Journal of Sustainable Smart Systems in Education & Environment

Journal of Sustainable Smart Systems in Education & Environment (JSSSEE) | eISSN: 2979-143X

The Double-Edged Sword of AI Pair Programmers: A Systematic Literature Review of Security Vulnerabilities in AI-Generated Code and Agentic Development Environments

Mahmoud E. Farfoura, Cybersecurity Department, Faculty of Science and Information Technology, Al-Zaytoonah University of Jordan, Amman, Jordan, Email: m.farfoura@zuj.edu.jo, ORCID: [0000-0002-9010-6989]

Mohammad Alia, Cybersecurity Department, Faculty of Science and Information Technology, Al-Zaytoonah University of Jordan, Amman, Jordan, Email: dr.m.alia@zuj.edu.jo, ORCID: [0000-0002-1821-7756]

Ibrahim Mashal, Cybersecurity Department, Faculty of Science and Information Technology, Al-Zaytoonah University of Jordan, Amman, Jordan, Email: i.mashal@zuj.edu.jo, ORCID: [0000-0002-8899-4155]

Adnan Hnaif, Cybersecurity Department, Faculty of Science and Information Technology, Al-Zaytoonah University of Jordan, Amman, Jordan, Email: adnan_hnaif@zuj.edu.jo, ORCID: [0000-0001-8523-4321]

Mahmoud Odeh, Department of Cybersecurity, Faculty of Information Systems, Zarqa University, Jordan, Email: Modeh@zu.edu.jo, ORCID: [0000-0002-2949-0475]

Corresponding Author: Mahmoud E. Farfoura, Email: m.farfoura@zuj.edu.jo

JSSSEE | eISSN: 2979-143X | Vol. 1 No. 2 (2026) | OPEN ACCESS
https://doi.org/10.66823/2egmd793

Abstract

The role of AI pair programmers has expanded from local code completion to active participation in the development environment. Contemporary tools can interpret repository context, edit multiple files, call package managers, execute terminal commands, and communicate with external services. This review synthesizes security evidence concerning GitHub Copilot, ChatGPT-based coding, code large language models, Cursor-style agentic editors, command-line coding agents, and Model Context Protocol ecosystems. A protocol-driven search, supplemented by backward and forward snowballing, covered literature and technical evidence available through 12 July 2026. The verified corpus comprised 216 verified records, including peer-reviewed studies, preprints, benchmarks, standards, and clearly identified technical disclosures. The evidence supports useful roles in vulnerability discovery and repair, test generation, and secure-coding guidance, but it also documents recurring injection flaws, unsafe memory and file handling, weak cryptography, authentication errors, secret

exposure, hallucinated dependencies, and incomplete patches. Whether these weaknesses persist depends partly on human factors, including expertise, prompt framing, review effort, automation bias, and the authority users assign to the assistant. Most generated-code defects remain familiar CWE classes. Agentic systems add orchestration-level concerns: indirect prompt injection, context poisoning, tool and protocol supply-chain attacks, permission amplification, approval spoofing, cross-file persistence, and autonomous execution. We synthesize these findings in a unified taxonomy, an evidence-based threat model, a lifecycle security model, and a continuous-assurance architecture built on structured context, least privilege, sandboxing, provenance, conventional SAST/SCA/secret scanning, security tests, and mandatory human authorization for high-impact actions. Without such governance, faster production may be offset by accumulating validation debt and downstream incident risk.

Keywords: AI-assisted software development, GitHub Copilot, Cursor, Code large language models, Secure code generation, Prompt injection, agentic AI, Model Context Protocol, Software supply chain, Systematic literature review.

1. Introduction

Generative AI is moving into the control plane of software development. Early assistants offered local suggestions, often no more than a line or a function. Current systems can retrieve repository context, revise several files, run tests, call package managers, and execute commands. This wider role gives errors a wider reach. A vulnerable completion may remain confined to one function; an agentic error may alter a build script, install a package, expose credentials, or create a persistent rule that shapes later sessions. Security evaluation must therefore look beyond the generated snippet. It should also examine context selection, the trust assigned to instructions, tool access, approval behavior, and independent verification of the resulting changes.

Empirical studies consistently show that credible, compilable code may still be exploitable. Pearce et al. demonstrated this problem in CWE-grounded scenarios, while later comparisons of ChatGPT, Copilot, and other models associated security outcomes with prompt quality, programming language, model family, temperature, and scenario complexity [35]-[50]. Human behavior is just as influential in the resulting risk. Developers using assistants may complete tasks sooner yet examine unfamiliar code less carefully, place excessive trust in model competence, or accept unsafe suggestions [37]-[39]. Thus, rapid generation may move inadequately examined code into testing and review, even when the same tools introduce secure patterns early.

LLMs have also been applied to vulnerability detection, localization, explanation, repair, penetration testing, and test generation. The reported gains are uneven. Performance on curated examples often declines for repository-scale or semantically complex defects, self-correction may yield incomplete patches or alter intended behavior, and conventional static analysis continues to identify defects missed by language models [51]-[105]. Model output remains untrusted code, and agent actions remain privileged operations.

We make five contributions. The review consolidates evidence on code completion, conversational generation, repository-aware assistants, and agentic IDE and CLI systems. Established software weaknesses are distinguished from orchestration-layer attacks, and model and tooling risks are connected with human factors. The available mitigation evidence is organized as a continuous-assurance architecture. A research agenda is set out for Cursor-like environments, coding agents, and MCP-enabled tool ecosystems, for which the peer-reviewed record remains less mature than the Copilot literature.

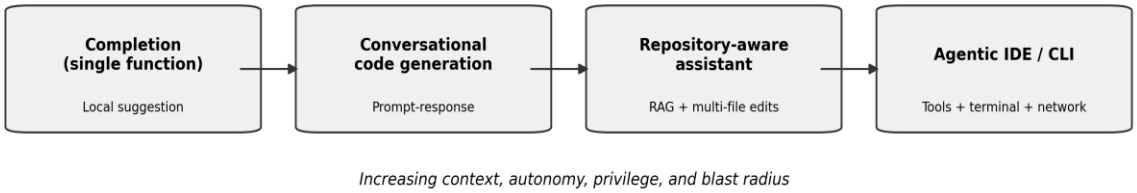

Figure 1: Evolution of AI-assisted software development and the corresponding increase in context, autonomy, privilege, and blast radius.

2. Background And Conceptual Scope

2.1 From Code Models to Agentic Development Environments

Modern code assistants build on pretrained language models specialized through code corpora, instruction tuning, reinforcement learning, infilling objectives, tool use, and repository retrieval. Codex, CodeGen, InCoder, AlphaCode, Code Llama, StarCoder, WizardCoder, CodeT5, CodeBERT, GraphCodeBERT, PLBART, and CodeRL established the trajectory from token completion to synthesis, repair, and reasoning over code [15]-[27]. CodeXGLUE provided common understanding and generation tasks, while SWE-bench and software-engineering agents shifted evaluation from isolated functions to issue resolution in real repositories [28]-[34]. This shift is security-relevant because repository context and execution tools expand the trusted computing base beyond the model.

A product may fall into more than one category depending on its mode and configuration. Throughout this paper, we use four operational categories defined by capability rather than brand. Completion assistants propose local code from nearby context. Conversational generators respond to explicit prompts, often outside an IDE. Repository-aware assistants retrieve information across files and make corresponding edits. Agentic development environments additionally plan multistep work and operate tools such as terminals, browsers, Git, package managers, cloud APIs, or MCP servers.

2.2 What Counts as a Security Failure?

We classify as a security failure any model output, context-selection decision, or tool action that weakens confidentiality, integrity, availability, authenticity, accountability, or supply-chain trust. Code-layer failures generally correspond to established CWE categories. Orchestration failures need not contain a vulnerable generated line: an untrusted README can redirect the agent, a poisoned MCP description can shape a tool call, and a malicious repository rule can trigger secret exfiltration. The distinction matters operationally because conventional scanners may detect the code defect while remaining silent about the orchestration failure.

2.3 Shift Left, Shift Right, and Continuous Assurance

AI-assisted development needs controls throughout the lifecycle: policies on what can be prompted for, generation within constraints, verification before merging, bounds on agent execution, and deploy-time telemetry. It is a natural extension of the idea of "shifting left" (moving security-relevant activities to earlier points in the lifecycle), which already describes practices like security requirements specification, threat modeling, secure design, static analysis, dependency checking, and testing. While I can help shift left by automatically generating tests, understanding and responding to warnings, proposing safer APIs, and fixing defects while

authoring code, it threatens to do the opposite when the rate of code production vastly exceeds thoughtful human review. The result is an argument for continuous assurance, which is consistent with existing secure-software and risk-management practices [163] - [167].

3. Review Methodology

The review followed established software-engineering guidance for systematic reviews and mapping studies. PRISMA concepts were used to make study identification, screening, and synthesis transparent [1]-[4]. The search strings covered AI code generation, Copilot, Cursor, coding agents, code LLMs, secure code, CWE, vulnerability, prompt injection, repository context, terminal execution, MCP, and software supply chain. Searches were completed on 12 July 2026. We covered eight databases and portals: IEEE Xplore, ACM Digital Library, USENIX, SpringerLink, ScienceDirect, arXiv, Google Scholar, and authoritative portals for standards and vulnerability disclosures. We then used backward and forward snowballing from earlier systematic reviews and major empirical studies [5]-[13]. The final reference base contained 216 records after identifiable preprint and published versions of the same work were merged. Claims described as established were grounded in peer-reviewed research. Preprints were retained mainly for rapidly changing areas, particularly agentic IDEs and MCP security, where the publication record remains immature. Standards and benchmarks supplied control definitions, while technical disclosures documented concrete attack paths in products. Vendor disclosures served this narrower purpose and were not given the same evidentiary weight as replicated academic findings.

Table 1: Review Questions And Synthesis Logic

RQ	Question	Primary evidence
RQ1	Which vulnerability types occur in AI-generated code?	CWE-grounded experiments, repository mining, SAST and dynamic testing.
RQ2	Which model, prompt, language, and task factors affect security?	Cross-model benchmarks, prompt studies, ablations, secure tuning.
RQ3	How do developer trust and workflow affect realized risk?	Controlled user studies, field studies, surveys, interaction analyses.
RQ4	Can LLMs improve vulnerability detection and repair?	Detection, localization, explanation, patching, validation studies.
RQ5	What changes in Cursor-like and agentic environments?	Prompt-injection studies, agent benchmarks, MCP research, disclosures.
RQ6	Which mitigations are supported by evidence?	Secure generation, SAST/SCA, sandboxing, privilege control, governance.

Studies were eligible when they examined code generation, AI-assisted programming, vulnerability detection or repair, developer interaction with coding assistants, agent security, or a software-security control relevant to these environments. We excluded duplicates, unsupported opinion pieces, generic LLM-safety studies without a transferable mechanism, and work in which ChatGPT was used only to edit prose. Quality appraisal considered the clarity of the research questions but did not stop there. We also examined whether prompts and model versions were reported well enough for reproduction, whether the tests and vulnerability or security oracles supported the stated claims, and whether sampling leakage was controlled. Repository realism, nondeterminism, and the alignment between evaluation design and security conclusion were considered as well.

Reporting a single insecurity rate would obscure rather than summarize this literature. The studies differ in task, model, temperature, programming language, vulnerability definition, and outcome measure; those differences are part of the evidence, not incidental noise. We therefore used narrative and taxonomic synthesis. Recurring mechanisms were traced across studies, claims were interpreted in relation to their designs, and findings that remain preliminary or disputed were kept separate from the more stable conclusions.

4. Evidence Landscape

4.1 Generated-Code Security

Across the empirical literature, code that compiles and appears syntactically correct cannot be assumed to be secure. Evaluations have used CWE scenarios, natural-language programming tasks, GitHub-mined code, web-development exercises, Python and C/C++ problems, smart contracts, and language-specific security suites [35]-[50]. The reported rates vary, as do prompts, model snapshots, sampling parameters, libraries, and security oracles. Even with this methodological diversity, the same families of weakness continue to surface: injection, unsafe resource management, file and path errors, misuse of cryptography, weak authentication, missing authorization, insecure defaults, and inadequate error handling. The variation is therefore meaningful, but it does not erase the recurrence of these underlying defects.

Higher model capability creates an awkward security tradeoff. Output may be more functional and more persuasive at the same time, making unsafe code less likely to attract suspicion during a quick review. Capability alone is therefore not evidence of safety. Securityoriented prompting and instruction tuning can reduce particular weakness classes, but reliable generalization has not been demonstrated [52]-[55]. Benchmarks such as SecurityEval, CodeLMSec, FormAI-v2, SALLM, and CyberSecEval improve repeatability. Their results still need careful interpretation because contamination, synthetic tasks, changing hosted models, and inconsistent CWE labels can affect the reported scores [41]-[45], [181].

4.2 Detection, Review, and Repair

Repository-scale analysis depends on information that an isolated snippet seldom provides. Reliable judgments may require interprocedural flows, call graphs, configuration, build context, environmental assumptions, and business rules. In this setting, LLMs can interpret warnings, prioritize suspicious code, draft candidate tests, and propose repairs. Their strengths and failures differ from those of static analysis. Language models can reason about intent and describe complex behavior, whereas static analyzers provide deterministic coverage, explicit data-flow reasoning, and auditable rules [72]-[97]. Deep-learning baselines and earlier automated-repair methods remain necessary comparators; without them, it is difficult to tell whether an LLM adds a new capability or merely replaces one imperfect predictor with another [99]-[105].

Execution feedback, validation, constrained decoding, and multiple input sources can improve a candidate repair, but they do not make it reliable. Three problems recur in the literature: the patch may fail to compile, address a symptom without correcting the cause, or change intended behavior while introducing another weakness. On the available evidence, unsupervised repair cannot be justified across arbitrary repositories [51]-[71].

4.3 Human and Organizational Evidence

Security outcomes in AI-assisted development depend as much on the user as on the model. Controlled studies disagree about the magnitude, and sometimes even the direction, of this effect. They nevertheless return to the same factors: expertise, task familiarity, confidence, explanation

quality, and review behavior [37]-[39]. Developers also move between exploratory use and rapid acceleration, while novices may accept an implementation they cannot explain independently [168]-[177]. Productivity can therefore improve even as assurance weakens, particularly when generation outpaces review [170]-[178].

Accountability ultimately remains with the human committer. That principle has little practical value, however, unless the committer has enough time, suitable validation tools, and the security knowledge needed to examine the change. Automation bias is particularly likely when an answer is polished, uses a familiar library, or contains reassuring comments. Compilation, passing unit tests, and provider-side filters can all encourage confidence, but none constitutes a security review. A workable process must therefore state clearly who is responsible for which decision and what evidence is required before acceptance.

