AI SAST

Combining Agents, Program Analysis, and Rules for High-Confidence Code Security

Robert Haynes
Principal Technical Marketing Engineer

ENDOR LABS

© 2025 Endor Labs.
All rights reserved.

Table of Contents

© 2025 Endor Labs.
All rights reserved.

I. Introduction

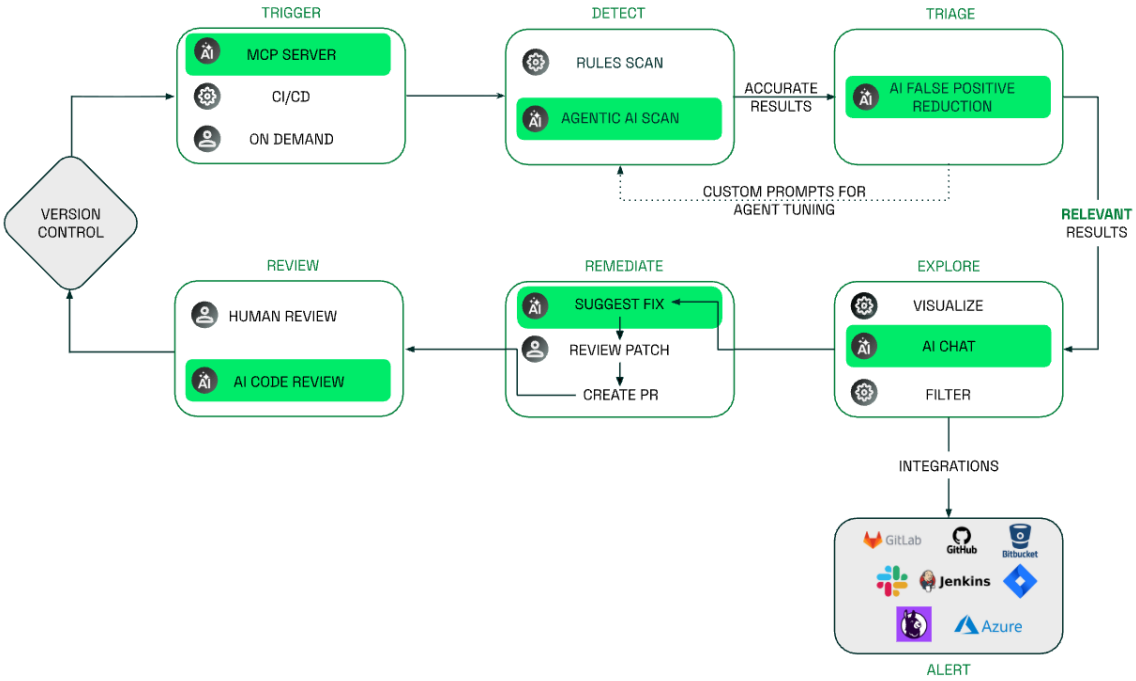
For an industry perspective, background, and value proposition of Endor Labs' AI-native static application security testing (SAST) tool, please read the excellent announcement blog. There you'll find an overview, along with compelling reasons to choose Endor Labs as your security testing and remediation partner.

If, however, you're interested in getting more details on the 'how', either because you've been convinced by the 'why' or because you don't believe anything a vendor says without the technical details (my brothers and sisters in arms!), then please read on.

As a recap, Endor Labs is launching a significantly upgraded AI SAST engine. It combines accurate and tested managed rules with AI detection and triage to deliver precise, high-confidence vulnerability findings prioritized by real risk. This multi-stage approach cuts through the noise, providing AppSec teams and developers with a clear path to action.

In this whitepaper, we'll aim to dive deeper into each component, explain what's happening, and why it works in sufficient detail to allow you to decide to take the next steps.

Figure 1: The Endor Labs AI SAST Engine



© 2025 Endor Labs.
All rights reserved.

II. An Open Source Engine with a Production-Ready Ruleset

At the heart of a SAST offering is the engine that finds potential flaws in source code. Deterministic (rule-based) checks are excellent at catching specific patterns with near-zero runtime cost. For example, a rule can quickly flag the use of an outdated encryption algorithm or the absence of input validation on a critical API endpoint.

