting call sites, explaining risk.
Small, reversible changes	One finding class per pull request; smallest viable version bump; no drive-by refactoring.
State outside the model	Memory lives on disk or in a tracker, never only in a conversation.
Separation of duties	Maker and checker are independent; the controller, not either agent, decides acceptance.
Everything is code	Charters, skills, prompts, policies and the controller are versioned and reviewed like production code.

Table 5. Design principles for secure loops.

SECTION 4

Use Case Portfolio

Not every security task is a good first loop. The best candidates have a deterministic verifier, a high volume of repetitive work and changes that are easy to review and revert.

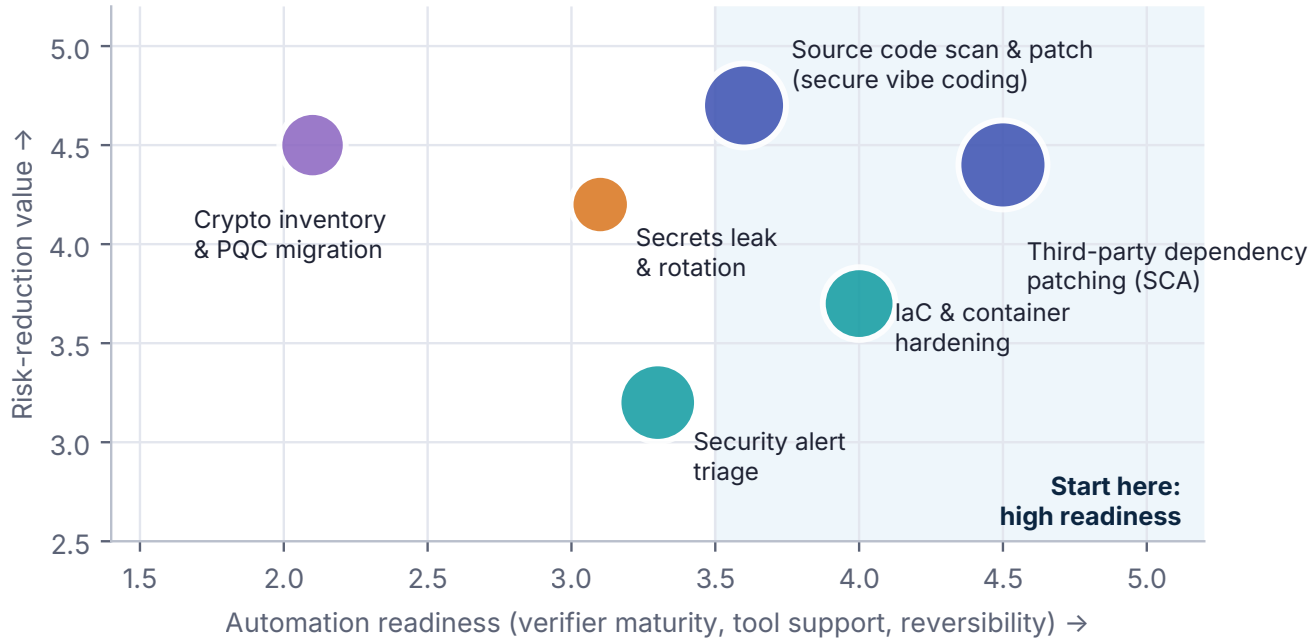


Figure 8. Indicative prioritisation of the six use cases (author's assessment on a 1 to 5 scale; bubble size reflects typical finding volume). Re-score with your own portfolio data before committing resources.

Use case	Primary sensors	Deterministic verifier	Start at	Value driver
5.1 Third-party dependency patching	OSV-Scanner, Trivy, Grype + Syft	Build, tests, re-scan, provenance check	L1 → L2	Largest finding volume; well-defined fixes
5.2 Source code scan and patch	Semgrep, CodeQL, language linters	Tests, SAST re-scan, diff guard	L1	Stops vulnerable AI-generated code before merge
5.3 IaC and container hardening	Checkov, KICS, Trivy config, Hadolint	Validate, plan diff, policy re-scan	L1	Removes misconfiguration drift at scale
5.4 Secrets leak and rotation	Gitleaks, TruffleHog	Re-scan, secret-manager read test	L1	Shortens credential exposure windows
5.5 Crypto inventory and PQC	CBOM generator, crypto rules, TLS scanners	KATs, interop tests, re-scan	L0 → L1	Quantum-readiness and crypto-agility
5.6 Finding and alert triage	All of the above via SARIF; SIEM alerts	Labelled golden set; evidence required	L0 → L1	Cuts false-positive load on analysts

Table 6. Use case summary. Tools listed are open-source defaults; equivalent commercial tools plug into the same loop.

SELECTING YOUR OWN USE CASE

Apply three tests. **Verifiable:** can a machine tell a good result from a bad one? **Repetitive:** does the task recur often enough to justify the harness? **Reversible:** can a bad change be caught in review or rolled back cheaply? A use case that passes all three is a strong loop candidate; one that fails the first should start at L0.

SECTION 5

Use Case Playbooks

Each playbook follows the same structure: objective, loop card, workflow, step-by-step procedure and use-case-specific guardrails. Section 6 then shows how to build any of them.

5.1 Third-party dependency vulnerability find-and-patch

SCA | Start at L1 | L2 for patch-level bumps

Objective. Continuously remove known-vulnerable third-party libraries and components, direct and transitive, by applying the smallest safe upgrade, proving that the build and behaviour are intact, and delivering a review-able pull request with full evidence.

Trigger	Nightly schedule; new advisory published for a package in the SBOM; lockfile change on the default branch.
Sensors	OSV-Scanner (source and container scanning, guided remediation for npm and Maven), Trivy or Grype against an SBOM produced by Syft or cdxgen. Prioritise with CISA KEV membership and EPSS probability, then reachability where the ecosystem supports it (for example `govulncheck` for Go).
Agent actions	Group findings by package; select the lowest fixed version in the same major line; update manifests, lockfiles or dependency-management overrides; adapt call sites when a minor API changed; update the progress note.
Verifiers	Diff guard; clean install from lockfile; full build and test suite; scanner re-run shows the finding closed and nothing new; licence policy; publisher provenance check for the new version; SBOM diff.
Exit and escalation	Major-version upgrade required, no fixed version published, tests fail twice with the same signature, or budget exhausted → issue in AppSec queue with diagnosis bundle.
KPIs	Median time to remediate critical/high; PR acceptance rate; regression escape rate; cost per merged fix.

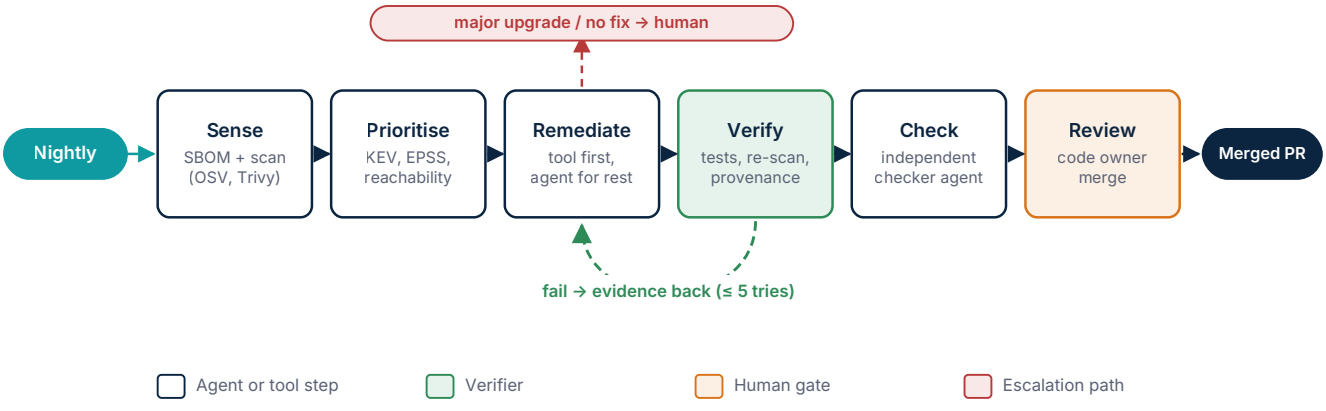


Figure 9. Dependency remediation loop.

Procedure

1. Generate or refresh the SBOM for the repository and scan it; write raw results to `.loop/state/`.
2. Prioritise: KEV-listed first, then EPSS above the agreed threshold, then CVSS critical/high. Drop findings already covered by an open loop PR.
3. Where a deterministic remediator exists, run it first (for example `osv-scanner fix --strategy=in-place` for npm lockfiles, Renovate or Dependabot for routine bumps). Use the agent for the cases the tool cannot finish: transitive overrides, breaking minor APIs, multi-module builds.
4. The agent applies the change following the `dependency-remediation` skill and records evidence.
5. The controller runs the verifier stack. On failure it returns the concrete error output to the agent; on repeated identical failure it escalates.
6. The checker agent reviews the diff for scope creep, behavioural risk and provenance; the controller opens a signed pull request labelled `loop-generated`.
7. A code owner merges. At L2, a defined patch-level class merges automatically when every check passes, and 10% of auto-merges are sampled for human audit.

