The Token Economics of AI AppSec Agents

A controlled benchmark of AI agents with and without access to deterministic tools for performing common security tasks

Matt Brown, Cris Staicu, Andrew Stiefel

ENDOR LABS

Executive Summary

Security teams are adding AI agents into AppSec workflows, and one architectural question sits underneath every deployment: should the agent derive security facts from the repository, or read precomputed, deterministic evidence?

To measure the impact on token costs and time, we ran the same 34 realistic application security prompts against 13 scopes — 12 open-source projects plus one tenant-wide view — through two isolated agent configurations. One received compact evidence from Endor Labs (findings, reachability, upgrade recommendations, license risk, policy posture). The other worked from local repository inspection and public web research. Same model, same prompts, same step budget; 884 measured responses in total.

We observed that the agent equipped with deterministic security tools:

Used	Completed the benchmark	Required	Kept costs predictable across projects
91.7%	**2.8x**	**77.6%**	**1.8x**
fewer tokens (6.6M vs 79.5M)	(133 vs 370 minutes)	fewer tool calls (1,380 vs 6,165)	vs a 4x swing

None of this should be shocking. Give an agent the answer as context and of course it performs better. What surprised us was the magnitude: 12x fewer tokens. That tells us a lot about the benefits of giving agents access to security tools: most of the tokens an unequipped agent burns go to codebase reconnaissance and context sharing. That's why agent experience — the quality and capability of tools you hand the model — will matter as much as the agent or AI model itself.

Methodology

To see how much tool access and precomputed data actually helps, we ran every prompt through two completely isolated testing tracks using claude-sonnet-4 (both capped at a 30-recursion limit and a 10-step budget):

- **The "With Evidence" Track:** The agent gets a neat, precomputed packet of Endor Labs data including SCA findings, reachability from call-graphs, upgrade paths, license risks, and policies, and uses it to reason out an answer.
- **The "LLM Alone" Track:** The agent starts from scratch. It has to scan the local repo (file reads, greps) and search the web to figure everything out on its own.

We ran against 12 large open-source projects spanning six languages: JavaScript, Python, Java, Go, Rust, and C#. For each language we picked one CMS and one widely-used library or framework (for example: Ghost and socket.io for JavaScript, Umbraco and Dapper for C#, OpenCMS and Thymeleaf for Java), plus a tenant-wide pass across all the codebases at once.

Each project got the same 34 prompts, built around the questions an AppSec team actually asks day to day: what's reachable, what to fix first, what to batch into one upgrade PR, what a CI gate should block, who owns the fix, why a vulnerable transitive dependency is even there, and whether a license is going to be a problem. There were 12 scenarios in all (the full list is in Table 2).

For every run we measured tokens in and out, wall-clock time, and tool calls. We also kept the complete reasoning traces; that's where the case studies later in this paper come from.

Results

Metric	With evidence	LLM alone	Delta
Total tokens (in / out)	**6.28M / 0.31M**	**78.7M / 0.82M**	**91.7% fewer**
Wall time	**2.2 hours**	**6.2 hours**	**64.0% faster**
LLM/tool-loop calls	**1,380**	**6,165**	**77.6% fewer**

Table 1. Aggregate results across 884 measured responses (442 prompts × 2 legs).

Scenario	Tokens saved	Time saved
Transitive root cause	**97.0%**	**72.4%**
Policy compliance	**96.6%**	**68.3%**
Risk acceptance	**96.5%**	**70.3%**
License compatibility	**96.4%**	**73.3%**
Upgrade batching	**95.8%**	**73.6%**
CI gate simulation	**95.7%**	**71.9%**
Owner routing	**94.2%**	**76.4%**
Reachability	**94.0%**	**67.3%**
Blast radius	**93.2%**	**66.0%**
Prioritization	**87.2%**	**50.2%**
CVE explanation	**81.4%**	**52.7%**
Upgrade impact	**72.5%**	**27.8%**

Table 2. Per-scenario aggregates across all 13 scopes, sorted by token savings.

Where time savings come from

The LLM-alone agent averaged ~14 tool calls per question, almost all low-level repository primitives. Across the run it issued 1,985 web searches, 849 grep calls, 783 manifest reads, 689 directory listings, and 547 file reads — hand-rolling software composition analysis from scratch: grepping for version strings, walking package.json files, counting imports. And because an agent re-sends its entire accumulated context at every reasoning step, each of

those repo dumps gets paid for again and again. The token drain isn't the agent thinking; it's the constant re-reading of raw repository data the model had to gather itself.

The evidence-equipped agent answered the same questions with ~3 tool calls per question, and they were high-level, security-aware constructs: a reachability summary, a CVE exposure report, an upgrade impact analysis. One call returns the answer because it isn't reasoning — it's a lookup. The 91.7% token reduction and the 4.5× drop in tool calls are the same phenomenon measured two ways: a few high-level calls against a deterministic, security-aware engine instead of many low-level calls against a raw filesystem.