Table 2: Representative Empirical Findings

Evidence line	Representative studies	Synthesis
Insecure generation	Pearce et al. [35]; Khoury et al. [36]; Toth et al. [44]	Plausible code repeatedly contains established CWE classes; rates depend strongly on task and prompt.
Human reliance	Perry et al. [38]; Sandoval et al. [39]; Barke et al. [168]	AI assistance changes speed, confidence, and review behavior; realized risk is not determined by model output alone.
Secure prompting/tuning	He and Vechev [52]; He et al. [53]; Tony et al. [54]	Security instructions and tuning reduce selected flaws but are brittle and incomplete.
Detection/repair	Pearce et al. [51]; Fang et al. [75]; Ding et al. [83]	Useful for triage and candidate patches, but performance degrades on repository-scale and semantic defects.
Poisoning/supply chain	Schuster et al. [107]; Aghakhani et al. [108]; Spracklen et al. [112]	Training/context poisoning and package hallucinations can convert assistance into a supply-chain channel.
Agentic execution	Liu et al. [130]; Storek et al. [131]; Huang et al. [133]	Repository context and tools enable indirect injection to become cross-file modification or command execution.

4.4 Cross-Language, Framework, and Task Heterogeneity

Even with the same model family, the security of a program differs depending on the programming language. A summary score fails to differentiate between the failure surfaces and safety default of C, C++, Java and Python. In non-memory-safe languages, a reasonable completion might contradict unexpressed constraints on bounds, lifetimes or integer behavior. Managed languages don't suffer from some of those risks, but do give more weight to injection, authorisation, unsafe deserialisation, framework configuration and dependency selection. Furthermore, models can suggest out-of-date APIs when new safer language features are available [205].

Repository context may make suggestions more relevant of suggestions, but it can also introduce obsolete examples, unsafe conventions, and attacker-controlled artifacts. Framework conventions add further variation. A SQL query that is safe within one data-access layer may be dangerous in another. Similarly, an authentication fragment may hash passwords correctly but omit the session rotation required by its framework. Syntactically valid cloud deployment code

may still violate an organization's IAM policy. A function signature alone does not reveal many production security obligations from a function signature.

Security findings are more informative when stratified by task, privilege, data sensitivity, and attacker interaction. Task complexity also changes the meaning of a security result. Small algorithmic exercises primarily assess functional completion and expose only a narrow range of input-validation defects. Web applications, identity workflows, cryptographic code, parsers, and infrastructure-as-code instead depend on negative requirements and environmental assumptions. High HumanEval-like performance provides little evidence about tenant isolation, race conditions, confused-deputy behavior, or deployment policy.

This qualification is particularly important for commercial assistants because their underlying models, filters, and retrieval policies may change without an archival identifier. The current evidence does not support a stable ranking of models. Performance varies with model version, sampling settings, system prompts, context length, retrieved material, and security instructions. A measurement from one hosted snapshot therefore characterizes a configured system at a particular time, not a durable property of its brand.

4.5 Iterative Refinement and Security Drift

Successive refinement does not necessarily make code safer. Requests for additional features, refactoring, performance improvements, and fixed tests and explanations change the implementation and the context for the next turn. A subsequent request may have different goals, and sneakily strip away a previous security invariant. Apparently constructive improvement prompts have been found by Shukla et al. [192] to improve critical findings. Each refinement should therefore be considered as a new change that affects security, not as an improvement in monotonic fashion.

If changes are not verified then a secure first answer indicates little about the security of the final artifact. There are three reasons for this drift. Objective substitution can first be used in such a way that the newest instruction which replaces the original control, be it in terms of speed or brevity, can take the place of a control that was only implicit. Second, it may forget to fill in requirements that had been set up previously in a lengthy conversation. Third, the use of patch layering will help to make sure that each change is assessed on its own merits rather than as part of the broader landscape. In that model, a cache might not require authorization, the revised error-handling could leak a secret, or a convenience wrapper could reintroduce unsafe string construction.

Thus, iterative workflows demand deterministic tests, exploit reproduction, tests based on explicit properties and a change-aware review after every material transformation. This problem does not go away if students do their own self review and can worsen the issue. While it is useful to get a model to give feedback on its own products to fix minor flaws, it is not independent evidence since the model is trained against the biases and blind spots, and may lack context. A second model might introduce more variations in the errors, but even then, if the models agree, this is not a security oracle.

An assistant can aid in converting security requirements into tests, schemata, policy files and review checklists. The enforcement, however, should not be based on the model having to remember those requirements from a previous exchange. Every change set should be judged individually in terms of the same durable artifacts. They capture security invariants in a way that they persist when conversation context is truncated/displaced.

4.6 From Controlled Prompts to Repositories and Production Code

Controlled benchmarks are unable to capture all of the conditions in which real incidents occur. Production environments encompass legacy code, undocumented assumptions, mixed-trust inputs, dependency graphs, secrets and time pressure. For causal comparison, simplifying is needed but it reduces ecological validity. The assistant at repository scale should be able to reason across files and configuration, and the vulnerability can be introduced by the user or by the model [79, 208].

The importance of ecological validity is that a small error rate at the local level can create a significant operational issue. A small number of defects per line may result in significant overall damage if the same pattern of defects are repeated in multiple projects. Scientists have started to compare contributions to production repositories by AI with contributions by humans, and attribution is hard. The code that is generated is often modified, authorship is not always easy, and reporting procedures are not consistent. Initial evidence at scale indicates that defects that stem from artificial intelligence can either be found in the repository or not, depending on the review depth and subsequent human modification [207].

Depth should be reviewed after security impact, not label of a file. Infrastructure definitions, automation artifacts and generated tests, in addition to application source, should be used for analysis in production studies. If the test is also based on the same misconception as the implementation, it can lead to false sense of security. In similar fashion, regular application tests can detect no security impact in the case of an agent modifying a CI workflow, container file, Terraform policy or dependency manifest.

4.7 Reading Conflicting Results Correctly

Some findings that appear contradictory address different questions. Static analyzers count rule violations rather than demonstrated exploitability. User studies evaluate the human-model pair, not the model alone. Secure-generation benchmarks cover selected CWE scenarios, while repository studies operate with incomplete and uneven context. Once the oracle, unit of analysis, and study setting are made explicit, a reduction in warnings can coexist with an increase in exploitable behavior because the measures describe different constructs.

Functionality and security belong in the same report, but not in the same outcome measure. A warning is not a confirmed weakness, and a confirmed weakness is not necessarily an exploit. Large-scale evidence shows that passing unit tests and scoring well on functional benchmarks do not imply a low density of security defects [206]. The strongest evaluations bring together independent forms of evidence: compilation and functional testing, vulnerability-specific exploitation attempts, static and dynamic analysis, and expert confirmation of the root cause.

We deliberately avoid pooled claims such as "LLMs are insecure X percent of the time." Such a percentage would combine incompatible prompts, models, languages, sample sizes, and definitions. A more defensible conclusion concerns mechanism: current systems repeatedly generate familiar weakness classes, while the observed frequency changes with the task, language, prompt, model version, and verification process. When agentic permissions are added, some of those model errors cease to be suggestions and become system actions, substantially increasing their potential impact.

Table 3: Evidence Interpretation Matrix

Evidence type	Primary strength	Typical limitation	Appropriate claim
CWE prompt suites	Controlled comparison and repeatability.	Synthetic context; limited business logic.	The configured model can generate selected weakness classes.
Static-analysis studies	Scalable measurement across many samples.	False positives/negatives; weak exploit evidence.	Outputs trigger specified security rules at a measured rate.
Dynamic/exploit studies	Demonstrates runtime consequence.	Expensive; narrower coverage.	A concrete generated behavior is exploitable under stated assumptions.
Developer experiments	Measures realized human-AI outcomes.	Small samples; learning and task effects.	Assistance changes behavior, confidence, or security performance in that setting.
Repository studies	Captures cross-file and ecosystem context.	Attribution and ground truth are difficult.	Performance differs from snippets under realistic dependencies and history.
Agent red-team studies	Exercises tools, permissions, and attack paths.	Fast product drift; testbed dependence.	A configured agent can convert untrusted influence into an effectful action.
Technical disclosures	Early evidence of concrete product flaws.	May lack replication or population estimates.	An attack path was feasible in a named version/configuration.

5. Taxonomy Of Security Vulnerabilities In Ai-Generated Code

5.1 Injection and Improper Input Handling

Security instructions can drive a model towards parameterized APIs and encoding relevant to the context in which it operates, but it is not always compliant, and it must be verified by a human. The types of injections are familiar: SQL, command, cross-site scripting, template, and LDAP injection, in addition to unsafe shell construction. In many instances, the model correctly meets the functional request, but it interpolates or concatenates untrusted data into a query, command or HTML response. The mechanism is nothing new and not unique to LLMs. It is a tried and trusted trust-boundary violation re-imagined by a new development process.

5.2 Memory, Type, and Resource Safety

C and C++ completions might involve unchecked buffers, integer truncation, use after free, absence of length checks, or error paths that leak resources. It's easy to miss these defects as a snippet may appear to be syntactically correct but may be wrong in some assumption it is making in its calling context. However, in a controlled study on low-level programming, Sandoval et al. did not see a significant overall increase in critical bugs when using AI assistance [39]. The result does not reflect general safety – it indicates the degree to which the outcome is dependent upon the design of the task and the skill of the user.

5.3 Authentication, Authorization, and Session Management

An authentication implementation might seem complete, yet they may fail to include rate limits, appropriate password-hashing parameters, session rotation, authorization checks, secure

cookie attributes, tenant boundaries, etc. While security requirements tend to be negative and nonlocal (code must not do something that was never asked for), functional prompts emphasize the successful path. Snippet benchmarks capture this form of business-logic authorization poorly, leaving an important gap in current evaluation design.

5.4 Cryptography, Randomness, and Secret Handling

Automated secret scanning is only helpful if it's scanning all the artifacts that are being generated as opposed to only scanning code that is already targeted for production. Familiar cryptographic mistakes are found from one model to the next such as the use of obsolete algorithms, hardcoded keys, certificate verification disabled, and predictable randomness. That's understandable, as it is common that the deprecated APIs are found in training data and in well-established security examples. API keys, tokens, and passwords are examples of credentials that can be generated and subsequently stored in caches, logs, or version control systems, effectively generating a supply-chain issue if a credential is accidentally generated locally.

5.5 File, Path, Serialization, and Deserialization

Giving an agent access to files will only “blow up” a seemingly innocuous path handling mistake. Natural-language instructions could result in path traversal, unsafe archive extraction, too much permissions, insecure temporary files, untrusted deserialization or pickle loading in code or configuration. Exploitability is influenced by the layout of the file systems, the boundaries of the containers and runtime privileges. Those conditions and the impact of them are seldom captured by a snippet-level test.

5.6 Dependency and Software-Supply-Chain Risk

A model may recommend a package that does not exist, has been abandoned, contains a known vulnerability, or closely resembles a typosquatted dependency. Unlike a code-level CWE, this weakness enters through dependency selection and therefore belongs to the software supply chain. Hallucinated dependencies have been reported across models and programming languages. The risk becomes markedly greater when an agent is permitted to install the suggested package without an independent check.

5.7 Logic, Configuration, and Insecure Defaults

Logic flaws remain difficult for both static analysis and model self-review. Missing invariants, races, confused-deputy behavior, unsafe defaults, and fail-open or information-revealing error paths require an understanding of program intent that current benchmarks represent only weakly. These defects can therefore survive two apparently different checks: a rule-based analyzer may lack the relevant semantic model, while the generating model may repeat the same mistaken assumptions during self-correction.

Table 4: Unified Code-Layer Taxonomy

Category	Typical weaknesses	Why assistants fail	Required verification
Input and injection	SQL/OS/XSS/template injection, missing validation	Prompt emphasizes functionality; trust boundary is implicit.	Parameterized APIs, taint/data-flow analysis, adversarial tests.
Memory/resource safety	Bounds, integer, lifetime, race, cleanup errors	Local completion lacks whole-program invariants.	Compiler hardening, sanitizers, static analysis, fuzzing.
Identity and access	Weak auth, missing authorization, session errors	Negative security requirements are underspecified.	Threat model, access-control tests, policy-as-code.
Crypto and secrets	Weak algorithms, bad randomness, hard-coded secrets	Models imitate examples and optimize plausibility.	Approved libraries, secret scanning, crypto review.
Files and serialization	Traversal, unsafe extraction/deserialization, permissions	Path and object trust is context dependent.	Sandbox, allowlists, canonicalization and format tests.
Dependencies	Hallucinated/typosquatted packages, vulnerable versions	Model predicts plausible names, not repository truth.	Registry verification, lockfiles, SCA, signed provenance.
Logic/configuration	Policy bypass, unsafe defaults, cloud/IAM exposure	Requires system intent and deployment context.	Architecture review, integration tests, IaC scanning.

6. The Human Factor: Trust, Automation Bias, And Review Debt

When you review something for a second time, you're not as unfamiliar with it. While a familiar pattern can be followed rapidly, less rapidly and less reliably in an unfamiliar domain, even when a suggestion seems plausible. AI pair-programming paradigm moves the cognitive effort from building an implementation to assessing an implementation that has been proposed. The studies relate acceptance of insecure suggestions to the combination of expertise and task familiarity, and not to either one of these factors alone [37]-[39], [168]-[177].

There are three reinforcing mechanisms. Fluency of language is often confused for competence and leads to authority bias. The apparent authority may mask the user's uncertainty and result in confidence bias, when the system provides a single answer to a user without even suggesting alternatives. A user can then just accept the suggestion and not have to rebuild a complicated operation from the documentation. All three factors, apparent expertise, less uncertainty, and less effort, mean that verification is less likely.

When it grows faster than the amount of evidence to support it, review debt accumulates. Five accepted completions, for example, could add five distinct security choices to a single diff, all of which might not be given the same level of consideration as a handwritten change of comparable complexity. According to the literature, this imbalance is one of the principal ways in which an AI assistant can undermine security [170]-[178] and not simply the possibility of an insecure suggestion.

Not that developers take AI-generated code wholesale. Rather, security results are a function of the interface of expert and task design. The outcomes are far from consistent and some research

has shown that expert developers are able to identify aspects of the model that novices cannot. Automation bias is increased by time pressure and can be decreased by the need to justify a security decision to accept it before using it.

6.1 Expertise, Calibration, and the Recognition Problem

Experience provides but partial and not uniform protection. If the developer has a solid mental model, he/she can quickly evaluate a well-known suggestion, but this might not be the case in a large-scale repository or if the security impact extends across components. Writing secure code takes one kind of judgment while spotting a tricky detail in an apparently viable solution takes another. Even a developer that could create a secure solution from scratch can miss a defect in the code that is generated.