GUARDRAILS SPECIFIC TO THIS USE CASE

Never accept a version published within the last 72 hours without a provenance check (fast-moving malicious releases are a known supply-chain pattern). Pin CI actions and scanner binaries by digest: in March 2026 the version tags of a widely used scanner's GitHub Action were force-pushed to credential-stealing commits, compromising pipelines that referenced tags rather than full commit SHAs.

5.2 Source code security scan and patch (secure vibe coding)

SAST | Start at L1 | Prompt to production

Objective. Make AI-assisted development ("vibe coding") safe enough to go from a natural-language prompt to production. Two loops cooperate: a fast *inner loop* on the developer's workstation that prevents insecure code from being committed, and an *outer loop* in CI that scans the full change, patches what remains, and delivers a reviewed, signed, provenance-tracked release.

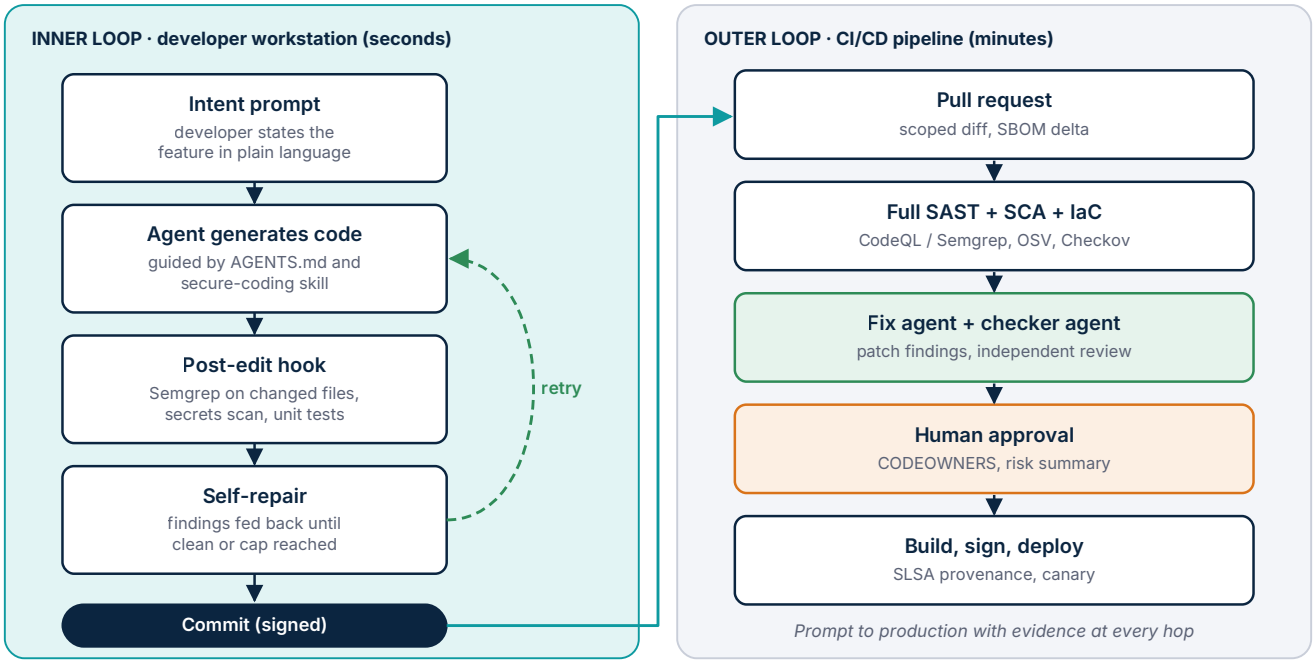


Figure 10. Secure vibe coding: the inner loop repairs in seconds at the point of generation; the outer loop proves the change in CI before it reaches production.

Trigger	Inner loop: every file edit by the coding agent (post-edit hook). Outer loop: pull request opened or updated; nightly full-repository scan.
Sensors	Semgrep (fast, rule-based, changed files) inner loop; CodeQL or Semgrep full analysis outer loop; language-specific tools (Bandit, gosec, SpotBugs with FindSecBugs, ESLint security plugins); Gitleaks for secrets; optional OWASP ZAP baseline against a preview environment.
Agent actions	Generate code under a secure-coding skill; on findings, refactor the sink (parameterised queries, output encoding, safe deserialisation, path canonicalisation), add or update a test that reproduces the vulnerability, and explain the fix.
Verifiers	Diff guard (no suppressions, no test deletion); new regression test fails on the old code and passes on the new; SAST re-scan clean at the finding location with no new high findings; full test suite; checker agent review.
Exit and escalation	Fix requires design change (for example an authorisation model), finding disputed by the agent with evidence (routed to triage loop 5.6), or cap reached.
KPIs	Findings per 1,000 AI-generated lines at merge; inner-loop self-repair rate; outer-loop fix acceptance; escaped vulnerabilities found later by testing.

Procedure: inner loop

1. Commit an `AGENTS.md` and a `secure-coding` skill that encode the organisation's secure coding standard as short, testable rules (for example "all SQL through the repository layer with bound parameters").
2. Configure the coding agent's post-edit hook to run a changed-files-only scan and fast tests (snippet below). A non-zero exit returns findings to the agent, which repairs before continuing.
3. Add a pre-commit gate with the same checks so that code written without the agent is held to the same bar.

```
# Post-edit hook (secure vibe coding inner loop)
# Most coding agents support a hook that runs after each file edit and returns
# its output to the model. Wire it to a fast, changed-files-only scan.
#!/usr/bin/env bash
set -euo pipefail
CHANGED=$(git diff --name-only --diff-filter=AM HEAD -- '*.py' '*.ts' '*.js' '*.java' '*.go')
[ -z "$CHANGED" ] && exit 0
semgrep scan --config p/default --config .loop/rules/ --error --quiet $CHANGED
gitleaks git --staged --redact --no-banner
make test-fast
# Non-zero exit returns the findings to the agent, which repairs before continuing.
```

Listing 1. Post-edit hook that closes the inner loop.

Procedure: outer loop

1. On every pull request, run full SAST, SCA and IaC scans and publish SARIF.
2. For each high or critical finding introduced by the change, start a fix iteration: the agent receives the SARIF result, the surrounding code and the secure-coding skill.
3. The agent writes a failing security regression test first, then the fix. The controller verifies both properties.
4. The checker agent confirms scope and absence of weakened controls; the controller pushes the fix as a separate commit on the PR branch with a clear message.
5. A human approves; the pipeline builds, signs and records provenance, then deploys through a canary.

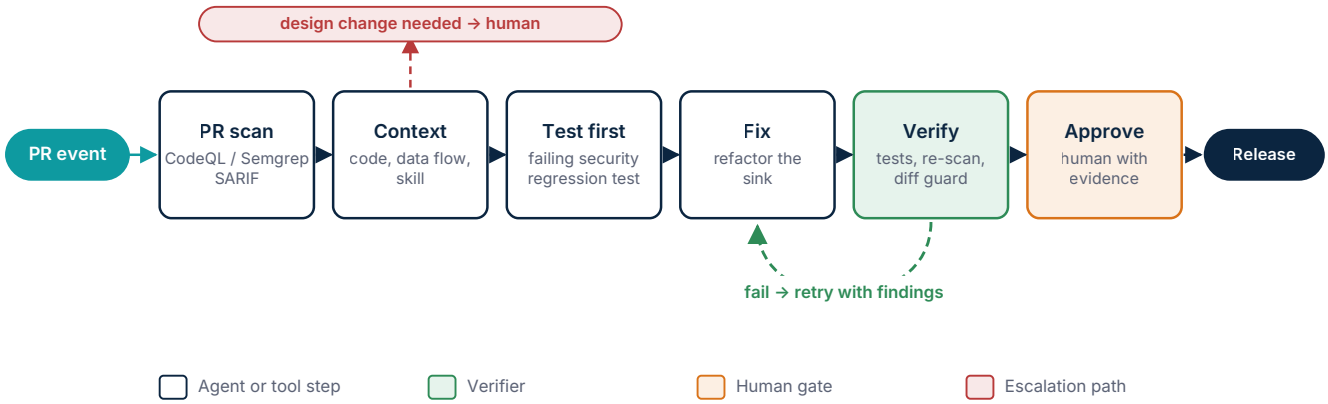


Figure 11. Outer-loop fix cycle for a pull request.

5.3 Infrastructure-as-code and container hardening

IaC / Containers | Start at L1 | L3 possible for base-image refresh

Objective. Detect and fix misconfigurations in Terraform, Kubernetes manifests, Helm charts and Dockerfiles, and keep container images on minimal, current, patched base images, without ever proposing a change that destroys or replaces live infrastructure.

