LEGIT

TECHNICAL ARCHITECTURE GUIDE

Fixing Code Issues Early to Protect Developer Flow

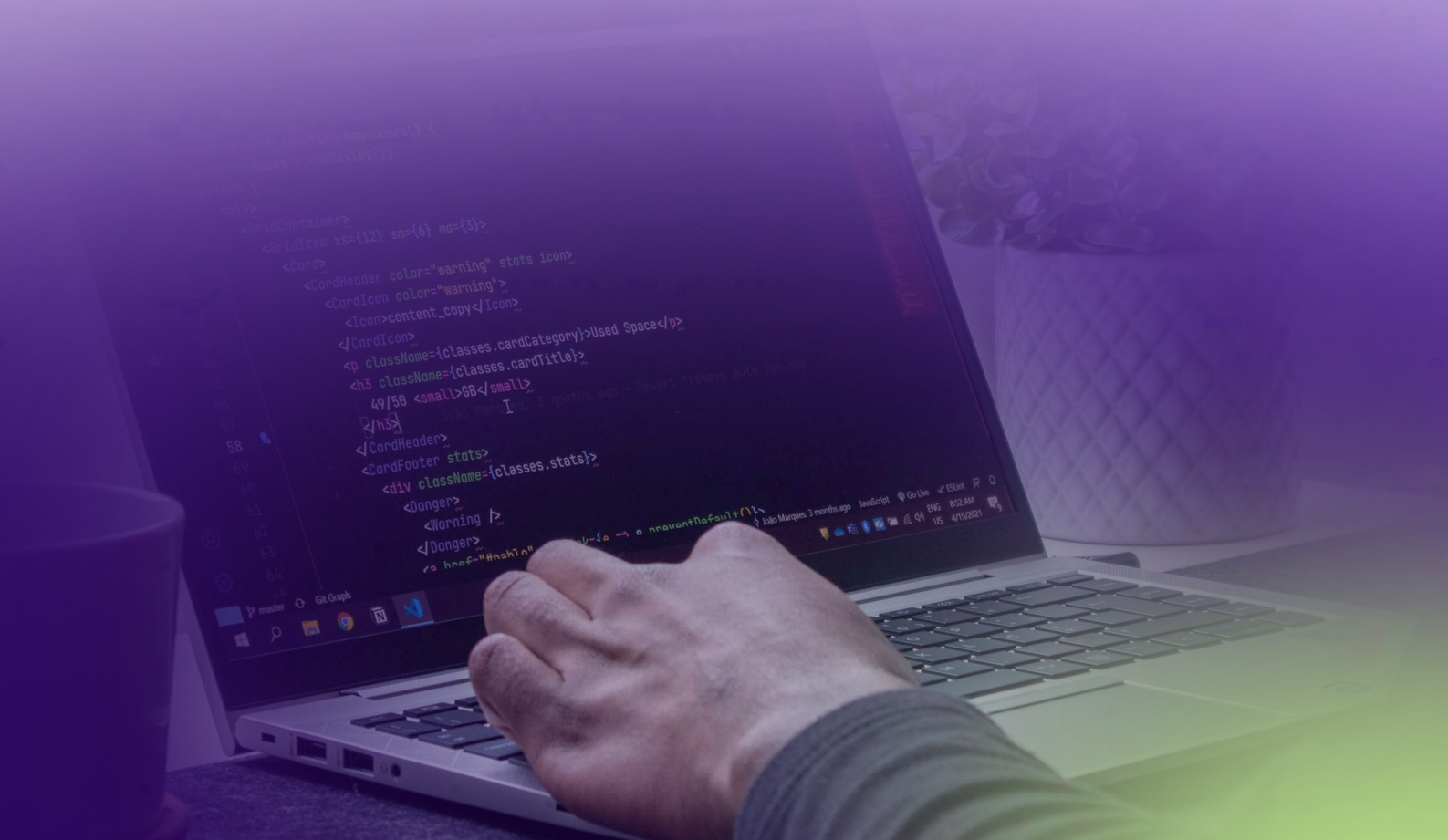
Learn how AI fundamentally changes development architectures and how pre-commit governance enables velocity without friction

Introduction

Developer productivity depends on flow: the state where engineers maintain focus, context and momentum. When issues are discovered late in the development process – after commits, code reviews or CI builds – teams pay the price in context switching, rework and broken momentum.

