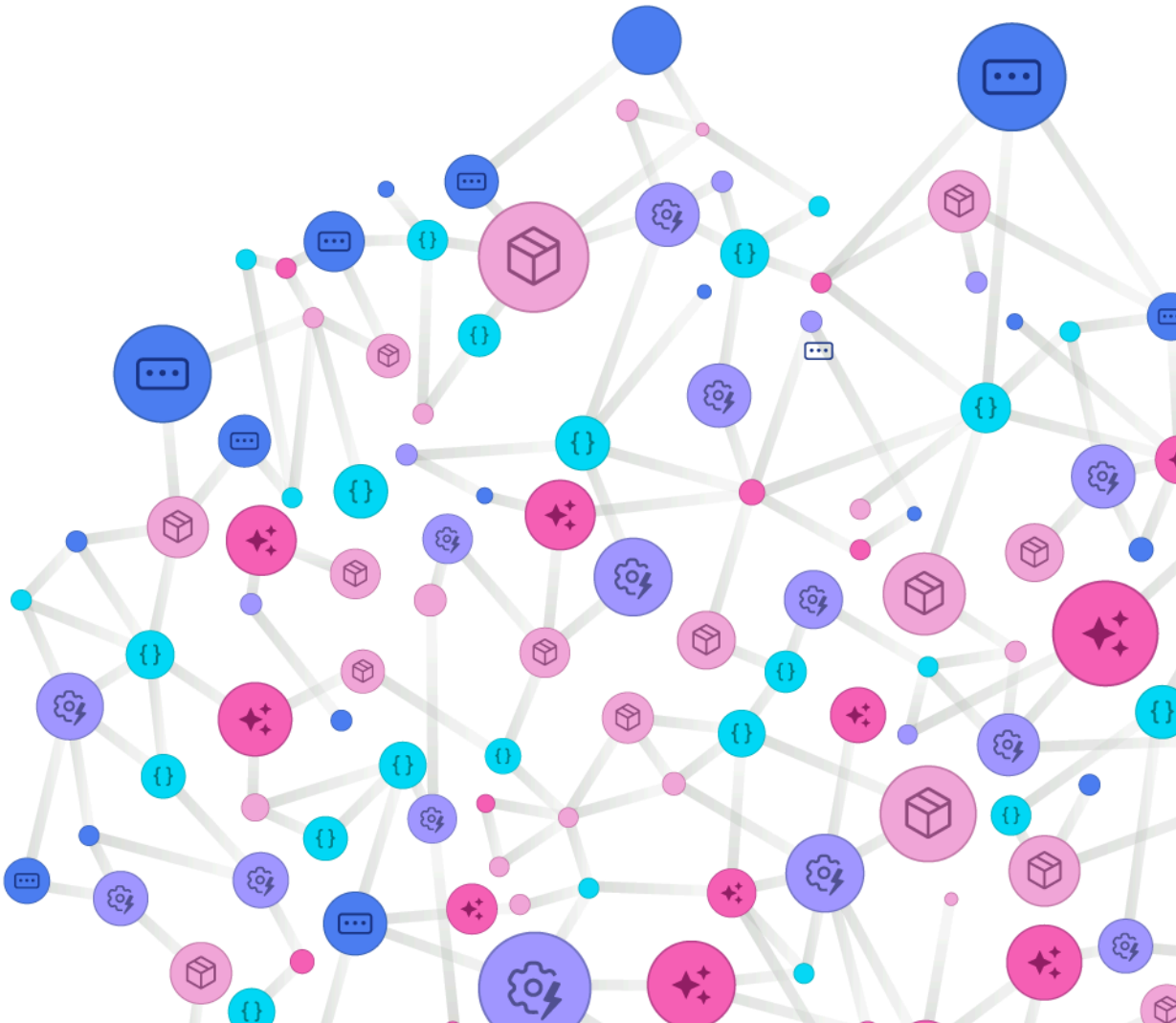
BACKSLASH

WHITEPAPER

Vibe Securing™: Addressing The Vibe Coding Tsunami of Security Risks



2025, v1.06

Vibe Securing™:
Addressing The Vibe Coding Tsunami of Security Risks

The rapid adoption of AI-augmented development, AI coding IDEs and agents, or "vibe coding" as it was coined by OpenAI co-founder Andrej Karpathy, is accelerating software development at an unprecedented pace. It is creating a new governance and security blind spot for security teams and engineering alike since it fundamentally changes the software development ecosystem. Concerns abound about the questionable quality, efficiency and security of these new practices and their resulting code at this early stage of its existence. And as we will demonstrate, at least from a security standpoint, the concerns are justified.

