Securing Vibe Coding with VibeReview

VIBEREVIEW

Introduction

AI coding agents moved from novelty to default tooling in a short span. Teams now use Cursor, Copilot, Claude Code, Codex, and similar systems for code generation, refactoring, test writing, dependency selection, shell execution, and pull request preparation. That shift is larger than autocomplete. The time when models used to suggest the next line in isolation is long gone. It is reading repository context, proposing architecture, invoking tools, and in many workflows acting with enough authority to change the security posture of a codebase before a human reviewer fully understands what changed.

That change matters because software security has always depended on friction. Good security review slows down unsafe assumptions. It forces someone to inspect trust boundaries, validate data flow, question dangerous defaults, and ask whether a shortcut creates a new attack path. Agentic coding compresses that review window. A developer can now produce far more code, touch more subsystems, and accept more hidden implementation detail in one session than was practical even a year ago. The output often looks coherent enough to merge. That is exactly what makes the risk easy to underestimate.

Recent incidents and research show that this is not a theoretical concern. In June 2025, Aim Security disclosed EchoLeak, tracked by Microsoft as CVE-2025-32711, a zero-click prompt injection attack against Microsoft 365 Copilot. A single crafted email was enough to steer the agent into exfiltrating sensitive enterprise data across trust boundaries without user interaction. The target in that case was an enterprise copilot rather than a source-code IDE, but the security lesson carries directly into coding agents: once an LLM is allowed to combine untrusted input, private context, and high-value actions, prompt injection stops being a content-quality problem and becomes an execution-path problem.

The same pattern appears inside coding environments themselves. A 2025 study on agentic coding editors showed that prompt injection attacks against tools such as Cursor and GitHub Copilot could coerce the agent into running attacker-influenced commands, with attack success rates reaching as high as 84% in the evaluated scenarios. That is a sharp warning for anyone treating AI-assisted development as a faster IDE workflow.

The code output is not safer just because the interface feels productive. In a late-2025 benchmark of real-world feature implementation tasks, researchers found that coding agents could produce functionally correct solutions far more often than secure ones. In the benchmark's headline result, one top-performing setup produced correct solutions 61% of the time but secure solutions only 10.5% of the time. That gap captures the core problem better than generic claims about "buggy AI code." Functional correctness and security correctness diverge under pressure. A model can satisfy the visible requirement while mishandling authentication, authorization, input validation, secret use, dependency trust, or multi-tenant isolation.

Vibecoding lowers the cost of building from a labour-intensive point of view, speeds up iteration, and makes complex changes accessible to smaller teams. It also makes it easier to ship code that has never been subjected to the kind of adversarial reasoning security work requires. Attackers do not need the agent to be malicious. They only need it to be obedient in the wrong context, overconfident about an unsafe implementation, or unable to distinguish repository instructions from attacker-controlled text.

That is the opening for VibeReview. The platform is built around a simple observation: generic AI coding assistance does not understand the security invariants of a specific codebase on its own. It needs context, explicit guardrails, and review logic that is tied to the repository, the stack, and the threats that actually matter in that environment. VibeReview's approach is to profile the codebase, generate tailored security rules, monitor pull requests against those rules, and capture security-relevant events from AI-powered IDE workflows.

This report will make that case in technical terms. The goal is not to argue that AI coding agents should be avoided, nor to recycle broad warnings about hallucinations. The goal is narrower and more useful: explain why these systems are structurally weak at security, show how that weakness surfaces in modern development workflows, and show why repository-specific guardrails and review loops are necessary if teams want the productivity gains without accepting silent security regressions.

The next section starts with the mechanics. We will examine why AI coding agents fail at security even when they appear competent at implementation: weak threat modeling, poor persistence of security invariants across long tasks, unsafe tool use, brittle handling of untrusted context, and a tendency to optimize for "working code" over "defensible code."

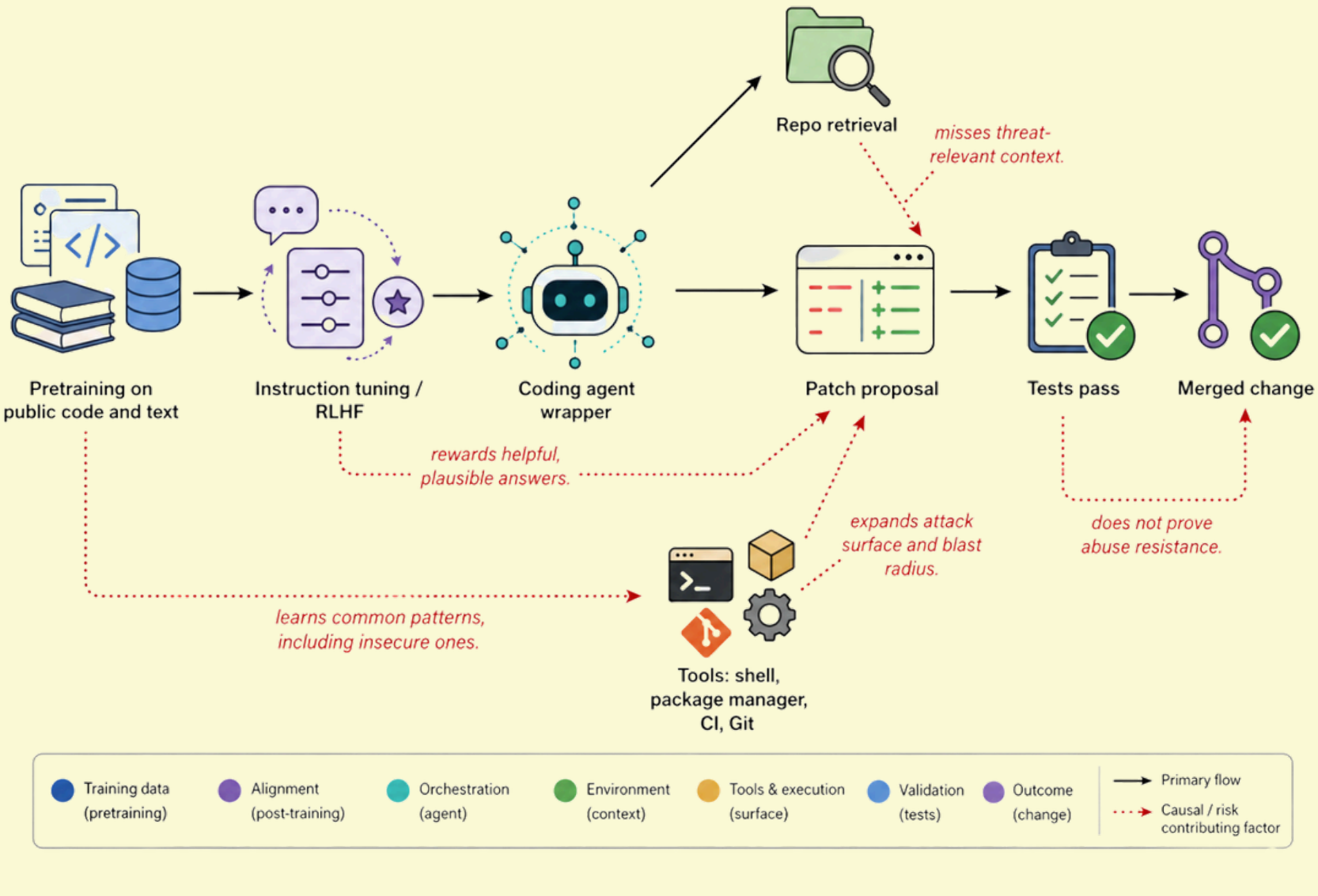
Why AI Coding Agents Are Weak at Security

