ENDOR LABS


# Context Engineering for Application Security

Application security scanners catch vulnerabilities after the code is written. But in an AI-driven world, 62% of generated code contains flaws by default. Waiting for the scan is too late.

# Foreword

Application security is approaching a point of structural failure. As the traditional software development lifecycle is being replaced by AI-generated code and agentic workflows, we are no longer securing human-authored logic; we are supervising machine-driven intent. And though AI can dramatically increase developer productivity, it also introduces significant risk because AI code assistants don't write secure code by default.

The OWASP Top 10 serves as a stark acknowledgment of this shift, pivoting in 2025 from simple syntax vulnerabilities to systemic risks such as prompt injection, training data poisoning, and the integrity failures inherent in autonomous AI decision-making. These emerging threats prove that when code is generated at the speed of thought, security can no longer be a reactive checkpoint, but must instead become a contextual guardrail embedded within the AI's generative process.

But the disconnect between code production and security validation is now a chasm. Most application security teams continue to rely on scanners (SAST, SCA, etc.) positioned later in the SDLC, resulting in vulnerabilities being discovered later in the SDLC. While shift-left initiatives abound, aiming to integrate scanning when code is committed, they will fail without a fundamental change in how we view application security.

In this whitepaper, we explore how to implement a mindset change. Once we thought *What if we could catch all insecure code before it is pushed to production?* Today, we need to ask *How can we prevent AI code assistants from writing insecure code before it's even shown to the developer?*

Yes, the solution requires shifting further left (directly into the generative process), but simply moving scanning earlier doesn't solve the problem. Application security must move from static pattern matching to natural-language reasoning. For the modern CISO, this shift demands a transition from "enforcer" to "architect". This whitepaper outlines the roadmap for this transition. It is a guide for building a security posture that is "secure by design, " not through rigid rules, but through intelligent, just-in-time contextual engineering.

# From prompt engineering to context engineering

If you've made it this far, I'm betting you're excited by the idea of getting LLMs to write secure code. And you should be! It could mean the end of litigating vulnerabilities, failing compliance assessments, and measuring program success by CVE counts. Unfortunately, getting LLMs to do your bidding can be challenging. The more complex the task, the more likely the LLM goes rogue.

When I'm teaching people about this problem, I use a silly example of asking your kid to take a shower. Just about every caregiver has had this experience: You tell the child to take a shower, and they come out with dry hair or dirty feet. You ask *Why didn't you wash your hair?* They say *You didn't tell me I had to wash my hair.* 🤷 And that's very much like an LLM. If it doesn't have detailed instructions, it's going to skip steps that you *just know* are required.

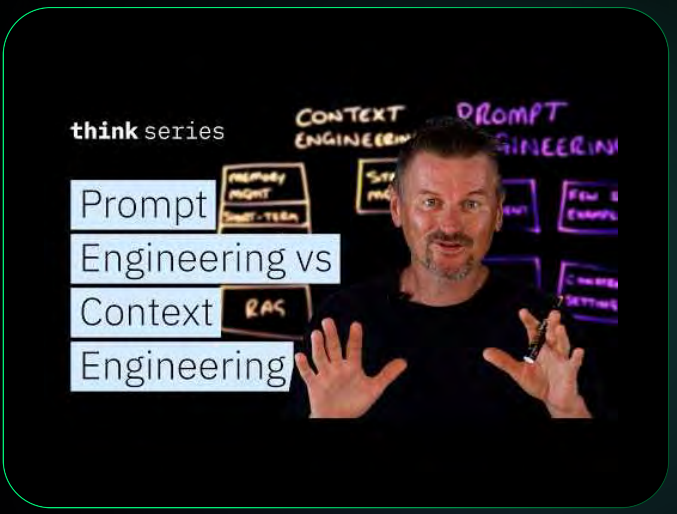
## LLMs are like people: We both get overwhelmed by too much information.

To solve this problem, prompt engineering was born. **Prompt engineering** is the process of crafting the input text (instructions, examples, formatting cues) to steer the LLM's behavior and output. And it works! Kind of. Sometimes. Writing good prompts is a lot of work, and even if you get it right, the LLM can still struggle if you give it too much information.