This whitepaper explores the growing security risks associated with AI-generated code in software development and the poorly governed environments in which it thrives, looks at the findings of our research into AI-generated code security, and presents a necessary shift in application security strategy. Backslash Security's capabilities were designed from the ground up to address these risks, using a purpose-built platform that provides immediate contextual risk assessment of findings. Starting with visibility and governance, it provides transparent guardrails that ensure a hardened environment, secure-by-default code, and empowers developers with actionable remediation guidance to further enhance the security of their code by leveraging AI-driven remediation. The result is a vibe coding ecosystem and workflows that are **secure by default.**

The Rise (and Rise) of Vibe Coding

AI-assisted development tools like Cursor, Windsurf, Claude Code and GitHub Copilot are quickly becoming the de facto tools for many engineers and non-engineers alike. This shift, driven by bottom-up adoption, allows developers to create code based on intuition and rapid iteration. The pace of this change is relentless; Google CEO says that 25% of Google code is already AI-written, while Gartner conservatively predicts that by 2028, 75% of developers will use AI assistant. This is also evidenced by the truly astounding growth of tools such as Cursor, reaching 100s of millions of dollars in revenue in just a few months. AI-augmented coding brings clear benefits to development teams: It increases efficiency, saves time, and frees developers to work on complex tasks like system architecture and design. However, these tools also introduce substantial security challenges.

Most code security scanning tools lack context around exploitability of security findings. Therefore, security teams continue to struggle with a high volume of vulnerabilities. Now with the widespread use of AI in coding, due to the explosion in open source usage and rapid development of cloud native apps. Many of the SAST and SCA scanners that have been around for years are already generating too much noise and false positives. Now, with AI accelerating code generation even further, security teams face the risk of falling further behind, struggling with even more findings, both real and false. Existing tools lack the insights, speed and scale to find, curb, and remediate these new vulnerabilities and weaknesses.

Lack of Visibility and Governance

Challenge #1 for most security teams and CISOs is lack of visibility, without which it is impossible to achieve governance. AI coders are either IDEs (Integrated Developer Environments) in their own right, or AI agents integrated into existing IDEs – and broadly speaking, **security teams have never had visibility into IDEs**. They didn't need it. The closest we've come to that in recent years are IDE extensions that provide developers with remediation advice as they code, a form of shift-left security. But this is usually unidirectional input *into* the IDE, not a way to centrally see what's going on within them. For security teams to even know where vibe coding is used, this must change.

> ***"How do we look at vibe coding? With horror!"***
> CISO, publicly traded company

Not only do we not know where developers are "vibe coding", we also don't know **which LLMs they are using,** and what other 3rd party components (e.g., MCP servers) have been introduced into their supply chain. The choice of LLM is in itself important, because as we'll see, even among the popular "safe" ones there is great variance in the security posture of their code output. There might be more esoteric LLMs out there that present significant risk.

In terms of 3rd party component risk, **MCP (Model Context Protocol) servers** is of primary concern. MCP servers are used to extend the LLM into external tools and data sources. They can really be used for anything that requires interacting with the AI service beyond its default interface. Already, there are tens of thousands of MCP servers available, and websites cataloguing them. Herein lies an additional risk - many of these MCPs are unofficial and of unvetted origin. Some can already inject malicious code or extract confidential data. Others simply expose the environment to 3rd party risk from unvetted data sources and a lack of understanding how the AI model might ingest them. They present unacceptable risk to application, infrastructure, and data security.

Are Coding LLMs Secure by Default?

Spoiler: The answer is no. Backslash conducted research evaluating seven versions of popular LLMs from OpenAI, Anthropic, and Google. We tested the code generated by these models against 10 Common Weakness Enumeration (CWE) use cases using three tiers of prompting techniques: "naive," "security-minded," and "rule-bound".

The findings revealed a common theme: **LLMs generally produce insecure code when not explicitly guided towards security**.

These are the highlights of the findings:

- **"Naive" Prompts:** Using simple prompts that only requested code for the application functionality, *all* tested LLMs and versions generated code vulnerable to **at least 4 out of 10 common CWEs**, and in some cases **9 out of 10 were vulnerable**.

- **Security-Minded Prompts:** Prompts that included a general security request ("make sure you are writing secure code") or requested compliance with OWASP best practices produced more secure results, but ***still yielded significant code vulnerabilities for 5 out of the 7 LLMs tested***. The effectiveness of these prompts also varied depending on the programming language.
- **Rule-Bound Prompts:** Only when prompts were bound to specific rules defined by Backslash to address particular CWEs did the LLMs consistently produce code that was secure and not vulnerable to the tested issues, *achieving a 100% score across all tested models*.