The one practical issue that is a problem for the individual is calibration – confidence must reflect functional correctness and security. An assistant can look well calibrated when it's just doing its job, and poorly calibrated when it reveals itself to be faulty under an adverse circumstance. Expertise can only help the developer to adjust the confidence to this hidden uncertainty, and only then does it improve outcomes. Interventions should consequently not be limited to trust or distrust but to whether or not confidence is justified through independent security evidence.

6.2 Security Salience and Prompt Framing

The typical scenario for a developer is to talk to an assistant with a functional goal in mind, not with a complete security specification. The model answers the question asked and omitted safeguards which are not explicitly set out by either party. Selected weakness classes can be reduced by concrete security instructions, but these are influenced by the position and order of the instructions given and which ones are stated [37]-[39]. A little prompting can help, but it can't substitute for an unidentified threat boundary by the developer.

Remind them to improve promptly, but it's not enough. The term “secure code” could mean a security approach is effective against one type of attack but not another, since no threat model is provided that establishes the boundaries. The developer will be required to specify the desired security attributes, and the model will have to take those directives on all applicable code sequences.

6.3 Review Debt, Change Amplification, and Team Throughput

Now, code can be generated quicker than a developer can comprehend it. Production outpaces review and unverified decisions on security remain dormant until an incident occurs. This is not just run-of-the-mill technical debt. The software can operate as intended, but the team's limited knowledge of the software's assumptions can cause complications in debugging, modifying, and incident response.

The number of adoptions should be weighed on the finite review capacity. This is proposed by several authors as three indicators: generated lines/reviewed lines, average review depth per change, and interval between generation and independent security check [170]-[178]. Each one focuses on a different aspect of the issue. When viewed as a whole, they indicate if the adoption is proceeding at a pace that is faster than the team can develop a credible assurance.

The study by Kozak et al. targets insecure behavior when performing an autonomous task and not one-shot completion. They report that their preliminary findings show that agents are more likely to engage in unsafe behaviors than completion-only assessments show. The finding should be interpreted with caution as the sample size is small and all tasks are from one environment.

The primary importance is to reveal a limit of scope of evaluation, not to determine a population rate.

6.4 Team, Educational, and Open-Source Externalities

AI assistance transforms work while boosting its pace. Faster generation for authors, bulkier and more diverse changes for reviewers to validate. The distribution of effort is structurally unequal since review is less visible and less rewarded than authorship. This imbalance results in a security risk prior to any given suggestion being evaluated.

Available evidence suggests that structured disclosure increases reviewer vigilance without imposing prohibitive overhead [175], [178]. This imbalance may therefore be reduced by identifying substantial AI assistance, documenting a concise threat rationale for each accepted change, and, where practical, reviewing generated segments separately from human-authored ones.

Before being allowed to use generation for original work, students should be required to find and fix errors in generated code, several authors have argued [102], [104]. Immediate explanations and examples are available through assistants, but surface patterns may be internalized by students without the analytical skills needed to evaluate those patterns.

Table 5: Conditions That Moderate Human-Ai Security Risk

Condition	Lower-risk configuration	Higher-risk configuration
User knowledge	Can explain the code and generate adversarial tests.	Cannot recognize insecure APIs or missing requirements.
Task framing	Security properties and misuse cases are explicit.	Only functional output or speed is requested.
Change size	Small, reviewable diff with narrow scope.	Repository-wide edits or unrelated refactoring.
Evidence	Independent tests, scanners, exploit checks, and review.	Model self-review or passing happy-path tests only.
Interface	Provenance, exact changes, and permission effects are visible.	Fluent answer and one-click acceptance obscure uncertainty.
Organization	Security gates and reviewer capacity scale with generation.	Output volume increases while assurance capacity is fixed.
Accountability	Named owner can justify and roll back the change.	Responsibility is diffused to the model or vendor.

7. The Positive Edge: Ai For Security Detection, Repair, And Education

7.1 Vulnerability Detection and Explanation

Developer-facing explanations of scanner findings, suggested data flows, and exploitability-ranked warnings are capabilities LLMs offer. Detection accuracy is best supported for known vulnerability patterns in curated datasets [72]-[97]; how well LLMs perform on zero-day patterns and repository-scale code is still open. The most immediately useful capability, in our view, is explanation, because it lowers the expertise threshold for acting on static-analysis results.

Related research on ML-assisted malware analysis suggests that security models can be made both accurate and computationally economical. Farfoura et al. demonstrated effective classification with a sharply reduced feature set [219], while HFDNet extended this line of work through a lightweight visual architecture based on harmonic feature decomposition [220].

7.2 Vulnerability Repair

No reviewed method consistently produces dependable patches without external validation. A reliable repair workflow must subject model-generated changes to compilation, testing, exploit reproduction, and renewed static analysis. Constrained decoding can reduce unintended changes by limiting the scope of a repair. It does not establish that the resulting program is semantically correct [51]-[71].

7.3 Secure Coding Education

Assistants can explain secure APIs, generate counterexamples, and tailor examples to a learner's level. For novices, this provides quick access to security-relevant patterns that might otherwise require an extensive documentation search. That immediacy is also a source of risk. A learner may reuse the generated pattern in another context without recognizing the assumptions that made it safe in the original one.

7.4 Threat Modeling and Test Generation

LLMs can shorten the early work of brainstorming abuse cases and translating requirements into test templates. Their threat models are less dependable when negative requirements or unusual edge cases matter, because those are precisely the points a security engineer is likely to probe and a model may omit. Preliminary evidence therefore supports using an LLM-generated threat model as working material for a structured threat-modeling session, not as a replacement for that session [104], [105].

7.5 Hybrid Security Analysis Workflows

Connecting security tools rather than replacing them is the most credible positive use case. Ingesting a static-analysis report, filtering false positives against project context, proposing a fix, invoking a test runner, and presenting a validated change for human approval is a workflow an LLM can support. The determinism of static analysis is preserved while the manual effort of triage and initial repair is reduced.

Tool independence should be preserved in hybrid systems. Suppressing a finding or bypassing a control must not be allowed for the LLM. Architectures in which the model proposes and deterministic tools verify, rather than architectures in which the model both proposes and judges, are supported by the reviewed evidence [72]-[97].

7.6 Security-Aware Retrieval and Controlled Generation

Secure code generation can be improved by retrieving approved patterns, vulnerability root causes, and organization-specific secure coding guidelines into the prompt. Retrieval-augmented generation has shown measurable improvements in reducing insecure patterns when the retrieval corpus is curated and current [110], [111]. The quality of retrieved context directly affects output security.

Retrieval is also an attack surface. A knowledge base that contains insecure examples, stale fixes, or adversarially planted patterns will degrade output security. The curation and maintenance of retrieval corpora should follow the same governance as other development dependencies.

7.7 Secure Tuning, Co-Decoding, and Prompt Optimization

Fine-tuning on vulnerability fixes, instruction tuning, constrained decoding, co-decoding, and prompt optimization have each shown reductions in specific weakness classes. The improvements are generally narrow: a method that reduces SQL injection may have no effect on cryptographic misuse. We regard constrained decoding as the most promising because it operates at inference time and can be combined with other controls, but it requires a well-defined specification to constrain against [52]-[55].

A high aggregate "secure generation rate" describes performance only within the conditions of the benchmark. Adversarial evaluation shows that a strong aggregate score does not prevent vulnerable output after adversarial prompting, distribution shift, or repeated interaction. Robustness to these conditions is a separate, stricter requirement that most current benchmarks do not measure [181].

7.8 Security Operations and Organizational Learning

LLMs can help incident response by summarizing vulnerable changes, locating similar patterns, generating rollback commands, and drafting disclosure text. Pilot studies in security operations report faster triage, but these findings are based on internal evaluations lacking independent replication and should be considered preliminary.

8. THE CURSOR ERA: AGENTIC IDEs AND A NEW ORCHESTRATION ATTACK SURFACE

Cursor-like editors, Copilot agent modes, Claude Code, Codex CLI, and MCP-enabled tools differ from completion assistants in three respects: they plan multi-step sequences, they invoke tools with real side effects, and they operate with varying degrees of human approval. This section examines the new attack surface created by these capabilities. Most of the evidence comes from preprints and technical disclosures, because the peer-reviewed record for agentic development environments is still forming [221], [222].

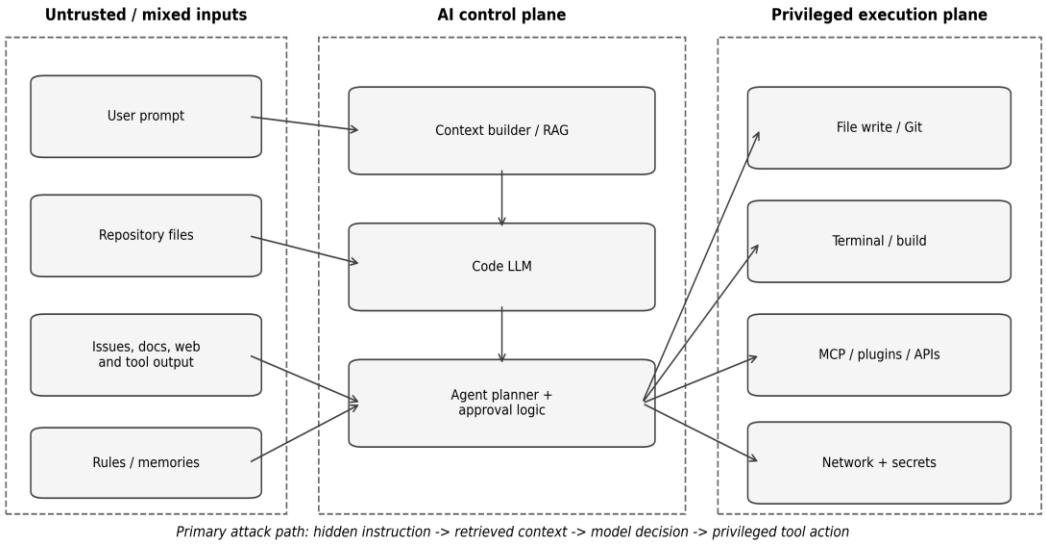

Figure 2: Trust boundaries and primary indirect-prompt-injection path in a repository-aware, tool-using coding agent.

8.1 Indirect Prompt Injection and Context Poisoning

Indirect prompt injection places an instruction inside material that the model is supposed to analyze rather than obey. In a repository-aware assistant, the carrier can be a README, a comment, a test name, a dependency description, or an MCP tool schema. The model cannot reliably distinguish instructions from data in its input, so any untrusted content that enters the context window becomes a potential instruction source.

8.2 Rules, Memories, and Persistence

Agentic editors often support project rules, persistent memories, skills, or instruction files. These mechanisms improve consistency across sessions, but they also create persistence for injected instructions. A rule written by an attacker survives session boundaries and affects every subsequent generation. The persistence model of agentic tools therefore requires the same trust assumptions as other long-lived configuration.

8.3 Terminal and Tool Abuse

Frequent approvals can turn into a security problem on their own. One user who is granted permission to perform routine functions on a day-to-day basis might end up granting permission to an illegitimate command without paying attention to it. Shell access means the consequences are instantaneous: One click can lead to a compromised suggestion, credential theft, file exfiltration, dependency manipulation, reverse-shell creation or an environment change.

8.4 MCP and Plugin Supply Chains

A protocol and descriptive schemas are used to separate models from tools in MCP. The modularity also exposes a software supply-chain attack surface: a malicious or compromised MCP server might return false data, execute unwanted actions, or include instructions in the description of the tools it provides. There is no published study that has considered MCP in production codebases. It does have, however, elements of tool integration architectures with less successful outcomes in other areas [223].

8.5 Cross-File and Cross-Origin Propagation

The impact of the context poisoning can only become apparent after multiple seemingly valid context changes across files or session. There is no need for an explicit request for malware. Just a single message to import, adjust a threshold, or tinker with the configuration can propel the system in the direction of the poisoned state.

8.6 Approval Spoofing and Permission Amplification

The system is only secure with human approval if it is informed. Its worth hinges on the interface's content and user's time to look at the operation. A natural-language description may obscure implications which would be apparent to the user in the actual command and the user might consent to something other than they would otherwise have consented to. Approval spoofing is a direct demonstration that provides a good explanation that appears benign but is followed by a malicious command [121], [125].

Table 6: Agentic Development Threat Taxonomy

Threat class	Attack carrier	Potential effect	Control priority
Indirect prompt injection	README, comment, issue, web page, tool output	Model follows attacker instruction instead of user intent.	Separate instructions/data; mark provenance; adversarial evaluation.
Context/RAG poisoning	Malicious or strategically placed repository artifact	Poisoned retrieval steers edits or suppresses warnings.	Trusted indexes, scoped retrieval, content signing, anomaly checks.
Persistent rule poisoning	Project rules, memories, skills, dotfiles	Cross-session or cross-user behavioral persistence.	Review and sign control files; protected paths; reset visibility.
Tool/terminal abuse	Shell, build tool, browser, Git, cloud API	RCE, secret theft, persistence, destructive changes.	Sandbox, allowlisted capabilities, explicit high-impact approval.
MCP/plugin supply chain	Tool descriptions, server update, package	Tool squatting, rug pull, confused deputy, data exfiltration.	Identity, pinning, least privilege, audit logs, egress policy.
Approval manipulation	Long diff, bundled command, misleading summary	Nominal human consent without informed understanding.	Action decomposition, exact command/diff preview, two-person rule.
Cross-file propagation	Generated helpers, tests, workflows, hooks	Weakness spreads and becomes future trusted context.	Protected files, provenance graph, rollback and repository scanning.

8.7 An End-to-End Agentic Exploit Chain

A transformation chain is a useful mental model for agentic risk. First, the attacker embeds a payload in a context source that the agent will ingest. Next, the agent reads the payload, treats it as an instruction, converts it into a tool call, and runs that call under the user's authority. Each transition—ingestion, interpretation, tool-call formation, and execution—is a point where a control can intervene: input filtering, instruction-data separation within the model, tool-call validation, and execution sandboxing, respectively.

The full exploit path is not covered by any single control. Semantically encoded payloads defeat input filtering. Because the model cannot deterministically distinguish instruction from data, model-level instruction-data separation is unreliable. Knowing the expected call—which depends on the original intent—is required for tool-call validation. A defense-in-depth strategy with controls at every stage of the chain is therefore necessary.

