AI SAST: Code Security for the Agentic SDLC

How Endor Labs AI SAST combines program analysis, agentic reasoning, and deterministic engineering to find and fix the flaws that matter in AI-written code.

Sarah Johnson

ENDOR LABS

Table of Contents

I. Introduction: Securing the Agentic SDLC

AI writes a large and growing share of production code, and that code carries the same vulnerability patterns as everything before it, often faster than human review can keep up. Almost half of companies now have codebases that are at least 50% AI-generated, up from 20% at the start of 2025. And in benchmark testing, only 10.5% of an AI coding agent's solutions came back secure, even when 61% were functionally correct. That means more code reaches production with vulnerabilities, leaked secrets, and unsafe dependencies baked in than any human review process was built to catch.

Securing the agentic SDLC has three main pieces: securing the code these agents produce, securing the software supply chain they draw from (packages, models, etc.), and securing the harness they hook into (hooks, skills, MCP servers). This paper is about the first part, and specifically about the check that's supposed to catch insecure code before it ships. Today that check is usually a static analysis tool built to match patterns. Still, these traditional SAST tools flag a lot of false positives (69% for C/C++ tools in NIST's 2018 study) and miss the bug classes that do the most damage: broken access control, insecure design, and business logic flaws.

Endor Labs AI SAST is built on code context, program analysis, and agentic reasoning instead of an LLM bolted onto a pattern-matching engine, so it reads, traces, and reasons about code the way a security engineer would, and triages away the noise before a finding reaches a developer. You can then hand off findings to pre-built Endor Labs remediation agents that propose the fix, so the same platform that catches an issue in the IDE follows it through PR review and into CI/CD.

We're not the only ones taking an AI approach to static analysis, and that's part of why this whitepaper exists: "AI SAST" now covers pretty different things depending on who's saying it. We'll cover how the analysis pipeline works, how the system controls for LLM non-determinism, and how it holds up against traditional SAST tools and frontier models on a ground-truth benchmark.

II. Why Existing Approaches Fall Short

In our benchmark results, we found that neither Claude nor Codex was the strongest at finding flaws, and both missed many of the hard-to-find ones (though we didn't have access to Mythos-class models for these results). The reason comes down to how the two common approaches work, and each fails in the opposite direction.

Pattern-based scanners are fast, deterministic, and reliable on well-understood bug classes like injection and hardcoded secrets. But they match known-bad patterns across the whole codebase without reasoning about how the application behaves, so they flag what code resembles, not what code does. A pattern matcher cannot tell whether user-controlled data actually reaches a vulnerable sink, or whether a sanitizer two files away already neutralizes the issue. So it buries teams in noise while staying blind to the vulnerabilities that live in application behavior rather than in any single line. Broken access control is OWASP's #1 application security risk, and 100% of applications OWASP tested had some form of it. Rule-based SAST has historically struggled to detect these types of flaws because it depends on application-specific authorization logic rather than fixed patterns.

Frontier models fail the other way. LLM agents like Claude and Codex are flexible enough to surface issues no rule anticipated, but a model only reasons about the code it actually examines. When pointed at a repository on its own, LLMs only reach a narrow slice. That gap widens as repositories grow: on a large enterprise Java codebase, the security-relevant code Claude and Codex examined dropped below 10%.

Closing that gap means giving the reasoning engine what a human security engineer would demand before an audit: a complete, structured picture of how the application fits together and how data moves through it. Endor Labs AI SAST does exactly that. Every function is triaged to judge whether it's worth a closer look, and only the interesting ones get in-depth AI analysis. The result is the breadth of traditional SAST and the flexibility of the agents, without the blind spot of either.

III. How Endor Labs AI SAST Works

AI SAST runs on AURI, Endor Labs' security harness for agentic development. On top of its proprietary program analysis tools and years of curated vulnerability data, AI SAST orchestrates specialized agents, each with a job a human reviewer would recognize. Each pipeline stage gathers context and hands it forward, so the final classification is informed by everything learned along the way.

Three design decisions make this hold up on large codebases:

1. The agents reason over a structured, queryable representation of your codebase rather than raw text, so the system never dumps entire repositories into an LLM. That keeps coverage complete and costs predictable.
2. Infrastructure context is read before any security analysis runs, so findings are scored against how your application is actually deployed.
3. Model capability is matched to task difficulty. Inexpensive models run detection over pre-built context, a stronger model validates candidate findings, and the most expensive reasoning is reserved for generating evidence and fixes on the findings that get confirmed.

