

MAPPING SECURITY RISKS AND MITIGATION STRATEGIES IN AIR ASSISTED SOFTWARE DEVELOPMENT ACROSS THE SOFTWARE DEVELOPMENT LIFECYCLE

Muhammad Afzal^{*1}, Fareed Hussain², Hadi Bux³, Allah Dino⁴, Meer Gul⁵, Ameer Hussain⁶

^{*1,2,3,4,5,6} Department of Information Technology, Sindh Agriculture University, Tandojam, Sindh, Pakistan

¹pyareafza086@gmail.com, ²fareeddars06@gmail.com, ³hadikhanhadikhan672@gmail.com
⁴salmanpalijo36@gmail.com, ⁵meergulchandio111@gmail.com, ⁶amavia035@gmail.com

DOI: <https://doi.org/10.5281/zenodo.23153041>

Keywords

AI-assisted software development; software security; large language models; software development lifecycle; vulnerability mitigation; evidence mapping.

Article History

Received: 25 June 2026

Accepted: 16 September 2026

Published: 05 October 2026

Copyright ©Author

Corresponding Author: *

Muhammad Afzal

Abstract

AI-assisted development creates a security problem that extends beyond whether generated code compiles: developers must judge the reliability of recommendations, dependencies, analyses, and repairs. Prior reviews already synthesize generated-code weaknesses, attacks and defenses for code language models, secure prompting, and threats across software and LLM lifecycles. This study therefore advances a narrower contribution: a claim-level evidence chain connecting an AI-assisted application-development activity and its software-development-lifecycle (SDLC) stage to the reported risk, vulnerability, mitigation, evaluation design, security and functional outcomes, and residual uncertainty. We specify a systematic-review protocol for peer-reviewed studies first published from 1 January 2020 through 12 September 2026. A purposively assembled set of thirteen primary studies is used only to pilot the framework; it is not the final systematically screened corpus. The pilot identifies insecure generation, poisoning, package hallucination, indirect prompt injection, unreliable detection and repair, and over-reliance on assistant output. Comparative studies support some prompting, learned-prefix, retrieval, fine-tuning, and repair strategies within their tested settings, while also showing that improvement in a security proxy may coexist with reduced functional correctness. All quantitative visuals are restricted to auditable pilot mappings or values reported in an individual source. Database exports, independent screening, completed appraisal, and full-version reconciliation remain prerequisites for presenting this work as a completed systematic literature review.

INTRODUCTION

AI assistance changes both the production of software artifacts and the way developers assess them. A model can propose an implementation, explain an analyzer warning, recommend a dependency, or generate a patch without establishing that the resulting artifact satisfies its security requirements. Empirical studies consequently examine different units of analysis: generated programs, developer–assistant workflows,

repository snippets, model behavior, and security-specific adaptation of code models [1, 2, 3, 4]. A generated-program vulnerability rate does not directly estimate the risk of a human–AI workflow, and a successful benchmark patch does not establish a safe production release.

The evidence is heterogeneous. Human-subject studies do not produce a uniform direction of effect [2, 5]; repository and benchmark studies measure different artifacts [3, 6]; and poisoning,

package hallucination, and indirect prompt injection concern distinct trust boundaries [7, 8, 9]. The literature therefore supports a differentiated synthesis rather than a single vulnerability rate for “AI-assisted development.”

Systematic reviews already cover major parts of this landscape. Negri-Ribalta et al. extract AI models, security concerns, verification procedures, vulnerabilities, exploits, and proposed mitigations for code generation [10]. Criollo Ramírez et al. review the security of LLM-generated code [11, 12]. Chen et al. organize CodeLM security around attacks, defenses, tasks, models, datasets, metrics, and tools [13]; Tony et al. combine a prompting review with controlled secure-generation experiments [14]. Lifecycle-oriented coverage is also established: Al-Hashimi et al. explicitly address generative AI and cybersecurity in the SDLC [15], while Moia et al. map threats and defensive strategies across software and LLM lifecycles [16]. Accordingly, neither a catalogue of CWEs nor a generic lifecycle taxonomy is a defensible novelty claim.