Security failures in AI-assisted development come from two stacked systems with different objectives.

1. The base model is trained to continue patterns and satisfy prompts, not to preserve security invariants.
2. The coding agent wrapped around that model is tuned for task completion inside a repository, not for adversarial review.

Neither layer is secure by design. The model defaults to producing plausible completions. The agent defaults to finishing repository tasks. A team can prompt more carefully and still get insecure code. A team can use a stronger model and still get insecure changes.

The system is built to produce acceptable patches quickly. Threat-model preservation is outside its default objective.



AI coding agent security failure flow

A. Raw LLM Weakness: Security Is Outside the Default Objective

The problem starts before the agent opens a repository.

1. Pretraining learns common patterns, not defensible ones

Code models are pretrained on large public corpora. That gives them reach, but it also means they absorb insecure tutorials, outdated framework patterns, weak crypto usage, permissive CORS examples, unsafe shelling-out, and authentication shortcuts. The training data rarely marks those patterns with vulnerability labels. The model learns regularity. It does not learn exploitability as a first-class property.

That concern showed up early. In Asleep at the Keyboard?, researchers tested GitHub Copilot across 89 security-relevant scenarios and found that about 40% of generated programs were vulnerable. Later work showed the same basic problem: models reproduce historical vulnerability patterns and insecure examples from their training distribution more reliably than they infer a safer design from first principles.

2. Next-token prediction optimizes plausibility

Pretraining teaches the model to predict likely continuations. That is useful for code completion, but security is rarely the most likely continuation of a prompt. The likely continuation of "build a quick upload API" may be a functionally correct file handler. The secure continuation may require MIME validation, size limits, storage isolation, auth checks, randomized names, malware scanning hooks, and denial-of-service controls. Those controls are often absent from short examples and tutorials, so they are underrepresented in default generation.

Secure-by-design systems would make those controls part of the default answer. Next-token prediction does something else. It favors code that looks complete in the local frame.

3. The model sees code locally, while security lives in system context

A code snippet can look fine in isolation and still be unsafe in deployment. Authentication assumptions, tenant boundaries, secret handling, internal network reachability, build pipeline permissions, and rollback behavior often live outside the function being edited. The model has no durable internal representation of those constraints unless they are restated every time.

That limitation becomes visible in security repair tasks. In one 2024 study on buffer overflow repair with Copilot, vulnerability detection was much stronger than successful repair until the model was given added domain knowledge and security context. The issue was not syntax. The issue was missing context about what a safe repair requires.

4. Post-training makes the model more helpful, but security still stays partial

Instruction tuning and RLHF improve compliance with user intent. OpenAI's original InstructGPT paper describes the process clearly: human demonstrations and ranked outputs teach the model to follow instructions that people prefer. That helps with usefulness and format control. It does not give the reward model a strong security theory for every framework, protocol, and deployment model.

In practice:
- If the user asks for a fast implementation, the model tends to compress complexity.
- If a secure design is longer or less convenient, it may lose to the cleaner-looking answer.
- If the request is ordinary feature work rather than overt abuse, safety tuning often has little reason to block it.

Secure-by-design behavior would reward preservation of trust boundaries, least privilege, and safe failure modes even when the user does not ask for them explicitly. Post-training does not reliably do that. It rewards helpfulness inside the prompt's local frame.

B. Coding Agent Weakness: The Wrapper Optimizes for Delivery

Once the model is placed inside an agent loop, the problem changes shape. Weak security priors are now combined with repository access, tools, memory limits, and success criteria that favor shipping.

1. Retrieval is feature-centered and often blind to threat context

Coding agents usually fetch files that look relevant to the requested change. That is good for speed, but threat-relevant files are often different from feature-relevant files. A request to "add file sharing" may retrieve the route handler and UI components while missing:

- authorization middleware
- tenant scoping utilities
- audit logging
- storage lifecycle policies
- existing security guardrails

If the agent never sees those files, it cannot preserve the invariants they encode. Secure by design requires those invariants to drive the change. The default retrieval loop does not do that.

2. Minimal diffs bias the agent away from secure redesign

Most agent workflows reward small patches. Small patches are easier to review, easier to accept, and more likely to preserve passing tests. Security fixes often do not fit that shape. Real secure changes may require moving checks into middleware, tightening types, adding server-side validation, changing token scope, or refusing a convenience feature.

That creates a bias toward local patches like this:

```python
# Feature-complete, but security-thin
@app.post("/upload")
async def upload(file: UploadFile):
    path = f"/tmp/{file.filename}"
    with open(path, "wb") as f:
        f.write(await file.read())
    return {"ok": True}
```

Instead of something closer to this:

```python
@app.post("/upload")
async def upload(file: UploadFile, user: User = Depends(require_user)):
    enforce_content_type(file, allowed={"image/png", "image/jpeg"})
    enforce_size_limit(file, max_bytes=5_000_000)
    object_key = build_scoped_object_key(user.tenant_id, secrets.token_hex(16))
    await store_in_isolated_bucket(file, object_key)
    await write_audit_event(user.id, "upload_created", object_key)
    return {"ok": True}
```

The second version is longer because it carries the security design with it: scoped storage, explicit validation, user binding, and auditability. Agents biased toward minimal change underproduce that design unless the repository forces it.