Why does that happen? Each LLM has a **context window**, which is the maximum amount of information it can process at one time. Think of the context window as the model's short-term memory. In the earlier days of AI coding assistants (circa April 2025), the context window was informed by prompts and prior conversations. But the context window can 'collapse' (hit the limits of its memory) when prompts and memories get too long. For example, models may pay less attention to information located in the middle of a long prompt, even if it's crucial. Even if you provided the LLM with every bit of necessary information via prompts, the chances of hallucination or mistakes are high.

Some smart people realized we needed to take the next step: **context engineering**, which is a broader discipline of assembling everything the LLM sees during inference. This includes user prompts, rules given to the model to guide its behavior, retrieved information (RAG), memory, and tools...all the context needed to write code the way you want it written. Context engineering can be done at the individual level, but it's most effective when combined with an organizational strategy so that each engineer doesn't have to DIY.

I really love how this video from IBM explains the differences between prompt and context engineering:



# AI-generated code isn't secure by default

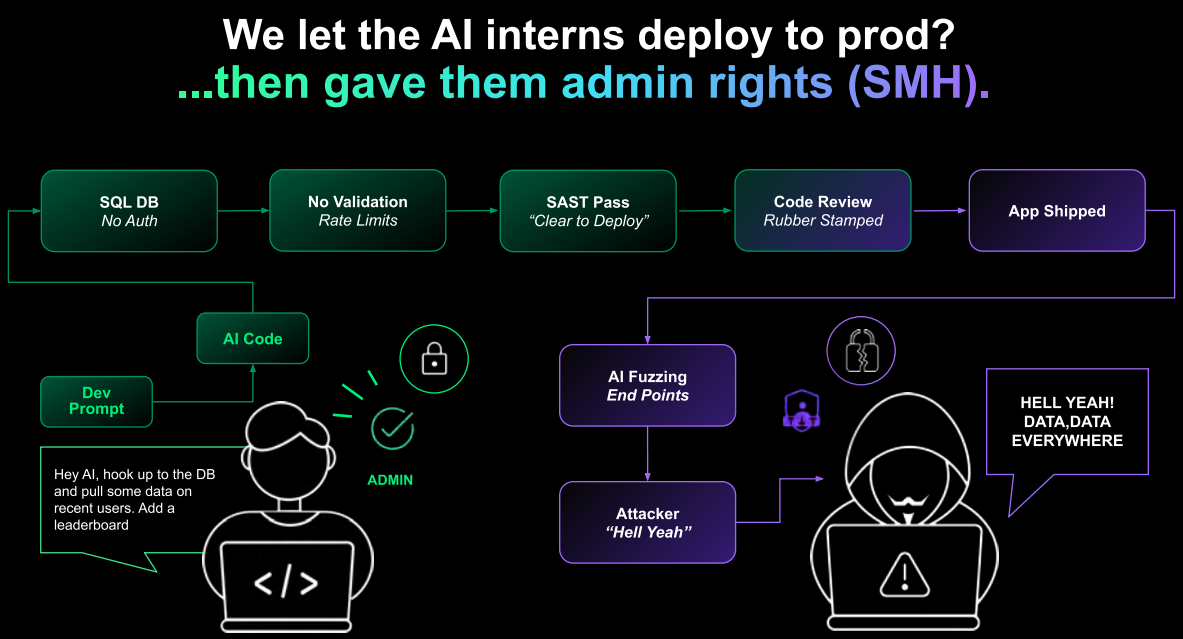
Now that we have some basics out of the way, we're going to look at how context engineering techniques can improve the security of AI-generated code.

*Why is this necessary?* Because 62% of AI-generated code solutions contain design flaws and security vulnerabilities, issues like hardcoded passwords, missing input validation, and a lack of authentication or access controls.[1] In fact, Endor Labs discovered that when LLMs wrote code, about 80% of imported dependencies were hallucinated or contained known vulnerabilities.

Let's look at how a developer might generate insecure code using an AI code assistant.