The narrower synthesis problem is evidential traceability. The accessible comparison literature does not establish a shared claim-level representation that simultaneously records the assisted activity, application-SDLC stage, risk origin and control point, reported weakness, mitigation status, comparator, security oracle, co-reported functional outcome, setting, and residual uncertainty. Because the full texts of the 2024 Criollo Ramírez conference paper and Al-Hashimi et al. were not retrieved for this draft, this gap statement is conditional, not an exhaustive absence claim.

Objective and research questions

The intended review asks:

RQ1 What security risks are reported in AI-assisted software development across SDLC stages?

RQ2 What vulnerabilities are associated with AI-assisted development, and where do they occur?

RQ3 What mitigation strategies have been proposed?

RQ4 How effective are those strategies under

the reported empirical evidence?

RQ1-RQ2 populate the stage-activity-risk-weakness portion of the chain; RQ3 identifies controls; and RQ4 assesses endpoint-specific effectiveness evidence. The contribution is the traceable representation and its pilot application, not a new model, algorithm, dataset, experiment, or industrial validation.

Scope and Analytical Framework

We use *AI-assisted software development* for activities in which an AI system produces, modifies, analyzes, or recommends software-related artifacts. Generative models, coding assistants, and AI-based security analysis are eligible when both the development activity and the security outcome are explicit. Generic phishing classification or network intrusion detection is out of scope unless it directly concerns the security of an AI-assisted development workflow [17, 18].

Three security objects must remain separate. First, the developed software can contain insecure code or incorrect controls. Second, the assistant and supporting infrastructure can be manipulated through poisoned models, untrusted context, or dependency recommendations. Third, assurance can fail when a detector, test oracle, or user judgment allows defects to pass. A model supply-chain attack and a hallucinated package name may both affect a program, but they originate at different trust boundaries.

Figure 1 uses the SDLC as an analytical organizing framework rather than a claim that development is linear. The shaded stages have direct mappings in the thirteen-study pilot; unshaded stages are search targets, not evidence of absence. Upstream model training is recorded as a risk origin, not mislabeled as requirements engineering for the application under development.

Secure-by-design principles such as least privilege and complete mediation remain useful when evaluating the authority granted to an assistant [19]. They are design arguments, however, not measured intervention effects in this pilot. Likewise, a scanner’s silence denotes no detected finding under a configuration, not proof of security.

The study’s core evidence chain is shown in Figure

2. Every link requires a source locator or an explicit reviewer-inference label. This prevents

unsupported transfer from a model-level improvement to an application-level security