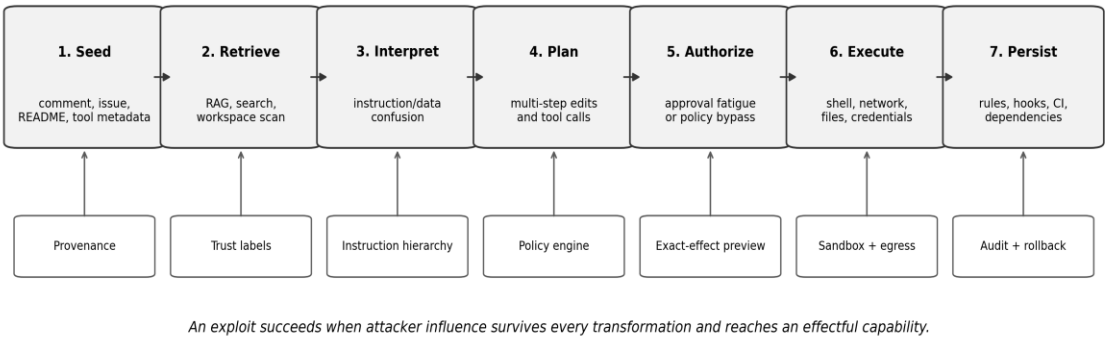

Figure 3: Agentic coding exploit chain and the corresponding control opportunity at each transformation stage.

8.8 Pull-Request, Issue, and CI/CD Agents

A particularly dangerous trust inversion is created by repository automation. Content the agent reads but the developer rarely inspects includes pull-request descriptions, issue comments, and CI logs. Instructions that the agent acts on with the developer's authority, without the developer's knowledge, can be injected by an attacker who controls any of these channels.

That prompt injection can reach agentic workflows through issue titles and comments is demonstrated by GitInject, which evaluates attacks in live, ephemeral repositories [121]. No special access to the developer's machine is required; the agent's automatic ingestion of repository events is what the attack exploits.

Large-scale analysis of agentic GitHub workflows identifies two important patterns. Prompt-to-agent injection through issue and PR content is the most common attack vector, and the success rate increases with the agent's autonomy. Agents that auto-approve their own changes are significantly more vulnerable than those that require explicit confirmation.

Defensive workflow design should separate untrusted-event ingestion from privileged execution. Read-only analysis of issues and comments can proceed without restriction; write operations triggered by that analysis should require independent confirmation through a channel the attacker does not control.

8.9 Agent Data Injection and Semantic Trust Confusion

Not every attack needs an explicit instruction such as "ignore previous rules." Agent data injection demonstrates that carefully crafted data records can manipulate an agent's tool calls without containing any imperative language. The injection is purely in the data layer, which bypasses content filters that look for instruction patterns.

Recent evidence demonstrates that data-style injections can trigger arbitrary navigation, code execution, and data exfiltration in web-browsing agents [147]. The attacks work because the agent treats retrieved content as factual context for decision-making, and a sufficiently crafted page can redirect that decision-making toward attacker-specified goals.

Contextual manipulation also challenges the assumption that separating "instructions" from "data" is sufficient for defense. When an agent reasons over data, the data itself becomes part of

the decision process. An attacker who controls the data can influence the decision without ever issuing an instruction.

8.10 Retrieval-Optimized and Query-Agnostic Injection

A hidden payload has no effect if it is never retrieved. Earlier demonstrations sometimes assumed that an attacker could place content directly in the agent's prompt. In practice, the payload must first survive ingestion into a retrieval index, then be selected by the retrieval algorithm for a relevant query. Query-agnostic injection, which increases the probability of retrieval regardless of the query, addresses this constraint.

Coding-agent attacks are also becoming less dependent on a specific user request. QueryIPI uses persistent perturbations in repository files that are triggered regardless of the developer's task. This means that the attacker does not need to predict what the developer will ask; the payload activates whenever the agent retrieves context from the compromised file.

The evaluation bar is further raised by automated attack optimization. Successful injections can be generated by black-box methods adapted to agentic settings without access to model weights or system prompts. A defense evaluated only against hand-crafted attacks may significantly overstate its robustness given that these methods exist.

8.11 Attribution, Dynamic Benchmarks, and Residual Risk

Whether an unsafe tool call is causally driven by untrusted input is what attribution-based defenses attempt to determine. Controlled-setting results are promising, but the approach depends on the model's internal representations—which may not be accessible for proprietary systems—and the adversarial adaptability of attribution methods has not been thoroughly evaluated.

Static benchmarks miss an important property that AgentDyn exposes: in real tasks, benign and malicious third-party content coexist, and the agent must separate them while operating. Isolating attack scenarios from normal operation, as most benchmarks do, may not capture this coexistence.

No available defense makes it safe to treat arbitrary repository or web content as trusted input for a privileged agent. Assuming that any untrusted content entering the context window may carry an instruction, and designing controls that limit the consequences of that assumption, is the pragmatic stance we recommend.

9. Are These New Vulnerability Classes?

At the code layer, these are generally not new vulnerability classes. SQL injection remains SQL injection whether written by a person or a model, and the same applies to hard-coded credentials. The vulnerability itself is unchanged; AI alters the production mechanism and may increase the scale at which the pattern appears.

Existing classifications are less complete at the orchestration layer. Prompt injection has effects similar to command injection, yet it works through semantic interpretation and is not addressed by conventional input validation. Context poisoning targets retrieved model context rather than application data flow, so it has no direct CWE counterpart. The underlying security principles are familiar, but the classification must represent how they operate in an agentic system.

9.1 Familiar Weaknesses, New Production Dynamics

Familiar defects still dominate generated source code: injection, missing authorization, unsafe memory handling, cryptographic misuse, and insecure defaults are all covered by CWE. AI primarily changes their production and propagation. The same vulnerable pattern can be generated independently in many projects, and correcting the model later does not remove versions already committed to repositories.

AI changes the propagation and persistence of vulnerable patterns. A weakness prevalent in training data may be reproduced independently across projects by different models. Moreover, model-level corrections do not alter generations that have already been committed.

9.2 Orchestration-Layer Weaknesses

Orchestration is the distinct layer. The system decides which context to trust, how to interpret it, which tools to call, and when human approval is needed. A weakness in this process cannot be reduced to one defective line of generated code. It arises from the interaction between a model designed to follow instructions and an environment that turns those instructions into actions.

These weaknesses remain connected to established security principles. Indirect prompt injection crosses a trust boundary, permission amplification is a form of privilege escalation, and approval spoofing reflects the confused-deputy problem. The purpose of a taxonomy is therefore organizational. It places known principles in the operating context of agentic development without claiming that the underlying failure modes are wholly new.

9.3 Criteria for Claiming a New Vulnerability Class

A weakness should be treated as a new agentic class only if three conditions hold. Its root cause is not fully covered by an existing CWE entry; exploitation depends on orchestration behavior rather than defective code alone; and mitigation requires controls beyond standard secure-coding practice. These tests help prevent new terminology from disguising vulnerabilities that are already well understood.

Model-generated SQL injection remains CWE-89. A repository comment that causes an agent to exfiltrate secrets is different because the failure occurs when untrusted context is interpreted as an instruction, not when a program statement is written incorrectly. No single CWE captures that mechanism adequately. We therefore regard it as a plausible entry in an agentic weakness enumeration.

9.4 Toward an Agentic Weakness Enumeration

The field needs a focused extension to CWE rather than a competing taxonomy. An agent-specific entry should identify the violated trust assumption, the capability required for exploitation, and the minimum preventive control. Because the attack surface is changing faster than standards typically evolve, the enumeration should remain lightweight and readily revisable.

10. Evaluation: What Should “Secure” Mean?

10.1 Beyond Pass@k

A secure-code benchmark should assess three distinct properties: build success, functional correctness, and security under an explicit threat model. Reporting pass@k or accuracy alone merges functionality with safety and hides the conditions in which the model fails.

10.2 Oracles and Ground Truth

No single oracle suffices. False positives and rule gaps affect static analyzers; only executed paths are covered by dynamic tests; specific vulnerabilities are confirmed by exploit-based evaluation, but absence of others is not guaranteed. Combining multiple oracles and reporting agreement and disagreement among them yields the strongest evaluation.

10.3 Repository and Agent Benchmarks

Because hidden configuration, generated files, CI definitions, dependencies, and infrastructure artifacts are all part of the attack surface for agentic systems, repository-scale benchmarks should include them. The files that agents are most likely to modify are the very ones missed by benchmarks that evaluate only application source code.

10.4 Adaptive Adversaries

Robustness is likely overstated by a defense evaluated only against a fixed set of strings. Injection payloads can be optimized by adaptive adversaries using gradient-free methods, and filters that were effective against hand-crafted inputs are often bypassed by the resulting attacks. An adaptive component should therefore be included in evaluation.

10.5 Repeated Sampling, Variance, and Confidence Intervals

Because a code generator is stochastic, a single completion cannot characterize it and may hide a wide distribution of outcomes. This variance is itself relevant to security. Evaluators should sample multiple completions for each prompt and report the full distribution of security outcomes rather than only the best or worst case.

When outcomes are widely distributed, a mean insecurity rate without a confidence interval can mislead. Prompts, random seeds, model versions, and temperatures may all contribute variation. Confidence intervals should therefore accompany the point estimate and reflect the relevant sources of uncertainty.

10.6 Temporal Validity, Versioning, and Reproducibility

Hosted models change continually, which makes reproduction especially difficult. Evaluations should record collection dates, model versions, API endpoints, system prompts, and every available configuration identifier. These details are essential evidence, not administrative metadata: without them, later researchers cannot reproduce a finding or determine whether two studies tested comparable systems.

A first-class design consideration should be temporal replication. Rerunning a benchmark monthly or quarterly with the same prompts and a fresh model snapshot produces a time series that reveals whether security outcomes are stable, improving, or degrading. Almost no longitudinal data of this kind exists in the reviewed literature.

10.7 Joint Security-Utility Metrics

A secure but non-functional output is not a successful generation; a functional output that is insecure is not a safe one. Joint security-utility metrics should report both dimensions and their trade-off. For repair, the unit should be a validated patch: the original exploit no longer works, intended behavior is preserved, and no new weakness is introduced.

For repair, the unit should be a validated patch: the original exploit no longer works, intended behavior is preserved, and no new weakness is introduced. Partial repairs that suppress a symptom without removing the root cause should be reported separately.

10.8 Agent and Workflow Metrics

Agentic systems need metrics beyond code findings. Relevant measures include unauthorized-action rate, context-payload detection rate, approval-bypass success rate, and the agent's compliance with stated capability boundaries. These metrics are specific to the orchestration layer and are not covered by conventional secure-code benchmarks.

Comprehension, not merely clicks, is what approval metrics should capture. Whether users notice a malicious action embedded in an approval request, and whether the approval interface design affects detection rate, can be tested by researchers. That the approval step is not providing meaningful safety is indicated by a high approval rate with low comprehension.

10.9 Minimum Reporting Checklist

Table 7: Minimum Reporting Checklist For Security Evaluation

Dimension	Required disclosure
System identity	Model and product version, date, mode, system policy, temperature, and decoding settings.
Context	Prompt templates, repository state, retrieval corpus, trust labels, and context-window handling.
Capabilities	File, shell, network, browser, Git, MCP/plugin tools, credentials, and sandbox boundaries.
Tasks	Language, framework, privilege, data sensitivity, source, contamination controls, and attacker model.
Sampling	Number of scenarios and repetitions, seed handling, failures, and confidence intervals.
Oracles	Compilation, tests, static/dynamic tools, exploit checks, manual review, and adjudication process.
Outcomes	Functionality, confirmed weaknesses, exploitability, severity, utility, and false-positive analysis.
Agent traces	Retrieved items, plans, tool calls, approvals, effects, persistence, and rollback behavior.
Artifacts	Prompts, code, containers, rules, queries, logs, and scripts sufficient for replication.
Limitations	Hidden components, provider drift, unsupported generalizations, and evidence classification.

11. Mitigation: A Defense-In-Depth Reference Architecture

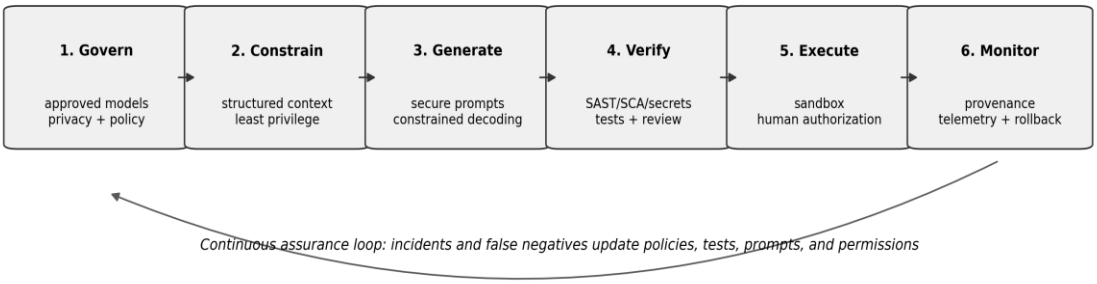

Figure 4: Continuous-assurance architecture for secure AI-assisted software development.

11.1 Governance and Model/Tool Inventory

An inventory of approved assistants, model endpoints, extensions, MCP servers, and coding agents should be maintained by organizations. Untracked tools create blind spots in threat modeling and incident response, the reviewed studies collectively show [163]-[167]. Model version, data handling, network access, and approval scope for each tool should be recorded in the inventory.

11.2 Structured Context and Provenance

Every context item should be labeled with its origin and trust level, and that provenance should be preserved through the generation and execution pipeline. Provenance tracking is the minimum infrastructure needed to support context-governance controls, several authors have argued [163], [167]. Trusted project documentation cannot be distinguished from untrusted web content or attacker-injected material without provenance.

11.3 Least Privilege, Sandboxing, and Capability Control

An agent's accessible scope should be limited to precisely those files, commands, network destinations, and credentials that the current task demands. Least privilege, a cornerstone of access-control design, applies with equal force to agent capabilities. Current tool implementations fall far short of this principle, in our assessment—most agents are granted broad workspace access by default [222].

11.4 Independent Code Verification

At least the same pipeline as human-written code should be applied to generated code. Conventional static and dynamic analysis identify defects that model self-review misses, the reviewed evidence confirms [72]-[97]. SAST, SCA, secret scanning, dependency review, and security-focused tests may all be included in the pipeline.

11.5 Approval Design

Comprehensible and decomposed approvals are essential. The exact command, expanded aliases, affected files, and security-relevant flags should all be displayed. That vague or summarized approvals are exploitable has been demonstrated in the attack literature [121], [125]. Approval design is a security-critical user-interface problem, not merely a workflow convenience, we argue.