AI-assisted development amplifies this challenge. Code assistant like GitHub Copilot, Claude Code, Cursor and others generate code 3-5x faster than traditional, human-led development. This speed advantage disappears when security findings arrive hours, days or weeks later, forcing developers to context-switch back to code they've already moved past. And at this point, memories of the code context has begun to fade.

The opportunity isn't more scanning; that’s proven to be a flawed approach. It's earlier signals. This guide explains how AI changes the SDLC architecture, why traditional security controls break developer flow and what architectural patterns enable enforcement without friction.



THE FLOW PROBLEM

Why Late Feedback Is Expensive

Consider a typical scenario:

A developer uses Claude to generate a database query function at 9:30 AM. The generated code includes a hardcoded API key for convenience. The developer commits the code at 9:45 AM and moves on to the next feature. At 2:00 PM – 4.5 hours later – the CI pipeline flags the exposed credential. The developer must now:

1. Stop current work and context-switch back to the database query functionality
2. Remember what the original code was supposed to do and why
3. Investigate which credential was exposed and why
4. Fix the issue and re-test
5. Commit again and wait for CI
6. Return to interrupted work (if possible)

Total cost: 30-60 minutes of developer time, broken flow state and delayed feature work.

Alternative with earlier feedback: At 9:30 AM, as Claude generates the code, in-IDE capabilities detect the hardcoded credential in real-time. The developer sees an immediate warning in their IDE: 'API key detected. Use environment variable instead.' They fix it in 30 seconds, commit clean code at 9:32 AM and continue working. Flow state never broken.

Total cost: 30 seconds. No context switch. No flow disruption.

This isn't a hypothetical. Studies by Google's DevOps Research and Assessment (DORA) team found that teams with fast feedback loops deploy 200x more frequently and recover from incidents 24x faster. The difference is architectural: where feedback happens and when.

The Compounding Cost of Late Feedback

When security issues are found late, costs compound across multiple dimensions:

Feedback Timing	Developer Time	Delivery Impact	Developer Impact
Pre-generation	30 seconds (inline fix)	None – prevented issue	Positive (prevented problem)
Post-commit	30-60 min (context switch + rework)	PR delays, review cycles	Frustrating (feels like failure)
Post-CI	2-4 hours (investigation + fix + retest)	Pipeline blocked, release delays	Demoralizing (public failure)
Production	Days/weeks (incident response + hotfix)	Customer impact, emergency rollback	Crisis mode

The difference isn't just time, it's the compound effect of context loss, flow disruption and team morale. A 30-second inline fix preserves momentum. A 4-hour post-CI investigation destroys it.

Where AI Changes SDLC Architecture

Traditional SDLC security architectures assume a linear flow: developer writes code → commits to repository → automated controls run. AI breaks this assumption by generating code before the repository knows about it.

Traditional SDLC Control Points

In the traditional model, security controls activate at these points:

Repository commit	Pre-commit hooks scan for secrets, but only see what's being committed right now
Pull request	Code review and automated checks, but context is already lost
CI/CD pipeline	SAST, SCA and container scanning occur, generally hours after code was written
Runtime	DAS and runtime protection executed days or weeks after code entered production

The problem

The above control points are all post-creation. They detect problems after code already exists, which means developers must context-switch back to fix them. Expensive, time consuming, frustrating – none of which advance the engineering and business agendas.

AI-Driven SDLC: The New Control Point

AI fundamentally changes where code is created:

Traditional	Developer types code character by character in IDE → commits to repository
AI-Driven	Developer prompts AI → AI generates complete code blocks → developer accepts → commits to repository

The critical difference: there's a new decision point between 'AI generates' and 'developer accepts.' This is where controls must exist to preserve flow.

SDLC Layer Comparison:

SDLC Layer	Traditional Model	AI-Driven Model
Code Creation	Human types in IDE → commits	Human prompts AI → AI generates → human accepts → commits
Visibility Start	Repository (post-commit)	Developer endpoint (IDE, browser, terminal)
Security Feedback	CI/PR (hours later)	Inline (immediate, < 200ms)
Context Preservation	Lost (developer moved on)	Preserved (developer still in context)

The Architectural Implication: To preserve flow, security controls must operate at the same layer where AI operates – the developer endpoint, not the repository.