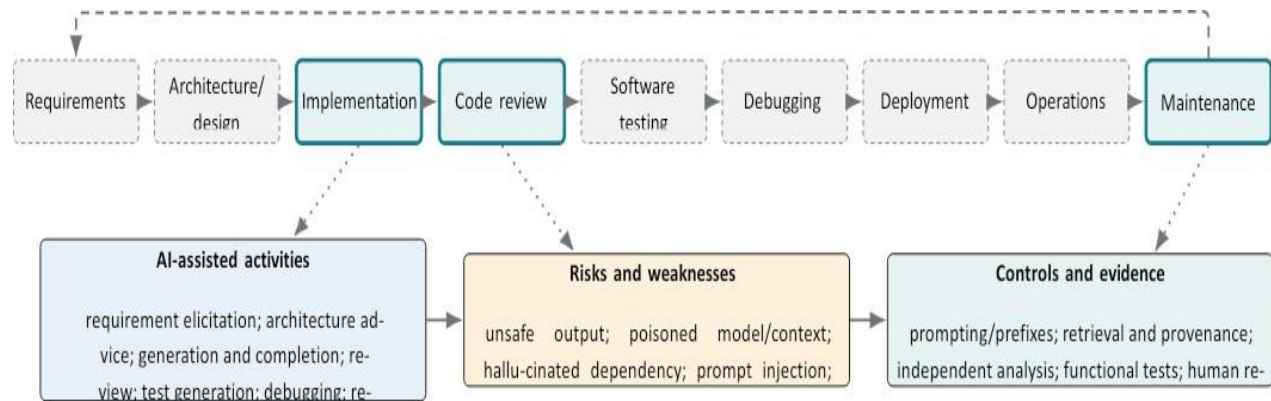


Figure 1: AI-assisted software-development security across the SDLC. The lifecycle is an analytical organizing framework; it does not assert a sequential process or empirical coverage at every stage. Source: authors' synthesis of the review scope and pilot records [10, 15, 16]. No external figure is reproduced.

Related Work and the Bounded Research Gap

Table 1 compares the closest security-focused study families. It makes the overlap explicit and records the precise remaining gap. Cells labeled "not verified" do not support an absence claim.

The defensible gap is therefore not an absence of generated-code reviews, CodeLM defense taxonomies, prompting experiments, or lifecycle mappings. It is incomplete integration at the level of an individual security claim. The completed review will record stage and activity; risk origin and control point; vulnerability or assurance failure; whether a mitigation is proposed or evaluated; comparator, dataset, model, tool, metric, and oracle; functional outcome; environment and temporal scope; and residual uncertainty. This representation is designed to expose cases in which studies use the same label for different endpoints, a mitigation is evaluated with the same tool that guided it, or upstream model defense is generalized to downstream software assurance without direct evidence.

Research Methodology

Protocol and execution status

The intended design combines systematic mapping and narrative synthesis, drawing on

established software-engineering review and mapping guidance [22, 23]. Backward and forward snowballing are planned as complements to database searches [24]. PRISMA 2020 is used as a reporting framework, not as a quality score or evidence that the review has been completed [25]. The operational window is peer-reviewed literature first published from 1 January 2020 through 12 September 2026. Earlier methodological and foundational sources are retained as background, not counted as current primary studies. First-online and issue dates must both be stored to avoid excluding eligible early-access work or admitting future publications without evidence.

Executed work consists of web-assisted scholarly discovery, targeted paper lookup, publisher-page checking, and source consultation sufficient to assemble a purposive pilot of thirteen studies. Not yet executed are complete native database searches and exports, exhaustive deduplication, independent title/abstract and full-text screening, completed appraisal, and protocol registration. Figure 3 reports this status without manufacturing PRISMA counts.

Search strategy

The conceptual search combines three blocks:

- **Technology:** “large language model” OR LLM OR “generative AI” OR “AI coding assistant” OR “neu-ral code completion” OR Copilot OR “AI-assisted programming”;
- **Activity:** “software development” OR “software engi-neering” OR “code generation” OR “code review” OR “vulnerability detection” OR “vulnerability repair” OR “requirements engineering” OR “threat modeling” OR “software testing” OR debugging OR deployment OR maintenance;
- **Security:** security OR secure OR insecure OR

vulnera-bilit y OR poisoning OR “prompt injection” OR “pack-age hallucination.”

The main query is Technology AND Activity AND Secu-rit y. Stage-specific searches will add each lifecycle term, including infrastructure as code. IEEE Xplore and ACM Digital Library are prioritized for security and software-engineering conferences, with ScienceDirect, Springer-Link, Wiley, and Taylor & Francis as journal sources. Discovery engines may locate candidates but do not replace publisher/DOI verification. Exact strings, fields, filters, timestamps, result counts, exports, and pagination limits

Table 1: Research-gap comparison with the closest study families. Source: compiled by the authors from the cited original reports and publisher records.