11.6 Monitoring, Provenance, and Recovery

User request, retrieved context, model/version, tool calls, approvals, file changes, and runtime effects should all be connected in audit logs. Sparse though the literature on forensic analysis of AI-assisted development may be, the general principle from software supply-chain security is that traceability from change to cause is essential for incident response [163]-[167].

Table 8: Control Matrix For Ai-Assisted Development

Layer	Minimum controls	Higher-assurance controls
Policy	Approved tools; data classes; user accountability.	Risk assessment; provider audit; restricted projects; exception workflow.
Context	Task-scoped retrieval; untrusted-data labels.	Signed control files; protected indexes; remote-content quarantine.
Model	Secure system prompt; security-specific instructions.	Secure fine-tuning; constrained decoding; multi-model validation.
Tools	No autonomous high-impact actions; allowlists.	Capability tokens; disposable sandbox; egress and secret broker.
Code	Tests, SAST, SCA, secret scanning, review.	Fuzzing, sanitizers, formal checks, specialist review for critical code.
Supply chain	Pinned dependencies; registry verification.	Signed packages, SBOM, SLSA provenance, private mirrors.
Operations	Audit logs, protected branches, rollback.	Behavioral monitoring, provenance graph, incident simulations.

11.7 Secure Retrieval and Context Governance

Rather than assembling context as a flat string, it should be built as a typed, provenance-preserving object. Each item should carry a source label, a confidence score, and a trust classification. Including these labels in the model's prompt enables it to reason about which inputs are reliable and which need verification.

A compromised retrieval index degrades every generation that depends on it. Retrieval indexes should therefore be governed by the same security lifecycle used for dependencies. Ingestion should authenticate each source, strip executable content wherever possible, and record every addition. Later review is equally important because stale or newly insecure material must be removed from the index.

11.8 Capability Security and Policy Enforcement

Tool schemas should define capability boundaries explicitly, and the protocol should enforce them independently of model discretion. Narrow, typed operations such as read_file, propose_patch, run_tests, and query_package_metadata make authority visible and auditable. Broad calls such as execute_shell or write_file place excessive power behind one interface and should be replaced with smaller operations whenever possible.

Capability tokens should grant only the authority needed for the declared task, and that authority should end automatically with the task. Tokens therefore need to be short-lived, task-bound, and non-transferable. After reviewing a pull request, for example, an agent should retain no permission to modify the main branch.

11.9 Meaningful Human Authorization

Authorization for a high-impact action should present the exact operation and its likely consequences. A nominal human-in-the-loop control offers little protection if the user sees only a vague intention and must decide under time pressure. Evidence reviewed in this study indicates that approval fatigue and imprecise summaries materially reduce the security value of human approval [121], [125].

The authorization path must remain outside the model's own control flow. The model may explain a proposed action, but natural-language persuasion must not be able to confer authority. The model should not approve its own request or reformulate a denied operation to evade the control.

11.10 Monitoring, Incident Response, and Recovery

Monitoring should prioritize events with clear security relevance: access to secrets, contact with unexpected network destinations, deviation from the declared task, and repeated attempts after denial. Agent telemetry should enter the same operational processes as other security signals. Although practical experience remains limited, behavioral anomaly detection may usefully complement code-level controls [121], [125].

Agentic incidents leave less time for response because the system may continue acting until its authority is revoked. Conventional vulnerability handling may therefore be too slow. Incident playbooks should enable immediate token revocation and workspace isolation, with dependency rollback or model downgrade available when required.

11.11 Assurance Profiles

A single control level is unsuitable for tasks with very different consequences. Risk-tiered assurance can preserve usability without granting unsafe autonomy: low-risk autocomplete may need little friction, while high-risk agentic actions should require provenance, sandboxed execution, and informed human approval. Assurance should be proportional to potential impact.

Table 9: Risk-Based Assurance Profiles For Ai-Assisted Development

Profile	Typical work	Permitted autonomy	Required assurance
Low	Documentation, local examples, non-sensitive tests.	Read workspace; propose bounded edits.	Normal review, tests, provenance, no secrets or external execution.
Moderate	Application features, refactoring, dependency updates.	Edit branch; run approved build/test tools in sandbox.	SAST/SCA/secret scan, dependency verification, reviewer ownership, restricted egress.
High	Authentication, crypto, parsers, IaC, CI/CD, data access.	Proposal only; no direct merge or privileged terminal.	Threat model, specialist review, exploit tests, policy-as-code, signed artifacts.
Critical	Production credentials, release, incident response, safety-critical code.	Read-only analysis or tightly scripted actions.	Two-person authorization, isolated environment, full logging, formal or independent validation, rehearsed rollback.

12. Security Across The Ai-Assisted Software Development Lifecycle

Governance shouldn't start and stop with code generation. From requirements to decommissioning, security relevant decisions are found throughout the lifecycle [163]-[167]. Responsibility, controls, supporting evidence should not only be in the implementation stages.

12.1 Requirements and Security Intent

Security stories should explicit the threats the implementation should be able to withstand and the assistant should review the output of the story in relation to these threats. The requirements phase is the first place where you can avoid a missing safeguard rather than find it later. Some authors claim that the best lever to decrease insecure generation is the explicit specification of security requirements in the prompt [37]-[39].

Policies on the use of AI are also required. Teams should agree on the following: What is acceptable as source code leaving the environment? What models are acceptable? What validation should follow any changes that are created? According to our assessment, most organisations currently don't have those policies, and adoption is going ahead without governance.

12.2 Architecture and Threat Modeling

The agent's maximum blast radius is determined by architecture. The model, retrieval index, and tool environment should be treated as part of the trusted computing base in threat modeling. All context sources, all tool interfaces, and all approval mechanisms compose the attack surface.

A useful design artifact is the capability budget, which lists the minimum reads, writes, commands, endpoints, and credentials required for each agent task. Additional authorization should be required for any request exceeding the budget. The architectural analog of least privilege for agentic systems is what we regard capability budgeting as.

12.3 Implementation and Local Development

Proposal mode should be the secure default during implementation. Task-scoped context may be retrieved and changes suggested by the assistant, but writing files, running commands, or modifying configuration should require explicit approval. The developer's authority over all side-effecting operations is preserved by this mode.

A disposable workspace with restricted mounts and egress is where local execution should occur. Per-task injection and post-use revocation should apply to environment secrets. Cross-task contamination is prevented by discarding the workspace after each session.

12.4 Verification and Security Testing

Generation and verification must be independent. Intended behavior is established by functional tests; the absence of specific weaknesses by security tests; deterministic coverage by static and dynamic analysis; and judgment that automated tools cannot replicate by human review. That model self-review is not a substitute for independent verification is consistently shown by the reviewed evidence [72]-[97].

A security evidence bundle should accompany the change. The threat or requirement identifier, the test that verifies the security property, the static-analysis result, and the human-review record can all be included. Traceable and auditable security claims result from this bundle.

12.5 CI/CD and Release

CI functions as both an assurance mechanism and an agent attack surface. Isolating untrusted pull-request content from the CI execution environment is necessary. Agent-triggered runs

should receive the same isolation as untrusted contributions, given that the agent may have been compromised by injected instructions.

AI-produced components should carry provenance records alongside those of human-authored components so that incident responders can trace a defect to its origin. A verifiable release bundle should connect the source, dependency lockfiles, build environment, tests, and resulting artifacts.

12.6 Deployment and Operations

Because deployment agents operate where the consequences are greatest, they should have less autonomy than other agents. Any change to production configuration, secrets, or network policy must require human authorization, regardless of the agent's confidence.

When an incident affects a component last changed with AI assistance, the code diff is only part of the evidence. Responders should also examine the generation context and the tool calls made during that session. Runtime telemetry is most useful when it can connect observed behavior to the provenance of the AI-assisted change that may have caused it.

12.7 Maintenance, Model Drift, and Decommissioning

Maintenance includes the assistant environment and application code. A provider can change the model, an extension can be given new permissions or an MCP server can change its schema. Any of these changes may change the way the application behaves regarding security without changing the actual application. It's important for organizations to track them and review their threat assumptions following major changes.

An assistant is not fully decommissioned until it has had its credentials, extensions, local services, or cached repository context removed. These remnants maintain an attack surface after it was formally used. When decommissioning, tokens should be revoked, connected components should be removed, all cached data deleted and an auditable record of access exercised kept.

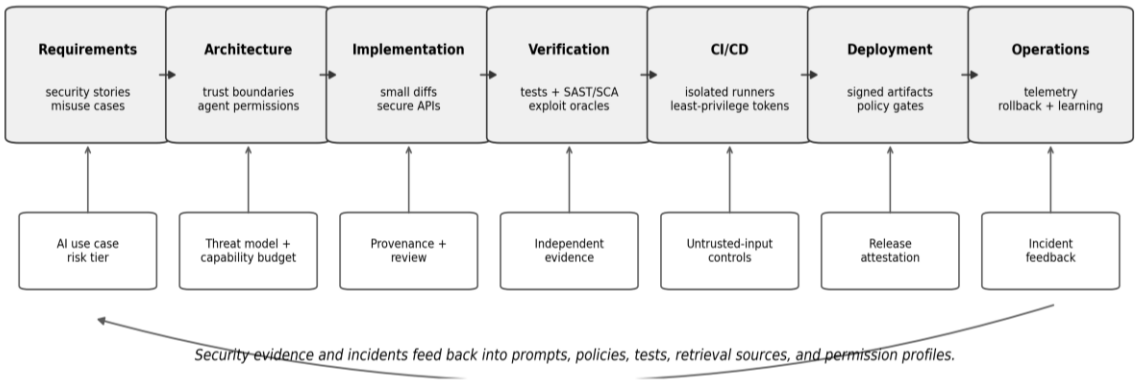

Figure 5: Security gates across the AI-assisted development lifecycle and the feedback loop from operations.

Table 10: Lifecycle Risks And Required Security Evidence

Phase	Representative AI-specific risk	Minimum evidence before progression
Requirements	Functional prompt omits abuse cases, privacy, or policy.	Validated security stories, data classification, AI-use constraints.
Architecture	Agent or retrieval service receives excessive trust.	Threat model, capability budget, trust-boundary and data-flow review.
Implementation	Insecure code, package hallucination, secret exposure.	Small diff, provenance, approved APIs and dependencies, sandboxed execution.
Verification	Model-generated tests confirm the model's own assumptions.	Independent tests, SAST/SCA/secrets, exploit or dynamic checks as needed.
CI/CD	Untrusted PR or issue content reaches privileged agent/token.	Separated workflows, least-privilege tokens, typed validation, protected branches.
Release	Generated justification substitutes for verifiable assurance.	Signed artifacts, attestations, policy gates, review and rollback readiness.
Operations	Persistent agent state or generated defect causes incident.	Telemetry, provenance graph, response playbooks, revalidation and lessons learned.

13. Governance, Privacy, And Compliance

13.1 AI System and Tool Inventory

Governance starts with knowing which tools are deployed. IDE assistants, model endpoints, browser extensions, MCP servers, command-line agents, and any service that forwards code or context to an external model should all appear in the inventory. That shadow AI use is widespread and that organizations regularly underestimate how many AI tools operate in their development environment is demonstrated by the reviewed studies [163]-[167].

Capability-oriented rather than brand-oriented inventory is needed. A single product may operate as a completion assistant in one mode and as an agentic development environment in another. The capability, not the product name, determines the security requirements.

13.2 Data Protection and Repository Confidentiality

Proprietary code, credentials, customer data, vulnerability details, and business logic can all be contained in prompts and context. Significant data-protection implications attend the sending of this content to external models. Data classification should determine which models and endpoints are approved for each category of content, several authors have argued [163], [167].

Before model submission, data minimization should occur. The volume of sensitive material leaving the development environment can be reduced through task-scoped retrieval, redaction, synthetic context, and local inference. Most current deployments do not implement systematic data minimization, in our assessment.

13.3 Accountability and Human Responsibility

All accepted changes, and their security implications, should have a human owner, either an individual developer or a team. The model can suggest the edit, but cannot be the person responsible for the edit. The literature holds the human decision to accept the change responsible and not the previous model output [163-167].

The evidence in favor of this is that of a shared responsibility, where the ownership is clearly defined at every level. Vendors design products which are as secure as possible, they disclose the product, platform teams approve configurations and guardrails, and developers determine whether suggestions are accepted or rejected.

13.4 Third-Party, Extension, and MCP Risk

The supply-chain exposure of extensions and MCP servers is similar to that of any other development dependency and deserves the same supplier scrutiny. Compromised component may be used to inject instructions, to export information, or to modify the behaviour of the tools. For this reason, a number of authors suggest governed extension registries which ask for security attestation in order to be admitted [121] [125].

Once you receive registry approval, you have only established trust at one moment. A legitimate tool can be substituted or changed after it has been registered, but prior to use. To ensure that the executing component matches its registered manifest, runtime attestation is required. Governed registries can then share vetted manifests and the policy profile.

13.5 Auditability, Record Retention, and Evidence

Audit records must enable the reconstruction of an agent action without any unnecessary retention of prompt content that is sensitive. The key items of evidence are the request, retrieved context, made tool calls, and the changes. Full prompts should be avoided if the prompts contain proprietary or otherwise sensitive information that does not need to be investigated.

Model conversation logs and security audit logs should be distinguished by organizations. Conversation logs for debugging and improvement, audit logs for incident response/compliance. There are various differences in retention periods, access controls and encryption requirements.

13.6 Metrics and Risk Acceptance

Governance metrics should be linked to outcomes - adoption. Measures like change rate, insecure-suggestion acceptance rate, time-to-detect AI-introduced defects, and generated code to independently verified code ratio are helpful. The literature reviewed indicates that most organizations are not monitoring security results and are monitoring adoption.

The correct level must accept residual risk. Low-risk autocomplete with standard review may be acceptable to a team, while explicit risk acceptance from the security team or a designated authority should be required for agentic operations on production systems. Risk acceptance is frequently implicit and informal, in our assessment, making it difficult to audit.

14. Research Gaps And Agenda

Cursor-like systems have a far smaller evidence base than Copilot. What the field needs are longitudinal studies tracking security outcomes under agentic development and evaluations that isolate the effect of individual controls rather than measuring adoption as a whole. The current literature offers risk correlates but few controlled tests demonstrating that a given mitigation actually works.