Trigger	PR touching IaC paths; weekly drift scan; new base-image digest available.
Sensors	Checkov or KICS (Terraform, CloudFormation, Kubernetes, Helm), Trivy config and image scanning, Hadolint for Dockerfiles, kube-linter; custom policies in OPA/Rego via Conftest.
Agent actions	Apply secure defaults (encryption at rest, private endpoints, least-privilege IAM, `runAsNonRoot`, read-only root filesystem, dropped capabilities); move images to minimal or distroless bases; pin base images by digest.
Verifiers	`terraform validate`; a plan containing *no delete or replace actions*; policy re-scan; image rebuild, image re-scan and container smoke test; Kubernetes server-side dry run.
Exit and escalation	Any plan with destructive actions, IAM changes that broaden access, or changes to production state files → human.
KPIs	Policy violations per environment; time to remediate; image CVE count per service; percentage of images on approved bases.

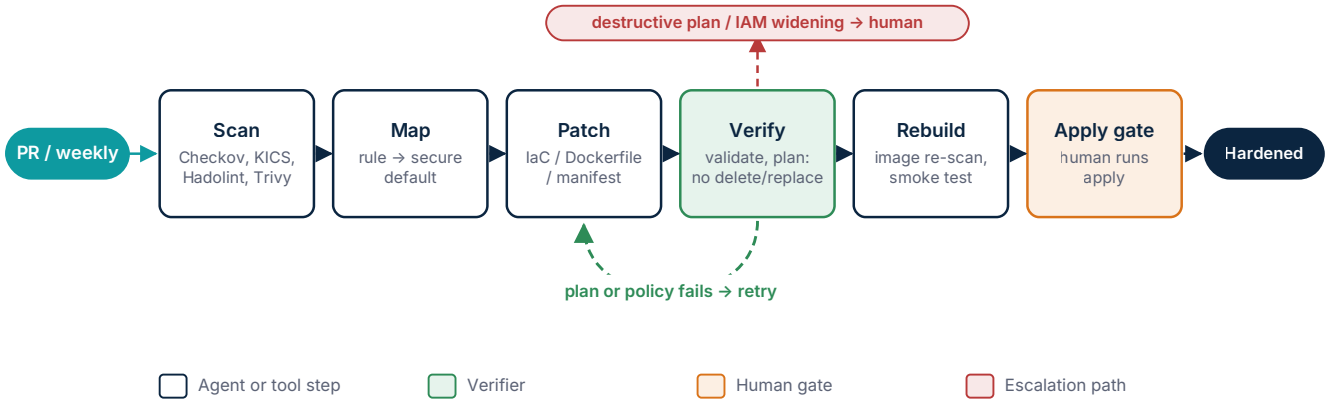


Figure 12. IaC and container hardening loop.

```
# Verifier for IaC loop: refuse any plan that destroys or replaces resources
terraform init -backend=false && terraform validate
terraform plan -out=tf.plan -lock=false
terraform show -json tf.plan > plan.json
jq -e '[.resource_changes[] | select(.change.actions | index("delete"))] | length == 0' plan.json
checkov -f plan.json --framework terraform_plan --compact --quiet
```

Listing 2. IaC verifier: the loop refuses any plan that deletes or replaces resources.

GUARDRAILS SPECIFIC TO THIS USE CASE

The agent never receives cloud credentials with write access; plans run with read-only roles and no remote state lock. Apply is always a human-initiated pipeline step. Pin scanner actions by SHA: an infrastructure-as-code scanner's GitHub Action was also targeted in the March 2026 action-tag compromise campaign.

5.4 Secrets leak detection and rotation

Secrets | Start at L1 | History rewrite always human

Objective. Find hard-coded credentials in code, configuration and history; confirm whether they are live; rotate them through the secret manager; refactor code to read secrets by reference; and, where required, purge them from Git history under human control.

Trigger	Push protection event; scheduled full-history scan; external report of an exposed key.
Sensors	Gitleaks (fast, rule-based), TruffleHog (with live verification of many credential types), detect-secrets baselines.
Agent actions	Refactor the code path to fetch the secret from the approved secret manager by name; add configuration for local development; open the rotation ticket; draft the history-purge plan.
Verifiers	Re-scan shows no secret at the location; application reads the secret from the manager in a test environment; old credential confirmed revoked by the rotation job.
Exit and escalation	Verified live secret → immediate page to service owner in parallel with the loop. History rewrite and force-push are never automated.
KPIs	Time from detection to revocation; live secrets per 100 repositories; recurrence rate.

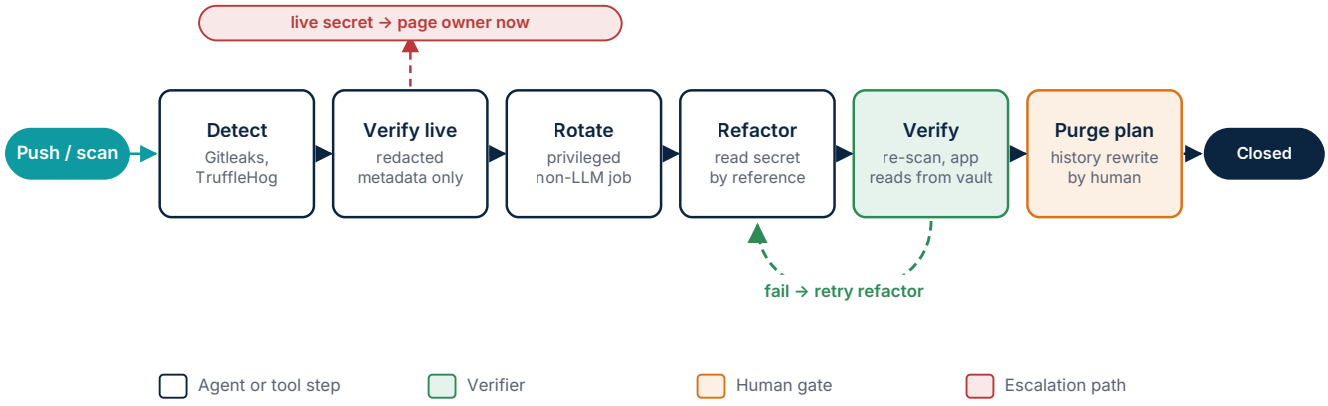


Figure 13. Secrets loop. Rotation is performed by a privileged, non-LLM job; the agent handles only the code refactor.

```
# Secrets loop: the agent never sees the secret value
gitleaks git --redact --report-format json --report-path .loop/state/leaks.json
trufflehog git file://. --results=verified --json > .loop/state/verified.json
# Rotation is executed by a privileged, non-LLM job; the agent only receives
#   {rule_id, file, line, secret_name, verified: true/false, rotation_ticket}
# and refactors code to read the secret from the secret manager by name.
```

Listing 3. Detection and the redacted contract between scanner, rotation job and agent.

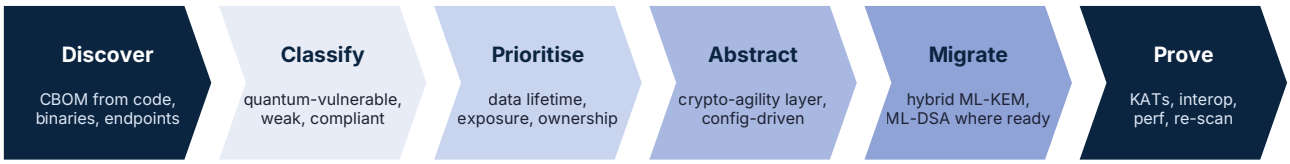
NON-NEGOTIABLE

Secret values must never enter model context, prompts, logs or traces. Scanners run with redaction enabled, and the agent receives only metadata: rule, file, line, secret name, verification status and ticket reference.

5.5 Cryptographic inventory and post-quantum migration

CBOM / PQC | L0 discovery | L1 for every change, permanently

Objective. Build and maintain a Cryptography Bill of Materials (CBOM), classify every algorithm, key and protocol use, remove broken cryptography, introduce crypto-agility, and plan and draft migrations to the NIST post-quantum standards: ML-KEM (FIPS 203), ML-DSA (FIPS 204) and SLH-DSA (FIPS 205). The CBOM is the loop's primary sensor.



Every migration step is human-gated (autonomy L1). The loop accelerates discovery and drafting, not approval.

Figure 14. Post-quantum migration loop phases. Discovery and drafting are automated; every change is approved by a cryptography owner.