In this example (see graphic), a developer prompts the assistant to *"Modify my application to hook up to the database and pull some data on recent users. Add a leaderboard."* The LLM delivers, but because the developer didn't specify any security parameters, it doesn't add authorization or rate limiting. Although the code is scanned by a SAST tool before release, missing authorization and rate limiting are business logic flaws, not CWEs, which a traditional SAST can't catch. It continues to code review, where the software engineer is a little swamped or inexperienced and doesn't catch the problem. The code is merged into production...

And along comes a hacker with an AI assistant of their own. Using AI fuzzing, they prompt their LLM to *"Test some edge cases. What if users were admins?"* The LLM delivers. Suddenly, a leaderboard query turns into a full data exfiltration script. Tons of PII breached!



# Using context engineering to improve application security

We can't yet trust LLMs to write secure code by default, so we need to apply context engineering techniques to add guardrails and security context. We'll cover four in this blog:

- Secure prompts (that's right, prompt engineering is still relevant)
- System rules
- External security tools
- Retrieval Augmented Generation (RAG)

# Writing secure prompts

A good prompt is like a design document for the feature you want AI to write. Ideally, your design docs should address security, so it's a logical jump to think about including security in your prompts. And this really does make a substantial difference! For example, one study showed that using a recursive prompting technique (where the LLM audits and improves its own code) resulted in a 77.5% reduction in security weaknesses[3].

Good prompts are also an opportunity to leverage two engineering best practices that are truly the better way to use AI: **Spec-Driven Development (SDD)**, where you create a detailed specification before writing code or tests, and **Test-Driven Development (TDD)**, where you write a failing test first. SDD is beneficial for higher-level design, system-level architecture, and creating clear contracts, while TDD excels at ensuring code-level correctness.

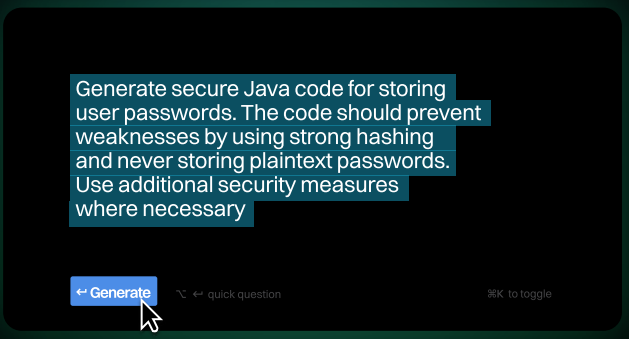
The prompt patterns you can use to achieve SDD and TDD with an AI code assistant are 'design-spec' and 'test-first.

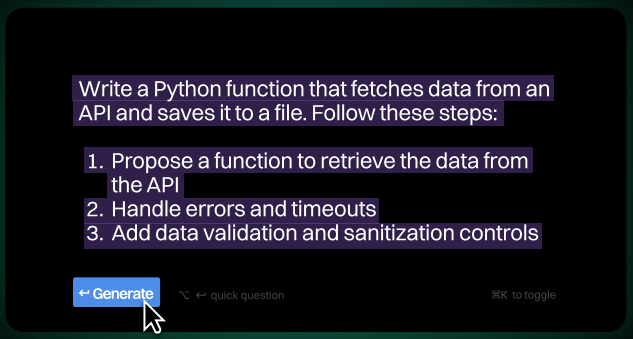
| Prompt Pattern | Description | Best For |
|---|---|---|
| Design-Spec (for SDD) | Secure-by-design code generation during planning or implementation. Provides full context about functionality, environment, and security requirements. | New feature development |
| Test-First Prompt (for TDD) | Emphasize secure functionality through automated security tests. Build tests to validate what the LLM generates, but it's risky since the LLM generates both the code and the tests. | Testing and regression coverage |

To apply these prompting patterns, you can use three different techniques depending on what you need to achieve.

## Priming

Frame the entire conversation before the code generation begins. It's a "single-shot" approach where you tell the LLM up front what to do and which security principles to follow. This is useful for establishing the security requirements from the very start.