Architecture Pattern: Pre-Commit Enforcement

Pre-commit enforcement shifts security controls to the generation point. Here's how it works technically:

Component 1

Endpoint Integration

AppSec tooling integrates at the IDE level through lightweight extensions, such as:

VS Code	Native extension intercepts Copilot, Cursor, and inline AI completions
JetBrains Suite	Plugins for IntelliJ IDEA, PyCharm, WebStorm, etc.
AI-Native IDEs	Direct integrations with Cursor, Windsurf, Zed
Terminal/CLI	SDK for wrapping AI coding tools like Aider, Claude Code

Performance

Extensions add < 5MB memory footprint and < 200ms latency per evaluation. Developers don't notice the overhead.

Component 2

Real-Time Policy Enforcement

When a developer prompts an AI assistant, AppSec integrated into the IDE:

Intercepts the prompt and generated code	The extension captures both the developer's prompt ('write a function to query the database') and the AI's generated response.
Returns inline feedback	If code passes all policies, it's auto-approved and inserted into the editor. If violations are detected, the developer sees inline feedback with specific guidance.
Creates audit record	Every generation event – approved or blocked – generates an immutable audit record stored in the VibeGuard backend.

Performance characteristics

Latency	< 200ms end-to-end (prompt → policy evaluation → response)
Throughput	10,000+ evaluations/second per policy engine instance
Scalability	Auto-scales to handle enterprise-wide deployment (tested with 10,000+ developers)

Component 3

Transparent Evidence Collection

Every AI generation event creates an audit record with:

Context	Developer ID, timestamp, tool used, project/repository
Prompt	What the developer asked the AI to do
Generated code	Full code generated by the AI
Policy evaluation	Which policies were checked, pass/fail status
Developer action	Accepted, rejected, or modified the suggestion

Critical

No additional developer workflow steps required. Evidence collection is completely transparent and automatic.

IMPLEMENTATION PATTERNS

What Works and What Doesn't

Following are the patterns that preserve developer flow:

Pattern

Policy-Based Auto-Approval

What it is	95%+ of AI-generated code passes policy checks and is auto-approved without developer intervention. Only violations require action.
Why it works	Developers trust the system because it rarely interrupts them. When it does, the feedback is specific and actionable.
Example	A developer generates 20 functions per day. 19 are auto-approved. 1 is flagged for a hardcoded credential. The developer fixes it inline in 30 seconds and moves on. Flow preserved.

Anti-Pattern

Blocking All AI Code Pending Manual Review

What it is	Every AI-generated code block requires security team review before the developer can commit.
Why it fails	Creates massive bottleneck. Developers disable AI tools or route around controls. Security becomes the enemy of productivity.
Observed outcome	90% of developers bypass the control within 2 weeks. Security team is overwhelmed and gives up.