Previous study	Objective and method	Scope / SDLC coverage	Risk and vulnerability coverage	Mitigation coverage	Effectiveness / metrics	Main limitation for this synthesis	What this study adds / exact gap
Negri-Ribalta et al. [10]	SLR of AI code generation and software security using PICO, database searches, snowballing, quality assessment, and paired extraction.	Code suggestion is eligible beyond a narrowly named implementation phase; no application-activity lifecycle chain.	Models, concern identification, verification, vulnerabilities, exploits, and languages.	Suggested mitigations and levels of intervention.	Heterogeneous source procedures; no common claim-level comparator plus functional-outcome representation.	Proposed interface and oversight effects remain largely untested; evidence is organized around generated-code security.	Preserve its risk-weakness-mitigation coverage while adding stage/activity provenance, evaluation conditions, and residual uncertainty.
Criollo Ramírez et al. [11, 12]	2024 SLR with a linked multivocal extension.	Generated-code security; full 2024 conference text not retrieved for feature-by-feature verification.	Publisher material verifies vulnerability categories.	Publisher material verifies mitigation/best-practice and tool categories.	<i>Not verified from the 2024 full text.</i>	The accessible material cannot establish whether the complete evidence chain is present.	Requires full-text reconciliation; no novelty claim is based on an assumed omission.
Chen et al. [13]	SLR of 67 CodeLM-security papers, organized by attack and defense strategy.	CodeLM tasks and system assets rather than application-SDLC activity outcomes.	Backdoors, adversarial attacks, models, datasets, metrics, and tools.	Defense taxonomy and open challenges.	Includes attack/defense and model-performance measures.	Does not center developed-software artifacts and application-stage assurance.	Link model-level threats and measures to development activities, artifact outcomes, and assurance limits.
Tony et al. [14]	Prompting SLR plus controlled comparison using 150 LLMSEval tasks and GPT-3/3.5/4.	Prompted code generation, principally implementation.	CWE-oriented security findings and task-validity checks.	Four evaluated prompting families, including recursive criticism and improvement.	Bandit, CodeQL, manual validity assessment, and comparative weakness measures.	Bounded to prompting; model randomness, oracle differences, and production transfer remain limitations.	Compare prompting with other mitigation families while retaining the exact oracle, task validity, functionality, and setting.
Al-Hashimi et al. [15]	Publisher indexing describes a systematic review and empirical component.	Explicit generative-AI–cybersecurity–SDLC scope.	<i>Not verified from full text.</i>	<i>Not verified from full text.</i>	<i>Not verified from full text.</i>	No negative feature claim is defensible without the full report.	Establishes that combining GenAI, cybersecurity, and the SDLC is not itself novel; full-text reconciliation must test finer overlap.
Moia et al. [16]	Systematic review plus scenario, severity, STRIDE, and threat/risk analyses.	Complete software and LLM lifecycles; development and operation scenarios.	Threats, design choices, attack strategies, severity, and phase-specific mappings.	Defensive strategies mapped to lifecycle phases and attacks.	Does not determine the efficacy, implementation complexity, cost, or combined protection of defenses.	Targets LLM-based systems broadly rather than evidence about AI-assisted SE artifacts.	Narrow the unit to development-assistance claims and trace controls to empirical comparators, oracles, functional outcomes, and uncertainty.

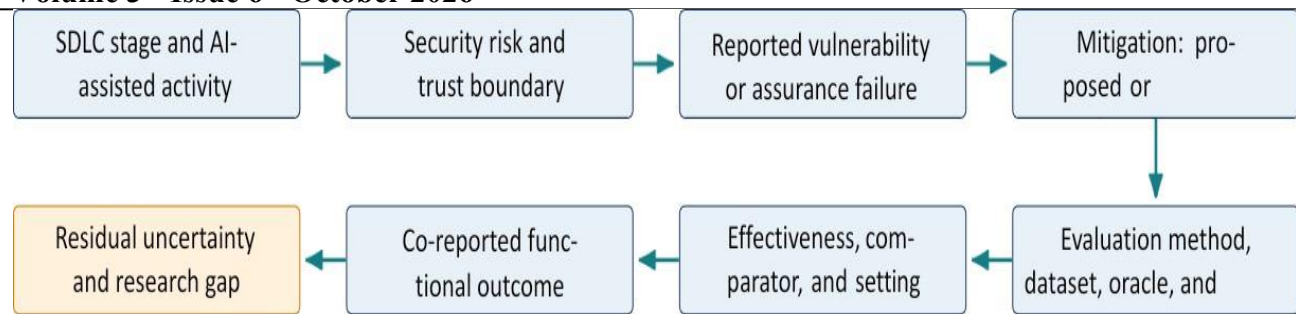


Figure 2: Conceptual evidence-mapping framework. Source: authors' claim-level synthesis of the reviewed literature [10, 13, 14, 16]. The diagram is a data-extraction schema, not a claim that every connection already has empirical support.