In 2024, Endor Labs, along with other industry organizations, announced OpenGrep, an open-source fork of Semgrep. We set out on a mission to build the most advanced static analysis engine—and to make it fully open source.

OpenGrep is a key component of Endor Labs' AI SAST service. It's the engine that analyzes source code for flaws, but uses an enhanced ruleset maintained by the Endor Labs security research team. Endor Labs' rules leverage our team's experience and the latest research to recognize these red flags in code. But unlike generic linters, the rules are tuned for accuracy in an enterprise codebase setting.

Each rule is tested against real-world code to verify that it finds the intended weakness and minimizes false positives. In effect, the engine's rule layer acts as a high-precision net, instantly capturing obvious security bugs, including many that default open-source rule sets would not catch, without inundating developers with trivial or irrelevant warnings.

In addition to enhanced detection rules, Endor Labs has augmented the rule findings with additional context and remediation information, making findings not only more accurate but also more useful. It also comes with additional benefits for enterprise-scale AI triage, namely, reducing the tokens needed to parse a code base.

Combining the speed and simplicity of OpenGrep with a managed (yet also customer-editable) production-ready ruleset is the best of both worlds, expanding OpenGrep's capabilities. This engine provides the 'raw material' for the eventual list of high-priority flaws. Because no matter how good your ruleset is, finding pieces of code with security flaws is just the start of creating actionable findings.

III. Agentic AI to Refine and Triage

Establishing Reachability: Remove everything that's not exploitable

When asked about the process of sculpting his famous statue of David, Michelangelo reportedly said, "Remove everything that is not David".

Removing (or at least de-prioritizing) the flaw findings that are not exploitable helps reveal the flaws that pose a real threat because their input or output is accessible to an attacker. A SQL statement built using variables supplied by a user is more vulnerable to exploitation, compared to one that accepts, say, a fixed range of flags from another internal function. To identify exploitable vulnerabilities, we must trace all inputs back to their source.

© 2025 Endor Labs.
All rights reserved.

Endor Labs uses AI agents that can analyse the data flow through the code in the tested source file to perform a taint analysis, tracing input from sources to sinks. If a dangerous function or vulnerable API is never reachable via any source of untrusted input, it's likely not exploitable from the outside. In those cases, we can deprioritize or even suppress the finding. Conversely, when there is a clear path from an external entry point all the way to a dangerous operation, we have strong evidence of an exploitable vulnerability. Those are the issues that deserve immediate attention.

Let's take a real-world example: Suppose a rule flags the usage of a vulnerable encryption function in two different places in the code. One occurrence is in a utility that processes user-uploaded data (i.e., external input); analysis reveals that an attacker could reach this code path with carefully crafted input. The other occurrence is in an offline admin report tool that only ever reads internal data. The analysis shows that no external data can flow into this path. With this knowledge, the engine can mark the first finding as a high-priority, likely exploitable vulnerability, and perhaps lower the priority of the second or tag it as requiring no immediate fix. This **"**reachability" filtering can significantly reduce the volume of issues that security teams must manually triage. In our prior work on dependency security, reachability analysis was able to cut alert noise by over 90%. We are seeing similar noise reduction for code vulnerabilities by focusing on what's actually reachable.

This evidence-based style of static analysis ensures that when we alert you to a vulnerability, we can also show you the path an attacker could take to exploit it, which builds developer confidence in the finding. It's not just "this line of code is dangerous" – it's "here's how this could be exploited in your application's context," turning hypothetical issues into actionable knowledge.

Triage: Contextual Analysis and Continuous Learning

After applying the first two functions (rules and analysis), we've typically narrowed the field to a set of *potential* vulnerabilities that are likely relevant; static analysis has discovered a security weakness, and it appears exploitable.

But is it really a high-priority finding?