The immediate conclusion is that developers using AI coding tools without specific, security-focused prompts will create insecure and vulnerable code 40%-90% of the time. **Relying on developers, who largely are not security experts, to craft effective security prompts is both unrealistic and impractical**.

> ***"I asked a developer to automate a test case for me, he told me he did it in 2 hours with ChatGPT, and now I'm worried."***
> GRC Manager, MedTech Company

But aren't such results a temporary "growing pain"? Don't AI-generated code improve over time? It will. But many organizations can't risk waiting for as long as this might take, and it will take a long time. We only tested 10 very simple scenarios against 10 common CWEs. There are 10,000s of potential vulnerabilities out there, and numerous threat vectors. We need to understand the long-term implications of these findings, not just these point-in-time results.

The Tsunami of New Risks

As the use of AI coding practices spreads, we're facing two main types of security risks: Environmental or infrastructure risks, and vulnerabilities in code.

Infrastructure Risk

When AI agents are presented into IDEs or the IDEs themselves are replaced, and when MCP servers are connected willy-nilly to developer environments, this presents a slew of new risks to the security posture and governance of these environments. It might lead to the introduction of malware and vulnerabilities into code, and might also simply serve to infiltrate the developer machines, network, and source code management systems. Here are a few examples of infrastructure risk:

- **Poor IDE/Agent Configuration:** The various tools now available were developed rapidly, and each have different settings that might affect their security posture. For example, they might have allowlisting and denylisting capabilities, open to configuration by the user. We found that some of these features do not work as advertised (e.g., in Cursor), creating a false sense of security.
- **Vulnerable MCP servers:** Unvetted MCP servers can be vulnerable to abuse, depending on their permissions and network settings. They can also be malicious using obfuscation and spoofing techniques.

- **3rd party risk:** When connecting any external data source or agent via MCP, or even using proprietary integration, this can potentially open the door to context poisoning and data poisoning. The AI model is unaware of the identity or context in which the 3rd party tools or data sources are used, leading to unexpected behaviors. For example, the AI model can be fooled into training itself on the wrong dataset.
- **Prompt poisoning via rules:** Rules are a useful tool that can provide consistency to prompting and are available in some form in most AI coders. However, developers can download pre-made tools off the internet, and those tools can be poisoned to make the coder introduce backdoors or vulnerabilities into code.

Weaknesses in Code

In order to understand why “insecure by default” AI-generated code is not just a fluke, we need to get to the root causes, and they stem from the way LLMs are created and trained:

- **Outdated Security Knowledge:** LLMs are trained on large historical datasets. The security posture of the training data reflects the knowledge available at the time of training. This means the AI may recommend or generate code based on deprecated or vulnerable packages, lacking real-time awareness of new Common Vulnerabilities and Exposures (CVEs).
- **Replicating Third-Party Vulnerabilities into First-Party Code:** AI can sometimes extract and embed open-source code directly into an application instead of referencing the package. This action brings the third-party code, including its flaws, into the proprietary codebase, making traceability and remediation harder and undermining the modularity benefits of open source.
- **Questionable Code Quality:** The quality of AI-generated code often reflects the *average* quality of the data it was trained on. If trained on a vast base that includes code not meeting enterprise-grade design or policy standards, the AI will generate code with hidden security and maintainability issues ("smelly code"). There is no immediately effective way to enforce organizational quality control standards on AI output.
- **Insecure Code Generation at Scale:** AI makes code easily reusable and accessible. A poor prompt that results in vulnerable code can be widely shared and adopted by other teams, rapidly spreading insecure patterns across an organization.
- **Gen AI Hallucinations:** LLMs can "hallucinate," generating outputs that are nonsensical or incorrect. This can lead to vulnerabilities that humans wouldn't typically introduce – such as non-existent open-source packages – potentially an entirely new genre of subtle security flaws that are difficult to find.
- **LLM Poisoning:** It is alarmingly simple to "poison" the training data or inputs of LLMs. This form of supply chain attack could embed flawed logic or insecure code patterns directly into the models themselves, causing widespread generation of vulnerable code without users even knowing.

Not only that, but AI is simply creating **more code**, **more exploitation opportunities for attackers**, and **more vulnerabilities** (the number of which has been growing exponentially anyway).

These issues compound the existing challenges for security teams, who are often outnumbered by developers (a ratio as high as 1:100 in some organizations). The volume of potential vulnerabilities will increase dramatically, while the traditional "find the needle in a haystack" approach to managing risks and distinguishing real threats from noise becomes unsustainable.