Table 11: Priority Research Agenda

Priority	Research question	Recommended design
Agentic IDE security	How do context scope, approval mode, and permissions change attack success?	Cross-product testbed with replayable repositories, terminal traces, and controlled secrets.
Context poisoning	Which repository artifacts dominate retrieval and persist across sessions?	Adversarial corpus with comments, docs, issues, images, rules, generated files, and semantic variants.
Human authorization	Can developers understand and safely approve agent actions under time pressure?	Controlled studies with exact-command previews, risk labels, fatigue, and expertise strata.
Secure generation	Do security tuning and prompting generalize to unseen CWEs and libraries?	Time-split, contamination-aware benchmarks with repeated sampling and exploit oracles.
Repair assurance	When does an apparently fixed patch introduce semantic regression?	Real repositories, vulnerability reproduction, regression suites, differential and mutation testing.
MCP supply chain	How can tool identity, scopes, updates, and descriptions be made trustworthy?	Protocol-level experiments with signed metadata, pinning, rug pulls, and least-privilege tokens.
Economics and governance	Do productivity gains exceed review, incident, and compliance costs?	Longitudinal organizational studies measuring review debt, escaped defects, and remediation effort.
Provenance	How should AI contribution and tool-use lineage be represented?	Open provenance schema linked to commits, SBOMs, model versions, prompts, actions, and evidence.

Multilingual prompts and comments, underrepresented programming ecosystems, and the security of generated infrastructure-as-code deserve further research. Non-English contexts are especially understudied, and the assumption that security findings generalize across languages and frameworks has not been tested.

15. Practical Guidance

15.1 For Developers

Any code generated is to be treated as any outside code that is not trusted. Prompting should be included in the prompt, but this is just the first control. Each suggestion will have to be evaluated by the developers, and accepted code should go through the same security pipeline as hand-written code. The studies reviewed indicate that explicit security instructions help, but by themselves are not sufficient to prevent, insecure output [37]-[39].

15.2 For Security and Platform Teams

Model access should be done via organizational infrastructure and not via individual accounts. The successful integration should also be user friendly and not avoidable. Accordingly, several authors prefer a platform based enforcement of controls rather than policies that require each developer to implement them on a uniform basis [163] - [167].

15.3 For Tool Vendors

Security enforcement should be done underneath the level of the model so it is not reliant on the model behavior. Instructions should be separated from data in the vendor layer; capability should be communicated in tool schemas; agent actions should be exposed in audit interfaces. The

evidence points towards controls that are effective even when the model is unpredictable [121,125].

15.4 For Educators

Several authors argue that students should learn to evaluate generated output independently before relying on assisted generation [102], [104]. Teaching should therefore require them to audit generated code, map flaws to CWE, design adversarial tests, and compare the results with secure reference implementations.

16. Threats To Validity

Four limitations apply to this review. Successful attacks and prominent tools may be favored over negative results and less visible products by publication and novelty bias. Completeness for a rapidly evolving field cannot be guaranteed by the search, despite its systematic nature. Subjective judgments about study design and evidence strength are involved in quality appraisal. Finally, the available evidence as of the cutoff date shapes the synthesis; some conclusions may be altered by new findings.

Reference verification was performed at the record level, but bibliographic metadata for very recent preprints and technical disclosures may be incomplete or subject to revision. We have flagged preprint status where applicable and consolidated identifiable preprint and published versions.

17. Conclusion

AI pair programmers are a double-edge security technology. They can take weakness knowledge, repair capability, and secure-coding guidance to the point of writing; they also generate recognized weakness classes at rates that vary based on task, language, prompt, and model version. This shift from suggestion to agency alters the nature of the security problem, and is the most important finding of this review. Autocomplete adds defects and agents add incidents.

The change from suggestion to agency is the key. Controls against insecure snippets are still necessary, but not agents! The autocomplete can provide a weak piece, this agentic system can perform a series of tool invocations that updates the repository, installs dependencies, modifies configurations, and sets up an ongoing back-door for the user to have access to the target system.

Creating code early or quickly isn't enough to move security left. Secure adoption demands continual assurance: Structured context, limited authority, independent verification, true authorization to take high impact action, and context model, tools and context sources as part of the trusted computing base. Otherwise, the security debt gets pushed downstream where the cost to remediate gets higher and it becomes more dangerous.

References

[1] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," EBSE Technical Report EBSE-2007-01, Keele Univ. and Durham Univ., 2007.

[2] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," Information and Software Technology, vol. 64, pp. 1-18, 2015.

[3] M. J. Page et al., "The PRISMA 2020 statement: An updated guideline for reporting systematic reviews," BMJ, vol. 372, Art. no. n71, 2021.

[4] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in Proc. 18th Int. Conf. Evaluation and Assessment in Software Engineering (EASE), 2014, pp. 1-10.

[5] E. Basic and A. Giaretta, “From Vulnerabilities to Remediation: A Systematic Literature Review of LLMs in Code Security,” arXiv:2412.15004, 2024.

[6] C. Negri-Ribalta, R. Geraud-Stewart, A. Sergeeva, and G. Lenzini, "A systematic literature review on the impact of AI models on the security of code generation," Frontiers in Big Data, vol. 7, Art. no. 1386720, 2024.

[7] Y. Chen et al., "Security of language models for code: A systematic literature review," arXiv:2410.15631, 2024.

[8] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large language model for vulnerability detection and repair: Literature review and roadmap," arXiv:2404.02525, 2024.

[9] X. Hou et al., "Large language models for software engineering: A systematic literature review," ACM Transactions on Software Engineering and Methodology, 2024.

[10] J. Zhang, H. Bu, H. Wen, Y. Chen, L. Li, and H. Zhu, "When LLMs meet cybersecurity: A systematic literature review," arXiv:2405.03644, 2024.

[11] H. Xu et al., "Large language models for cyber security: A systematic literature review," arXiv:2405.04760, 2024.

[12] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, "A survey on large language model security and privacy: The good, the bad, and the ugly," High-Confidence Computing, vol. 4, no. 2, Art. no. 100211, 2024.

[13] X. Jiang et al., "A survey on large language models for code generation," arXiv:2406.00515, 2024.

[14] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, "No need to lift a finger anymore? Assessing the quality of code generation by ChatGPT," IEEE Transactions on Software Engineering, 2024.

[15] M. Chen et al., "Evaluating large language models trained on code," arXiv:2107.03374, 2021.

[16] J. Austin et al., "Program synthesis with large language models," arXiv:2108.07732, 2021.

[17] E. Nijkamp et al., "CodeGen: An open large language model for code with multi-turn program synthesis," in Proc. Int. Conf. Learning Representations (ICLR), 2023.

[18] D. Fried et al., "InCoder: A generative model for code infilling and synthesis," in Proc. Int. Conf. Learning Representations (ICLR), 2023.

[19] Y. Li et al., "Competition-level code generation with AlphaCode," Science, vol. 378, no. 6624, pp. 1092-1097, 2022.

[20] B. Roziere et al., "Code Llama: Open foundation models for code," arXiv:2308.12950, 2023.

[21] R. Li et al., "StarCoder: May the source be with you!" Transactions on Machine Learning Research, 2023.

[22] Z. Luo et al., "WizardCoder: Empowering code large language models with Evol-Instruct," in Proc. Int. Conf. Learning Representations (ICLR), 2024.

[23] Y. Wang et al., "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in Proc. EMNLP, 2021, pp. 8696-8708.

[24] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in Findings of EMNLP, 2020, pp. 1536-1547.

[25] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in Proc. Int. Conf. Learning Representations (ICLR), 2021.

[26] W. U. Ahmad et al., "Unified pre-training for program understanding and generation," in Proc. NAACL-HLT, 2021, pp. 2655-2668.

[27] H. Le, Y. Wang, A. Gotmare, S. Savarese, and S. Hoi, "CodeRL: Mastering code generation through pretrained models and deep reinforcement learning," in Advances in Neural Information Processing Systems, vol. 35, 2022, pp. 21314-21328.

[28] S. Lu et al., "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in NeurIPS Datasets and Benchmarks, 2021.

[29] C. E. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?" in Proc. Int. Conf. Learning Representations (ICLR), 2024.

[30] J. Yang et al., "SWE-agent: Agent-computer interfaces enable automated software engineering," in Advances in Neural Information Processing Systems, 2024.

[31] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying LLM-based software engineering agents," arXiv:2407.01489, 2024.

[32] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," in Proc. Int. Conf. Learning Representations (ICLR), 2023.

[33] N. Shinn et al., "Reflexion: Language agents with verbal reinforcement learning," in Advances in Neural Information Processing Systems, vol. 36, 2023.

[34] A. Madaan et al., "Self-Refine: Iterative refinement with self-feedback," in Advances in Neural Information Processing Systems, vol. 36, 2023.

[35] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in Proc. IEEE Symposium on Security and Privacy, 2022, pp. 754-768.

[36] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How secure is code generated by ChatGPT?" in Proc. IEEE Int. Conf. Systems, Man, and Cybernetics, 2023, pp. 2445-2451.

[37] O. Asare, M. Nagappan, and N. Asokan, "Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code?" Empirical Software Engineering, vol. 28, no. 6, Art. no. 129, 2023.

[38] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?" in Proc. ACM SIGSAC Conf. Computer and Communications Security, 2023, pp. 2785-2799.

[39] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at C: A user study on the security implications of large language model code assistants," in Proc. 32nd USENIX Security Symposium, 2023, pp. 2205-2222.

[40] R. Elgedawy et al., "Occasionally secure: A comparative analysis of code generation assistants," arXiv:2402.00689, 2024.

[41] M. L. Siddiq, J. C. da Silva Santos, S. Devareddy, and A. Muller, "SALLM: Security assessment of generated code," in Proc. IEEE/ACM ASE Workshops, 2024, pp. 54-65.

[42] H. Hajipour, K. Hassler, T. Holz, L. Schonherr, and M. Fritz, "CodeLMSec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models," in Proc. IEEE Conf. Secure and Trustworthy Machine Learning, 2024, pp. 684-709.

[43] M. L. Siddiq and J. C. S. Santos, "SecurityEval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques," in Proc. Workshop on Mining Software Repositories Applications for Privacy and Security, 2022, pp. 29-33.

[44] R. Toth, T. Bisztray, and L. Erdodi, "LLMs in web development: Evaluating LLM-generated PHP code, unveiling vulnerabilities and limitations," in SAFECOMP Workshops, 2024, pp. 425-437.

[45] N. Tihanyi, T. Bisztray, M. A. Ferrag, R. Jain, and L. C. Cordeiro, "Do neutral prompts produce insecure code? FormAI-v2: Labelling vulnerabilities in code generated by large language models," arXiv:2404.18353, 2024.

[46] S. Hamer, M. d'Amorim, and L. Williams, "Just another copy and paste? Comparing the security vulnerabilities of ChatGPT-generated code and Stack Overflow answers," in Proc. IEEE Security and Privacy Workshops, 2024, pp. 87-94.

[47] M. F. Rabbi, A. I. Champa, M. F. Zibran, and M. R. Islam, "AI writes, we analyze: The ChatGPT Python code saga," in Proc. Mining Software Repositories, 2024.

[48] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, and J. Yu, "Security weaknesses of Copilot-generated code in GitHub," arXiv:2310.02059, 2023.

[49] J. Gong, N. Duan, Z. Tao, Z. Gong, Y. Yuan, and M. Huang, "How well do large language models serve as end-to-end secure code producers?" arXiv:2408.10495, 2024.

[50] A. Mohsin et al., "Can we trust large language model generated code? A framework for in-context learning, security patterns, and code evaluations across diverse LLMs," arXiv:2406.12513, 2024.

[51] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in Proc. IEEE Symposium on Security and Privacy, 2023, pp. 2339-2356.

[52] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in Proc. ACM SIGSAC Conf. Computer and Communications Security, 2023, pp. 1865-1879.

[53] J. He, M. Vero, G. Krasnopolska, and M. Vechev, "Instruction tuning for secure code generation," in Proc. Int. Conf. Machine Learning, vol. 235, 2024, pp. 18043-18062.

[54] C. Tony, N. E. D. Ferreyra, M. A. Huq, S. S. S. Das, and D. Spinellis, "Prompting techniques for secure code generation: A systematic investigation," arXiv:2407.07064, 2024.

[55] S. Y. Kim, Z. Fan, Y. Noller, and A. Roychoudhury, "Codexity: Secure AI-assisted code generation," arXiv:2405.03927, 2024.

[56] T. K. Le, S. Alimadadi, and S. Y. Ko, "A study of vulnerability repair in JavaScript programs with large language models," in Companion Proc. ACM Web Conf., 2024, pp. 666-669.

[57] L. Zhang, Q. Zou, A. Singhal, X. Sun, and P. Liu, "Evaluating large language models for real-world vulnerability repair in C/C++ code," in Proc. ACM Workshop on Security and Privacy Analytics, 2024, pp. 49-58.

[58] Y. Nong, H. Yang, L. Cheng, H. Hu, and H. Cai, "Automated software vulnerability patching using large language models," arXiv:2408.13597, 2024.

[59] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "VulRepair: A T5-based automated software vulnerability repair," in Proc. ACM Joint European Software Engineering Conf. and Symposium on Foundations of Software Engineering, 2022, pp. 935-947.

[60] B. Berabi, A. Gronskiy, V. Raychev, G. Sivanrupan, V. Chibotaru, and M. Vechev, "DeepCode AI Fix: Fixing security vulnerabilities with large language models," arXiv:2402.13291, 2024.

[61] D. de Fitero-Dominguez, E. Garcia-Lopez, A. Garcia-Cabot, and J.-J. Martinez-Herraiz, "Enhanced automated code vulnerability repair using large language models," Engineering Applications of Artificial Intelligence, vol. 138, Art. no. 109291, 2024.

[62] Z. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in C code," IEEE Transactions on Software Engineering, vol. 49, no. 1, pp. 147-165, 2023.

[63] X. Zhou, K. Kim, B. Xu, D. Han, and D. Lo, "Large language model as synthesizer: Fusing diverse inputs for better automatic vulnerability repair," arXiv:2401.15459, 2024.

[64] Y. Wu et al., "How effective are neural networks for fixing security vulnerabilities?" in Proc. ACM ISSTA, 2023, pp. 1282-1294.

[65] Y. Nong, M. Aldeen, L. Cheng, H. Hu, F. Chen, and H. Cai, "Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities," arXiv:2402.17230, 2024.

[66] P. Liu et al., "Exploring ChatGPT's capabilities on vulnerability management," in Proc. 33rd USENIX Security Symposium, 2024, pp. 811-828.

[67] P. Liu, H. Wang, C. Zheng, and Y. Zhang, "PromptFix: Vulnerability automatic repair technology based on prompt engineering," in Proc. Int. Conf. Computing, Networking and Communications, 2024, pp. 116-120.

[68] U. Kulsum, H. Zhu, B. Xu, and M. d'Amorim, "A case study of LLM for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback," in Proc. ACM AIware, 2024, pp. 103-111.