This is also why the architecture beats frontier models on their own turf. A model pays for every token it reads. Turn one loose on a whole repository, and it reads only a sample, and whatever it never reads, it never finds. Program analysis pre-computes what actually needs analyzing, so coverage competes with neither cost nor speed: handed deterministic context from Endor Labs, an agent completed the same security tasks with the same model 2.8x faster and with 91.7% fewer tokens.

ENDOR LABS

ENDOR LABS AI SAST

Build a knowledge base

Build code context graph

- Data flows
- Caller/callee
- Cross-file/repo/service

Index & segment the code

- Embeddings
- Mantic index -> Semantic index
- File/function hashesse

Add framework & infra context

- Kubernetes
- Dockerfile
- Terraform

Results

A queryable model of your codebase and how it's actually deployed. Built once; PR/diff scans analyze what changed.

- Maximizes coverage
- Minimizes false negatives
- Context-aware prioritization

Find what's real

Run the security analysis

- Detection agents
- Source → sink
- Multi-repo

Triage & prove exploitability

- True/false/unknown
- Exploit payload
- Triage agents

Classify & score

- CWE label
- Severity
- App context

Results

Ranked, validated findings from a deterministic knowledge base of your codebase, analyzed by agents.

- Minimizes false positives
- Validates every finding
- Deterministic, agent-augmented

Ship the fix

- PR review
- MCP server
- IDE / CLI flows

Results

- Delivered where code is writen
- Each finding with evidence and a fix
- Automated remediation with AURI agents

Context handed forward

Step 1: Build the code context graph

Before any agent runs, Endor Labs' program analysis builds a code context graph of your application: caller/callee relationships, data flows, and the connections that cross files, services, and repositories. Endor Labs wrote the analyzers that produce it, with language-specific analysis tuned per ecosystem over years of software composition analysis work. No LLM takes part in building the graph. The agents downstream query it rather than infer it, which is why the same commit yields the same call paths every time, and why the evidence behind a finding holds up in an audit.

This is the same call-graph and reachability infrastructure that has powered Endor Labs' software composition analysis for years, now pointed at first-party code.

Step 2: Index and segment the code

In parallel, Endor Labs builds a semantically searchable index of the codebase, using file hashes, function hashes, and embeddings that capture intent. Rename a method from getDog() to getCanine(), and the index recognizes the same logic; change what it does and the index registers new behavior. Indexing targets maximum coverage, which is what limits false negatives: a finding can only be missed if the code was never examined.

The first scan builds this index and is the longest. Subsequent scans are incremental, analyzing what changed rather than re-deriving the graph.

Step 3: Add framework and infrastructure context

Before vulnerability analysis runs, the system reads how and where the code runs: Dockerfiles, Kubernetes manifests, Terraform, Helm charts, CI configuration, and the frameworks in use. An internet-exposed service scores higher for the same flaw than an internal CLI tool. And framework protections count: Spring Security, the standard security framework for Java applications, blocks cross-site request forgery (CSRF, an attack that rides a logged-in user's session) by default, so a CSRF flag there is recognized as a false positive instead of reported as risk.

Step 4: Run the security analysis

Detection agents trace user-controlled data from source to sink across multi-file, multi-function architectures, drawing on the call graph, function summaries, and dependency context. Because the graph spans repository boundaries, a flow that starts in one repository and terminates in another (the case single-repo scanners treat as a black box) is followed end to end, which is what keeps the analysis working on sprawling multi-repo estates and complex monorepos.

Step 5: Triage the findings

Triage agents check every raw finding against explicit criteria and classify it as true positive, false positive, or unknown, with the reasoning recorded for audit. Each candidate is re-checked against the actual code: a sanitizer upstream that neutralizes a flagged injection, or a

command-execution call inside a terminal emulator that's the product working as intended. For confirmed findings, triage attaches an attack vector and exploit payload, the specific input or call sequence that reaches the sink, so "this looks exploitable" becomes a reproducible artifact that travels with the finding as evidence.

Step 6: Classify and score

Confirmed findings get a CWE label, a severity scored against your application's context, and the evidence behind them: the exact data flow and the function-level call path from source to sink.

Step 7: Fix the vulnerability

For confirmed findings, AI SAST proposes a code fix: a specific change that resolves the flaw where it lives, written against the same context graph the analysis ran on. Because the fix targets the exact path the earlier steps traced, it addresses the real data flow instead of applying a generic template. A developer reviews and applies it, and the evidence from Step 6 travels with the fix, so whoever signs off can see what's being changed and why.

Coverage (steps 1 through 3) feeds recall. Triage (step 5) protects precision. The code context graph is what lets the models reason about behavior instead of surface patterns. That combination is the architecture behind the benchmark results in Section VI.

IV. From Fix to Shipped

A fix only reduces risk if a developer sees it, trusts it, and ships it. Endor Labs delivers the fix into the way developers already work, and scales that across an organization.