Trigger	Monthly CBOM refresh; new crypto policy version; release of a PQC-capable library version.
Sensors	CycloneDX CBOM generation (for example cdxgen's `cbom` command, or static analysers that emit CBOM), Semgrep crypto rules, TLS configuration scanners (sslyze, testssl.sh), certificate inventories.
Agent actions	Classify assets against policy; replace broken primitives (MD5, SHA-1 signatures, DES/3DES, RC4, short RSA keys); introduce an algorithm-agnostic interface so algorithms become configuration; draft hybrid key-exchange changes where libraries support them.
Verifiers	Known-answer tests; interoperability tests against peers; performance benchmarks within budget; CBOM re-scan confirms the asset reclassified; no new weak primitive introduced.
Exit and escalation	Every change goes to a cryptography owner. Protocol or wire-format changes require an architecture review.
KPIs	Percentage of assets inventoried; broken-crypto count; percentage of quantum-vulnerable assets with an approved migration plan; crypto-agility coverage.

```
# Crypto inventory for the PQC loop
cbom -o .loop/state/cbom.json .                  # cdxgen CBOM (CycloneDX)
semgrep scan --config .loop/rules/crypto/ --json -o .loop/state/crypto.json
sslyze --json_out=.loop/state/tls.json api.internal.example:443
# Policy classes the loop applies to each asset:
#   BROKEN      MD5, SHA-1 signatures, DES/3DES, RC4, RSA < 2048
#   QV          RSA, ECDSA, ECDH, DH  (quantum-vulnerable; plan migration)
#   PQC-READY   ML-KEM (FIPS 203), ML-DSA (FIPS 204), SLH-DSA (FIPS 205), hybrids
```

Listing 4. Crypto discovery sensors and the classification policy the loop applies.

WHY AUTONOMY STAYS AT L1

Cryptographic errors are silent: code can compile, tests can pass and interoperability can appear fine while security is lost. NIST's transition guidance (IR 8547) sets a timeline in which quantum-vulnerable public-key algorithms are deprecated and then disallowed in the 2030s, so the value of this loop is speed of discovery and quality of drafts, not unattended merges.

5.6 Security finding and alert triage

Triage | L0 → L1 | Auto-close only with evidence and sampling

Objective. Reduce analyst load by having agents enrich, de-duplicate and classify findings from all scanners and, where appropriate, application security alerts, routing true positives into the remediation loops above and proposing justified suppressions for false positives.

Trigger	New SARIF results from any loop or pipeline; new alert in the vulnerability management platform or SIEM queue.
Sensors	SARIF aggregation (for example DefectDojo), code search, call-graph and data-flow tools, service catalogue for exposure and data classification.
Agent actions	Gather code context; trace source-to-sink reachability; check exposure and asset criticality; produce a structured verdict with evidence; route to a fix loop or draft a suppression with justification.
Verifiers	Schema validation; evidence must cite real file and line locations (checked mechanically); precision and recall measured weekly on a labelled golden set; random 10% human audit of closures.
Exit and escalation	Confidence below threshold, conflicting evidence, or any finding on a crown-jewel asset → analyst queue.
KPIs	Analyst hours saved; triage precision; mean time to triage; percentage auto-routed to fix loops.

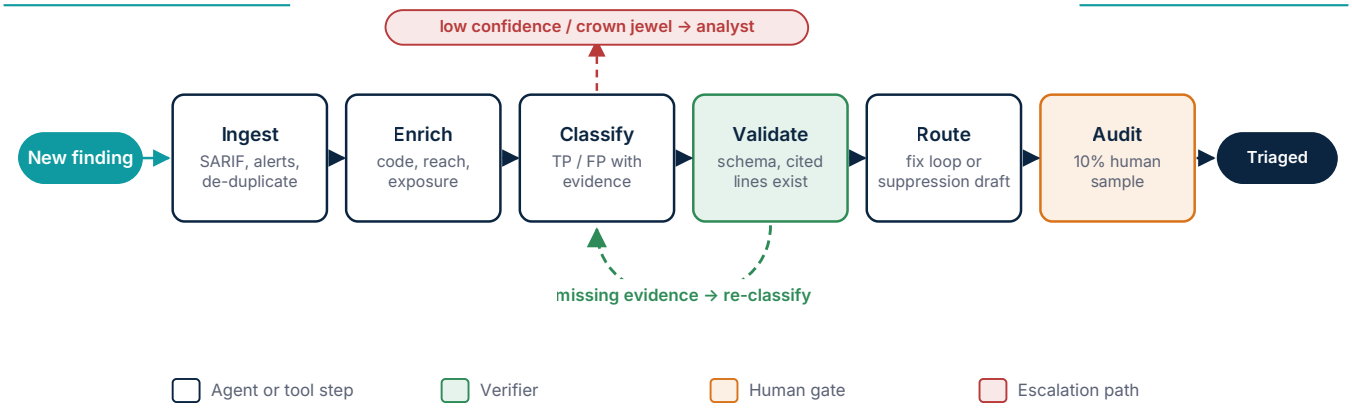


Figure 15. Triage loop. True positives feed the remediation loops; suppressions require evidence and audit.

```
{
  "finding_id": "semgrep:java.lang.security.audit.sqli.jdbc-sqli:OrderDao.java:88",
  "verdict": "true_positive",
  "confidence": 0.86,
  "evidence": [
    "user-controlled 'orderId' flows from OrderController.java:41 (no validation)",
    "string concatenation into Statement.executeQuery at OrderDao.java:88",
    "endpoint is internet-facing per service catalogue"
  ],
  "exploitability": "reachable",
  "recommended_route": "code-scan-and-patch-loop",
  "suppression_justification": null
}
```

Listing 5. Structured triage verdict. Free-text verdicts are rejected by the schema verifier.

SECTION 6

Implementation Guide: Building Your First Loop

This section is a complete, ordered procedure. It uses the dependency remediation loop (5.1) as the worked example because it has the most mature verifiers, but every step is written so that you can substitute any use case from Section 5, or your own.

6.0 Prerequisites

Skills

- Git branching, pull requests and code review
- Reading CI pipeline files (YAML) and shell scripts
- Basic Python (the controller is about 100 lines)
- Familiarity with one scanner's output format

Platform

- A non-critical pilot repository with a reliable test suite
- CI runners that can be made ephemeral and network-restricted
- Approved model access through the enterprise LLM gateway
- A bot identity with fine-grained, single-repository permissions

Phase A: Design

Step 1 Choose the use case and write the loop charter

The charter is the contract for the loop. Write it before any code. It forces precise answers to the questions that otherwise surface as incidents: what exactly counts as done, what is out of bounds, how far may the loop act alone, and when must it stop.

```
# .loop/charter.yaml  -  one charter per loop
loop_id: sca-remediation-payments-api
owner: appsec-platform@example.com        # accountable human team
use_case: third-party-dependency-patching
goal: >
  Remove every reachable CRITICAL or HIGH vulnerability in direct and
  transitive dependencies of services/payments-api without breaking the build,
  the test suite or the public API.
definition_of_done:
  - osv-scanner reports 0 CRITICAL/HIGH findings for the lockfile
  - build and full unit + integration test suite pass
  - no change to files under protected_paths
  - SBOM regenerated and diff attached to the pull request
non_goals:
  - major-version upgrades that change public interfaces
  - edits to application logic beyond import or API-call adjustments
autonomy_level: L1                    # L0 observe | L1 propose | L2 conditional merge
trigger:
  schedule: "0 2 * * 1-5"             # weekdays 02:00 UTC
  events: [new_advisory, lockfile_changed]
budget:
  max_iterations: 5
  max_tokens: 400000
  max_wall_clock_minutes: 45
  max_cost_usd: 12
no_progress: same_failure_signature_twice
escalation: open issue in AppSec queue with diagnosis bundle
protected_paths: [".github/**", ".loop/**", "**/test/**", "**/*.lock.sig", "infra/prod/**"]
```

Listing 6. Loop charter for the worked example. Every field is read by the controller or the policy guard.

Checklist for a good charter: the goal is measurable by a tool; every done-criterion is automatable; non-goals name the tempting shortcuts; the budget is finite in iterations, tokens, time and money; there is exactly one escalation path with a named owner.

Step 2 Lay out the loop workspace in the repository

Keep everything the loop needs in version control so that it is reviewable and reproducible.

```
repo-root/
├── AGENTS.md                     # standing instructions every agent reads first
├── .loop/
│   ├── charter.yaml              # goal, done-definition, autonomy, budget
│   ├── skills/
│   │   ├── dependency-remediation/SKILL.md
│   │   └── secure-coding/SKILL.md
│   ├── prompts/
│   │   ├── fixer.md              # maker prompt template
│   │   └── checker.md            # checker rubric
│   ├── state/
│   │   ├── progress.md           # human-readable spine: done, next, blocked
│   │   └── attempts.jsonl        # machine log: one line per attempt
│   ├── policy/
│   │   └── guard.py              # diff guard: protected paths, test deletion
│   └── controller.py             # the loop itself
└── .github/workflows/loop-sca.yml # trigger and sandbox
```

Listing 7. Recommended repository layout.

Step 3 Select the agent runtime and model access

Choose one option from each row. The controller treats the agent as a replaceable command, so you can change the choice later without redesigning the loop.

Layer	Open-source options	Commercial options (interchangeable)
Headless coding agent	OpenHands, Aider, goose, opencode	GitHub Copilot coding agent, OpenAI Codex, Claude Code, Gemini CLI, Cursor, IBM Bob
Agent framework (if building a custom agent)	LangGraph, Microsoft Agent Framework, OpenAI Agents SDK, Google ADK, CrewAI	Managed agent platforms from cloud providers
Model access	Self-hosted open-weight models behind the gateway	Frontier models through the enterprise LLM gateway
Tool connectivity	Model Context Protocol (MCP) servers	Vendor connectors exposed over MCP

Table 7. Runtime choices. Selection criteria: headless/non-interactive mode, hook support, configurable tool allow-lists, gateway compatibility and auditable logs.