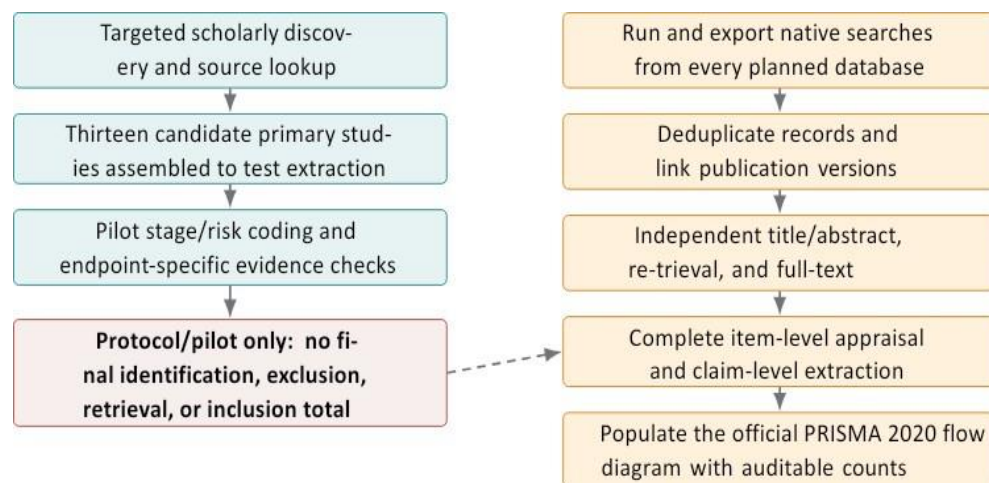


Figure 3: Review-execution status using PRISMA-aligned terminology. This is not a study-selection flow diagram and contains no screening statistics. Source: original status diagram; terminology informed by PRISMA 2020 [25]. The official PRISMA flow should be added only after auditable counts exist.

must be retained. Deduplication will use normalized DOI followed by title/author/year adjudication; publication versions will be linked as study families.

Eligibility, screening, and appraisal

Eligible reports are peer-reviewed journal articles, conference papers, or substantive workshop papers that identify an AI-assisted development activity and a software-security outcome, threat, or evaluated mitigation. Empirical risk characterization is eligible without an intervention. Conceptual controls may be retained only when their assumptions and mechanism are explicit.

Reviews support related-work comparison and discovery; they do not enter primary-study risk-frequency graphics.

Duplicates, non-development AI applications, reports with no security connection, opinion-only material, and non-peer-reviewed records are excluded from the primary corpus. Improved correctness alone is not treated as a security outcome. If full text cannot be retrieved, the record is marked "report not retrieved" rather than appraised from an abstract.

Two reviewers should independently pilot and apply the criteria, reconcile disagreements, and record one primary exclusion reason per report. Proposed appraisal items cover clear objectives; reproducible design; justified sampling; artifacts; model/tool versions; explicit threat model; validated oracle; appropriate comparator; denominators and uncertainty; threats to validity; and functional/security trade-offs. Each applicable item may be scored 1, 0.5, or 0

with a written rationale; unexamined fields remain “verification required.” No arbitrary total-score exclusion threshold is imposed.

Evidence labels describe design, not certainty: L1 conceptual proposal; L2 limited prototype; L3 controlled benchmark; L4 explicit baseline/alternative comparison; and L5 real-world or industrial evidence. Setting is recorded separately because a retrospective repository evaluation or controlled student study must not be promoted to prospective industrial effectiveness.