1. **Findings surface where code is written.** They show up on PRs during code review, and through baseline and incremental scans in the IDE or CLI. For AI coding agents, an MCP server, hooks, and skills expose the same findings and evidence inside the agent loop.
2. **Findings arrive fix-ready.** Each finding pairs its evidence with a suggested fix, so a developer can verify and resolve in minutes instead of researching for hours, even for findings that span multiple files and layers.
3. **Pre-configured agents automate remediation.** The AURI Agent Hub is a catalog of open-source, pre-configured security agents that use Endor Labs' API and contextual data to autonomously handle tasks like AI SAST triage, remediation, and opening up PRs.
4. **Policy scales your workflow.** AI SAST runs on Endor Labs' API-first, Rego-based policy engine: define a policy once and enforce it across repositories, teams, CI/CD stages, and any coding agent your developers adopt. Block PRs only on high-confidence critical findings, suppress a known false-positive class, route by team. Teams can also provide their own context, from design principles to project-specific guidance, scoped to a single project, a namespace, or the entire tenant.

V. Engineering Determinism into an AI System

The most common objection to AI-powered static analysis is that LLM reasoning is probabilistic. Findings can drift across line numbers, shift between sibling CWEs, or vanish between runs. Endor Labs enforces determinism in layers.

Caching comes first: on a main branch at the same commit, a rerun returns cached results and no LLM is invoked at all. The model runs only on a change, an error, or a low-confidence result, and PR scans evaluate the diff against the indexed baseline rather than rescanning from scratch. Findings anchor to file path and function rather than line numbers, with function-level hashes and a uniqueness key, so a vulnerability stays the same finding even as the code around it moves. And model output is normalized on the way out: sibling CWEs collapse into families, line proximity is tolerated, prior results serve as anchors, and temperature is tuned near zero for tasks that must be repeatable.

The result: the depth of AI reasoning, with output you can reproduce and defend.

VI. How It Compares: Benchmarking Against SAST and Frontier Models

We benchmarked Endor Labs AI SAST against four traditional SAST tools (Semgrep OSS, OpenGrep, Bearer, and CodeQL) and two frontier models prompted to find vulnerabilities directly (Claude and Codex), across 8 public projects with a hand-verified ground truth.

Tool	TP	FP	FN
Endor Labs AI SAST	**192**	**192**	**249**
Semgrep OSS	80	464	345
OpenGrep	87	478	338
Bearer	74	132	351
CodeQL	84	53	314
Claude (raw LLM)	74	29	351
Codex (raw LLM)	55	9	370

TP = true positives, FP = false positives, FN = false negatives.

Endor Labs AI SAST found 192 real vulnerabilities, more than double any other tool in the test, and the most vulnerabilities no other tool could find: 63 of its findings were caught by nothing else, versus a next-highest of 18 (Claude Opus 4.7). Coverage is why. It detected 64 of the 106 CWE types in the benchmark, the widest range of any tool and nearly double the next-highest of 36 (Claude Opus 4.7). That breadth comes from examining more of the codebase and reasoning across more weakness classes than a fixed rule set can match or a frontier model can hold in context at once. That breadth is what becomes raw volume: 2.6x the true positives Claude (Opus 4.7) caught, 3.5x Codex (GPT-5.5), and 2.4x Semgrep OSS. See the blog post for full methodology and the per-tool results table of exactly what each tool caught.

The results hold up outside the lab, too. Pointed at buffa, Anthropic's Rust protobuf library, AI SAST traced untrusted wire data to an unbounded allocation that became CVE-2026-55407, a

roughly 22x memory-amplification denial of service. That's a zero-day found by following a data flow in a memory-safe language, in heavily reviewed code from a frontier-model lab.

VII. Conclusion

AI coding agents changed the question application security has to answer. It is no longer "how do we scan the code humans wrote last sprint" but "how do we secure the code agents are writing right now, and the supply chain they write it with."

Answering it takes a control that sits outside the agents it checks. One whose findings hold up in an audit because they come from program analysis, not a second opinion from a model. One wired into the development loop tightly enough that engineering never has to choose between shipping and security.

That is what Endor Labs AI SAST is built to be: the most real vulnerabilities found in our ground-truth benchmark, noise removed in triage before it reaches a developer, evidence behind every finding, and policy-driven enforcement across every agent, repository, and pipeline stage, on the reachability engine that already cut SCA noise by 92%. And it does it in about a minute per pull request, fast enough to run on every PR, not once a release, so shipping and security stop being a trade-off.

To see how Endor Labs AI SAST performs on your own codebase, request a demo at endorlabs.com/demo-request.