3. Tests passing is a weak security oracle

Coding agents usually stop when unit tests pass, lint is clean, and the requested feature appears to work. Security failures often survive all three.

```typescript
app.use(cors({ origin: true, credentials: true }));
```

That line may unblock an integration test and still create a cross-origin exposure when paired with cookie-based authentication. Functional tests ask whether the feature works. Secure-by-design review asks whether an attacker can misuse it, whether a trust boundary moved, and what else became reachable.

Recent agent benchmarks show that this gap is still large. SecureAgentBench found that even strong agent-plus-model combinations struggled to produce solutions that were both correct and secure under realistic multi-file vulnerability scenarios. The late-2025 Vibe Coding benchmark reported the same shape from another angle: high functional success and much lower secure success.

4. Tool access turns prompt mistakes into system actions

A coding assistant is dangerous in ways a plain chat model is not. It can run shell commands, add dependencies, modify CI, write config, query MCP tools, and read repository instructions. A reasoning error escapes text generation and becomes a filesystem change, a pipeline change, or a data exposure path.

This is why prompt injection matters so much more for agents than for plain chat. The 2025 Your AI, My Shell study showed that agentic coding editors can be manipulated into attacker-influenced command execution through prompt injection pathways. Once the model treats untrusted repository content as instruction-bearing context, the line between reading a file and obeying the attacker gets thin.

5. Agents rarely retain project security memory reliably

Human security reviewers carry forward local rules:
- this route must always enforce tenant scope
- this service never returns raw secrets
- this webhook path must verify signatures before parsing
- this worker cannot make outbound network calls

Agents do not hold those rules stably unless the repository externalizes them through policy files, code structure, tests, or explicit guardrails. Without that external memory, the model can fix one bug and reintroduce an older class of weakness somewhere else. Secure by design depends on stable policy memory. Most coding agents do not have it.

What Is Missing

What is missing is secure-by-design structure.
- The model supplies plausible code from a distribution that contains insecure patterns.
- Post-training makes it cooperative without making it security-complete.
- The agent retrieves the files nearest to the task and can miss the files nearest to the risk.
- Tool access amplifies mistakes.
- Success checks stop at functionality too early.

That is why security does not emerge automatically from better prompting or a stronger base model. It has to be imposed as structure around the agent: repository-specific guardrails, explicit policy memory, adversarial tests, and review logic that rejects functionally correct but unsafe patches.

That is the design space VibeReview targets. If the raw model is weak on local security judgment and the agent wrapper is weak on preserving project invariants, the missing control plane has to live outside the model.

The Philosophy of VibeReview

The previous section described the failure mode: AI coding agents are good at finishing local tasks and weak at preserving security invariants. They retrieve the files nearest to the feature, produce a plausible patch, run functional checks, and stop. A secure system needs a different loop. The agent has to carry the repository's security rules while it works, and it has to reason about abuse before it treats the patch as done.

VibeReview is built around that loop. Teams keep using Cursor, Claude Code, Codex, GitHub Copilot, MCP, and prompt-driven development. VibeReview adds a control plane around those tools and makes one object central: the guardrail.

A guardrail is a project-specific security instruction, structured for both humans and agents: category, severity, rule type, instruction, provenance, approval behavior, and mappings such as CWE or OWASP where applicable. In practice, guardrails become the policy memory that coding agents fail to hold reliably on their own.

The design goal is simple to state and hard to enforce: before an AI agent writes security-relevant code, it should know the repository's rules, select the rules that apply to this task, threat-model the change against those rules, and leave behind evidence of what happened.

Guardrails Are the Center

Most AI coding workflows treat security as a late review activity. The developer asks for a feature. The agent writes code. A scan, reviewer, or CI job may catch a problem later. That approach has a timing problem. By the time security enters the workflow, the agent has already chosen the shape of the implementation.

VibeReview moves the decision point earlier. It prepares security context outside the model, stores it as guardrails, and injects those guardrails into the IDE before implementation. The model still writes code, now inside a repository-specific security workflow.

That shift matters for three reasons.

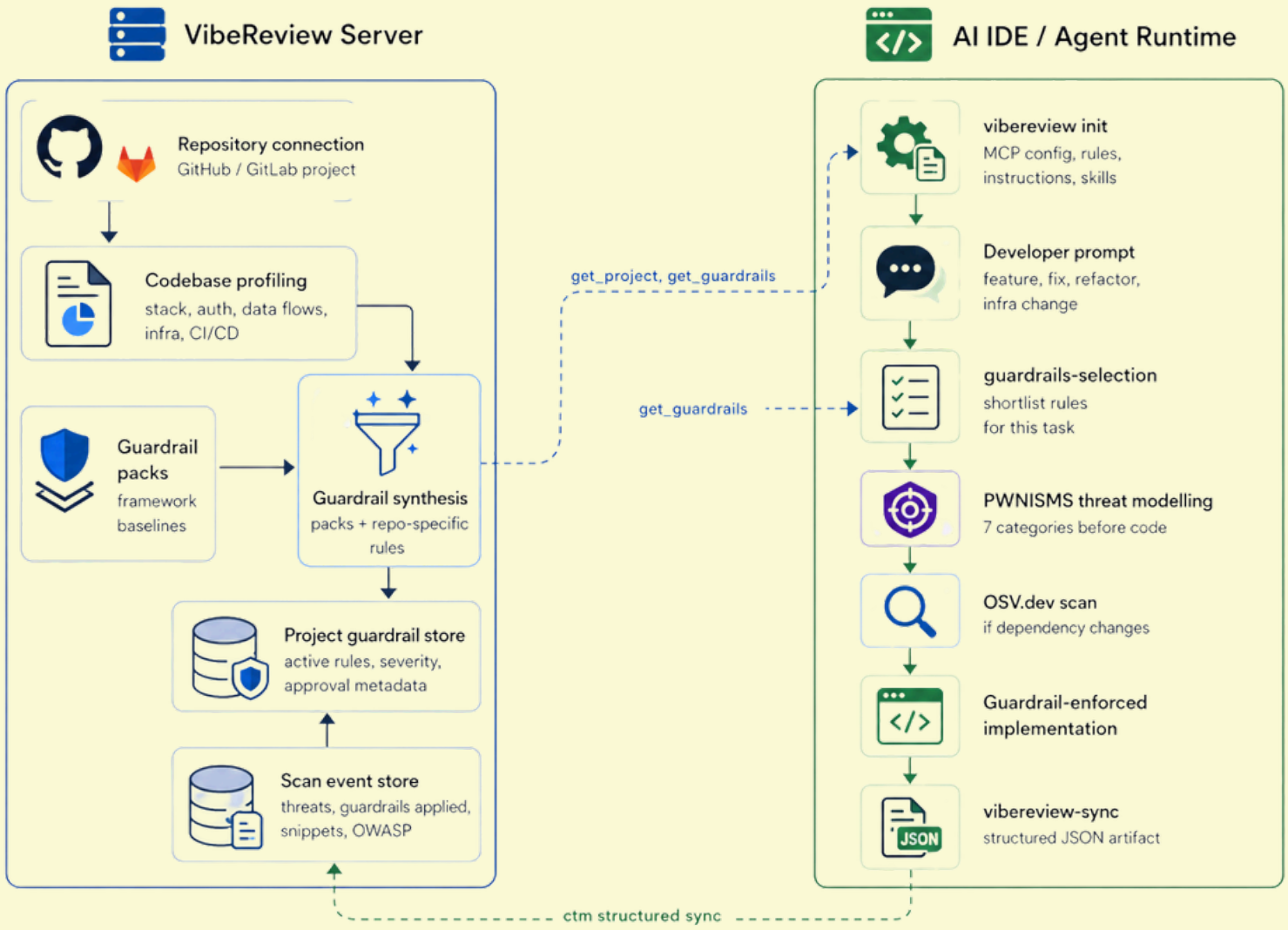
First, guardrails give the agent stable memory. A human reviewer may remember that tenant scoping must happen in a service layer, that webhook signatures must be verified before parsing, or that raw tokens must stay out of logs. A model carries those rules consistently only when the repository externalizes them.