The AI Dev Security Imperative: A New Mindset

Given the speed and scale of AI code generation, it is clear that relying on existing AppSec tools and methods will not suffice. Such tools, like traditional SAST and SCA, already cause frustration and lack of trust among developers, and cannot be simply shored up to handle AI code. Furthermore, expecting developers to become security experts capable of crafting sophisticated security prompts is unrealistic. It has never worked before, and there's no reason to believe that will change.

Won't the use of AI as a security tool itself be the savior, finally allowing understaffed teams to catch up? That's unlikely. The pace of code creation is accelerating so much that it would never level the playing field, and it is also an extremely wasteful approach that will spin up endless compute workloads. The answer cannot be just using more "Security AI" to battle "Coding AI".

What about runtime security for AI? It is needed, but just as in other areas of security, we must take a layered approach and cannot expect the nascent AI runtime tools to be our single point of failure.

> ***"The gap between AI tooling and AppSec is widening fast."***
> Head of Platform Eng., SaaS company

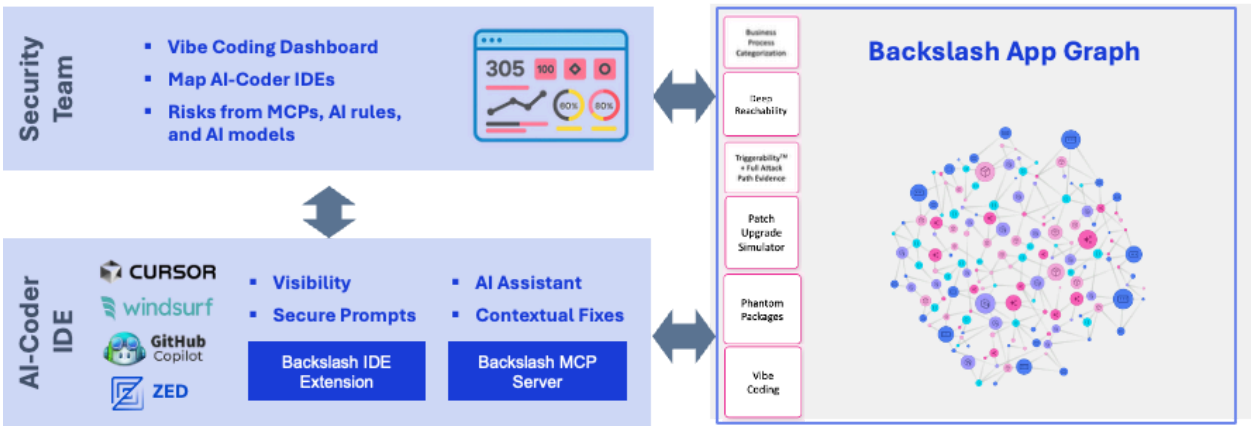
Instead, security needs a fundamental shift in mindset. **The focus must move from merely detecting vulnerabilities *after* code is written, to preventing their creation *before* code is generated.** It must understand context, threat vectors, and exploitability from the outset in order to be efficient and avoid false positives.

The Opportunity: Achieving Security by Design

Despite the challenges, we have a significant opportunity: **Security by design** of AI-generated code, an objective that has always eluded DevSecOps initiatives. By embedding the right controls and security guardrails into the AI coding process, security and engineering teams can ensure that vulnerability-free code is generated from the start, effectively embedding secure coding best practices directly into every piece of LLM-generated code. Not only is the code more secure by *default*, there's also a set of processes that ensure this is maintained on an ongoing basis – hence, by design.

Equally, we must ensure the security posture of the entire vibe coding ecosystem by enforcing consistent policies across multiple teams and tools, and monitoring the ecosystem continuously for deviations.

Backslash Security proposes a fresh approach to vibe coding security. Our approach centers on integrating security directly into the developer's tools and workflow using a combination of new capabilities. We call this "vibe securing".



