



RIDGE SECURITY

Close the Loop: Find, Fix, and Prove It

Auto-Remediation with RidgeGen™
and Google CodeMender

Architecture, validation evidence, and the productized remediation workflow

TECHNICAL WHITE PAPER

EXECUTIVE SUMMARY

Security teams do not lack vulnerability reports. They lack a dependable way to turn a proven risk into a verified fix, especially when more application code is written by AI assistants that optimize for working features, not attested security.

This paper describes a customer-facing auto-remediation process. RidgeGen finds and validates exploitable risk on the running application. Google CodeMender generates a targeted source patch from that evidence. RidgeGen re-attacks the patched system to confirm the hole is closed. Security and engineering share one find → fix → re-verify loop instead of another ticket queue.

Ridge Security has run this loop end-to-end in controlled validations across real and deliberately difficult targets. High-severity findings on a production-style web application fell from eight to zero across four remediation rounds. Two independently AI-built applications of the same product specification both reached zero critical and zero high findings after the loop. On a public benchmark, a textbook JWT fix that passed review still failed at runtime, and only re-attack revealed why.

Ridge is now productizing the remaining manual steps so customers can run the same pattern as an enabled remediation workflow: validated finding in, reviewable patch out, runtime attestation before promotion, with a human gate retained for merge and release.



Results from Ridge Security internal validation runs, 2026. They show what the loop does in practice; they are not a guarantee of identical counts on every customer application.

01 THE PROBLEM BUYERS ACTUALLY HAVE

AI ships code faster than teams can attest to it

AI coding assistants can generate routes, data layers, UIs, and auth scaffolding in a matter of hours, and that code often runs. A login screen, role tags, or middleware in the repo does not mean access is enforced across every data path. In our RidgeGen validations, we see apps with complete auth components in the codebase and UI while their backend APIs sit open without requiring credentials.

Findings without proof create work; findings without fixes create backlog

Traditional scanners optimize for coverage and tolerate false positives. Humans triage. That model breaks when an automated fixer is placed downstream: repairing a non-issue changes correct code, adds churn, and can introduce new risk.

Even validated findings stall when the slow half of the cycle — localization, patch writing, regression checking, and re-test — stays manual. Cadence stretches from hours to weeks. By then, the release has already shipped.

“Looks fixed” is not enough

Three signals are commonly treated as proof that a security issue is resolved. Each is incomplete:

- **Code review** reasons about intent in the diff.
- **Fixer confidence** asserts that a change was meant to address the issue.
- **A successful compile and merge** prove the patch is well-formed, not that the exploit fails.

THE PRINCIPLE

A security fix is a runtime property. The only dependable way to know a patch held is to replay the attack against the running system after deployment.

02 TWO CAPABILITIES, ONE CLOSED LOOP

RidgeGen™: runtime validation and attestation

RidgeGen is Ridge Security’s agentic offensive for targets where multi-step exploitation and business-logic risk matter. It reasons through multi-step attack paths, surfaces business-logic and non-CVE risk that signature scanners miss, and promotes findings only when reproducible exploit evidence supports them (Ridge Security, 2026a).

In the remediation loop, RidgeGen plays two roles:

1. **Detect and validate.** Prove the vulnerability on the live target with a reproducible attack path.
2. **Re-attack and attest.** After a patch is deployed to a candidate environment, replay the exploit and report whether the fix held.

RidgeGen is model-agnostic and designed for on-premises deployment within the customer security boundary, with safety postures that keep destructive actions outside default production testing.

Google CodeMender: context-aware code remediation

Google CodeMender is an AI code-security agent, originating in Google DeepMind research and available to Google Cloud customers in preview through Gemini Enterprise Agent Platform. It can find issues in source, verify exploitability in a customer-managed sandbox, and generate tested patches delivered as code diffs for developer review (Popa & Flynn, 2025; Gerstenhaber, 2026).