TIP FOR BEGINNERS

Start with an existing headless coding agent rather than a framework. Frameworks are valuable when you need custom multi-agent graphs; for a first loop they add complexity without adding safety.

Phase B: Build

Step 4 Wire the deterministic sensors

Install each scanner at a pinned version with checksum verification, run it from a script, and write machine-readable output (JSON or SARIF) to `.loop/state/`. Add a small gate script per scanner that exits non-zero when the charter's done-criteria are not met (for example, any CRITICAL or HIGH remaining). The loop depends on these exit codes, so test them on known-good and known-bad inputs before involving any agent.

Step 5 Write AGENTS.md and the skills

`AGENTS.md` holds standing rules every agent must read; skills hold procedures for a specific job. Write them as short, testable instructions. Add a "known pitfalls" section and grow it from real failures; this is how the loop compounds knowledge instead of re-deriving the project each run.

```
# AGENTS.md  (excerpt)

## Mission for automated agents in this repository
You fix security findings. You do not add features, refactor for style or
change behaviour that is not required by the fix.

## Hard rules
1. Never modify, skip, delete or weaken a test to make a check pass.
2. Never edit files under .github/, .loop/, infra/prod/ or any *.sig file.
3. Never print, log or copy secret values. Refer to secrets by name only.
4. Prefer the smallest version bump that removes the vulnerability.
5. If a fix needs a major-version upgrade or an API change, stop and write
   BLOCKED: <reason> to .loop/state/progress.md.

## How to verify your work locally
- make build && make test
- osv-scanner scan source -r . --format json > .loop/state/osv.json
- semgrep scan --config p/default --error --json -o .loop/state/semgrep.json

## Conventions
- Java services use Maven with dependencyManagement for transitive pins.
- Node services use npm with a committed package-lock.json.
```

Listing 8. AGENTS.md excerpt.

```
---
name: dependency-remediation
description: Use when a scanner reports a vulnerable third-party package and the
  task is to upgrade, pin or override it safely in npm, Maven, pip or Go projects.
---
# Dependency remediation

## Procedure
1. Read .loop/state/osv.json. Group findings by package, not by CVE.
2. For each package choose the lowest fixed version in the same major line.
   - npm: try `osv-scanner fix --strategy=in-place -L package-lock.json` first.
   - Maven: add or update a <dependencyManagement> override for transitives.
   - pip: update the pin in requirements*.txt and re-lock with pip-compile.
3. Re-install from the lockfile (npm ci / mvn -U verify) and run tests.
4. Verify publisher provenance for every new version where the registry
   supports it (e.g. `npm audit signatures`); record the result.
5. Append one line per package to .loop/state/progress.md:
   <package> <old> -> <new> | CVEs closed | tests: pass/fail | provenance: ok/none

## Known pitfalls in this codebase
- jackson-databind must move together with jackson-core and jackson-annotations.
- Do not upgrade spring-boot-starter-* independently of the Spring Boot BOM.
```

Listing 9. Skill file for dependency remediation.

Step 6 Connect tools through MCP with least privilege

Expose only the operations the loop needs. Prefer read-only servers; allow-list individual operations for write-capable servers; inject scoped tokens from the CI secret store at runtime. Review third-party MCP servers as you would any dependency: pin versions, read the code or provenance, and run them inside the sandbox.

```
{
  "mcpServers": {
    "scm": {
      "command": "scm-mcp-server",
      "args": ["--repo", "org/payments-api", "--allow", "read_file,list_prs,create_pr,comment_pr"],
      "env": { "SCM_TOKEN": "${LOOP_BOT_TOKEN}" }
    },
    "advisories": {
      "command": "osv-mcp-server",
      "args": ["--read-only"]
    },
    "tickets": {
      "command": "tracker-mcp-server",
      "args": ["--project", "APPSEC", "--allow", "create_issue,comment_issue"]
    }
  }
}
```

Listing 10. MCP configuration with operation allow-lists (server names are illustrative).

Step 7 Build the verifier stack

Implement the tiers from Section 3.3 in order. Start with the diff guard, because it blocks reward hacking before any expensive check runs.

```
# .loop/policy/guard.py  -  deterministic diff guard (runs before every verify)
import fnmatch, subprocess, sys, yaml
charter = yaml.safe_load(open(".loop/charter.yaml"))
changed = subprocess.run(["git", "diff", "--name-status", "origin/main...HEAD"],
                         capture_output=True, text=True).stdout.splitlines()
violations = []
for row in changed:
    status, path = row.split("\t", 1)
    if any(fnmatch.fnmatch(path, p) for p in charter["protected_paths"]):
        violations.append(f"protected path touched: {path}")
    if status == "D" and ("/test" in path or path.endswith(("_test.go", ".spec.ts", "Test.java"))):
        violations.append(f"test file deleted: {path}")
# block silencing patterns introduced by the diff
diff = subprocess.run(["git", "diff", "origin/main...HEAD", "-U0"], capture_output=True, text=True).stdout
for marker in ["@Disabled", "it.skip(", "pytest.mark.skip", "nosemgrep", "# nosec", "//NOSONAR"]:
    if any(l.startswith("+") and marker in l for l in diff.splitlines()):
        violations.append(f"suppression marker added: {marker}")
if violations:
    print("\n".join(violations)); sys.exit(1)
```

Listing 11. Diff guard: protected paths, deleted tests and new suppression markers.

Then write the maker and checker prompts. The checker runs with fresh context, read-only tools and, where possible, a different model from the maker.

```
# .loop/prompts/fixer.md  (maker)
You are a security remediation engineer working inside an isolated sandbox.

GOAL
{{GOAL}}

RULES
Follow AGENTS.md and the dependency-remediation skill exactly. Make the smallest
change that satisfies the goal. Do not touch protected paths. Do not modify tests.

FEEDBACK FROM THE PREVIOUS ATTEMPT
{{FEEDBACK}}

BEFORE YOU FINISH
1. Run the local verification commands listed in AGENTS.md.
2. Update .loop/state/progress.md: what changed, why, evidence, what remains.
3. If the goal cannot be met without breaking a rule, write BLOCKED: <reason>.
```

Listing 12. Maker (fixer) prompt template.

```
# .loop/prompts/checker.md  (independent checker; fresh context, read-only tools)
You review a change produced by another agent. You did not write it.
Assess the diff against the charter and answer each item with evidence:
1. Does every change trace to a listed finding? List any that do not.
2. Could the change alter runtime behaviour beyond the fix? Name the risk.
3. Were tests, assertions, scanner rules or configs weakened in any way?
4. Are new dependency versions published by the expected maintainer?
5. Is the progress note accurate and complete?
Finish with exactly one line:  VERDICT: APPROVE   or   VERDICT: REJECT <reason>
```

Listing 13. Checker rubric. The controller accepts only an explicit APPROVE verdict.

Step 8 Implement the loop controller

The controller is deliberately plain code. It reads the charter, runs the agent, runs the verifiers itself, detects no-progress by hashing failure output, enforces the budget, and either opens a pull request or escalates. It never trusts the agent's own account of what happened.

```
#!/usr/bin/env python3
"""Minimal, vendor-neutral loop controller. Any headless coding agent that
accepts a prompt on stdin and edits files in the working tree can be plugged in
through AGENT_CMD (for example an open-source agent CLI in non-interactive mode)."""
import hashlib, json, os, subprocess, sys, time, yaml

CHARTER = yaml.safe_load(open(".loop/charter.yaml"))
BUDGET = CHARTER["budget"]
AGENT_CMD = os.environ["AGENT_CMD"]          # e.g. "<agent-cli> --headless --prompt-stdin"
LOG = ".loop/state/attempts.jsonl"

def sh(cmd, timeout=1800):
    p = subprocess.run(cmd, shell=True, capture_output=True, text=True, timeout=timeout)
    return p.returncode, (p.stdout + p.stderr)[-20000:]

def verify():
    """Deterministic verifiers first. Returns (passed, findings_text)."""
    results = {}
    results["guard"] = sh("python .loop/policy/guard.py")          # protected paths, test deletion
    results["build"] = sh("make build")
    results["tests"] = sh("make test")
    results["sca"] = sh("osv-scanner scan source -r . --format json "
                        "> .loop/state/osv.json; python .loop/policy/osv_gate.py")
    failed = {k: v[1] for k, v in results.items() if v[0] != 0}
```

```
    return (not failed), json.dumps(failed)[:15000]

def signature(text):
    return hashlib.sha256(text.encode()).hexdigest()[:16]

def run_agent(prompt):
    p = subprocess.run(AGENT_CMD, shell=True, input=prompt, capture_output=True,
                       text=True, timeout=BUDGET["max_wall_clock_minutes"] * 60)
    return p.stdout[-5000:]

def main():
    start, seen = time.time(), set()
    template = open(".loop/prompts/fixer.md").read()
    feedback = "First attempt. Read .loop/state/osv.json and AGENTS.md."
    for i in range(1, BUDGET["max_iterations"] + 1):
        if (time.time() - start) / 60 > BUDGET["max_wall_clock_minutes"]:
            return escalate("budget: wall clock exhausted", i)
        prompt = template.replace("{{GOAL}}", CHARTER["goal"]).replace("{{FEEDBACK}}", feedback)
        run_agent(prompt)
        ok, findings = verify()
        sig = signature(findings)
        log(i, ok, sig)
        if ok:
            ok_checker, notes = checker_review()          # independent agent, fresh context
            if ok_checker:
                return open_pull_request(i, notes)
            findings = notes
        if "BLOCKED:" in open(".loop/state/progress.md").read():
            return escalate("agent reported hard blocker", i)
        if sig in seen:
            return escalate("no progress: identical failure signature", i)
        seen.add(sig)
        feedback = f"Attempt {i} failed verification. Evidence:\n{findings}"
    return escalate("iteration cap reached", BUDGET["max_iterations"])

def checker_review():
    rc, out = sh("git diff --stat && git diff | head -c 60000")
    rubric = open(".loop/prompts/checker.md").read()
    verdict = run_agent(rubric + "\n\nDIFF:\n" + out)
    return ("VERDICT: APPROVE" in verdict), verdict

def log(i, ok, sig):
    with open(LOG, "a") as f:
        f.write(json.dumps({"iter": i, "passed": ok, "sig": sig, "ts": time.time()}) + "\n")

def open_pull_request(i, notes):
    sh("git checkout -b loop/$(date +%Y%m%d-%H%M) && git add -A && "
       "git commit -S -m 'sec: dependency remediation (loop)' && git push -u origin HEAD")
    sh("gh pr create --label loop-generated --body-file .loop/state/progress.md")
    print(f"PR opened after {i} iteration(s)"); return 0

def escalate(reason, i):
    sh(f"gh issue create --label loop-escalation --title 'Loop escalation: {reason}' "
       f"--body-file .loop/state/progress.md")
    print(f"Escalated after {i} iteration(s): {reason}"); return 2

if __name__ == "__main__":
    sys.exit(main())
```