Token usage scales with codebase size

There is a second-order cost problem hiding in the averages: the LLM-alone agent's price tag is a function of your codebase, not your question. Asked the same 34 questions, it spent 2.7M tokens on a small project (Kotti) and 10.5M on a large one (OpenCMS), a 4x swing driven by how much code it had to crawl and how many findings it had to sift. Most of what the model does in this configuration is reconnaissance: digging through the repository to find relevant facts before it can reason about them.

The evidence-equipped agent answered the identical questions in a 0.33M–0.61M token band (1.8x swing), because lookup costs about the same whether the repo is ten files or ten thousand.

In budgeting terms: across the 13 scopes, the LLM-alone agent's per-scope cost varied with a 37% coefficient of variation, versus 15% with evidence, making the LLM-alone approach more than twice as unpredictable. A security bill that scales with the size of every repository you point it at, and swings 4x depending on which one it lands on, gets worse as codebases grow and alert counts climb.

Why time savings trail token savings

Evidence calls aren't free, and that is exactly why time saved (64%) trails tokens saved (91.7%). A deterministic tool call still does real work: it runs against the actual project, it carries backend latency, and it returns an output the model still has to read. Even with evidence in hand, the agent ingested roughly 14,000 tokens of it per question on average, and far more on the heavy tasks.

As a result, the larger the evidence packet a task produces, the smaller the win, and wall-clock degrades faster than token count. Upgrade impact analysis returned ~52,600 tokens of evidence per question, and its savings fell to 72.5% on tokens and 27.8% on time. CVE explanation, at ~33,000 tokens of evidence, saved 81% of tokens but only 53% of time. Compare lean scenarios like policy compliance, where compact 5–6K outputs translated into 96–97% token savings and roughly 70% time savings.

Case Study 1

The right answer at 4x the price

We asked both agents to build a remediation plan for Flask. Both landed on upgrading Werkzeug, which clears CVE-2024-34069 and two other findings. The difference was what it cost to get there. With evidence, one lookup returned the pinned version, the three CVEs it clears, and the safe upgrade set — and the agent wrote the plan. Without it, the agent rediscovered everything from the repo: it crawled manifests, counted "50+ Werkzeug imports across app.py, cli.py, helpers.py, testing.py," and inferred versions over a dozen reasoning turns, burning 4x the tokens (309K vs 78K) to reach the same conclusion.

Worse, it still got a detail wrong: it guessed Werkzeug 3.1.x when the project pins 2.3.3. Even when the model gets the right idea, it pays a premium to reconstruct what a tool already knows, and "approximately right" on a version number is simply wrong when you are the one cutting the remediation ticket.

Case Study 2

Confidently wrong about reachability, at 22x the cost

The more expensive failure is when the model is confidently wrong. We asked, against Ghost, for the single function-reachable, highest-severity finding, which is exactly the reachability question that determines patch priority. With call-graph evidence, the answer was grounded and calibrated: one medium-severity reachable finding, and zero critical or high findings actually reachable.

Without it, the agent spent 22x the tokens (723K vs 32K, across 45 tool calls and 29 reasoning turns on that one question) and produced a confident, neatly-cited, false answer: it declared an Express XSS (CVE-2024-43796) "function-level reachable" and "High/Critical."

Its evidence? That Express is imported and used throughout the codebase. But import presence is not reachability. With no call graph, the agent substituted "the package is used" for "the vulnerable function is actually reachable," then dressed the guess in tool citations, down to a fabricated version number (4.22.2; the project runs 4.18.2).

Limitations

A few caveats about this data:

- **This is a cost benchmark, not a correctness benchmark.** We deliberately kept answer-quality scoring out of scope for this paper, so the two case studies above are illustrative.
- **The evidence-equipped agent's economics depend on work done upstream.** Call graphs and finding databases are precomputed before they're given to the agent. That work isn't free. What we measured — and what matters for an agent fleet — is marginal cost per question.
- **The win gets thin on evidence-heavy tasks.** Upgrade impact analysis saved only 27.8% of wall time on average, and on the single worst project–scenario pair (Flask, upgrade impact) the time saving fell to 3.9%. Where the tool returns large evidence packets, the advantage narrows.
- **One model, one environment.** Both setups ran claude-sonnet-4 against open-source projects in a staging tenant. Different models, projects, or projects will have different outcomes.

Implications

For teams deploying agents at scale, the practical takeaways are simple. Deterministic evidence is the substrate for agent reasoning, not a competitor to it. Agents are good at synthesis and planning once the facts are in front of them, and expensive when asked to excavate the facts themselves.

Token economics deserve a line in the architecture review, because an agent whose cost scales with repository size and varies 4x between projects is hard to budget at fleet scale. For companies like Endor Labs building these tools, the next round of gains are in agent experience design: right-sizing evidence packets to the question to reduce both time and token costs based on the agent's query.