A rules-only scanner might flag a potential vulnerability in a file without understanding what other compensating controls might be in place. Consider a scenario where a rule triggers on a function constructing a SQL query using string concatenation, and the function processes user-supplied data. By itself, that appears to be an exploitable SQL injection flaw. But what if earlier in the call flow, a utility class sanitizes all the inputs to that function? A file-limited SAST tool wouldn't know about that sanitization and would report a vulnerability, a classic false positive. This lack of cross-file awareness is a significant reason why legacy SAST alerts so often waste developers' time. It's also why simply adding more and more rules can backfire; more patterns yield more findings, but without context, many of those findings will be of low

© 2025 Endor Labs.
All rights reserved.

quality. As one study highlights, dealing with a high volume of SAST alerts can consume weeks of effort, much of it spent sifting out noise.

So, the final—and arguably most powerful—layer of analysis is agentic AI-based triage. Endor Labs AI agents review the findings in the context of the entire codebase and any additional metadata, much like a human security expert would, to make final determinations and prioritizations. Even with reachability filtering, some findings still lack the full context necessary to assess their impact or validity. Static analysis may not be aware of specific runtime configurations, the relationships between components, or the nuances of business logic that could mitigate a vulnerability. An *agentic AI* system can interpret the findings in a broader context, essentially adding a layer of reasoning on top of the deterministic analysis.

Taking the example above, an AI agent can be prompted to look at configuration files, related classes, or even documentation to determine if a mitigating control exists. If it finds a separate sanitizing class or a comment indicating the endpoint is for internal use only, it can flag that context. This doesn't necessarily mean the issue is a false positive, but it might downgrade the severity or add a note that exploitation requires breaching another layer of defense.

In essence, the AI triage behaves like a diligent security analyst: correlating information across files and systems to paint a fuller picture of each finding. This agent brings a *holistic perspective*. It knows the list of vulnerabilities from the static analysis, and it can traverse your repository, reading code, comments, and even design docs or tickets if provided, to see what might contextualize those vulnerabilities. The outcome is a further refined set of results, where truly high-risk issues are separated from those that are theoretically vulnerable but effectively mitigated by design.

Another advantage of using an AI layer is the ability to learn from developer feedback continuously. Endor Labs' agentic AI doesn't operate in a vacuum; it learns from the decisions your team makes. When a developer marks a finding as a false positive or adds a suppression comment with a rationale (e.g., *"Not an issue – this input is validated by Service X"*), the AI absorbs that information. The next time it encounters a similar pattern, it will recall the prior context and can automatically suppress or de-prioritize the issue if the same mitigating conditions are present. Over time, this means the engine gets customized to your codebase. Recurring safe patterns will no longer trigger alerts, and new findings will come pre-triaged with knowledge of past resolutions. This feedback loop turns static analysis from a one-way report into an evolving conversation between the tool and your team's expertise.