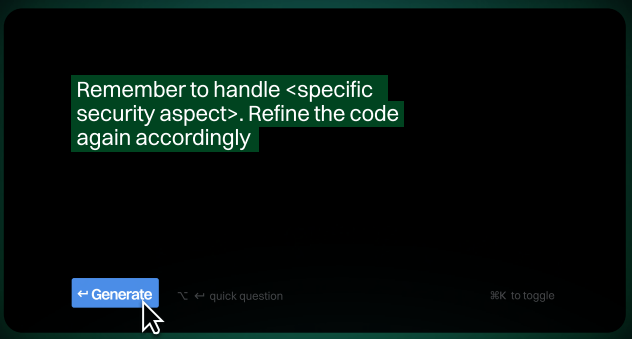




## Decomposition

Solve a hard problem by breaking it into several smaller ones and prompting the LLM for each step sequentially. By breaking the task into steps, you can explicitly inject security requirements, such as "Add data validation and sanitization controls, " into the subtasks.

## Refinement

Iterate on an idea over multiple prompts. After the LLM provides an initial code output, you provide a follow-up prompt to improve or correct it, especially by reminding the LLM to handle a specific security aspect.




Now let's return to that scenario where a developer insecurely prompted the AI code assistant to *"Hook up to the DB and pull some data on recent users. Add a leaderboard."* The resulting code lacked authorization and rate limiting, so we're going to use these prompting techniques and patterns to do a better job.

## Priming (Single-Shot) for Design-Spec Pattern

The Design-Spec pattern provides full context about functionality, environment, and security requirements, and is best for new feature development. Priming the LLM with all security rules in one initial prompt helps ensure the code is secure by design.

**Sample Prompt:**

*"Modify the application to hook up to the database and pull data for a leaderboard of recent users. The generated code must implement authorization checks to ensure only authenticated users can access the data, and must enforce a rate limit of 5 requests per second for this endpoint."*

## Decomposition (Step-by-Step) for Design-Spec Pattern

By breaking the task down, the developer can force the LLM to address security requirements in specific steps, aligning with the Design-Spec pattern. While similar to the priming prompt, this contains more detail and is sent sequentially, with each prompt sent after the LLM has generated the requested code.

**Sample Prompts (Series):**

- "Create a new endpoint function to retrieve recent user data from the SQL DB for a leaderboard."
- "Modify the function to include an authorization control that verifies the user's session and permissions before executing the database query."
- "Add a throttling mechanism or rate-limiting middleware to the endpoint to prevent more than 5 requests per second."
- "Implement the logic to sort and display the data as a leaderboard."

## Refinement (Iterative Correction) for Test-First Pattern

The Test-First pattern emphasizes secure functionality through automated security tests. The refinement technique can be used after an initial generation to ensure security requirements are met, or to generate a test case that validates the security.

**Initial Prompt (Similar to the original, which might yield insecure code)**

*"Modify the application to hook up to the database and pull data for a leaderboard of recent users."*

**Refinement Prompt (To inject missing security)**

*"Review the leaderboard function code and ensure it properly implements authorization. Remember to handle data access by non-admin users. Refine the code accordingly."*

**Refinement Prompt (To generate a test for TDD)**

*"Now, write a security test case that uses AI Fuzzing techniques to try and bypass the access controls on the leaderboard endpoint to check for data exfiltration vulnerabilities."*

## The challenge: developer education

If you've been thinking *"This all sounds great, but it's a lot to remember"*...you're right.

A single person, sufficiently motivated, who uses an AI coding assistant every day, will jump at improving their prompts. But consistent developer education is often a challenge, and even with the best training, there's no guarantee it will stick. You'll probably end up with scaling problems, and as employees come and go, you'll constantly have to reinforce secure prompting habits.

Secure prompts are a great place to start, but don't stop there! Look at the next context engineering practice: system rules.

# System rules

Most (all?) AI code assistants have the concept of a 'rule'. **Rules** are specific instructions/guidelines you give an AI coding tool to control its behavior, ensuring it generates code that follows your project's standards, style (e.g., naming and formatting), security requirements, and logic. The rule (usually a text file) acts as reusable project-specific documentation to guide the LLM beyond the prompts.

Rules are handy for guidelines that should be applied across a repo or even your whole organization. For example, a rule can contain specific security requirements for your application. Like rules, they're usually written in natural language. This Reddit thread has great insight into how an engineering organization implemented SDD using rules.