DeepMind reported that, during early research development, CodeMender-related work upstreamed 72 security fixes to open-source projects, including large codebases. Google Cloud’s preview positioning emphasizes remediation at machine scale while keeping developers in control of commit decisions.

CodeMender can also accept vulnerability input from external security tools (Google Cloud, 2026). That import path is what makes a RidgeGen-seeded remediation loop practical: the fixer receives an exploit-validated seed instead of a static guess.

Why the combination matters

Figure 1 contrasts operating each product alone with the closed find → fix → re-verify loop.

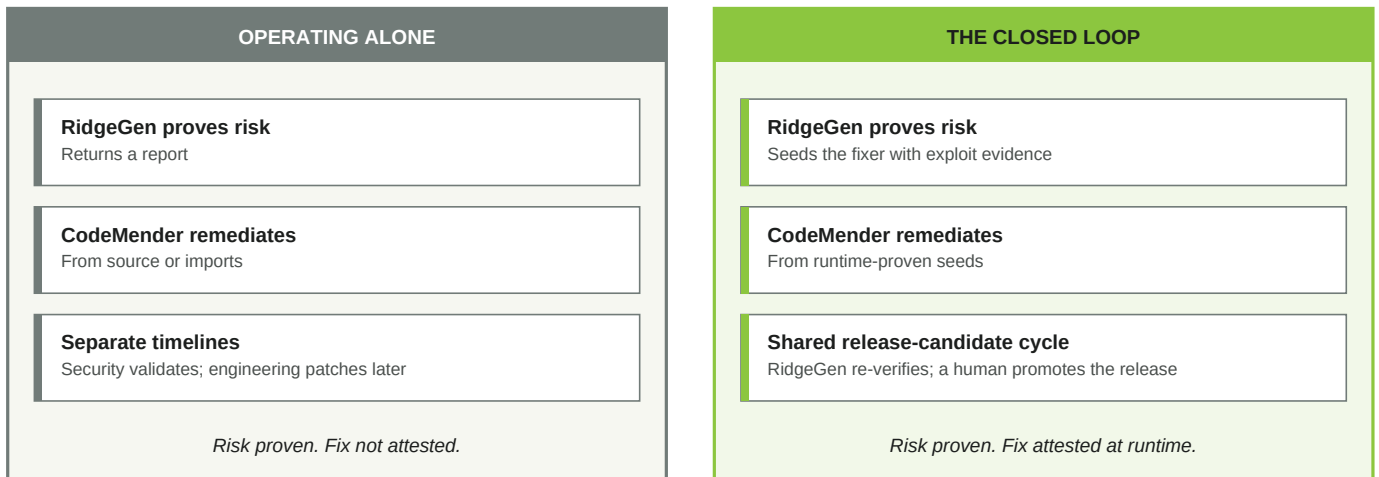


Figure 1 — Operating alone versus the closed loop. The left column is the common buyer pattern today: a validated report on one timeline and a patch on another.

The right column is the loop we productize: the same runtime evidence seeds the fixer, then re-attack decides whether the candidate is ready for a human promotion gate. That difference is why we keep RidgeGen as the attestation authority even when CodeMender, or another authorized fixer, writes the diff.

Precision without attestation tells you what is wrong, not whether you fixed it. Remediation without precision forces triage and unsafe auto-edits. Together, exploit-validated detection and runtime re-attack reduce the chance that auto-remediation edits unverified findings, and they confirm each material patch against the running candidate.

03 HOW THE CUSTOMER WORKFLOW WORKS

The loop is deliberately engine-neutral on the fix side. RidgeGen attests; CodeMender, or another authorized fixer, patches source. Nothing silently merges to production. Figure 2 shows the blue/green candidate path: attack the current site, remediate into a patch candidate, then clear both functional and security gates before a human promotes the release.