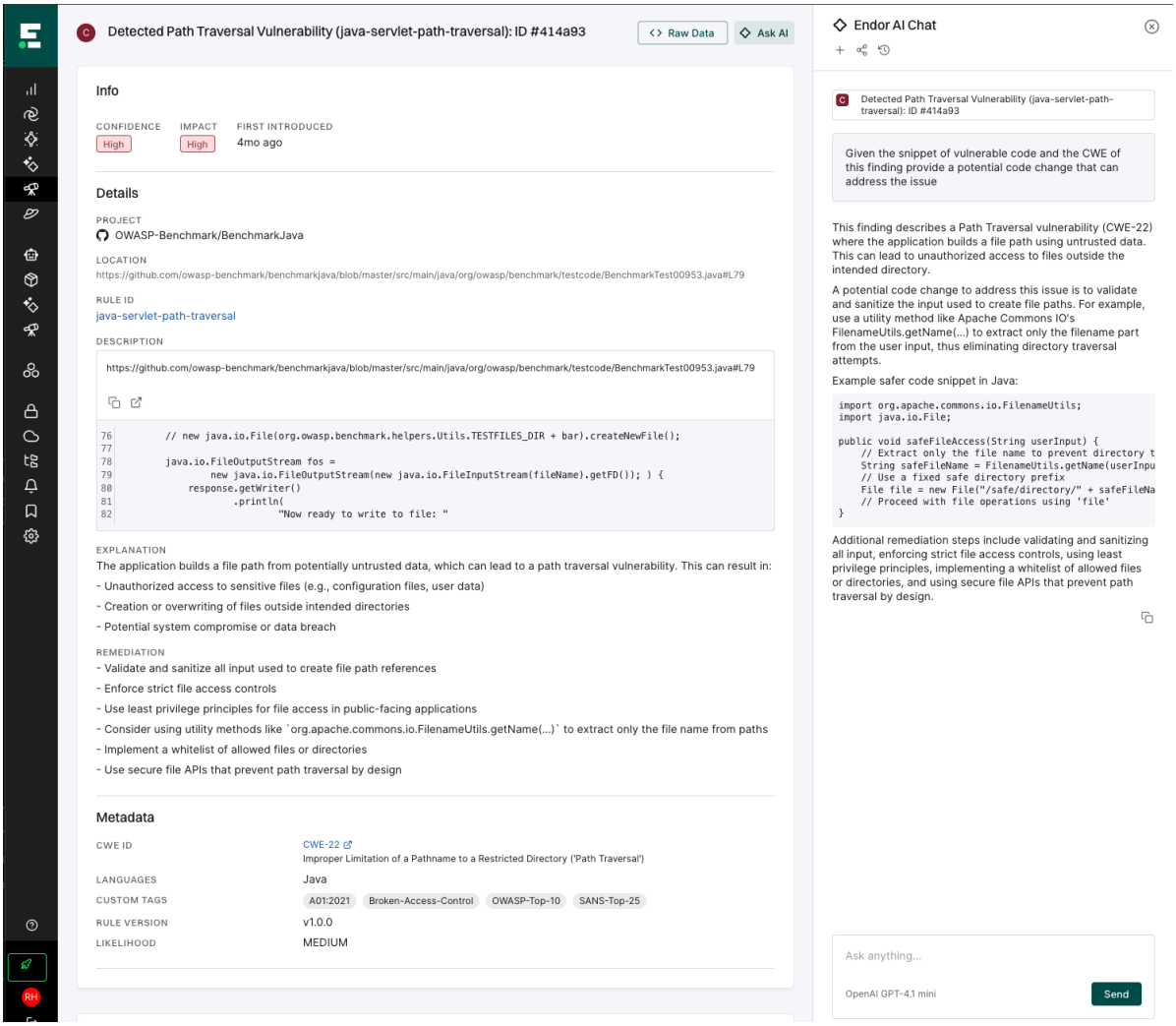
Importantly, we do *not* use this AI agent as a generic code reviewer on every pull request (that's a separate capability, outside the scope of this discussion). Instead, here the AI is focused on triaging static analysis results at scale. It acts as an automated security analyst that can handle thousands of findings, filter out the ones you've deemed acceptable, and escalate the ones that truly need attention. This agent-based triage incorporates policies and past learnings, which brings us to another key point: scaling the triage process through policy.

© 2025 Endor Labs.
All rights reserved.

IV. Efficient Remediation

Finally, all findings from this engine come with actionability built in. Each result includes a description of the issue, the exact evidence of the problem (such as the call trace showing how data flows to a vulnerability), and remediation guidance or code-fix suggestions. Agentic chat, for instance, can provide specific code snippets to resolve the identified issue. The aim is not only to inform you of the problem, but also to provide guidance on where it is located and how to resolve it. For example, a finding for a dangerous deserialization might include a recommendation to switch to a safe serialization library or implement an allowlist of classes. By providing this level of detail, we further reduce the cognitive load on developers. Instead of spending time diagnosing the issue, they can move straight to fixing it correctly. It also helps build trust: when developers see that a vulnerability alert comes with clear proof and guidance, they recognize it as a serious and legitimate concern, not a vague linter gripe.

Figure 2: Resolving security flaws from Endor Labs Agentic Chat



© 2025 Endor Labs.
All rights reserved.

V. Scaling with Policy and Automation

One of the ultimate benefits of this multi-modal SAST approach is how it enables policy-driven, large-scale triage of security findings. High-velocity teams simply do not have the bandwidth to click through each alert in a huge SAST report manually, nor should they have to. Because our engine attaches rich context, risk metadata, and confidence levels to each finding, you can define policies to handle them in bulk. For example, you might set a policy that any finding with low severity or low exploitability is automatically placed in a deferred queue or marked as "informational". Conversely, a policy could say that any critical finding with high confidence (as determined by the AI triage) should immediately create a JIRA ticket or page the on-call security engineer. The multi-stage analysis makes these kinds of distinctions reliable, so you can trust the automation to do the initial sorting.

