# AI Generated Code Review A Practitioner Security Methodology

A mature methodology for AI generated code review goes beyond a generic OWASP checklist and matches the review technique to the code's function: threat model authentication and payment logic explicitly, chain multiple SAST and SCA tools rather than relying on one, gate AIflagged pull requests in CI/CD before human review and reserve manual attention for the vulnerability classes automated tools consistently miss business logic errors, contextual authorization and multistep data flow across files. Treating every AIgenerated function with an identical shallow pass is the main reason experienced teams still ship preventable flaws.
Most guidance on reviewing AI generated code including our own step by step AI generated code review process is written to establish a solid general baseline. That is a reasonable floor for teams just getting started, but it is insufficient as a standalone methodology for practitioners reviewing production systems at volume. The categories that actually cause incidents in AI generated code server side request forgery, insecure deserialization, business logic bypass, race conditions in concurrent access sit outside the basics most checklists cover and they require a review technique tailored to how AI models actually fail, not a generic pass.


## Why a Generic Checklist Is not Enough

AI models fail in patterned, predictable ways, but those patterns are not evenly distributed across vulnerability classes. Research into confirmed real world flaws in AI generated code has repeatedly found that server side request forgery and injection family weaknesses: path

traversal, command injection, NoSQL injection account for a disproportionate share of serious findings, far more than a flat OWASP Top 10 pass would suggest. A methodology built around equal attention to all ten categories spends too much time on lower frequency issues and not enough on the handful of patterns doing most of the damage.
The other gap in generic checklists is codetype blindness. A checklist item like "check for broken access control" means something different for an internal admin tool than for a public facing payment endpoint. Practitioners need a methodology that scales review depth to the sensitivity and exposure of the code being reviewed, not a flat pass applied identically everywhere.

## Threat Modeling AIGenerated Code by Function

Rather than reviewing every AIgenerated function the same way, categorize it first and apply a review depth appropriate to that category.
Authentication and session logic deserves the deepest scrutiny of any category, because AI models frequently generate flows that look at complete password checks, token generation, session handling while missing a single enforcement point that renders the rest irrelevant. Explicit threatmodel: can a request bypass this check by hitting a different endpoint, reusing a token past its intended lifetime, or exploiting a timing difference in comparison logic?
Authorization and access control requires reasoning the model rarely performs correctly, because ownership and permission rules are usually implicit in your system's architecture rather than explicit in any single prompt. Enterprise analysis of AI assisted development at scale found privilege escalation paths appearing 322% more often than in human written code within the same codebases, a gap wide enough that authorization logic should never be accepted on the strength of a single automated pass.
Data handling and external requests: anything constructing a URL, file path, or query from a variable should be reviewed specifically for SSRF and injection patterns, since these categories have shown up as the single most common flaw types in confirmed AI generated code vulnerabilities in independent research.
Serialization and deserialization logic deserves specific attention that generic checklists tend to skip entirely. AI models trained on a mix of old and current code sometimes reproduce deserialization patterns that were acceptable years ago and are now well understood attack vectors for remote code execution.
Concurrency and state matters most in code handling shared resources, queues, or multistep transactions. AIgenerated code is prompted and generates one function at a time, which makes it especially prone to race conditions that only appear when multiple requests interact with shared state simultaneously, a failure mode a single function review will never surface.

## Building AI Code Review Into CI/CD

Methodology only scales if it is enforced structurally rather than left to individual reviewer discipline. A practical CI/CD gate for AI generated code includes: automatically flagging commits or pull requests that touch AIassisted files (many editors and IDE integrations expose this metadata), requiring static and dependency analysis to pass before a human reviewer is

even assigned and blocking merge on any of the high severity categories injection, exposed secrets, missing authorization on new endpoints regardless of how clean the rest of the diff looks.

This structural approach matters because manual discipline degrades under deadline pressure in ways automated gates do not. A reviewer three pull requests behind schedule is more likely to skim; a CI gate blocking merge on a flagged dependency doesn't get tired. The same logic underlies a proper web application security testing program: test coverage that depends entirely on manual effort erodes over time, while automated gates hold the line consistently.

It's also worth tracking, at a team level, which categories of AIflagged issues recur most often. If your team's tool chain repeatedly catches the same class of missing authorization check across different AI generated pull requests, that is a signal worth acting on directly either through a shared prompt template that explicitly requests authorization logic, or through targeted training rather than relying on review alone to catch the same gap indefinitely.

## Vulnerability Classes That Deserve Extra Attention

Beyond the OWASP basics, a few specific patterns recur often enough in AIgenerated code to warrant explicit methodology rather than incidental discovery.

Business logic by passing a model asked to implement a discount, refund, or access tier rule may implement the happy path correctly while leaving an obvious sequencing exploit open, such as applying a discount code multiple times or accessing a premium feature by manipulating a clientside flag the server trusts unconditionally.


Insecure defaults left over from scaffolding debug modes, verbose error messages and permissive CORS configurations that AI tools generate as reasonable defaults for a working prototype but that should never survive into a reviewed, productionbound pull request. These sit

adjacent to the broader set of web app security threats that a mature program tracks continuously rather than catching only at review time.

Crossboundary validation gaps input validated correctly in one function but reused, unvalidated, by a second function elsewhere in the same file or a related module. This is one of the clearest arguments for tracing data flow manually rather than reviewing functions in isolation, since AI models generate each piece correctly without reasoning about the full path data takes through the system.