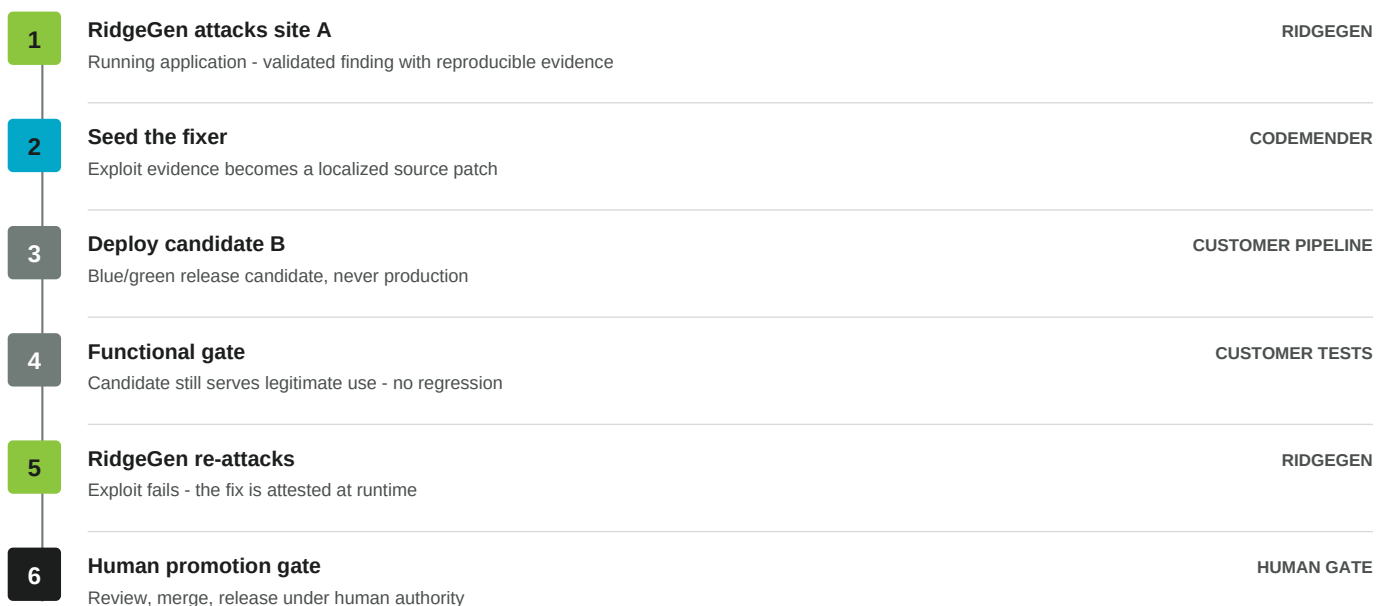


Figure 2 — The find → fix → re-verify customer workflow, with the owner of each step.

Figure 2 is a control path. It does not assign owners or name the artifact each step must have. In customer deployments, those ownership boundaries are where loops stall: security validates, engineering patches, and platform owns the candidate environment, but the handoffs are rarely written down. Table 1 maps each workflow step to an owner and a concrete output so the loop is operable in a real release process.

STEP	OWNER	OUTPUT
Attack the running target	RidgeGen	Validated finding with reproducible exploit evidence
Localize and patch source	CodeMender	Diff aligned to the confirmed attack path
Deploy to a candidate environment	Customer pipeline	Parallel site or release candidate (blue/green style)
Confirm no functional break	Customer tests and smoke checks	Candidate still serves legitimate use
Confirm the hole is closed	RidgeGen	Runtime attestation: exploit blocked
Merge and release	Human gate	Authorized promotion

Table 1 — Workflow owners and outputs.

Safety invariants customers should require

- Never modify production code without authorization.
- Never treat “compiled and committed” as “fixed.”
- Label a change *looks-fixed* / *runtime-unverified* until re-attack succeeds.
- Isolate remediation when possible — one finding or one file per pass — so later patches do not silently undo earlier ones.