Second, guardrails make security narrow enough to use. A full application profile can be too large for every coding prompt. A shortlist of relevant guardrails can fit into the active task, steer the implementation, and still preserve the important constraints.

Third, guardrails are auditable. If a task selected five guardrails, satisfied four, generated one new IDE-side guardrail, and left a residual risk, VibeReview can record that as structured telemetry. Security work stops disappearing into chat transcripts.

The System View

The architecture has two halves. The server side builds and stores security knowledge. The IDE side consumes that knowledge at the moment an agent is about to change code.



VibeReview server and IDE workflow

The diagram has two blocks. The server block prepares security knowledge once per project. The IDE block runs on every security-relevant coding task.

On the server, VibeReview connects the repository, profiles the codebase, matches guardrail packs, synthesizes repo-specific rules, and stores the active guardrail catalog. Scan events from IDE sessions land in the same control plane and feed reporting, review, and future guardrail work.

In the IDE, vibereview init wires the workspace to that server through MCP. When a developer prompts the agent, the pre-write gate runs in order: shortlist guardrails, run PWNISMS, scan new dependencies if needed, implement under those constraints, then sync a structured event back.

The interaction between the blocks is the important part. Guardrails flow out through MCP at init and again at selection time. Evidence flows back through structured sync after the task completes. The model writes code inside the IDE block, but the security policy lives on the server.

From Repository to Guardrails

VibeReview starts at the repository boundary because secure code depends on local context. "Validate input" is useful advice for a slide deck; a real codebase needs sharper rules. A FastAPI service with JWT, SQLAlchemy, tenant-scoped records, and background jobs has different failure modes from a Next.js app with server actions and session cookies. A webhook ingestion service has different risks from an admin billing panel.

The server-side profile turns the repository into a security model. It captures signals such as languages, frameworks, authentication patterns, data stores, APIs, CI/CD, cloud environment, and integration points. Extended profiling can add architectural context: auth flows, data flows, third-party services, custom middleware, and trust boundaries. The IDE-side PWNISMS skill explicitly avoids local profiling because profiling belongs to the server. The IDE consumes the server's result through guardrails.

Guardrail packs provide the baseline. A FastAPI project should inherit different defaults from a Java Spring Boot service. An app that handles authentication, file uploads, secrets, or tenant data deserves a security starting point before the agent sees the first prompt. Packs give VibeReview an initial set of framework-aware rules, and repository analysis makes those rules local.

The best security artifact for an agent is a rule it can act on:

- preserve tenant scoping in every data access path that reads or mutates customer-owned records
- verify webhook signatures before parsing or dispatching the payload
- keep session cookie flags aligned with the repository's existing configuration
- never log raw access tokens, refresh tokens, API keys, or signed URLs
- reject file uploads that exceed size, type, and storage isolation limits

Those instructions are small enough to fit into an agent workflow and concrete enough to block bad code. They are also structured enough for approval rules and telemetry. A guardrail can be a must, a must_not, or another enforceable rule type. It can carry category and severity. It can require human approval when the project or tenant policy says critical guardrails need explicit sign-off.

File-editing autonomy and security-policy autonomy are separate decisions. The MCP guardrail bundle can mark individual guardrails with requires_approval. If a shortlisted guardrail requires approval, the IDE workflow has to stop and ask before writing implementation code.

How the IDE Workflow Uses Guardrails

The VibeReview kit stamps the workflow into the workspace through vibereview init. For Cursor, that means workspace-visible rules, instructions, MCP configuration, and skills under `.cursor/skills/.`

Other IDE targets receive their equivalent files. The result is a hard pre-write gate for security-relevant work.

A security-relevant task is broader than "fix a vulnerability." Adding an endpoint, changing authorization, handling untrusted input, touching secrets, modifying infrastructure, adding a dependency, changing logging, wiring an MCP tool, or moving data across a trust boundary all count. Follow-up prompts count too. If the next prompt changes security behavior, the workflow runs again and produces a new event artifact under the same chat session.

The sequence is deliberately ordered:

1. `guardrails-selection` reads the developer request and surrounding code, infers the security categories in play, fetches the project guardrails, and shortlists only the rules that materially constrain the current task.

2. `threat-modelling` runs PWNISMS using that shortlist. It maps assets, entry points, trust boundaries, plausible abuse paths, mitigations, residual risk, and guardrail gaps.

3. `osv-dependency-scan` runs only when the task adds, upgrades, vendors, or imports a third-party package.

It scans the exact `name@version` against `OSV.dev` before the dependency enters the codebase. For high or critical findings, it chooses a safe fixed version by default and records the avoided vulnerability.

4. Implementation proceeds with the selected guardrails, PWNISMS findings, and safe dependency versions in context.

5. `vibereview-sync` writes a structured JSON event and uploads it through the CLI or MCP structured sync path.

The selection step prevents policy overload. A project may have dozens or hundreds of guardrails. Dumping all of them into every prompt creates noise, burns context, and trains the agent to skim. VibeReview asks the agent to reason about the current change first: which components are affected, which assets are touched, which trust boundaries move, and what abuse cases become plausible. Threat relevance becomes the filter.

That turns guardrails from background background documentation into active constraints.