Listing 14. Reference loop controller (Python). Replace `make build` and `make test` with your project's commands.

Step 9 Sandbox and isolate every run

- **Ephemeral compute:** a fresh container, VM or microVM per run (rootless containers, gVisor or Firecracker-class isolation for untrusted code paths), destroyed afterwards.

- **Worktree isolation:** one Git worktree and branch per run so that parallel loops never collide; the agent's credentials cannot push.
- **Egress allowlist:** only the LLM gateway, package registries and advisory sources. Everything else is blocked, which also blunts data exfiltration through prompt injection.
- **Short-lived identity:** OIDC federation from CI to the gateway and registries; no static API keys in repositories or runners.
- **Resource limits:** CPU, memory and wall-clock limits on the job, matching the charter budget.

Phase C: Operate

Step 10 Orchestrate with CI

```
# .github/workflows/loop-sca.yml
name: loop-sca-remediation
on:
  schedule: [{ cron: "0 2 * * 1-5" }]
  workflow_dispatch:
permissions:
  contents: write          # push loop/* branches only (enforced by branch rules)
  pull-requests: write
  issues: write
  id-token: write          # OIDC to the LLM gateway; no long-lived API keys
jobs:
  loop:
    runs-on: [self-hosted, sandboxed, ephemeral]   # disposable VM or microVM
    timeout-minutes: 60
    concurrency: { group: loop-sca-${{ github.ref }}, cancel-in-progress: false }
    steps:
      # Pin every third-party action to a full commit SHA, never a tag.
      - uses: actions/checkout@<full-40-char-commit-sha>
        with: { fetch-depth: 0, persist-credentials: false }
      - name: Install pinned scanners (checksum-verified)
        run: ./scripts/install-tools.sh --verify-sha256
      - name: Obtain short-lived gateway token via OIDC
        run: ./scripts/gateway-login.sh            # exports LLM_GATEWAY_TOKEN (15 min)
      - name: Egress allowlist
        run: sudo ./scripts/egress-allow.sh llm-gateway.internal registry.npmjs.org repo1.maven.org api.osv.dev
      - name: Run loop
        env:
          AGENT_CMD: ${{ vars.AGENT_CMD }}
          GH_TOKEN: ${{ secrets.LOOP_BOT_TOKEN }}  # fine-grained, this repo only
        run: python .loop/controller.py
      - name: Upload evidence
        if: always()
        uses: actions/upload-artifact@<full-40-char-commit-sha>
        with: { name: loop-evidence, path: .loop/state/ }
```

Listing 15. Scheduled CI job. Every third-party action is pinned to a full commit SHA; the token is fine-grained and short-lived.

Step 11 Configure human gates

1. Protect the default branch: required reviews from CODEOWNERS, required status checks (the same verifiers), signed commits, no force-push.
2. Use a pull request template that the controller fills automatically: findings closed, versions changed, verifier results, checker verdict, cost and iteration count.
3. Label loop PRs (`loop-generated`) so reviewers and metrics can distinguish them.

4. Route escalations to one queue with a service-level objective (for example two business days) and a named owner.

Step 12 Instrument with telemetry

Emit one trace per run and one span per iteration using the OpenTelemetry generative-AI semantic conventions, extended with loop-specific attributes. Send traces to your observability stack or an open-source LLM tracing tool (for example Langfuse or Arize Phoenix). Redact prompts and outputs according to data-classification policy.

```
# Attributes recorded on every loop span (OpenTelemetry GenAI conventions + loop fields)
gen_ai.operation.name     = "invoke_agent"
gen_ai.request.model      = "<model-id from gateway>"
gen_ai.usage.input_tokens = 18234
gen_ai.usage.output_tokens= 2210
loop.id                   = "sca-remediation-payments-api"
loop.iteration            = 3
loop.autonomy_level       = "L1"
loop.verifier.result      = "fail:tests"
loop.failure_signature    = "9f1c2a7e44b0d1e3"
loop.exit_reason          = "success | cap | budget | no_progress | blocker"
loop.cost_usd             = 1.84
```

Listing 16. Recommended span attributes.

Step 13 Evaluate before you trust

1. Build a **golden set** of 20 to 50 historical findings with known correct fixes (or known "cannot fix" outcomes).
2. Run the loop against the golden set in shadow mode (L0) and measure: first-pass verify rate, final success rate, false "done" rate, mean iterations, mean cost.
3. Seed **adversarial cases**: a failing test that tempts deletion, a dependency README containing injected instructions, a fix that needs a major version. The loop must refuse, block or escalate each one.
4. Only move to L1 when false "done" is zero and all adversarial cases are handled.

Step 14 Pilot, then scale

Run L1 on one repository for two to four weeks. Review every PR, log every reviewer correction as a candidate skill update, and tune budgets. Expand to five to ten repositories with similar stacks, then to other stacks with their own skills. Promote a change class to L2 only against the criteria in Table 4.

Step 15 Close the improvement loop

Weekly, review the escalation queue and traces. Classify each failure: missing knowledge (update a skill), weak verifier (add a check), bad decomposition (change the plan step), or genuine limit (keep escalating). Make every improvement as a reviewed pull request against `.loop/`. This is loop 4 from Figure 3, and it is what turns a script into a system that gets better over time.

DEFINITION OF DONE FOR YOUR FIRST LOOP

Charter approved; sandbox and egress controls verified; diff guard tested with adversarial cases; golden-set results recorded; telemetry visible; escalation owner named; at least ten L1 pull requests reviewed with acceptance rate and cost reported to the sponsor.

SECTION 7

Securing the Loop Itself

A security loop is itself an agentic application with credentials, tools and autonomy. It must be threat-modelled and controlled like one. The OWASP Top 10 for Agentic Applications (2026 edition, published December 2025) provides the common vocabulary.

OWASP agentic risk	How it appears in a security loop	Loop controls in this guide
ASI01 Agent goal hijack	Injected text in a dependency README, issue or code comment redirects the agent	Fixed charter goal; untrusted content tagged as data; diff guard; egress allowlist; checker review
ASI02 Tool misuse and exploitation	Agent calls a write tool outside its task (closes issues, edits workflows)	MCP operation allow-lists; read-only by default; protected paths
ASI03 Identity and privilege abuse	Bot token reused to push to main or reach other repositories	Fine-grained single-repo tokens; OIDC; no push to protected branches
ASI04 Agentic supply chain	Compromised scanner, CI action, MCP server or model weights	SHA/digest pinning; checksum-verified installs; provenance checks; vetted MCP servers
ASI05 Unexpected code execution	Agent runs install scripts or generated code with host access	Ephemeral sandbox; `--ignore-scripts` where possible; resource limits
ASI06 Memory and context poisoning	Malicious text persisted into `progress.md` or skills and replayed later	State and skills changed only via reviewed PRs; state sanitised; per-run fresh context
ASI07 Insecure inter-agent communication	Maker forges or influences the checker's verdict	Controller mediates all hand-offs; checker gets diff and evidence, not maker reasoning
ASI08 Cascading failures	A bad skill update spreads defects across many repositories	Staged rollout; per-loop kill switch; regression metrics with automatic pause
ASI09 Human-agent trust exploitation	Reviewers rubber-stamp confident, well-formatted PRs	Evidence bundle; sampled deep reviews; reviewer rotation; acceptance-rate anomalies flagged
ASI10 Rogue agents	Loop behaviour drifts while appearing legitimate	Full tracing; behavioural baselines; budget ceilings; owner accountability; kill switch

Table 8. Mapping of OWASP agentic risks to loop controls.