Extraction and synthesis

The extraction schema follows Figure 2. A study may contribute multiple claim-level records. Source-reported results are separated from reviewer classifications such as the assigned SDLC stage. A CWE is retained only when the source supplies it; analyzer warnings, manually confirmed defects, benchmark targets, and plausible downstream exploits remain distinct fields. Synthesis is narrative and tabular, stratified by activity, risk mechanism, control family, comparator, oracle, and setting. Quantitative summaries are restricted to auditable pilot membership or within-study values with compatible definitions. Meta-analysis is not attempted because tasks, model versions, sampling budgets, labels, denominators, and uncertainty reporting differ materially.

Pilot Evidence Map and Preliminary Results **Study characteristics**

Table 2 identifies the thirteen primary-study records used to exercise the extraction framework. Their purposive composition is not a census of the field. Reviews and methods papers do not enter the pilot graphics, and the review component of S12 is not counted as a separate primary study.

RQ1–RQ2: lifecycle, risks, and vulnerabilities

The pilot maps eleven records to implementation, two to code review, and three to maintenance; multiple labels are permitted. Figure 4(a) reports counts, not percentages, and uses the explicit denominator of thirteen unique pilot studies. A zero means “not identified in this purposive pilot,”

not “no research exists.” Figure 4(b) shows categorical risk-family assignments. Filled cells are source-supported mappings, not equal-severity events or independent vulnerability counts.

Reported artifact weaknesses include injection, memory-safety defects, weak cryptographic choices, and insecure handling of data or resources [1, 3, 5, 7]. These remain conventional software weaknesses that can occur in AI-produced artifacts. Model poisoning and prompt injection are mechanisms that can induce or steer outcomes; package hallucination is a dependency-provenance risk; over-reliance is a human-process risk. Table 3 preserves those distinctions.

RQ3–RQ4: mitigation and endpoint-specific effectiveness

The consulted experiments evaluate security-aware prompting, learned prefixes, analyzer-informed repair, zero-shot patching, package-grounded retrieval, self-refinement, fine-tuning, and combinations of controls [4, 3, 20, 8, 14].

They support particular endpoints under particular oracles; they do not establish a universal ranking. Table 4 distinguishes what was measured from what remains unproved.

The strongest quantitative synthesis available here is deliberately narrow. Figure 5 redraws values reported in Tables 4–5 of Spracklen et al. [8]; it does not reproduce the source’s graphical design. The panels use different datasets and denominators. Panel (a) reports the percentage of package recommendations classified as hallucinated in each source-defined run; the exact run-level counts are not printed in the source table, so no counts or confidence intervals are reconstructed. Panel (b) reports pass@1 over the 164 HumanEval problems.

For DeepSeek Coder, fine-tuning lowers the reported hallucination rate from 16.14% to 2.66%, while HumanEval pass@1 falls from 51.4% to 25.3%. CodeLlama shows the same direction with a smaller functionality change. These values do not reveal how often an individual artifact is both secure and functional because the endpoints are measured on separate tasks. S09 directly motivates joint security/functionality

assessment on the same artifacts [6].

Human evidence is likewise context dependent. S02 re-ports less secure output on several tasks and greater con-fidence, whereas S03 finds a small and metric-dependent security effect for its C task [2, 5]. These studies should not be collapsed into a universal claim that AI either makes pro-grammers insecure or has no effect. Negative defense evi-dence also remains visible: S07's tested poisoning defenses do not reliably eliminate the attack, and S13 demonstrates an attack surface without a broadly validated defense effect [7, 9].

Discussion

The appropriate unit of synthesis is a security claim under stated conditions, not an assistant brand. A paper evaluates particular model versions, tasks,

Table 2: Characteristics of the pilot primary studies. Source: compiled by the authors from S01–S13.

These are candidate evidence records, not final systematic-review inclusions.