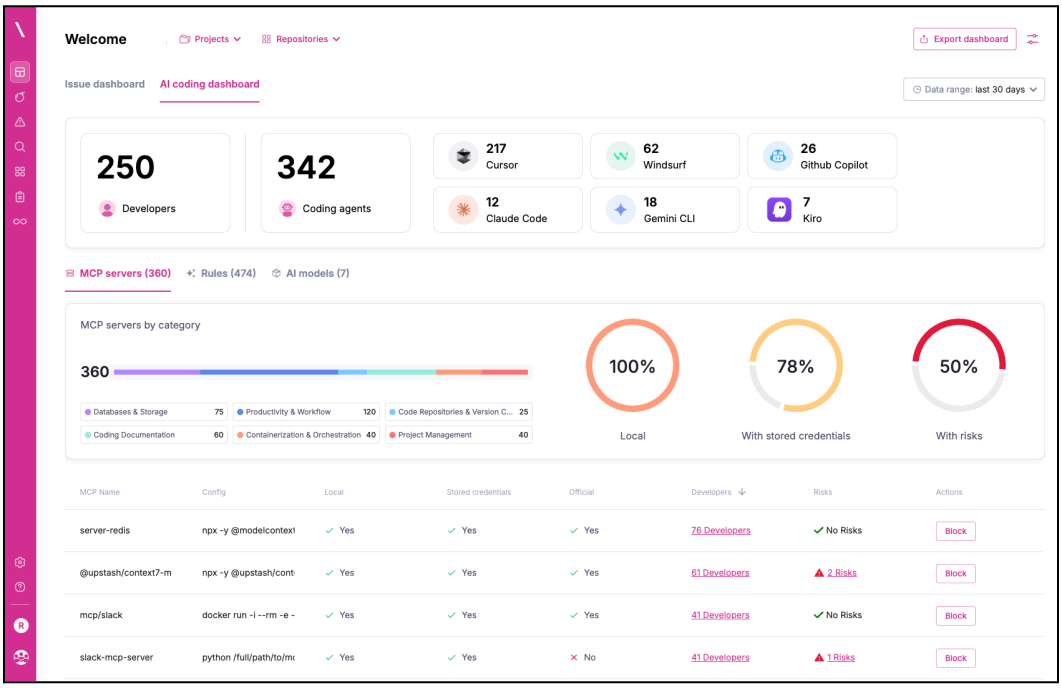
The Backslash Vibe Securing Solution

Backslash's Vibe Securing™ solution includes:

- **Vibe Coding Security Dashboard:** Giving security teams a comprehensive view of AI coders across their developer estate, unofficial MCP server use, what rules and models are used, and click-to-secure rule activation.
- **Backslash AI Prompt Rules & Policies:** These provide out-of-the-box, language-specific, and CWE-specific machine-readable rules that then govern the AI coder's prompt creation. Security teams can enforce which rules are active via policies. The rules are completely transparent to developers since they automatically enhance all prompts, and they are updated dynamically by the platform as new threats emerge.
- **Developer Transparency:** The majority of the benefits are gained without requiring the developers to do anything out of their ordinary workflow - it all works in the background.

- **Backslash IDE Extension:** This extension integrates directly into popular Agentic IDEs such as Cursor, Windsurf, and GitHub Copilot. It enables developers to receive Backslash security reviews on code written by *both* humans and AI directly within their workspace.
- **AI Coder Hardening:** Backslash enforces uniform configuration of AI agents and IDEs to ensure that user permissions, OS limitations, and other settings are securely set.
- **Backslash App Graph Model:** At the core of the Backslash platform is a proprietary code security engine that models the application into an AI-enabled App Graph. The App Graph ensures that all findings are contextual from the start - meaning that it doesn't flag vulnerabilities that have no real risk attached, avoiding a barrage of false positives.
- **Backslash MCP Server Security Assistant:** Provides further interactive, contextual security assistance by answering specific developer questions about code and packages, such as whether a package is vulnerable, if code exposes a vulnerable package, or what changes are needed for a safe upgrade.

By controlling the LLM prompt through AI rules and policies, integrating security reviews into the IDE via the extension, connecting with AI tools through the MCP server, and leveraging the contextual understanding of the App Graph, the Backslash platform aims to leverage AI-coding to drive secure code from the get-go, enabling true "security by design" for the first time. This approach provides built-in guardrails within a context-aware system created for both security teams and developers.



The Backslash Vibe Coding Security Dashboard

Conclusion and Action

The era of AI-generated code is here, bringing with it unprecedented speed in software development but with it a wave of new security risks. Our research shows that popular LLMs frequently generate insecure code unless specifically prompted with detailed security instructions. Root causes for risks like outdated security knowledge, vulnerability replication, questionable code quality, hallucinated flaws, and potential LLM poisoning are here to stay for the foreseeable future. This in addition to significant risks introduced into the environment by a set of disconnected and misconfigured tools.

Backslash empowers organizations to govern and secure the vibe coding environment, ensure AI-generated code is secure by default, and provide developers with interactive security guidance where they work.

This approach provides the necessary controls to enable security teams to gain control from the start and achieve the long-sought goal of "security by design". Modernizing application security for the AI era is essential to accelerate time-to-market safely, and avoid the perennial vulnerability chasing, experienced by so many AppSec and development teams to date.

To see Backslash in action, **book your demo now!**