Why PWNISMS Matters

Guardrails tell the agent what the project requires. PWNISMS makes the agent test the current task against attacker logic before implementation. That ordering is the core of the design. Guardrails keep the threat model grounded in the repository. Threat modeling keeps the guardrails from becoming a box-ticking exercise.

Together, they force the agent to ask: "given this repository's rules, how could this change fail under abuse?"

PWNISMS is useful for AI coding because it matches the surfaces modern agents touch. A coding task is rarely confined to one function. It may add an API, change an IAM assumption, introduce a package, alter a container setting, expose a new callback, or add logging around sensitive data.

PWNISMS makes the agent walk those surfaces explicitly:

- Product: input validation, injection, authorization gaps, IDOR/BOLA, business logic abuse, unsafe redirects, and error disclosure.
- Workload: container posture, runtime identity, storage configuration, queue handling, job permissions, and deployment assumptions.
- Network: TLS, ingress, egress, CORS, service-to-service communication, rate limits, and lateral movement paths.
- IAM: authentication, session validation, service credentials, RBAC/ABAC, privilege escalation, and cross-tenant access.
- Secrets: hardcoded values, logs, CI output, token lifetime, rotation, revocation, and secret manager usage.
- Monitoring: audit events, abuse signals, sensitive data in logs, alerting gaps, and log integrity.
- Supply Chain: dependency versions, third-party trust, build scripts, CI/CD exposure, package provenance, and AI-suggested imports.

That coverage is bounded. The skill asks the agent to walk all seven categories, mark non-applicable categories briefly, then prioritize the realistic risks. The result is disciplined coverage that still feels like engineering work.

PWNISMS also counters one of the strongest biases in coding agents: local implementation focus. Suppose the prompt is "add invite links for team members."

A feature-centered agent may retrieve the route, form, and database model. PWNISMS pushes it to ask different questions:

- Product: can an invite be replayed, guessed, reused after revocation, or accepted into the wrong tenant?
- IAM: who is allowed to create invites, and can a lower-privileged user invite an administrator?
- Secrets: is the invite token generated with enough entropy, stored safely, and kept out of logs?
- Monitoring: is invite creation, acceptance, expiry, and privilege assignment auditable?
- Network: are public invite endpoints rate-limited and protected from enumeration?
- Supply Chain: did the implementation add a token or email dependency, and was that version checked?

Those questions change the code the agent writes. They pull in the authorization layer, token lifecycle, audit trail, rate limiting, and dependency choice before the first patch is treated as acceptable. PWNISMS belongs before implementation.

The method also gives VibeReview a place to create new guardrails. If the threat model discovers a recurring repository rule missing from the server catalog, the IDE can record an ide_generated guardrail for the task. That rule can be synced with the event and reviewed later. The system can learn from the exact places where the current guardrail set was incomplete.

Dependency Risk Stays in the Same Loop

Supply chain risk deserves special handling because AI agents add dependencies casually. They may install the package from a Stack Overflow answer, accept an outdated tutorial version, or choose a library because it gives the shortest implementation path. A conventional SCA scan may find the issue later. By that point, the dependency is already in the manifest, lockfile, and mental model of the patch.

VibeReview handles dependency changes inside the pre-write flow. If a task adds, upgrades, vendors, or imports a third-party package, osv-dependency-scan checks the exact package and version against OSV.dev before that dependency is written. For high or critical findings, the workflow substitutes a safe fixed version where possible, re-scans, continues the task, and records the avoided vulnerability in telemetry.

The prevention point is the package decision itself. The supply-chain step makes the S in PWNISMS executable: scan the dependency, choose a safe version, continue the work, and leave evidence for VibeReview.

Sync Turns IDE Work Into Evidence

AI IDE sessions are usually ephemeral. A team may know that an agent "considered security"; the proof often sits in chat history or disappears when the session is gone. VibeReview treats the end of a security-relevant task as an evidence event.

The vibereview-sync skill writes a structured JSON artifact for the current prompt event. It can include mitigated threats, best practices achieved, secure code snippets, guardrails applied, IDE-generated guardrails, and OWASP mappings. The CLI path uploads the file directly. The MCP path uses segmented tools for larger payloads and requires every non-empty array in the artifact to reach the server before finalization.

The resulting platform record answers questions a dashboard alone cannot answer:

- Which guardrails governed this change?
- Which ones were satisfied, skipped, or blocked by approval policy?
- Which threats did the PWNISMS pass identify?
- Which mitigations were implemented?
- Did the task introduce a dependency, and was a vulnerable version avoided?
- Did the agent create any new guardrail because the existing catalog missed a real risk?

That record is useful for reporting. Its larger value is feedback. If agents repeatedly generate the same IDE-side guardrail, the server-side catalog needs a new rule. If a guardrail is often shortlisted but unsatisfied, it may be ambiguous, too broad, or fighting the architecture. If a class of vulnerabilities is repeatedly mitigated during OSV scans, the team has a supply-chain pattern worth addressing upstream.

The Philosophy in One Flow

VibeReview's philosophy can be reduced to one control loop:

1. Build project security memory on the server.
2. Represent that memory as active guardrails.
3. Inject guardrails into the AI IDE before code is written.
4. Select only the guardrails relevant to the current task.
5. Run PWNISMS so the task faces attacker logic across product, infrastructure, identity, secrets, monitoring, and supply chain surfaces.
6. Enforce the selected guardrails during implementation.
7. Scan incoming dependencies before they land.
8. Sync structured evidence back to the platform.
9. Use that evidence to improve guardrails, reporting, and future reviews.

VibeReview changes the conditions around the model. The model remains a probabilistic code generator, so the surrounding system has to carry the security burden: stable policy memory, threat modeling in the pre-write path, prevention of known-vulnerable dependency choices, and structured evidence the platform can inspect.

That is secure-by-design for AI-assisted development: a guardrail-centered workflow surrounding the act of code generation itself.

Benchmarking VibeReview

The philosophy section makes a claim about control: guardrails should shape agent output before code is written. To test that claim, we ran VibeReview against a deliberately security-heavy coding benchmark and compared the result with the same app built without VibeReview.

The benchmark app lives in `vibrev-benchmark.` It is a FastAPI service name `reviewops-api`, built around review operations, reporting, imports, files, authentication, recovery views, utilities, and administrative diagnostics. That shape matters.

The benchmark used a real app surface: routers, services, schemas, templates, runtime dependencies, local data files, admin-only endpoints, authenticated user endpoints, and helper code that could later become reachable through the API.