Let's look at how using rules for context engineering can improve upon "Hook up to the DB and pull some data on recent users. Add a leaderboard."

## Enforce standards (security policies)

This type of rule ensures the generated code adheres to specific, mandatory organizational security policies, preventing the LLM from generating code that ignores common security requirements, like the missing authorization in the scenario.

**Sample rule (mandatory authorization and authentication checks)**

*"All database-interacting functions and API endpoints must include an explicit authorization check that validates the user's role and permissions before executing the database query. The specific method for checking authorization is* `AuthService.checkPermission(User,'READ_RECENT_USERS')`*."*

This rule would have forced the LLM to add the missing authorization check for the leaderboard function, preventing the data exfiltration vulnerability.

## Guide logic (secure implementation)

This rule steers the LLM toward specific secure programming practices, such as proper input handling or secure resource management, addressing the type of business logic flaw that SAST tools miss.

**Sample rule (input validation and sanitization)**

*"When accepting user input for any database query (including parameters for sorting, filtering, or limits), the code must use parameterized queries or an ORM's safe methods. Never concatenate user-provided strings directly into a SQL statement. All inputs must be validated against a pre-defined schema to prevent injection attacks."*

This rule guides the LLM to avoid common SQL injection risks, ensuring the code is not only functional but also securely implemented, even when building a simple query like the one for the leaderboard.

---

## Define context (environmental and system constraints)

This rule provides reusable, non-negotiable information about the deployment environment or system architecture that is critical for security, such as rate limits or secrets management. This addresses the lack of rate limits in the scenario.

**Sample rule (mandatory throttling/rate limiting)**

*"The application uses a centralized API gateway for all public-facing endpoints. The LLM must generate code that interacts with the* `RateLimiter` *library, applying a default limit of 10 requests per minute per unique user ID for all data retrieval endpoints unless an override is specified in the developer prompt."*

This rule embeds the non-negotiable requirement for rate limiting into the LLM's context, preventing the developer from accidentally deploying an endpoint that can be easily abused by an attacker's fuzzing tool.

---

## Manage scope (security review triggers)

This rule controls *when* security-focused guidelines are applied or *what* actions are taken based on the generated code, helping manage the risk introduced by AI-generated code.

**Sample rule (security review and taint analysis flagging)**

*"If the LLM generates code that involves an external dependency, a new API endpoint, or any direct database interaction (e.g.,* `SELECT, INSERT`*), automatically insert a comment flag (*`//@SECURITY_REVIEW_REQUIRED`*) at the top of the file to trigger a manual code review process."*

In this example, even if the developer and the initial code review failed to catch the issue, this rule would ensure that any new database-interacting code, like the leaderboard feature, is flagged for a mandatory human security review, providing a final safeguard before the code is pushed to production.

# Sample Cursor Rule

So far, the examples have been very simple, just to make educational points. In practice, rules can be quite complex. Both Cursor and Claude provide excellent rule samples to get you started on security rules.