Pattern

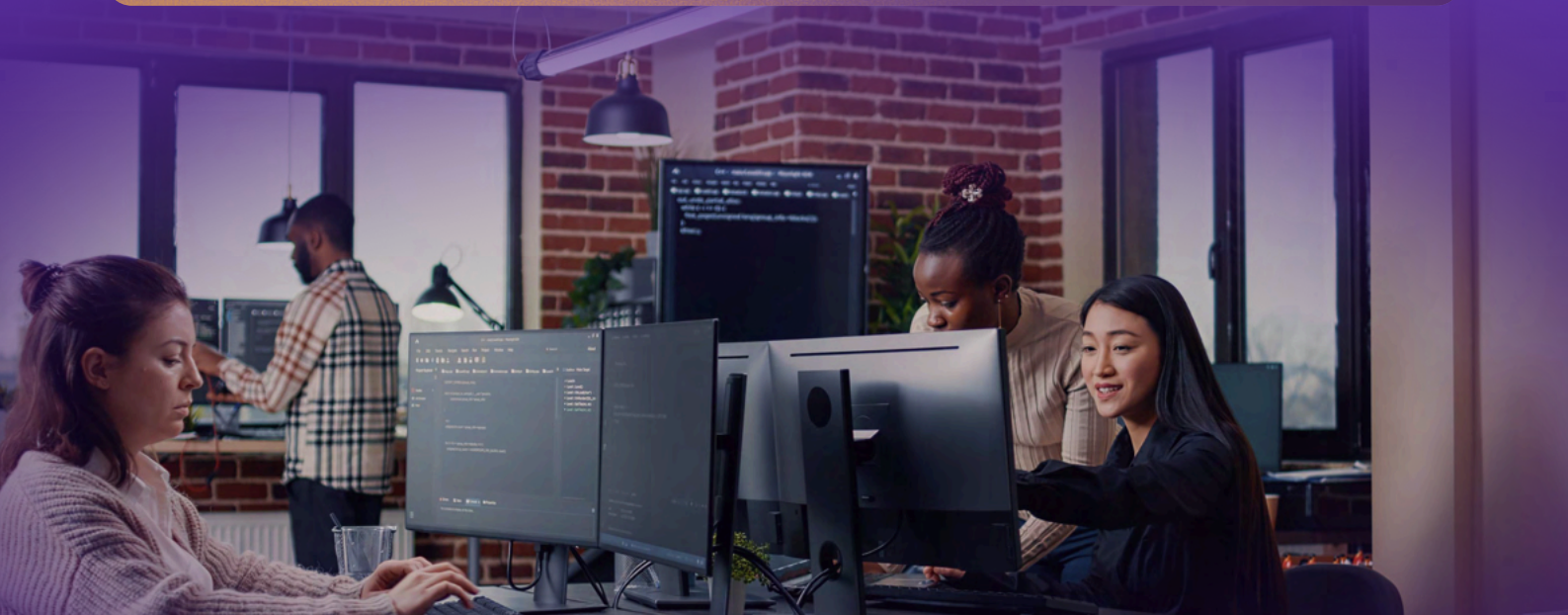
Inline Warnings at Generation Time

What it is	When a policy violation is detected, developers see immediate feedback in their IDE – while they're still looking at the code.
Why it works	Context is fresh. Developers can fix issues in seconds without breaking flow.
Example	AI suggests code with a SQL injection vulnerability. Developer sees inline warning: 'Potential SQL injection. Use parameterized queries.' Fixes it immediately. Total time: 15 seconds.

Anti-Pattern

Slack/Email Alerts After Commit

What it is	Security tool scans commits and sends Slack alerts or emails when it finds issues.
Why it fails	Context loss. Developer has moved on. Alert fatigue sets in. Notifications are ignored or queued for 'later' (which may never come).
Observed outcome	Mean time to remediation: 3-7 days. Many alerts are never addressed.



TRACKING RESULTS

What Organizations Achieve

Organizations that implement pre-commit AI governance report consistent improvements:

Developer Velocity Metrics

- **Significant reduction in rework time** (issues caught at generation vs. post-CI)
- **Nearly all security issues prevented before commit** (vs. 10-15% with traditional tools)
- **Improvement in developer satisfaction scores** (security becomes helpful vs. an obstacle)

Security Outcomes

- **Reduction in secrets committed to repositories**
- **Material decrease in license compliance violations**
- **Time to remediate AI-introduced vulnerabilities:** minutes instead of days

Business Impact

- **Audit readiness:** Complete AI governance evidence on demand
- **Deployment frequency:** Increases substantially due to fewer security-related delays
- **Security team efficiency:** Material reduction in manual code review time

CONCLUSION

Flow Isn't Optional

The fastest development teams don't fix more issues. Rather, they prevent disruptive ones from ever reaching the repo. AI-assisted development makes this principle more critical than ever.
Traditional security architecture was designed for a world where humans wrote code one line at a time. AI generates code 3-5x faster, but traditional controls still operate at the same slow, post-commit layer. This mismatch creates flow disruption at scale.

The architectural solution is straightforward: shift controls to where AI operates, at the developer endpoint. Evaluate code at generation time, provide inline feedback while context is fresh and collect evidence transparently. This preserves flow while improving security outcomes.

Organizations that make this shift report significant reductions in rework time with most issues prevented before commit. The difference isn't more security, it's applying an architecture that aligns with the way code is build and delivered today.

Flow isn't optional. It's how modern software gets built. Security architecture must adapt to preserve it.

Build fast with AI. Secure with AI-powered ASPM.

LEGIT

Legit Security is the AppSec platform purpose-built to secure AI-powered development. Our AI-native ASPM secures modern software development, including AI-first pipelines, code assistants, agents, and vibe coding. With unmatched visibility across the SDLC and from code to cloud, Legit makes it easy to identify, prioritize, and fix AppSec issues that matter most to the business.