04 EVIDENCE FROM VALIDATION RUNS

The following results come from Ridge Security internal validation runs in 2026, including intentional negative results (Ridge Security, 2026b). They show what the loop does in practice. They are not a guarantee of identical counts on every customer application.

4.1 Real application: four rounds to zero high severity

Target: a live AI video-generation web application with an Express/TypeScript backend. Figure 3 tracks HIGH severity, total validated findings, and health score across the four remediation rounds.

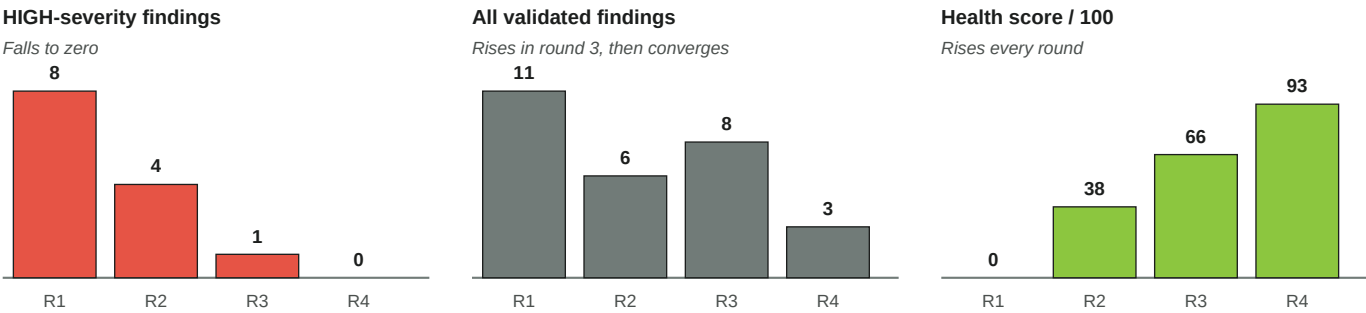


Figure 3 — Four remediation rounds: HIGH severity falls to zero as health score rises.

Buyers still need the operational story behind each round: which classes closed, which gaps were deployment-only, and what remained for a human product decision. Table 2 lists round-by-round counts with a short note on what the loop demonstrated in that pass. Read it as the evidence log behind Figure 3.

ROUND	VALIDATED FINDINGS	HIGH	WHAT THE LOOP SHOWED
Round 1	11	8	SSRF, broken access control, IDOR, provider-key abuse and source disclosure validated, patched and redeployed
Round 2	6	4	Serious exploited issues closed; source still exposed on the running site — a deployment-mode gap invisible to code review alone
Round 3	8	1	Deployment leak closed; subtler issues remained (provider-baseURL SSRF, headers, client-side key exposure)
Round 4	3	0	Remaining high-severity, directly exploitable server issues closed

Table 2 — Round-by-round evidence log.

Across the same sequence, health score moved 0 → 38 → 66 → 93 out of 100. Raw finding totals were intentionally non-monotonic (11 → 6 → 8 → 3): as high-severity holes closed, later rounds surfaced lower-severity issues that then converged. Severity trend is the correct convergence signal.

The three residuals after round 4 — two medium, one low — shared one product design choice: bring-your-own-key browser custody of provider API keys. Closing them would require moving key custody server-side and changing the product experience. That is a product decision for a human gate, not an automatic refactor.

CUSTOMER TAKEAWAY

A correct source patch can still leave a live deployment exposed. Runtime re-attack on the candidate site is what catches that.

4.2 Same product specification, two AI builders

Ridge took one internal product specification — an HR immigration-case tracker holding personal data — and had two different AI coding assistants build it independently. Both initial systems exposed personal data through broken access control. After validated findings and iterative remediation, both reached zero critical and zero high findings.

Two results matter for buyers evaluating AI-generated software:

1. **Author is not a security proxy.** The more functionally complete build initially looked secure — login UI, role badges, middleware present — while data APIs remained open. Adding cosmetic role-based access control without wiring enforcement to routes increased the finding count until middleware was actually applied. Only runtime attack and re-verify established the truth.
2. **Residuals diverge.** After hardening, leftover issues differed by build: CSP and credentialed CORS on one side; login rate-limiting and input-error handling on the other. Same specification, same loop, different blind spots. Only runtime testing tells you which residual set you have.