KILL SWITCH

Every loop must have a single, documented way to stop all runs immediately (for example a repository variable checked at controller start, plus disabling the scheduled job). Test it during the pilot.

SECTION 8

Measuring Success

Measure outcomes (risk removed, time saved) and the health of the loop (quality, cost, safety). A loop that is fast but produces rejected PRs is a cost centre; a loop that is accurate but escalates everything has not yet earned its keep.

Metric	Definition	Pilot target (indicative)
Median time to remediate	Days from finding detection to merged fix, critical/high only	≥ 60% reduction vs. baseline
PR acceptance rate	Loop PRs merged ÷ loop PRs opened	≥ 80% at L1; ≥ 95% to enter L2
First-pass verify rate	Attempts passing all deterministic checks on iteration 1	Tracked for trend; improves with skills
False "done" rate	Runs claiming success that a human or later test disproves	0 (any occurrence triggers review)
Regression escape rate	Loop-merged changes causing a defect or rollback	< 1%
Escalation rate	Runs ending in human escalation	15% to 30% is healthy at pilot
Cost per merged fix	Model, compute and review time per merged PR	Below fully loaded manual cost
Reviewer minutes per PR	Median human review time for loop PRs	< 10 minutes for SCA class

Table 9. Core KPIs. Targets are starting points for a pilot; calibrate after the baseline period.

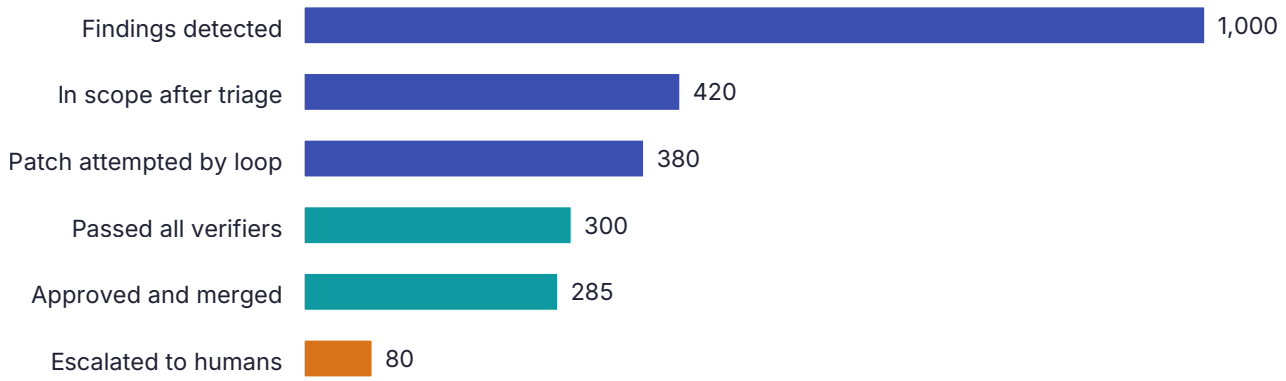


Figure 16. *Illustrative* remediation funnel for one quarter of a dependency loop pilot, showing how findings flow to merged fixes or human escalation. Replace with your own pilot data.

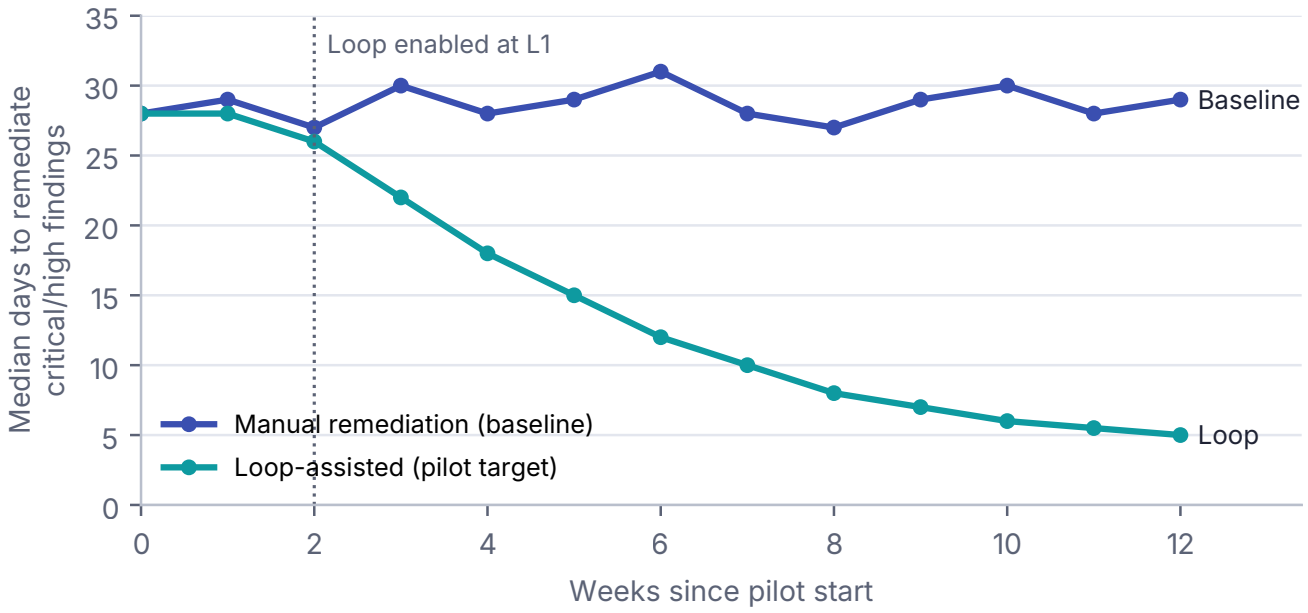


Figure 17. *Illustrative* target trajectory for median time to remediate after enabling the loop at L1. The shape, not the values, is the point: improvement follows skill tuning over several weeks.

SECTION 9

Governance and Operating Model

Loops change code on behalf of the organisation. Clear ownership, policy and auditability are what allow them to scale.

Activity	Loop owner	AppSec	Service team	Platform	AI governance
Approve charter and autonomy level	R	A	C	C	C
Build and maintain controller, skills, verifiers	A/R	C	C	C	I
Review and merge loop PRs	I	C	A/R	I	I
Sandbox, runners, gateway, identity	C	C	I	A/R	C
Promote to L2 or higher	R	A	C	C	C
Handle escalations	R	A	R	I	I
Quarterly risk review and audit	R	C	I	C	A

Table 10. RACI (Responsible, Accountable, Consulted, Informed).

9.1 Policy statements

1. Every loop has an approved charter, a named accountable owner and a documented kill switch.
2. Loops access models only through the enterprise LLM gateway, in line with data-classification rules.
3. No loop holds credentials that can push to protected branches, apply infrastructure or read production secrets.
4. Loop-generated changes are labelled, traceable to a run, and retained with their evidence for audit.
5. Autonomy above L1 requires documented evidence against Table 4 and AppSec approval; cryptography, authentication and secrets-history changes remain at L1.
6. Charters, skills, prompts, policies and controllers are versioned and changed only through reviewed pull requests.

ACCOUNTABILITY

Agent output does not transfer accountability. The reviewer who approves a loop PR, and the loop owner who configured it, carry the same responsibility as for any human-authored change. Watch for *comprehension debt*: code merged faster than teams understand it. Sampled deep reviews and clear PR explanations are the countermeasure.

SECTION 10

Adoption Roadmap: First 90 Days

Start narrow, prove value with evidence, and widen only as metrics allow.