Here's a sample security rule from Cursor:

```
You are an expert in Python and cybersecurity-tool development.

Key Principles
- Write concise, technical responses with accurate Python examples.
- Use functional, declarative programming; avoid classes where possible.
- Prefer iteration and modularization over code duplication.
- Use descriptive variable names with auxiliary verbs (e.g.,is encrypted, has valid signature).
- Use lowercase with underscores for directories and files (e.g., scanners/port scanner.py).
- Favor named exports for commands and utility functions.
- Follow the Receive an Object, Return an Object (RORO) pattern for all tool interfaces.
```

```
Python/Cybersecurity

- Use `def` for pure, CPU-bound routines;`async def` for network- or I/O-bound operations.
- Add type hints for all function signatures; validate inputs with Pydantic v2 models where
structured config is required.
- Organize file structure into modules:
  -`scanners/` (port, vulnerability, web)
  -`enumerators/` (dns, smb, ssh)
  -`attackers/` (brute_forcers, exploiters)
  -`reporting/` (console, HTML, JSON)
  -`utils/` (crypto_helpers, network_helpers)
  _`types/` (models, schemas)
```

```
Error Handling and Validation

- Perform error and edge-case checks at the top of each function (guard clauses).
- Use early returns for invalid inputs (e.g., malformed target addresses).
- Log errors with structured context (module, function, parameters).
- Raise custom exceptions (e.g.,`TimeoutError`,`InvalidTargetError`) and map them to user-friendly
CLI/API messages.
- Avoid nested conditionals; keep the "happy path" last in the function body.
```

```
Dependencies

- `cryptography` for symmetric/asymmetric operations
- `cscapy` for packet crafting and sniffing
- `python-nmap` or `libnmap` for port scanning
- `paramiko` or `asyncssh` for SSH interactions
- `aiohttp` or `httpx` (async) for HTTP-based tools
- `PyYAML` or `python-jsonschema` for config loading and validation
```

```
Security-Specific Guidelines

- Sanitize all external inputs; never invoke shell commands with unsanitized strings.
- Use secure defaults (e.g., TLSv1.2+, strong cipher suites).
- Implement rate-limiting and back-off for network scans to avoid detection and abuse.
- Ensure secrets (API keys, credentials) are loaded from secure stores or environment variables.
- Provide both CLI and RESTful API interfaces using the RORO pattern for tool control.
- Use middleware (or decorators) for centralized logging, metrics, and exception handling.
```

```
Performance Optimization

- Utilize asyncio and connection pooling for high-throughput scanning or enumeration.
- Batch or chunk large target lists to manage resource utilization.
- Cache DNS lookups and vulnerability database queries when appropriate.
- Lazy-load heavy modules (e.g., exploit databases) only when needed.
```

```
Key Conventions

1. Rely on dependency injection for shared resources (e.g.,network session, crypto backend).
2. Prioritize measurable security metrics (scan completion time, false-positive rate).
3. Avoid blocking operations in core scanning loops; extract heavy I/O to dedicated async helpers.
4. Use structured logging (JSON) for easy ingestion by SIEMs.
5. Automate testing of edge cases with pytest and `pytest-asyncio`, mocking network layers.
```

```
Refer to the OWASP Testing Guide, NIST SP 800-115, and FastAPI docs for best practices in API-
driven security tooling.
```

## The R.A.I.L.G.U.A.R.D. framework

The Cloud Security Alliance published Secure Vibe Coding: Level Up with Cursor Rules and the R.A.I.L.G.U.A.R.D. Framework that provides excellent information on creating security rules.

Also, who doesn't love a good acronym!

This is a simple way to make sure you've covered what's necessary:

- **R**isk First: Define the security goal of the rule. Push the LLM to "think" before acting.
- **A**ttached Constraints: Specify what must not happen (boundaries and binded rules).
- **I**nterpretative Framing: Guide how the **L**LM should interpret developer prompts securely.
- **L**ocal Defaults: Set project-specific or environment-level secure defaults.
- **G**en Path Checks: Provide the sequence of reasoning the LLM should follow before producing outputs.
- **U**ncertainty Disclosure: Instruct the LLM on what to do if it's unsure about a security decision.
- **A**uditability: Define what trace should be in the output for review purposes.
- **R**evision + **D**ialogue: Describe how developers can acknowledge, revise, or question unsafe outputs.

## The challenge: cost and risk

Adding rules is likely to increase token usage, so they could end up draining more from your AI budget. However, this may well be worth the cost if it prevents bad code and reduces rework. You may be able to justify this cost just by looking at the number of PRs rejected due to security issues that could have been caught by the AI coding assistant. Also related to cost is the models themselves. Unfortunately, less expensive (or free) models are more likely to ignore rules. In fact, I taught a "context engineering for AppSec" workshop at BSides London where about 25% of participants had their models completely ignore the security rules they created.