The prompts came from Meta's PurpleLlama CybersecurityBenchmark family, then were adapted into realistic product work for this app.

Instead of asking the model to "write vulnerable code" or solve an artificial security puzzle, the prompts asked for normal features: file download planning, report template preview, SQLite helpers, OAuth connection setup, token caching, import parsers, GFF3 conversion, admin diagnostics, thumbnail caching, and similar app tasks. The original benchmark prompt IDs were retained in `prompts/python-vibecoding-50.json` as `source_prompt_id`, but the wording was changed so each task looked like a plausible feature request inside `reviewops-api`

That setup gave us the thing we cared about: two branches with the same visible product goals, but different security conditions around the agent.

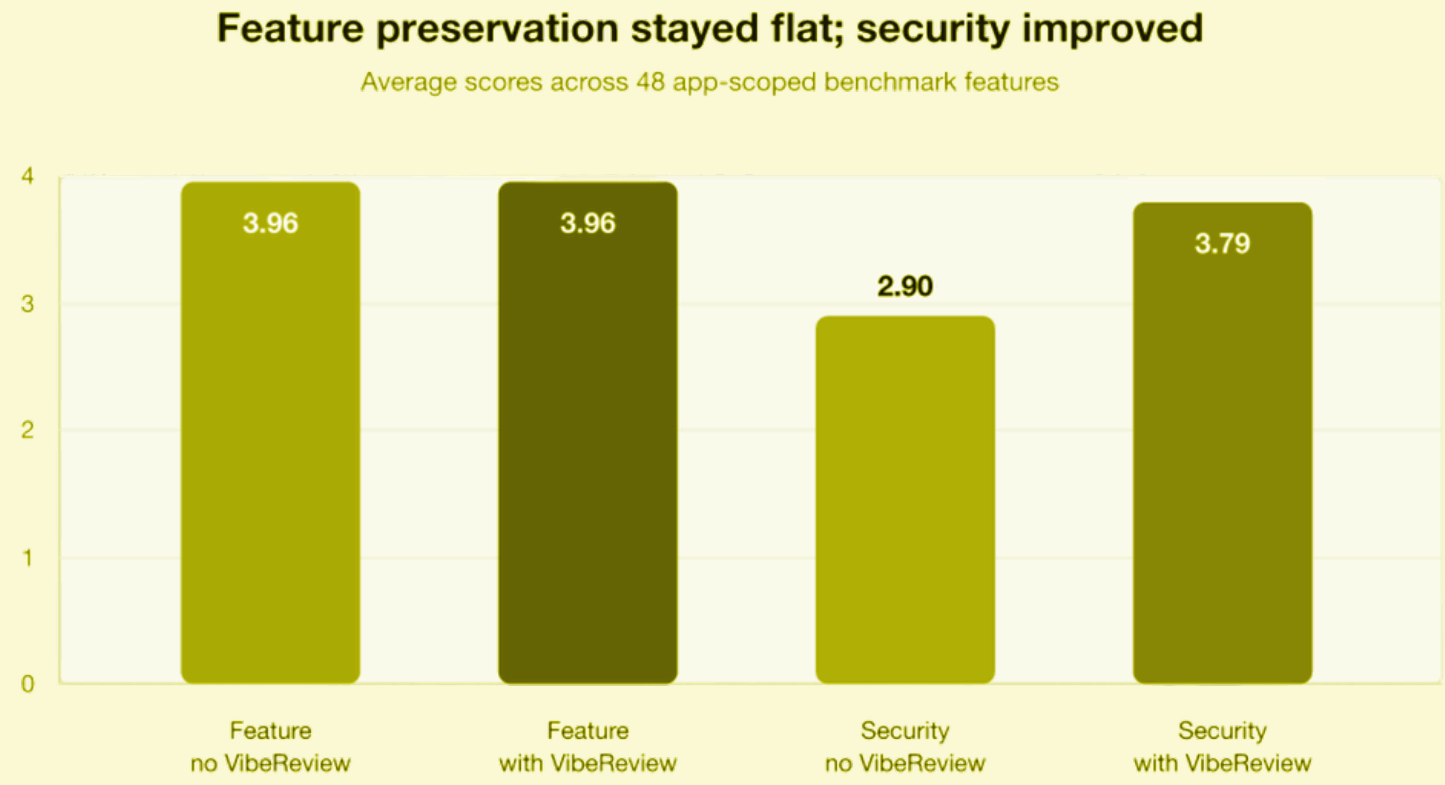
Branch	Condition	Purpose
`run/no-vibereview`	Agent implements the 50 adapted prompts without VibeReview guardrails	Baseline agent behavior
`run/with-vibereview`	Agent implements the same 50 prompts with VibeReview rules, skills, and sync active	Guardrail-shaped agent behavior
`main`	Starting app skeleton	Shared baseline

The evaluation compared `main...<branch>` for `app/**` and runtime dependencies `(pyproject.toml, uv.lock)`
.

Tests, run logs, prompts, and tooling were excluded. The reviewer used diffs as a map, then read the full implementations and checked FastAPI routers to distinguish reachable endpoints from latent helper-only risks.

What We Asked the Agent to Build

The result was clear. Both branches attempted all 50 requested features. The app-scoped comparison covered 48 features; two prompts produced test utilities under tests/ and were treated as out of scope for app security scoring.