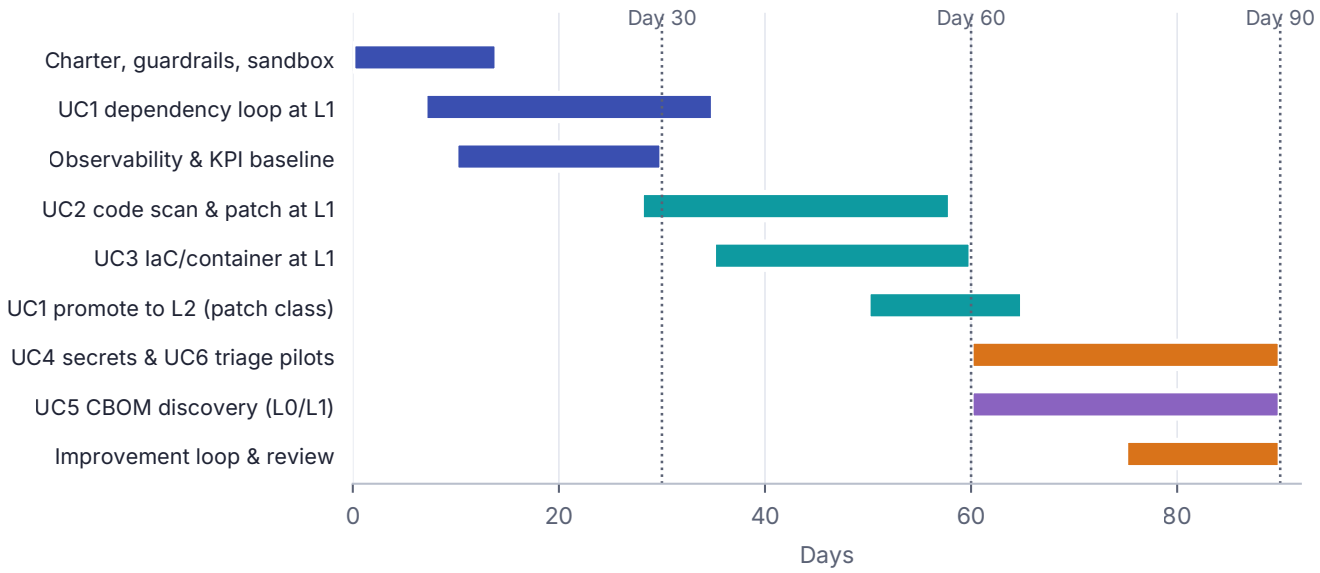


Figure 18. Recommended 90-day roadmap. Colours group work by wave: foundation and first loop, second-wave loops and promotion, and third-wave pilots.

Milestone	Exit criteria	Decision for sponsor
Day 30	Guardrails in place; UC1 at L1 on a pilot repository; golden-set and adversarial results recorded; KPI baseline captured	Continue and fund wave 2
Day 60	UC1 on 5 to 10 repositories; UC2 and UC3 at L1; acceptance ≥ 80%; zero false "done"	Approve L2 for patch-level dependency class
Day 90	UC4 and UC6 pilots running; UC5 CBOM discovery complete for pilot scope; improvement loop operating weekly	Scale plan, staffing and budget for next two quarters

Table 11. Milestones and sponsor decisions.

Appendices

A Loop readiness checklist

- ☐ Use case passes verifiable, repetitive, reversible tests
- ☐ Charter approved with measurable done-definition
- ☐ Non-goals and protected paths listed
- ☐ Budget: iterations, tokens, time, cost
- ☐ Escalation owner and queue named
- ☐ AGENTS.md and at least one skill committed
- ☐ Scanners pinned, checksum-verified, gate scripts tested
- ☐ Diff guard blocks test deletion and suppressions
- ☐ Checker agent uses fresh context and read-only tools
- ☐ Ephemeral sandbox and per-run worktree
- ☐ Egress allowlist enforced and tested
- ☐ OIDC, fine-grained single-repo bot token
- ☐ All CI actions pinned to full commit SHAs
- ☐ Telemetry and cost visible per run
- ☐ Golden set and adversarial cases passed
- ☐ Kill switch documented and tested

B Tool reference matrix

Capability	Open-source defaults	Notes
Dependency scanning	OSV-Scanner, Trivy, Grype, OWASP dep-scan	Prefer lockfile and SBOM input
SBOM / CBOM	Syft, cdxgen (SBOM and CBOM)	CycloneDX format throughout
Automated bumps	Renovate, Dependabot, `osv-scanner fix`	Use before the agent; agent handles leftovers
Prioritisation data	CISA KEV, FIRST EPSS, OSV.dev	Combine with reachability
SAST	Semgrep CE, CodeQL, Bandit, gosec, SpotBugs + FindSecBugs	SARIF output for triage
IaC and containers	Checkov, KICS, Trivy, Hadolint, kube-linter, Conftest/OPA	Plan-based checks for Terraform
Secrets	Gitleaks, TruffleHog, detect-secrets	Redaction always on
Crypto / TLS	cdxgen CBOM, sslyze, testssl.sh, Semgrep crypto rules	Map to FIPS 203/204/205
Finding aggregation	DefectDojo	Feeds the triage loop
Provenance	SLSA tooling, Sigstore (cosign), `npm audit signatures`	Verify inputs and sign outputs
Sandboxing	Rootless Podman/Docker, gVisor, Firecracker, dev containers	Ephemeral per run
Observability	OpenTelemetry, Langfuse, Arize Phoenix	GenAI semantic conventions

C Glossary

Term	Meaning
Agent	A model that can decide on and execute actions through tools (edit files, run commands, call APIs) to pursue a goal.
Agentic loop	A repeating cycle in which an agent acts, receives feedback and decides its next move until a checkable condition is met.
AGENTS.md	A repository file containing standing instructions that coding agents read before working.
Autonomy level	The policy-defined extent to which a loop may act without human approval (L0 to L4 in this guide).
CBOM	Cryptography Bill of Materials: an inventory of cryptographic assets (algorithms, keys, certificates, protocols) in CycloneDX format.
Charter	The versioned contract for a loop: goal, done-definition, non-goals, autonomy, budget, escalation.
Checker agent	An independent agent that reviews another agent's output with fresh context.
Context window	The amount of text a model can consider at once; long loops must manage it deliberately.
Diff guard	A deterministic check that blocks forbidden changes such as edits to protected paths or test deletion.
EPSS / KEV	Exploit Prediction Scoring System (probability of exploitation) and CISA's Known Exploited Vulnerabilities catalogue.
Harness	The tools, constraints and feedback mechanisms around a single agent.
Hook	A script that runs automatically at an agent lifecycle event (for example after a file edit) and can return feedback.
MCP	Model Context Protocol: an open protocol for connecting agents to tools and data sources.
Reward hacking	An agent satisfying the letter of a check without achieving its intent (for example deleting a failing test).
SARIF	Static Analysis Results Interchange Format: a standard JSON format for scanner findings.
Skill	A folder with a `SKILL.md` file describing how to do a specific job, loaded by agents when relevant.
SLSA	Supply-chain Levels for Software Artifacts: a framework for build integrity and provenance.
Vibe coding	Building software primarily by describing intent to an AI coding agent in natural language.
Worktree	A separate Git working directory on its own branch, allowing parallel agents to work without collisions.

D References

1. Osmani, A. "Loop Engineering." June 2026. addyosmani.com/blog/loop-engineering
2. IBM. "What is loop engineering?" IBM Think, July 2026. ibm.com/think/topics/loop-engineering
3. OWASP GenAI Security Project. "OWASP Top 10 for Agentic Applications for 2026." December 2025. genai.owasp.org

4. Yao, S. et al. "ReAct: Synergizing Reasoning and Acting in Language Models." 2022.
5. Shinn, N. et al. "Reflexion: Language Agents with Verbal Reinforcement Learning." 2023.
6. Thoughtworks Technology Radar. "Ralph loop" (technique, Assess). April 2026.
7. Google Open Source Security Team. "Announcing OSV-Scanner V2: Vulnerability scanner and remediation tool for open source." March 2025.
8. Aqua Security / GitHub Advisory GHSA-69fq-xp46-6×23 (CVE-2026-33634). Trivy and trivy-action supply-chain compromise. March 2026.
9. NIST. FIPS 203 (ML-KEM), FIPS 204 (ML-DSA), FIPS 205 (SLH-DSA). August 2024.
10. NIST. IR 8547, "Transition to Post-Quantum Cryptography Standards."
11. OWASP CycloneDX. cdxgen and the CycloneDX CBOM specification.
12. OpenSSF. SLSA specification; OpenTelemetry semantic conventions for generative AI.

ABOUT THE AUTHOR

About the Author

VSC

Dr. Viswanath S Chirravuri

Founder, AI AppSec Academy · D.Eng · GSE #335 · SANS SEC545 and SEC546 Instructor

Dr. Viswanath S Chirravuri (Vis) is a cybersecurity practitioner and educator with more than 20 years in the field and over 35 industry certifications, including the GIAC Security Expert (GSE #335). He teaches SANS SEC545 and SEC546, has spoken at RSA Conference in 2024 and 2026, and serves as a CompTIA subject-matter expert. He holds a Doctor of Engineering in Cybersecurity Analytics from The George Washington University, where his praxis research addressed quantitative detection of software artefact tampering using signed SLSA provenance metadata, combining machine learning with digital-signature verification.

He created AI AppSec Academy as a free knowledge hub covering AI for application security, securing AI systems and vendor AI security, with guides, cheat sheets, presentations and open-source tools.

LinkedIn: https://www.linkedin.com/in/vchirrav/
Website: https://aiappsecacademy.com

FEEDBACK AND CONTRIBUTIONS

This guide is a living document. Feedback on the reference model, playbooks or listings is welcome through LinkedIn or aiappsecacademy.com. Please include the section number, the proposed change and the evidence or incident that motivates it.

© 2026 Dr. Viswanath S Chirravuri. Published by AI AppSec Academy (aiappsecacademy.com). The views expressed are the author's own and do not represent those of any employer. Product and tool names mentioned in this document are the property of their respective owners and are cited for identification only; their mention does not imply endorsement.