[69] A. Storhaug, J. Li, and T. Hu, "Efficient avoidance of vulnerabilities in auto-completed smart contract code using vulnerability-constrained decoding," in Proc. IEEE ISSRE, 2023, pp. 683-693.

[70] Y. Yang et al., "DLAP: A deep learning augmented large language model prompting framework for software vulnerability detection," Journal of Systems and Software, vol. 219, Art. no. 112234, 2025.

[71] J. Bae, S. Kwon, and S. Myeong, "Enhancing software code vulnerability detection using GPT-4o and Claude-3.5 Sonnet: A study on prompt engineering techniques," Electronics, vol. 13, no. 13, Art. no. 2657, 2024.

[72] M. M. Mohajer et al., "Effectiveness of ChatGPT for static analysis: How far are we?" in Proc. ACM Int. Conf. AI-Powered Software, 2024, pp. 151-160.

[73] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Assisting static analysis with large language models: A ChatGPT experiment," in Proc. ACM Joint European Software Engineering Conf. and Symposium on Foundations of Software Engineering, 2023, pp. 2107-2111.

[74] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An LLM-integrated approach," Proceedings of the ACM on Programming Languages, vol. 8, no. OOPSLA1, pp. 474-499, 2024.

[75] C. Fang et al., "Large language models for code analysis: Do LLMs really do their job?" in Proc. 33rd USENIX Security Symposium, 2024, pp. 829-846.

[76] B. Steenhoek et al., "A comprehensive study of the capabilities of large language models for vulnerability detection," arXiv:2403.17218, 2024.

[77] J. Yu et al., "Security code review by LLMs: A deep dive into responses," arXiv:2401.16310, 2024.

[78] X.-C. Wen, C. Gao, S. Gao, Y. Xiao, and M. R. Lyu, "SCALE: Constructing structured natural language comment trees for software vulnerability detection," in Proc. ACM ISSTA, 2024, pp. 235-247.

[79] X.-C. Wen, X. Wang, Y. Chen, R. Hu, D. Lo, and C. Gao, "VulEval: Towards repository-level evaluation of software vulnerability detection," arXiv:2404.15596, 2024.

[80] J. Zhang et al., "An empirical study of automated vulnerability localization with large language models," arXiv:2404.00287, 2024.

[81] M. D. Purba, A. Ghosh, B. J. Radford, and B. Chu, "Software vulnerability detection using large language models," in Proc. IEEE ISSRE Workshops, 2023, pp. 112-119.

[82] Y. Liu et al., "VulDetectBench: Evaluating the deep capability of vulnerability detection with large language models," arXiv:2406.07595, 2024.

[83] Y. Ding et al., "Vulnerability detection with code language models: How far are we?" in Proc. IEEE/ACM ICSE, 2025, pp. 469-481.

[84] O. Cetin, E. Ekmekcioglu, B. Arief, and J. Hernandez-Castro, "An empirical evaluation of large language models in static code analysis for PHP vulnerability detection," Journal of Universal Computer Science, 2024.

[85] E. Pelofske, V. Urias, and L. M. Liebrock, "Automated software vulnerability static code analysis using generative pre-trained transformer models," arXiv:2408.00197, 2024.

[86] C. Chen et al., "When ChatGPT meets smart contract vulnerability detection: How far are we?" ACM Transactions on Software Engineering and Methodology, 2025, doi: 10.1145/3702973.

[87] Y. Sun et al., "GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis," in Proc. IEEE/ACM ICSE, 2024, pp. 1-13.

[88] Z. Gao, H. Wang, Y. Zhou, W. Zhu, and C. Zhang, "How far have we gone in vulnerability detection using large language models?" arXiv:2311.12420, 2023.

[89] C. Zhang et al., "Prompt-enhanced software vulnerability detection using ChatGPT," in Proc. IEEE/ACM ICSE Companion, 2024, pp. 276-277.

[90] A. Bakhshandeh, A. Keramatfar, A. Norouzi, and M. M. Chekidehkhoun, "Using ChatGPT as a static application security testing tool," ISeCure, vol. 15, no. 3, 2023.

[91] X. Zhou et al., "Comparison of static application security testing tools and large language models for repository-level vulnerability detection," arXiv:2407.16235, 2024.

[92] K. Tamberg and H. Bahsi, "Harnessing large language models for software vulnerability detection: A comprehensive benchmarking study," arXiv:2405.15614, 2024.

[93] D. Noever, "Can large language models find and fix vulnerable software?" arXiv:2308.10345, 2023.

[94] O. S. Ozturk, E. Ekmekcioglu, O. Cetin, B. Arief, and J. Hernandez-Castro, "New tricks to old code: Can AI chatbots replace static code analysis tools?" in Proc. EICC, 2023, pp. 13-18.

[95] Y. Guo, C. Patsakis, Q. Hu, Q. Tang, and F. Casino, "Outside the comfort zone: Analysing LLM capabilities in software vulnerability detection," in Proc. ESORICS, 2024, pp. 271-289.

[96] N. S. Mathews, Y. Brus, Y. Aafer, M. Nagappan, and S. McIntosh, "LLBezpeky: Leveraging large language models for vulnerability detection," arXiv:2401.01269, 2024.

[97] Q. Mao, Z. Li, X. Hu, K. Liu, X. Xia, and J. Sun, "Towards effectively detecting and explaining vulnerabilities using large language models," arXiv:2406.09701, 2024.

[98] A. Happe and J. Cito, "Getting pwn'd by AI: Penetration testing with large language models," in Proc. ACM Joint European Software Engineering Conf. and Symposium on Foundations of Software Engineering, 2023, pp. 2082-2086.

[99] M. Fu and C. Tantithamthavorn, "LineVul: A transformer-based line-level vulnerability prediction," in Proc. Mining Software Repositories, 2022, pp. 608-620.

[100] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in Advances in Neural Information Processing Systems, vol. 32, 2019.

[101] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" IEEE Transactions on Software Engineering, vol. 48, no. 9, pp. 3280-3296, 2022.

[102] Z. Li et al., "VulDeePecker: A deep learning-based system for vulnerability detection," in Proc. Network and Distributed System Security Symposium, 2018.

[103] H. Hanif and S. Maffeis, "VulBERTa: Simplified source code pre-training for vulnerability detection," in Proc. Int. Joint Conf. Neural Networks, 2022, pp. 1-8.

[104] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, "Transformer-based language models for software vulnerability detection," in Proc. Annual Computer Security Applications Conference, 2022, pp. 481-496.

[105] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," Proceedings of the ACM on Programming Languages, vol. 3, no. OOPSLA, Art. no. 159, 2019.

[106] C. Niu et al., "CodexLeaks: Privacy leaks from code generation language models in GitHub Copilot," in Proc. 32nd USENIX Security Symposium, 2023, pp. 2133-2150.

[107] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You autocomplete me: Poisoning vulnerabilities in neural code completion," in Proc. 30th USENIX Security Symposium, 2021, pp. 1559-1575.

[108] H. Aghakhani et al., "TrojanPuzzle: Covertly poisoning code-suggestion models," in Proc. IEEE Symposium on Security and Privacy, 2024, pp. 1122-1140.

[109] S. Oh, K. Lee, S. Park, D. Kim, and H. Kim, "Poisoned ChatGPT finds work for idle hands: Exploring developers' coding practices with insecure suggestions from poisoned AI models," in Proc. IEEE Symposium on Security and Privacy, 2024, pp. 1141-1159.

[110] S. Yan et al., "An LLM-assisted easy-to-trigger backdoor attack on code completion models: Injecting disguised vulnerabilities against strong detection," in Proc. 33rd USENIX Security Symposium, 2024.

[111] D. Cotroneo, C. Improta, P. Liguori, and R. Natella, "Vulnerabilities in AI code generators: Exploring targeted data poisoning attacks," in Proc. IEEE/ACM Int. Conf. Program Comprehension, 2024, pp. 280-292.

[112] J. Spracklen, R. Wijewickrama, A. H. M. N. Sakib, A. Maiti, B. Viswanath, and M. Jadliwala, "We have a package for you! A comprehensive analysis of package hallucinations by code-generating LLMs," arXiv:2406.10279, 2024.

[113] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open-source software supply chain attacks," in Proc. DIMVA, 2020, pp. 23-43.

[114] P. Ladisa et al., "SoK: Taxonomy of attacks on open-source software supply chains," in Proc. IEEE Symposium on Security and Privacy, 2023, pp. 1509-1526.

[115] N. Carlini et al., "Extracting training data from large language models," in Proc. 30th USENIX Security Symposium, 2021, pp. 2633-2650.

[116] N. Carlini, D. Ippolito, M. Jagielski, K. Lee, F. Tramer, and C. Zhang, "Quantifying memorization across neural language models," in Proc. Int. Conf. Learning Representations, 2023.

[117] K. Greshake et al., "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," in Proc. ACM Workshop on Artificial Intelligence and Security, 2023, pp. 79-90.

[118] S. Schulhoff et al., "Ignore this title and HackAPrompt: Exposing systemic vulnerabilities of LLMs through a global prompt hacking competition," in Proc. EMNLP, 2023, pp. 4945-4977.

[119] A. Zou, Z. Wang, N. Carlini, M. Nasr, J. Z. Kolter, and M. Fredrikson, "Universal and transferable adversarial attacks on aligned language models," arXiv:2307.15043, 2023.

[120] A. Wei, N. Haghtalab, and J. Steinhardt, "Jailbroken: How does LLM safety training fail?" in Advances in Neural Information Processing Systems, vol. 36, 2023.

[121] P. Chao, A. Robey, E. Dobriban, H. Hassani, G. J. Pappas, and E. Wong, "Jailbreaking black-box large language models in twenty queries," arXiv:2310.08419, 2023.

[122] A. Mehrotra et al., "Tree of attacks: Jailbreaking black-box LLMs automatically," in Advances in Neural Information Processing Systems, 2024.

[123] X. Liu, N. Xu, M. Chen, and C. Xiao, "AutoDAN: Generating stealthy jailbreak prompts on aligned large language models," in Proc. Int. Conf. Learning Representations, 2024.

[124] E. Bagdasaryan, T. Hsieh, B. Nassi, and V. Shmatikov, "(Ab)using images and sounds for indirect instruction injection in multimodal LLMs," arXiv:2307.10490, 2023.

[125] Y. Gong et al., "FigStep: Jailbreaking large vision-language models via typographic visual prompts," arXiv:2311.05608, 2023.

[126] W. Zou, R. Geng, B. Wang, and J. Jia, "PoisonedRAG: Knowledge corruption attacks to retrieval-augmented generation of large language models," in Proc. 34th USENIX Security Symposium, 2025.

[127] C. Clop and Y. Teglia, "Backdoored retrievers for prompt injection attacks on retrieval-augmented generation of large language models," arXiv:2410.14479, 2024.

[128] P. Cheng et al., "TrojanRAG: Retrieval-augmented generation can be a backdoor driver in large language models," arXiv:2405.13401, 2024.

[129] E. Hubinger et al., "Sleeper agents: Training deceptive LLMs that persist through safety training," arXiv:2401.05566, 2024.

[130] Y. Liu, Y. Zhao, Y. Lyu, T. Zhang, H. Wang, and D. Lo, "Your AI, my shell: Demystifying prompt injection attacks on agentic AI coding editors," arXiv:2509.22040, 2025.

[131] A. Storek, M. Gupta, N. Bhatt, A. Gupta, J. Kim, P. Srivastava, and S. Jana, "XOXO: Stealthy cross-origin context poisoning attacks against AI coding assistants," arXiv:2503.14281, 2025.

[132] A. Marzouk, "IDEsaster: Security vulnerabilities in AI-powered integrated development environments," Technical Report, 2025.

[133] S. Huang, Y. Huang, and F. H. Fard, "Are AI-assisted development tools immune to prompt injection? An empirical study of MCP-enabled coding systems," arXiv:2603.21642, 2026.

[134] S. Gaire, S. Gyawali, S. Mishra, S. Niroula, D. Thakur, and U. Yadav, "Systematization of knowledge: Security and safety in the Model Context Protocol ecosystem," arXiv:2512.08290, 2025.

[135] M. A. Ferrag, N. Tihanyi, D. Hamouda, L. Maglaras, and M. Debbah, "From prompt injections to protocol exploits: Threats in LLM-powered AI-agent workflows," arXiv:2506.23260, 2025.

[136] M. Bhatt, V. S. Narajala, and I. Habler, "ETDI: Mitigating tool squatting and rug-pull attacks in Model Context Protocol," arXiv:2506.01333, 2025.

[137] S. Jamshidi, A. M. Dakhel, K. W. Nafi, and F. Khomh, "Semantic Attacks on Tool-Augmented LLMs: Securing the Model Context Protocol Against Descriptor-Level Manipulation," arXiv:2512.06556, 2025.

[138] X. Hou et al., "Model Context Protocol: Landscape, security threats, and future research directions," arXiv:2503.23278, 2025.

[139] Y. Yang, D. Wu, and Y. Chen, "MCPSecBench: A systematic security benchmark and playground for testing Model Context Protocols," arXiv:2508.13220, 2025.

[140] E. Debenedetti, J. Zhang, M. Balunovic, L. Beurer-Kellner, M. Fischer, and F. Tramer, "AgentDojo: A dynamic environment to evaluate prompt injection attacks and defenses for LLM agents," in Advances in Neural Information Processing Systems, 2024.

[141] M. Andriushchenko et al., "AgentHarm: A benchmark for measuring harmfulness of LLM agents," in Proc. Int. Conf. Learning Representations, 2025.

[142] H. Zhang et al., "Agent Security Bench: Formalizing and benchmarking attacks and defenses in LLM-based agents," in Proc. Int. Conf. Learning Representations, 2025.

[143] S. Toyer et al., "Tensor Trust: Interpretable prompt injection attacks from an online game," in Advances in Neural Information Processing Systems, vol. 36, 2023.

[144] T. Yuan et al., "R-Judge: Benchmarking safety risk awareness for LLM agents," in Findings of EMNLP, 2024.

[145] Y. Ruan et al., "Identifying the risks of LM agents with an LM-emulated sandbox," in Proc. Int. Conf. Learning Representations, 2024.

[146] Q. Zhan, R. Fang, R. Bindu, A. Gupta, T. Hashimoto, and D. Kang, "InjecAgent: Benchmarking indirect prompt injections in tool-integrated LLM agents," in Findings of ACL, 2024.

[147] M. Mazeika et al., "HarmBench: A standardized evaluation framework for automated red teaming and robust refusal," arXiv:2402.04249, 2024.