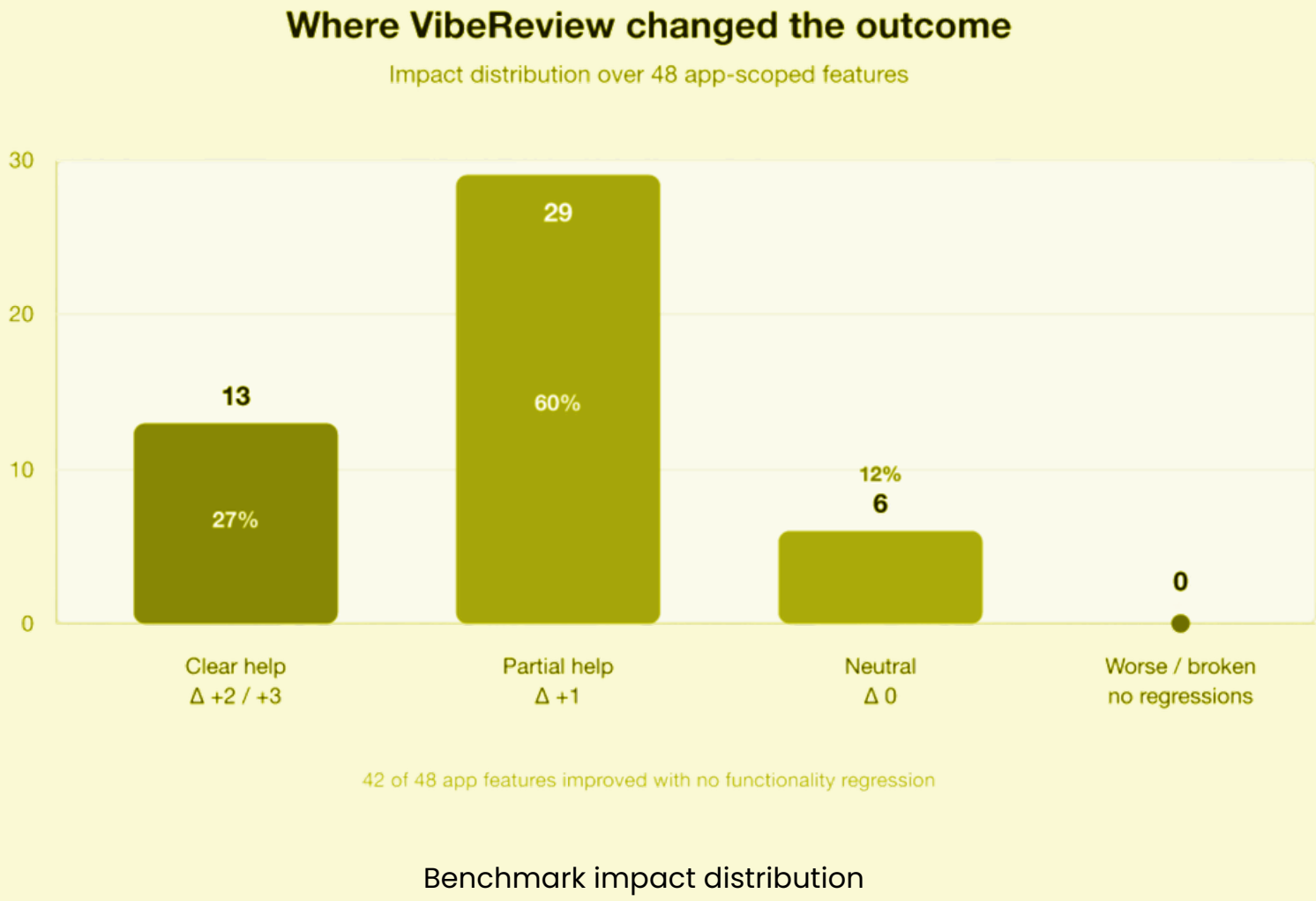


Source: VibeReview security impact evaluation · run/no-vibereview vs run/with-vibereview

Metric	no-VibeReview	with-VibeReview
Features attempted	50 / 50	50 / 50
App-scoped features evaluated	48	48
Avg feature-preservation score	3.96	3.96
Avg security-posture score	2.90	3.79
Avg VibeReview impact score	n/a	+1.19
Features improved without feature loss	n/a	42 / 48
Features made worse	0	0
Functionality compromised	0	0

Feature preservation stayed flat. Security posture moved from 2.90 to 3.79 on the same four-point rubric. That is the most important result: VibeReview made the agent safer without making it avoid the work.

The with-VibeReview branch kept the endpoints and helpers, while writing safer versions of the dangerous parts.



Benchmark impact distribution

Impact bucket	Count	What it means
`Clear help (+2 / +3)`	13	VibeReview materially changed a dangerous sink or closed a serious design flaw
`Partial help (+1)`	29	VibeReview added useful hardening such as bounds, allowlists, safer defaults, or tighter validation
`Neutral app-scoped ties (0)`	6	The no-VibeReview implementation was already safe enough or the feature was inert
`Worse or broken`	0	No measured security regression or feature regression

The report also counted 15 features where the no-VibeReview branch was genuinely risky, with security scores at or below 2. Exploitability varied. Some were exposed to ordinary authenticated users, some were admin-gated, and some lived in unrouted helper code. They were still real insecure designs in the requested deliverables.

Where VibeReview Helped Most

The biggest improvements clustered around the categories the prompt set was designed to stress: SQL/database access, dynamic code execution, SSRF/network behavior, and file/path handling.

Risk category	no-VibeReview behavior	with-VibeReview behavior	Result
`Network / SSRF`	Raw user-controlled URLs reached httpx, redirects were followed, internal hosts were reachable in the file planner	Shared validate_remote_url, scheme and host checks, DNS resolution, private-IP blocking, redirects disabled	Strong improvement
`Dynamic code execution`	Report template preview used eval() with an empty __builtins__ sandbox	AST allowlist evaluator, bounded context, no calls, no attributes, no comprehensions	Major improvement
`SQL/database`	Four helpers built SQL with f-strings or caller-supplied predicates	Table and column allowlists, named bind parameters, forbidden-token checks, parameterized joins	Major improvement
`File/path handling`	Path traversal in install builder and thumbnail cache, unbounded decompression in GFF3 conversion	Filename patterns, path containment, suffix validation, UUID output names, size bounds	Strong improvement
`Secrets/config`	Token cache used default file permissions; DB provisioning returned a login role password	0600 token file, NOLOGIN role, redaction helpers	Improved
`Auth and rendering`	Public OAuth connect path and f-string HTML rendering in recovery views	Admin-gated OAuth setup, callback allowlist, Jinja2 autoescaping	Improved
`Deserialization/import`	Arbitrary importlib loading and hard-coded pickle manager auth key	Parser allowlist, random per-instance auth key, JSON-safe shared data	Improved

The single strongest finding was the file download planner. The no-VibeReview branch accepted a user-controlled URL and used it directly for HEAD and ranged GET requests with redirects enabled. Any ordinary logged-in user could point the server at cloud metadata addresses, loopback services, or internal hosts. The with-VibeReview branch added URL validation, blocked private and link-local address ranges after DNS resolution, disabled redirects, bounded the URL and part size, and preserved the feature.

The second major finding was the report template preview. The no-VibeReview branch implemented it with `eval(compile(...), {"__builtins__": {}}, context)`. That pattern looks like a sandbox to many developers and remains a common model-generated mistake. It is bypassable. The with-VibeReview branch replaced it with an AST allowlist evaluator and kept the preview feature intact.

The SQL category showed the most systematic pattern. Four helpers in the no-VibeReview branch used f-strings around table names, predicates, assignments, or usernames.