Injectable output as well as input practitioners sometimes focus exclusively on inbound injection and underweight output encoding, which is where a large share of crosssite scripting vulnerabilities actually originate. Our detailed guide on how to prevent XSS attacks covers the specific encoding and sanitization patterns worth checking in AIgenerated templating and rendering code.

Cryptographic misuse and weak randomness AI models trained on years of public code sometimes default to older cryptographic primitives, fixed initialization vectors, or noncryptographic random number generators used for security sensitive values like tokens or password reset codes. This category is easy to miss precisely because the code runs correctly and produces output that looks like a properly generated secret; the flaw only becomes apparent when you check the specific function being used, not the output it produces.

Error handling that leaks internal state a generated function that returns a full stack trace, database error message, or internal file path to the client on failure. This rarely breaks functional tests, since tests usually check the happy path, which is exactly why it survives casual review and only surfaces when a practitioner deliberately triggers failure conditions during testing.

## Documenting Findings and Closing the Feedback Loop

A methodology that stops at finding vulnerabilities misses half the value practitioners can extract from the process. Documenting which categories of issues recur across AI generated pull requests, not just fixing them individually, turns review into a feedback loop that improves the next round of AIassisted development rather than repeating the same catch indefinitely.

In practice, this means tagging findings by category (missing authorization, hallucinated dependency, SSRF, business logic bypass) in whatever tracking system your team already uses and reviewing that tagged data periodically rather than only at incident postmortems. If authorization gaps keep surfacing in code generated from a particular prompt pattern or a particular AI tool, that's a signal worth acting on directly, whether through an adjusted prompt template that explicitly requests authorization logic or through a targeted training session focused on that exact gap rather than general AI code review awareness.

## Adjusting Methodology for Multi File and Agentic Generation

Everything above assumes reviewing a single AIgenerated function or file, but agentic coding tools now routinely generate, edit and apply changes across multiple files in a single pass, which changes the methodology in a meaningful way. A single function review technique doesn't scale to a change that touches a route handler, a database model and a configuration file

simultaneously, because the vulnerability might live in the interaction between those files rather than in any one of them individually.
This is closest in spirit to the risks covered in our breakdown of vibe coding security risks, where an entire feature gets prompted into existence rather than reviewed function by function. The practitioner adjustment is to review the full diff as a system change before reviewing individual files in isolation: map which files touch authentication, data access, or external requests first, apply the deepest scrutiny there and only then move to reviewing each file's internal logic. Reviewing files in isolation, in the order they happen to appear in a diff, is how crossfile authorization gaps and inconsistent validation slip through even a technically rigorous singlefile methodology.

## Practicing the Methodology Deliberately

A methodology only becomes reliable through repetition against real, exploitable code reading about SSRF or insecure deserialization in the abstract builds far less pattern recognition than finding and exploiting the same flaw in a controlled environment. Practitioners sharpening this skill often work through structured, handson labs. Our own sqlinjection exercises are built around exactly this kind of deliberate practice, isolating a single vulnerability class so the pattern becomes immediately recognizable in unfamiliar code later.
This same instinct assumes nothing, verifies everything, looking for what a defender missed is the connective tissue between AI code review and adjacent disciplines like bug bounty hunting, where practitioners apply identical reasoning to production applications rather than AI generated pull requests specifically. Teams building internal review competency at scale often draw directly on web security best practices already established for human written code, adapting emphasis rather than starting from scratch, since the underlying vulnerability classes haven't changed only the volume and speed at which they now appear.

## Conclusion

Practitioners looking to keep these instincts sharp outside of daytoday review work often use handson platforms like AppSecMaster to practice against deliberately vulnerable applications between projects, which keeps pattern recognition current as both AI tooling and attacker techniques continue to shift.
A practitioner-grade methodology for reviewing AI generated code treats the OWASP Top 10 as a floor, not a ceiling. Threat model by code function rather than applying identical scrutiny everywhere, chain multiple tools rather than trusting one scanner's clean result, enforce structural gates in CI/CD rather than relying on reviewer discipline alone and give specific, deliberate attention to the classes SSRF, business logic bypass, crossboundary validation gaps that generic checklists consistently underweight. The teams catching the most real vulnerabilities in AI generated code aren't reading more carefully; they are reading differently, with a methodology matched to how these models actually fail.

# Frequently Asked Questions (FAQs)

## Is the OWASP Top 10 sufficient for reviewing AI generated code?

It is a useful floor but not a complete methodology. Confirmed vulnerability research shows certain classes, like SSRF and injection family flaws, appear disproportionately often in AI generated code, meaning a flat pass across all ten categories misallocates review attention.

## How many static analysis tools should a mature program run?

‘More than one. Research comparing multiple SAST tools against confirmed AIgenerated vulnerabilities found a large share of real issues caught by only a single tool, meaning single tool coverage systematically misses findings that a second engine would catch.

## Should AIflagged pull requests get different CI/CD treatment than regular ones?

Yes. Gating AIassisted commits behind mandatory static and dependency analysis before human review rather than treating them identically to handwritten commits catches structural issues before a reviewer's time is spent on them.

## What vulnerability class do experienced reviewers most often underweight?

Business logic bypass and crossboundary validation gaps, since neither shows up as a clean pattern match for automated scanners. Both require manually tracing intent and data flow rather than patternmatching known bad code.

## Does reviewing AI generated code require different skills than reviewing human written code?

The underlying vulnerability knowledge is the same. What differs is emphasis and technique verifying dependencies exist, tracing crossfunction validation explicitly and threat modeling by code function rather than reading for general code quality.