In practice, this means an AppSec lead can manage by exception rather than poring over every issue. If your organization has compliance requirements, you can map the SAST findings to those and have the system automatically flag any violations of a specific standard. The combination of call graph evidence and AI context also provides the "why" behind each vulnerability, which is crucial for governance. Security managers can, for instance, not only view which vulnerabilities were found, but why the tool believes they are important (e.g., "user data flows into this function which then executes a shell command"). Having this level of explainability and assurance is key to getting developer buy-in as well. Developers are far more likely to fix an issue when the tool shows a clear exploit path or a concrete example of the problem.

Because the entire analysis runs quickly (our architecture parallelizes rule scanning and uses efficient graph algorithms, so scans remain fast even on large codebases), it can be integrated into the development workflow at various stages.

VI. Your Data, Your Control

As more and more engineers adopt AI coding assistants, moving scans from human-centric IDE plugins to AI-native MCP servers means that you can instruct your assistant to run a scan, present filtered findings, and even fix problems using natural language. You can instruct your assistant to run scans when specific events occur (such as updating a manifest file or when you commit code).

When it's time to merge code into your mainline, triggering scans with a more restrictive policy as part of your CI/CD process means security operations teams can be sure that they are in control of the risk levels in deployed applications, and that critical vulnerabilities in first-party code, dependencies, or container images don't get into production.

Of course, every environment is different, and not all applications are created equal; therefore, flexible, template-based policies can be easily configured to suit your security posture needs.

© 2025 Endor Labs.
All rights reserved.

We know trust is earned, especially when AI enters the picture. That's why AI SAST was built with a clear and minimal data usage model, focused on security and privacy from day one.

Here's exactly what we do (and don't) do:

- **Scope-limited analysis**: We only analyze the code snippets with an SAST rule match, with 50 lines before and after the match. We do not scan the full codebase of a project or entire files.
- **No code retention**: The complete code diff is not stored by Endor Labs. To support the dashboard experience, we may store context such as file or function names, or a brief snippet of the relevant code. We provide an option to opt out of displaying the code snippet as well. In that case, we only store and display the code location.
- **Your data is never used for training**: None of your data is used to train AI models, now or in the future.

AI SAST uses **Google Gemini and Azure OpenAI** for the LLM models, and is hosted within a **dedicated Endor Labs VPC** to ensure data is never sent to shared environments or public endpoints.

© 2025 Endor Labs.
All rights reserved.

VII. Conclusion

The upgraded multi-modal SAST engine from Endor Labs represents a new generation of static analysis, one built for the realities of AI-fueled, fast-paced development. By combining managed rules (for broad but precise detection of known issues), reachability (to assess real exploitability across the codebase), and agentic AI-based triage (to apply human-like context analysis and learning), we deliver findings that are both high-signal and high-priority. This multi-layered approach drastically reduces false positives and noisy findings (addressing the top complaint about SAST tools today) while also uncovering complex vulnerabilities that single-mode scanners would overlook. The result for security teams is fewer alerts to chase, and those that do appear come with rich context and evidence. Developers, in turn, receive timely, relevant, and actionable security feedback, allowing them to fix issues quickly without losing momentum.

In summary, a multi-modal SAST engine that “understands” your code through rules, graphs, and AI offers a powerful way to stay ahead of vulnerabilities without being overwhelmed by alerts. Early adopters have reported significant noise reduction and faster remediation times, validating the approach. By cutting alert fatigue and enabling policy-driven automation, we help security teams achieve effective triage at scale. And by providing deep code insight with evidence, we give developers and security engineers the confidence to act decisively on the findings. It’s SAST built for the era of complex, rapidly changing, AI-generated code, offering both speed and accuracy to meet the demands of modern AppSec.

To learn more about how Endor Labs can help you detect security design flaws and architecture changes, request a demo at endorlabs.com/demo-request.