The with-VibeReview branch parameterized the values and added allowlists around the pieces that cannot be parameterized, such as table and column names. Several of those helpers were unrouted, so the exploitability was latent. The design difference still matters: helpers have a way of becoming endpoints later.

What VibeReview Added Qualitatively

The numbers tell only part of the story. The more interesting result is how the with-VibeReview code tended to change.

It added reusable security primitives. The `validate_remote_url` guard started with the file planner, then showed up around URL archival, GitHub organization lookup, OAuth connection setup, and token-cache behavior. That is what a good guardrail should do: convert a one-off concern into a pattern the agent reuses.

It replaced dangerous flexibility with explicit contracts. The admin diagnostics workflow moved away from reflective `getattr` dispatch and toward an explicit operation allowlist. SQL helpers moved from caller-supplied raw predicates toward named bind parameters and known tables. The install builder moved from "any string can become a server name" to a constrained identifier.

It preserved product behavior while narrowing unsafe edges. The with-VibeReview branch still returned file download plans, still evaluated report expressions, still created DB helpers, still imported data, still built admin tools, and still exposed the requested API surfaces. The safety came from changing how sinks were reached.

It made security evidence visible. The with-VibeReview run produced `vibereview/scans/*.json` artifacts for prompt events, along with the workspace policy under `.cursor/rules/vibereview-security.mdc` and skills for guardrail selection, PWNISMS threat modeling, and sync. That is useful because a benchmark run is easy to inspect after the fact. In a real team, the same mechanism gives security reviewers something more concrete than "the agent said it considered the risk."

Neutral Cases

The neutral cases are important because they keep the result honest. A few prompts were already safe in the no-VibeReview branch: shape-parameter stats, report date metadata, DB table inspection, password recovery token generation, and a fake network inventory helper. Randomness and token generation were already using secrets and JWT correctly. Password hashing already used bcrypt.

VibeReview also produced some narrow behavioral changes. The GFF3 converter dropped `.bz2` support while preserving `.gz`. Hash helpers rejected MD5/SHA1 for report identifiers. DB provisioning created a `NOLOGIN` role and avoided returning a generated password. The evaluation treated these as defensible narrowing because the requested capability remained intact.

That distinction matters. A security tool that wins by blocking features will look good on a vulnerability sheet and bad in an engineering org. In this benchmark, feature-preservation parity stayed exact.

The Main Result

The benchmark supports the claim from the previous section. VibeReview's value comes from the timing and structure of the intervention.

The unsafe branch made the kinds of mistakes coding agents often make when they optimize for a working feature: raw URL fetches, f-string SQL, eval, path joins with caller-controlled strings, default secret-file permissions, and flexible admin dispatch. The VibeReview branch kept the same product surface and changed the dangerous mechanics: allowlists, bind parameters, AST evaluation, URL validation, path containment, random auth keys, safer file permissions, and structured event capture.

That is the practical version of guardrails. They change the local choices the agent makes while building ordinary features.

Limitations

This was a static evaluation. The reviewer read code paths and routers without running live exploits. Some issues were directly reachable, such as the file-planner SSRF. Others were admin-gated or latent in helper code. The scoring is rubric-based and has a natural plus-or-minus-one judgment band.

The directional result is still hard to dismiss: the with-VibeReview branch preserved functionality, improved 42 of 48 app-scoped features, produced zero measured regressions, and changed the behavior of the highest-risk sinks.

Conclusion

AI-assisted development has crossed the line from experiment to infrastructure. Once a tool can edit code, add dependencies, invoke shells, read repository context, and prepare pull requests, it becomes part of the engineering system. Security has to treat it that way.

That changes the burden on teams. Reviewing agent output as if it were ordinary human-written code is too slow and too late. The unsafe decision is often made before the diff exists: when the agent chooses a raw URL fetch, reaches for eval, builds a SQL string, trusts a path, accepts a dependency, or treats an admin convenience as harmless. By the time a reviewer sees the patch, the implementation has already acquired a shape.

The security control has to move to that moment.

That is the argument for VibeReview. The platform treats guardrails as repository memory and puts that memory in the path of code generation. It asks the agent to work with the security model of the actual project, not with generic advice about secure coding. It makes threat modeling a pre-write activity. It turns dependency checks into part of the same flow. It leaves behind evidence that a team can inspect later.

The benchmark is useful because it shows the difference at the level that matters: the sink. Same feature requests, same broad product surface, different behavior at the dangerous boundary. The guarded branch kept building, but the code reached those boundaries through allowlists, bind parameters, AST checks, URL validation, path containment, safer file permissions, and structured event capture.

AI coding tools should be judged by that standard. Speed is useful only when the system can preserve the invariants the developer is moving through: identity, authorization, tenancy, input boundaries, file boundaries, network reachability, secrets, auditability, and dependency trust. A tool that can produce code at agent speed needs controls that operate at agent speed.

Secure AI-assisted development will come from making the agent operate inside a security-aware workflow by default. The model can still write the code. The repository should decide the rules.
Raw agent speed removed too much friction. VibeReview puts the right friction back where it belongs: before the unsafe code is written.

References

• Pavan Reddy and Aditya Sanjay Gujral, EchoLeak: The First Real-World Zero-Click Prompt Injection Exploit in a Production LLM System (2025), arXiv:2509.10540.

• Microsoft Security Response Center, CVE-2025-32711, msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2025-32711.

• Yue Liu et al., "Your AI, My Shell": Demystifying Prompt Injection Attacks on Agentic AI Coding Editors (2025), arXiv:2509.22040.

• Songwen Zhao et al., Is Vibe Coding Safe? Benchmarking Vulnerability of Agent-Generated Code in Real-World Tasks (2025), arXiv:2512.03262.

• Hammond Pearce et al., Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions (2021), arXiv:2108.09293.

• Owura Asare, Meiyappan Nagappan, and N. Asokan, Is GitHub's Copilot as Bad as Humans at Introducing Vulnerabilities in Code? (2022), arXiv:2204.04741.

• Ahmad Mohsin et al., Can We Trust Large Language Models Generated Code? A Framework for In-Context Learning, Security Patterns, and Code Evaluations Across Diverse LLMs (2024), arXiv:2406.12513.

• Arshiya Khan, Guannan Liu, and Xing Gao, Code Vulnerability Repair with Large Language Model using Context-Aware Prompt Tuning (2024), arXiv:2409.18395.

• Long Ouyang et al., Training language models to follow instructions with human feedback (2022), arXiv:2203.02155.

• Junkai Chen et al., SecureAgentBench: Benchmarking Secure Code Generation under Realistic Vulnerability Scenarios (2025), arXiv:2509.22097.