4.3 Public benchmark: the fix that passed review and failed at runtime

On OWASP Juice Shop v20.2.0, RidgeGen validated 37 findings on a pristine instance, each with a working exploit in that evaluation. The Juice Shop series also shows why class-level comparison matters more than raw ticket totals across dynamic runs. We held the vulnerability class fixed and compared three remediation postures: unpatched, automated repair alone, and assistant-driven repair inside the RidgeGen loop.

VULNERABILITY CLASS	UNPATCHED	AUTOMATED REPAIR ALONE	INSIDE THE RIDGEGEN LOOP
JWT forgery (alg=none, RS256→HS256)	Critical	Still critical	Closed
Login SQL injection	Critical	Partial	Closed
Product-search SQL injection	Critical	Partial	Closed

Table 3 — Juice Shop class outcomes by remediation posture. Under automated repair alone, JWT forgery stayed critical while the SQL injection classes were only partial; inside the loop, those same classes closed.

The JWT row is the clearest single proof point: a review-passing textbook patch still failed until re-attack forced the real fix. Figure 4 walks that path from the textbook change to the runtime-attested repair.

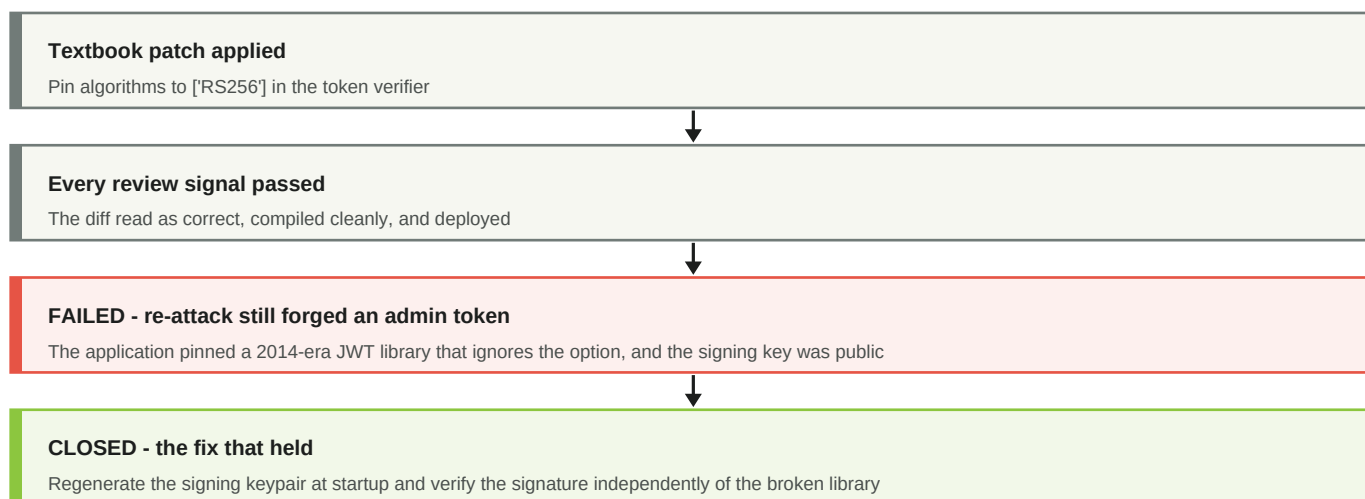


Figure 4 — The JWT looks-fixed failure, from textbook change to attested repair. The diagnosis came from the failed re-attack, not from source review.