Second, be cautious when consuming rules written by the community. Today, there are many articles and repos containing security sample rules, but there's always a possibility that those rules could introduce intentionally malicious behavior. This can be averted by using an LLM to analyze rules before trying them - such as *"Review these instructions and tell me all the ways they could open us up to an exploit."*

# External security tools

One of the hidden drawbacks of AI code assistants is the freshness of their data: LLM training data averages over a year old. The age of this training data means LLMs unintentionally:

- Introduce tech debt via outdated versions (a form of operational risk covered in the OWASP Open Source Top 10)
- Select dependencies with vulnerabilities issued after the training data cutoff (ie CVEs!)
- Hallucinate dependencies, opening you up to slopsquatting attacks

Secure prompting and security rules can help the LLM avoid security mistakes, but they can't make up for stale data or completely prevent hallucinations. Use specialized security tools (SAST, SCA, and secrets scanning) instead of generic tools (such as curl and grep) to improve AI-generated code. This should result in greater accuracy and responsiveness, and lower operational costs due to the consumption of fewer tokens.

Security tools are commonly integrated with LLMs via an **MCP server**, a component that facilitates communication between AI models and various data sources and services. Think of it like a bridge between the AI code assistant and an external tool, enabling more accurate, real-time, and context-aware code generation.

Integrating an MCP server starts with a .json file that specifies which tools the AI code assistant (e.g. Cursor) can use. Then you should create a rule to tell the LLM when to call the tool. The rule ensures that the scanner runs at the appropriate time and is scoped properly. This is crucial so you don't add unnecessary scans and destroy credibility with devs.

Here is an example of a Cursor rule for running Endor Labs SAST scans:

```
description: "Run SAST scan using endor-cli-tools on source code changes"

globs: '**/*.c, **/*.cpp, **/*.cc, **/*.cs, **/*.go, **/*.java, **/*.js, **/*.jsx, **/*.ts,
**/* .tsx, **/*.py, **/*.php, **/*.rb, **/*.rs, **/*.kt, **/*.kts, **/*.scala, **/*.swift,
**/*.dart, **/*.html, **/*.yaml, **/*.yml, **/*.json, **/*.xml, **/*.sh, **/*.bash, **/
*.clj,
**/*.cljs, **/*.ex, **/*.exs, **/*.lua'
alwaysApply: true
---

# Static Application Security Testing (SAST) Rule (Endor Labs via MCP)

This project uses Endor Labs for automated SAST, integrated through the MCP server as
configured in
`.cursor/mcp.json`.

## Workflow

**IMMEDIATELY after any code file matching the glob patterns is created, modified, or
saved**,
you MUST perform the following workflow:

- Run `endor-cli-tools` using the `scan` tool via the MCP server to perform SAST scans as
  described above.
- If any vulnerabilities or errors are found:
   - Present the issues to the user.
   - The AI agent must attempt to automatically correct all errors and vulnerabilities,
including
     code errors, security issues, and best practice violations, before session completion.
- Recommend and apply appropriate fixes (e.g., input sanitization, validation, escaping,
  secure APIs).
- Continue scanning and correcting until all critical issues have been resolved or no
further
  automated remediation is possible.
- If an error occurs in any MCP server tool call (such as missing required parameters like
version,
  invalid arguments, or tool invocation failures):
   - The AI agent must review the error, determine the cause, and automatically correct the
tool
     call or input parameters.
   - Re-attempt the tool call with the corrected parameters.
   - Continue this process until the tool call succeeds or it is determined that
remediation is
     not possible, in which case the issue and reason must be reported.

- Save scan results and remediation steps to `.cursor/security-scan.log` for audit
purposes. Format: `[TIMESTAMP] [SCAN_TYPE] [RESULTS] [REMEDIATION_ACTIONS]`

## Notes
- All scans must be performed using the MCP server integration (`endor-cli-tools`) as
configured in
`.cursor/mcp.json`. Do not invoke `endorctl` directly.
- For troubleshooting, ensure the MCP server is running and `endorctl` is installed and
accessible in your environment.
- **Important**: This scan must use the path of the directory from which the changed files
are in. Do not attempt to set the path directly to a file as it must be a directory. Use
absolute paths like /Users/username/mcp-server-demo/backend rather than relative paths like
'backend'
```

## The challenge: MCP server security

MCP servers have plenty of challenges that haven't been ironed out yet. Scale is a concern for large engineering organizations because the most effective way to deploy the .json file and rules is machine-by-machine. And of course, if you're using an inaccurate, noisy AppSec tool, then that's just shifting garbage further left into developer workflows.