[148] P. Chao et al., "JailbreakBench: An open robustness benchmark for jailbreaking large language models," in Advances in Neural Information Processing Systems, 2024.

[149] Y. Liu, Y. Deng, R. Xu, Y. Wang, and Y. Liu, "Formalizing and benchmarking prompt injection attacks and defenses," in Proc. 33rd USENIX Security Symposium, 2024.

[150] J. Yi et al., "Benchmarking and defending against indirect prompt injection attacks on large language models," arXiv:2312.14197, 2023.

[151] E. Wallace, K. Xiao, R. Leike, L. Weng, J. Heidecke, and A. Beutel, "The instruction hierarchy: Training LLMs to prioritize privileged instructions," arXiv:2404.13208, 2024.

[152] E. Debenedetti et al., "Defeating prompt injections by design," arXiv:2503.18813, 2025.

[153] Y. Wu, F. Roesner, T. Kohno, N. Zhang, and U. Iqbal, "IsolateGPT: An execution isolation architecture for LLM-based agentic systems," in Proc. Network and Distributed System Security Symposium, 2025.

[154] S. Chen, J. Piet, C. Sitawarin, and D. Wagner, "StruQ: Defending against prompt injection with structured queries," in Proc. 34th USENIX Security Symposium, 2025.

[155] S. Chen, A. Zharmagambetov, S. Mahloujifar, K. Chaudhuri, D. Wagner, and C. Guo, "SecAlign: Defending against prompt injection with preference optimization," in Proc. ACM SIGSAC Conf. Computer and Communications Security, 2025.

[156] T. Shi, J. He, Z. Wang, L. Wu, H. Li, W. Guo, and D. Song, "Progent: Programmable privilege control for LLM agents," arXiv:2504.11703, 2025.

[157] Z. Wang et al., "MELON: Provable defense against indirect prompt injection attacks in AI agents," in Proc. Int. Conf. Machine Learning, 2025.

[158] H. Inan et al., "Llama Guard: LLM-based input-output safeguard for human-AI conversations," arXiv:2312.06674, 2023.

[159] T. Rebedea, R. Dinu, M. N. Sreedhar, C. Parisien, and J. Cohen, "NeMo Guardrails: A toolkit for controllable and safe LLM applications with programmable rails," in Proc. EMNLP System Demonstrations, 2023, pp. 431-445.

[160] K. Hines et al., "Defending against indirect prompt injection attacks with spotlighting," Microsoft Research Technical Report, 2024.

[161] M. Nasr et al., "The attacker moves second: Stronger adaptive attacks bypass defenses against LLM jailbreaks and prompt injections," arXiv:2510.09023, 2025.

[162] F. Tramer, N. Carlini, W. Brendel, S. Madry, and A. Kurakin, "On adaptive attacks to adversarial example defenses," in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 1633-1645.

[163] National Institute of Standards and Technology, "Secure Software Development Framework (SSDF) Version 1.1," NIST SP 800-218, Feb. 2022.

[164] National Institute of Standards and Technology, "Artificial Intelligence Risk Management Framework (AI RMF 1.0)," NIST AI 100-1, Jan. 2023.

[165] OWASP Foundation, "OWASP Top 10 for Large Language Model Applications 2025," 2025.

[166] MITRE, "Common Weakness Enumeration (CWE)," 2026. [Online]. Available: https://cwe.mitre.org/

[167] MITRE, "2024 CWE Top 25 Most Dangerous Software Weaknesses," 2024.

[168] R. Paul, A. K. Turzo, and A. Bosu, "Why security defects go unnoticed during code reviews? A case-control study of the Chromium OS project," in Proc. IEEE/ACM ICSE, 2021, pp. 1373-1385.

[169] S. Barke, M. B. James, and N. Polikarpova, "Grounded Copilot: How programmers interact with code-generating models," Proceedings of the ACM on Programming Languages, vol. 7, no. OOPSLA1, Art. no. 78, 2023.

[170] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in CHI Conf. Extended Abstracts, 2022, pp. 1-7.

[171] A. Ziegler et al., "Measuring GitHub Copilot's impact on productivity," Communications of the ACM, vol. 67, no. 3, pp. 54-63, 2024.

[172] A. M. Dakhel et al., "GitHub Copilot AI pair programmer: Asset or liability?" Journal of Systems and Software, vol. 203, Art. no. 111734, 2023.

[173] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub Copilot's code suggestions," in Proc. Mining Software Repositories, 2022, pp. 1-5.

[174] S. Imai, "Is GitHub Copilot a substitute for human pair-programming? An empirical study," in Proc. ACM/IEEE Int. Symposium on Empirical Software Engineering and Measurement, 2022, pp. 319-321.

[175] J. Prather et al., "It's weird that it knows what I want: Usability and interactions with Copilot for novice programmers," ACM Transactions on Computer-Human Interaction, 2024.

[176] M. Kazemitabaar et al., "How novices use LLM-based code generators to solve CS1 coding tasks in a self-paced learning environment," in Proc. Koli Calling, 2023, Art. no. 19.

[177] B. Zhang et al., "Practices and challenges of using GitHub Copilot: An empirical study," arXiv:2303.08733, 2023.

[178] R. Choudhuri et al., "What Guides Our Choices? Modeling Developers' Trust and Behavioral Intentions Towards GenAI," arXiv:2409.04099, 2024.

[179] N. E. Díaz Ferreyra, M. S. Gurupathi, Z. Codabux, N. Arachchilage, and R. Scandariato, "Security concerns in generative AI coding assistants: Insights from online discussions on GitHub Copilot," arXiv:2604.08352, 2026.

[180] G. O'Brien, A. Parker, N. Eisty, and J. Carver, "A Survey of Generative AI Adoption and Perceived Productivity Among Scientists Who Program," arXiv:2512.19644, 2025.

[181] M. Bhatt et al., "Purple Llama CyberSecEval: A secure coding benchmark for language models," arXiv:2312.04724, 2023.

[182] P. E. Black, "SARD: A software assurance reference dataset," NIST, 2017.

[183] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," Proceedings of the IEEE, vol. 63, no. 9, pp. 1278-1308, 1975.

[184] G. McGraw, Software Security: Building Security In. Boston, MA, USA: Addison-Wesley, 2006.

[185] A. Shostack, Threat Modeling: Designing for Security. Indianapolis, IN, USA: Wiley, 2014.

[186] OWASP Foundation, "Application Security Verification Standard 4.0.3," 2021.

[187] OpenSSF, "Supply-chain Levels for Software Artifacts (SLSA) Specification, version 1.0," 2023.

[188] Check Point Research, "Cursor IDE's MCP Vulnerability: MCPoison," 2025. [Online]. Available: https://research.checkpoint.com/2025/cursor-vulnerability-mcpoison/.

[189] Cato Networks, "DuneSlide: Two Critical RCE Vulnerabilities," Jul. 2026. [Online]. Available: https://www.catonetworks.com/blog/duneslide-two-critical-rce-vulnerabilities/.

[190] Check Point Research, "Claude Code vulnerabilities: Project-file prompt injection and command execution risks," Technical Disclosure, 2025.

[191] GitHub Advisory Database, "OpenAI Codex CLI Enables Code Execution Through Malicious MCP Configuration," CVE-2025-61260, 2025. [Online]. Available: https://github.com/advisories/GHSA-xrxf-jgv3-qmrm.

[192] S. Shukla, H. Joshi, and R. Syed, "Security degradation in iterative AI code generation: A systematic analysis of the paradox," in Proc. IEEE Int. Symposium on Technology and Society (ISTAS), 2025; arXiv:2506.11022.

[193] M. Kozak, R. Z. Moghaddam, and S. Sivaraman, "When developer aid becomes security debt: A systematic analysis of insecure behaviors in LLM coding agents," arXiv:2507.09329, 2025.

[194] J. Isbarov, U. Suleymanov, I. Shumailov, and M. Kantarcioglu, "GitInject: Real-world prompt injection attacks in AI-powered CI/CD pipelines," arXiv:2606.09935, 2026.

[195] S. Wang et al., "Demystifying and detecting agentic workflow injection vulnerabilities in GitHub Actions," arXiv:2605.07135, 2026.

[196] W. Choi, J. Kim, T. Kang, J. Jeong, L. Xing, and B. Lee, "Agent data injection attacks are realistic threats to AI agents," arXiv:2607.05120, 2026.

[197] Y. He, H. Zhu, Y. Li, S. Shao, H. Yao, Z. Liu, and Z. Qin, "AttriGuard: Defeating indirect prompt injection in LLM agents via causal attribution of tool invocations," in Proc. 35th USENIX Security Symposium, 2026.

[198] H. Chang, E. Bao, X. Luo, and T. Yu, "Overcoming the retrieval barrier: Indirect prompt injection in the wild for LLM systems," in Proc. 35th USENIX Security Symposium, 2026.

[199] D. Hofer, E. Debenedetti, and F. Tramer, "Assessing automated prompt injection attacks in agentic environments," arXiv:2606.10525, 2026.

[200] N. Maloyan and D. Namiot, "Prompt injection attacks on agentic coding assistants: A systematic analysis of vulnerabilities in skills, tools, and protocol ecosystems," arXiv:2601.17548, 2026.

[201] F. H. Bappy et al., "From preventive to reactive: How AI coding assistants transform developers' security awareness," arXiv:2605.23130, 2026.

[202] A. Naiakshina, J. Speckels, A.-M. Ortloff, N. Jost, R. Serafini, and A. Yardim, "The AI tool can't make it any worse: Investigating developers' security behavior with AI assistants in a password storage study," in Proc. ACM CHI Conf. Human Factors in Computing Systems, 2026.

[203] J. H. Klemmer et al., "Using AI assistants in software development: A qualitative study on security practices and concerns," in Proc. ACM SIGSAC Conf. Computer and Communications Security, 2024, pp. 2726-2740.

[204] A. Kudriavtseva, N. A. Hotak, and O. Gadyatskaya, "My code is less secure with Gen AI: Surveying developers' perceptions of the impact of code generation tools on security," in Proc. 40th ACM/SIGAPP Symposium on Applied Computing, 2025, pp. 1637-1646.

[205] M. Kharma, S. Choi, M. AlKhanafseh, and D. Mohaisen, "Security and quality in LLM-generated code: A multi-language, multi-model analysis," arXiv:2502.01853, 2025.

[206] A. Sabra, O. Schmitt, and J. T. Sonar, "Assessing the quality and security of AI-generated code: A quantitative analysis," arXiv:2508.14727, 2025.

[207] B. Wang et al., "AI code in the wild: Measuring security risks and ecosystem shifts of AI-generated code in modern software," arXiv:2512.18567, 2025.

[208] V. Belozerov, P. J. Barclay, and A. Sami, "Secure Coding with AI—From Detection to Repair," arXiv:2504.20814, 2025.

[209] J. Li, A. Sangalay, C. Cheng, Y. Tian, and J. Yang, "Fine tuning large language model for secure code generation," in Proc. ACM/IEEE Int. Workshop on AI Foundation Models and Software Engineering, 2024, doi: 10.1145/3650105.3652299.

[210] M. Nazzal, I. Khalil, A. Khreishah, and N. H. Phan, "PromSec: Prompt optimization for secure generation of functional source code with large language models," in Proc. ACM SIGSAC Conf. Computer and Communications Security, 2024, doi: 10.1145/3658644.3690298.

[211] B. Lin, S. Wang, Y. Qin, L. Chen, and X. Mao, "Give LLMs a security course: Securing retrieval-augmented code generation via knowledge injection," in Proc. ACM SIGSAC Conf. Computer and Communications Security, 2025, pp. 3356-3370.

[212] M. Tessa, I. E. Olatunji, A. War, J. Klein, and T. F. Bissyande, "How secure is secure code generation? Adversarial prompts put LLM defenses to the test," arXiv:2601.07084, 2026.

[213] X. He, D. Li, H. Wen, Y. Zhu, C. Liu, M. Yan, and H. Zhang, "CoSEFA: An LLM-based programming assistant for secure code generation via supervised co-decoding," in Proc. ACM SIGSOFT Int. Symposium on Foundations of Software Engineering Companion, 2025, pp. 1198-1202.

[214] H. Li, R. Wen, S. Shi, N. Zhang, Y. Vorobeychik, and C. Xiao, "AgentDyn: Are Your Agent Security Defenses Deployable in Real-World Dynamic Environments?" arXiv:2602.03117, 2026.

[215] Y. Xie et al., "QueryIPI: Query-agnostic indirect prompt injection on coding agents," arXiv:2510.23675, 2025.

[216] S. Abdelnabi and E. Bagdasarian, "AI agents may always fall for prompt injections," arXiv:2605.17634, 2026.

[217] A. Souri and F. Al-Saqqar, "A comparative analysis of cloud service providers in performance, scalability, and reliability for smart and sustainable systems," Journal of Sustainable Smart Systems in Education & Environment, vol. 1, no. 1, pp. 1-13, 2026, doi: 10.66823/jsssee.10.

[218] M. S. Al-Jbour and D. Fuqua, "The role of AI in optimizing power consumption in mobile devices: A comprehensive review," Journal of Sustainable Smart Systems in Education & Environment, vol. 1, no. 1, pp. 40-55, 2026, doi: 10.66823/jsssee.8.

[219] M. E. Farfoura et al., "A low complexity ML-based methods for malware classification," *Computers, Materials & Continua*, vol. 80, no. 3, pp. 4833-4857, 2024, doi: 10.32604/cmc.2024.054849.

[220] M. Farfoura, "HFDNet: Harmonic feature decomposition for lightweight visual malware classification," *Journal of Computer Virology and Hacking Techniques*, vol. 22, Art. no. 55, 2026, doi: 10.1007/s11416-026-00637-w.

[221] X. Wang, K. Huang, B. Liang, H. Li, and X. Du, "Shadows in the Code: Exploring the Risks and Defenses of LLM-based Multi-Agent Software Development Systems," Proceedings of the AAAI Conference on Artificial Intelligence, vol. 40, no. 44, pp. 37970-37978, 2026, doi: 10.1609/aaai.v40i44.41134.

[222] A. Doshi, Y. Hong, C. Xu, E. Kang, A. Kapravelos, and C. Kästner, "Towards Verifiably Safe Tool Use for LLM Agents," in Proc. IEEE/ACM 48th International Conference on Software Engineering (ICSE-NIER), pp. 201-205, 2026, doi: 10.1145/3786582.3786839.

[223] T. Gasmi, R. Guesmi, J. Bennaceur, and I. Belhadj, "Bridging AI and software security: A comparative vulnerability assessment of LLM agent deployment paradigms," Information Sciences, vol. 740, Art. no. 123231, 2026, doi: 10.1016/j.ins.2026.123231.