Two independent fix approaches applied the textbook change: pin algorithms to ['RS256']. The change read as correct, compiled, and deployed. A forged administrator token still returned the full user table at runtime because the application pinned a 2014-era JWT library that ignores the option, and because the signing key was public. The fix that held required regenerating the signing keypair at startup and verifying the signature independently of the broken library. Failed re-attack drove that diagnosis, not source review alone.

CUSTOMER TAKEAWAY

Fix engines can be interchangeable. An independent runtime attestation authority is what makes their output trustworthy.

05 FROM PROVEN LOOP TO PRODUCTIZED AUTO-REMEDiation

The validation runs prove the loop. The remaining work is operational: less manual coordination between validate, seed, deploy, and re-verify, while humans still authorize merge and release. Figure 5 compares the validation model

we run today with the enabled remediation workflow Ridge is productizing.

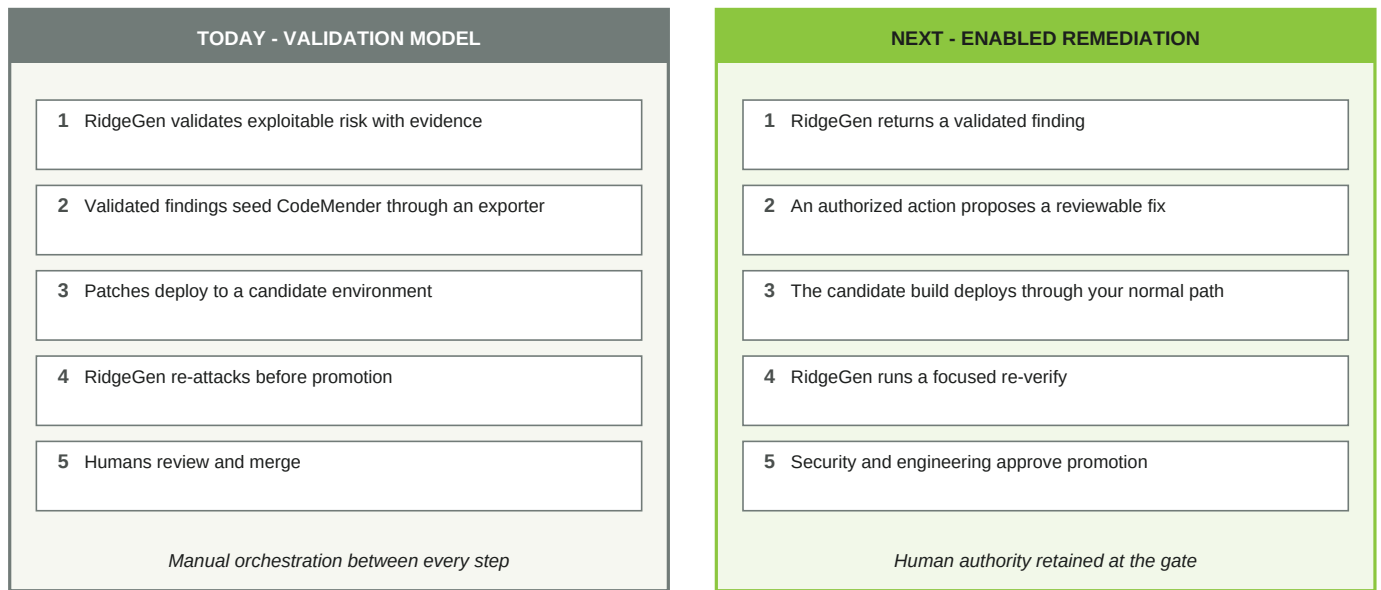


Figure 5 — The same attestation loop with fewer manual handoffs between validate, seed, deploy, and re-verify.

What Ridge is enabling

Ridge is automating the remaining manual orchestration so customers can enable a remediation workflow rather than assembling the loop by hand. This is remediation that runs as soon as a candidate is deployable, with human authority retained. It is not unattended production rewrite. Submission remains customer-authorized. RidgeGen's product role is attestation: decide whether the running system holds, whoever wrote it and whoever patched it.