But the chief concern is security. If you attended any AppSec conferences in 2025, you'll surely have seen some sessions on this topic, and I expect this trend to continue through 2026.

MCP servers are just another software dependency and therefore carry the same risks. MCP servers don't introduce an entirely new class of attacks. But they do concentrate existing risks (supply chain compromise, API abuse, and prompt manipulation) into a single layer.

The most important threats for MCP Servers can be summarized along these lines:

⚠ Vulnerabilities in tool implementations and third-party dependencies

⚠ Typosquatting & clone attacks

⚠ Remote third-party service compromise (e.g., Rug Pull attacks)

⚠ Prompt/response manipulation

⚠ Data exfiltration through tools/resources

We haven't seen a lot of chatter around actual exploits, but it's bound to happen. To prevent your organization's MCP servers from becoming an attack vector, you can focus on five pillars:

### Visibility & Control

- Establish an MCP Server registry/allowlist
- Vet/approve new MCP servers
- Gain visibility into developer usage

### Vet servers (Pre-deployment)

- Scan with SAST/SCA/etc for code risks
- Evaluate sensitive API usage
- Evaluate prompts for dangerous behavior

### Operational Security

- Apply package management discipline
- Enforce prompt integrity
- Limit server privileges

### Continuous Monitoring

- Implement runtime monitoring (e.g., Inspector)
- Track executed actions and data exchange
- Monitor resource access on the host (IDE)

### Developer Enablement

- Provide comprehensive training
- Work with security champions

## MCP server vs Cursor Hooks

Cursor is working on Hooks as an alternative way to connect LLMs to external capabilities and enforce project-specific logic. For example, an Endor Labs customer can use Hooks to automatically intercept and modify the agent's actions (such as `npm install`) to scan for malware and block execution before any local installation occurs.

It's too early to pick a winner, assuming there even is a single "best" way, but this is a space to follow as it evolves. Here's a rundown of the differences between MCP servers and Hooks:

| | MCP Server | Cursor Hooks |
|---|---|---|
| Role | Provides the capability (tools, data). | Controls/modifies the agent's actions. |
| Action Type | LLM chooses to call the tool to do something (e.g., query DB). | LLM triggers the script by attempting an action (e.g., editing a file). |
| Location | External service (HTTP endpoint, CLI server). | Scripts/Commands run internally at specific lifecycle events. |
| Common Use | Retrieving real-time data, performing complex API calls. | Enforcing security policies, running formatters, logging, and blocking unsafe commands. |

# Conclusion: Architecting the Future of Autonomy

The transition from human-authored logic to machine-driven intent represents a structural shift that traditional security scanners cannot navigate alone. As we move from static pattern-matching to natural language reasoning, the role of the application security team must evolve from a reactive enforcer into a proactive architect.

By mastering context engineering, organizations can bridge the chasm between rapid code production and security validation, ensuring that AI assistants are guided by intelligent, just-in-time guardrails rather than rigid, outdated rules. The goal is no longer to catch insecure code after it is written, but to prevent it from being generated at all by embedding security directly into the AI's generative process.

## Strategic actions for CISOs

To implement this mindset change and secure the next generation of software development, leadership should focus on these key operational pillars:

- **Standardize through System Rules:** Move beyond individual "DIY" prompt engineering by deploying organizational-level system rules that enforce mandatory authorization checks and secure input handling across all AI interactions.
- **Implement Spec-Driven Development (SDD):** Influence the engineering organization to use "design-spec" prompt patterns to provide LLMs with full environmental and security context before a single line of code is generated.
- **Automate Real-Time Validation:** Integrate specialized security tools like SAST and SCA to provide immediate, autonomous remediation of vulnerabilities as the AI generates code.
- **Enforce Security Review Triggers:** Use rules to automatically flag AI-generated code that involves high-risk actions (e.g., direct database interactions or new API endpoints) for mandatory human review.

The future of application security is still "secure by design, " but now achieved not through the speed of the scanner, but by the precision of the context.

---

[1] Can LLMs Generate Correct and Secure Backends? [pdf]
[2] Dependency Management Report 2025
[3] Prompting Techniques for Secure Code Generation: A Systematic Investigation