Different stakeholders need that workflow for different reasons. A CISO cares about evidence cadence. Platform teams care about a release-candidate gate. Engineers care about reviewable diffs tied to attack paths. Table 4 maps each audience to the outcome that group should demand from the workflow.

AUDIENCE	OUTCOME TO DEMAND
CISO and security leadership	Evidence-backed risk reduction with a measurable verify → fix → verify cadence
DevSecOps and platform engineering	A release-candidate security gate fed by exploit-proven seeds, not scanner noise
Application and product engineering	Reviewable diffs tied to concrete attack paths; less triage of theoretical findings
Compliance and audit stakeholders	Reproducible before/after attack evidence for material classes of risk

Table 4 — Audience outcomes for the remediation loop.

06 SCOPE AND CUSTOMER CONTROL

The automation stays inside clear bounds. You retain product and release authority.

- **Runtime validation scope.** RidgeGen verifies exploitable risk on the running application with reproducible evidence. Your team still owns business logic and feature requirements.
- **Production versus benchmark pacing.** Multi-layer synthetic targets such as Juice Shop are useful stress tests. Common vulnerabilities on production apps often close in fewer remediation rounds.

- **Class-based reporting.** We evaluate dynamic results by vulnerability class and verified proof of concept, not by raw ticket volume alone.
- **Architectural decisions stay with you.** Choices such as key-management strategy or UX-driven security policy remain human decisions. Automated patches should not silently rewrite product design.
- **Human promotion gate.** Discovery, patch proposal, and re-attack can be automated. Merge and release stay with your engineering and security owners.

07 WHAT TO DO NEXT

If your organization is already generating or heavily assisting application code with AI, treat security attestation as a release property:

1. Validate on the running system with exploit evidence, not confidence scores alone.
2. Remediate from proven seeds so automated fixing does not amplify false positives.
3. Re-attack after every material patch before you call the issue closed.
4. Enable the RidgeGen and CodeMender remediation path when you are ready to replace weeks of manual localization and re-test with an authorized, repeatable workflow.

RidgeGen's role in that practice stays fixed: the precise, exploit-grounded, engine-neutral attestation authority, independent of whoever writes the fix. CodeMender supplies context-aware remediation at machine scale. Together they run the find → fix → re-verify loop end to end.

REFERENCES

Gerstenhaber, M. (2026, July 21). *Now in preview: Find and fix software vulnerabilities with CodeMender*. Google Cloud.
<https://cloud.google.com/blog/products/identity-security/find-and-fix-software-vulnerabilities-with-codemender>

Google Cloud. (2026). *CodeMender overview*. Gemini Enterprise Agent Platform documentation.
<https://docs.cloud.google.com/gemini-enterprise-agent-platform/codemender>

Popa, R. A., & Flynn, F. (2025, October 6). *Introducing CodeMender: An AI agent for code security*. Google DeepMind.
<https://deepmind.google/blog/introducing-codemender-an-ai-agent-for-code-security/>

Ridge Security. (2026a). *RidgeGen*. <https://ridgesecurity.ai/ridgegen/>

Ridge Security. (2026b). *RidgeGen and CodeMender validation runs* (Internal technical reports). Ridge Security.

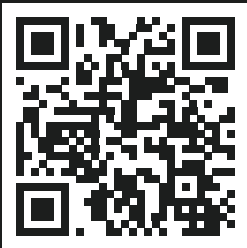


THE PRACTICE

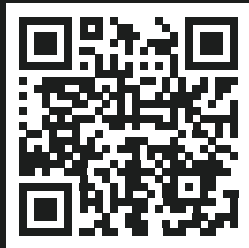
**Validate on the running system.
Remediate from proven seeds.
Re-attack before you call it closed.**

See how the loop runs against your own application.

<https://ridgesecurity.ai>



LINKEDIN
Ridge Security



YOUTUBE
/ridgesecurity



INSTAGRAM
@ridgesecurity



X
@ridgesecurityai