ID	Authors: year; venue	AI model/tool	Activity and reviewer-assigned stage	Security task	Dataset / method	Security outcome and oracle	DOI / record
S01	Pearce et al.; 2022; IEEE SP [1]	GitHub Copilot / Codex	Code completion; implementation	Generate code for CWE-oriented scenarios	Security-targeted generation experiment	Vulnerable outputs; CodeQL and scenario-specific checks	doi:10.1109/SP46214.2022.9833571
S02	Perry et al.; 2023; CCS [2]	Codex-based assistant	Assisted programming; implementation	Security and user confidence	Controlled developer tasks in Python, C, and SQL	Security/correctness labels and confidence measures	doi:10.1145/3576915.3623157
S03	Sandoval et al.; 2023; USENIX Security [5]	LLM code assistant	Assisted C programming; implementation	Memory/pointer weaknesses	Student-programmer linked-list task	CWE-oriented manual/task checks and correctness	No DOI; USENIX proceedings
S04	He and Vechev; 2023; CCS [4]	CodeGen with SVEN	Security-controlled generation; implementation	Harden or adversarially steer generation	CWE scenarios; HumanEval; controlled comparisons and ablations	CodeQL-defined security rate; pass@k	doi:10.1145/3576915.3623175
S05	Fu et al.; 2025; TOSEM [3]	Copilot, Code Whisperer, Codeium; Copilot Chat	Generated snippets and repair; implementation/maintenance	Detect and repair weaknesses	733 attributed Python/JavaScript repository snippets; prompt comparison	Affected snippets, CWE counts, issue-fix rate; scanners plus adjudication	doi:10.1145/3716848
S06	Pearce et al.; 2023; IEEE SP [20]	Multiple code LMs	Vulnerability repair; maintenance	Zero-shot security patching	Synthetic, hand-crafted, and historical cases	Valid, functional, and safe repair counts	doi:10.1109/SP46215.2023.10179324
S07	Schuster et al.; 2021; USENIX Security [7]	Neural code completion models	Poisoned completion; implementation	Targeted and untargeted poisoning	Poisoned and original code completion corpora	Bait confidence and top-1/top-5 completion accuracy	No DOI; USENIX proceedings
S08	Spracklen et al.; 2025; USENIX Security [8]	16 LLMs: DeepSeek Code 6.7B and CodeLlama 7B mitigation subset	Package recommendation; implementation	Package hallucination and mitigation	576,000 code samples overall; two-model RAG/self-refine/fine-tune experiment	Registry-defined hallucination rate; HumanEval pass@1	No conference DOI; USENIX record
S09	Dai et al.; 2026; ICSE [6]	Multiple secure-generation methods	Secure-generation evaluation; implementation	Joint security and functionality	BigCodeBench and SecCodePLT+	Secure-Pass@k, SAFE@k, and unit tests	doi:10.1145/3744916.3773217
S10	Belozorov et al.; 2026; EMSE [26]	GPT-4.1, GPT-5, Claude Opus 4.1	Detection and repair; implementation/code review/maintenance	Find and repair developer-code vulnerabilities	DevGPT C/C++/C# snippets; retrospective comparison	Manual review plus Flawfinder, Sengrep, Snyk, and Cppcheck	doi:10.1007/97810664-026-10812-8
S11	Ding et al.; 2025; ICSE [27]	Code-LM classifiers	Vulnerability detection; code review	Evaluate realistic detection	PrimeVul and comparison benchmarks	F1, paired-function, and temporal evaluation outcomes	doi:10.1109/ICSE55347.2025.00038
S12	Tony et al.; 2025; TOSEM [14]	GPT-3, GPT-3.5, GPT-4	Prompted secure generation; implementation	Reduce generated-code weaknesses	150 LLMSecEval tasks; prompting comparison	Bandit, CodeQL for C, and manual validity inspection	doi:10.1145/3722108
S13	Greshake et al.; 2023; AISec [9]	LLM-integrated applications, including a Copilot case	Injected completion context; implementation	Indirect prompt injection	Controlled attack demonstrations	Qualitative feasibility; no pooled rate extracted	doi:10.1145/3605764.3623985

prompts, artifacts, users, and oracles. The evidence map asks what was protected, against which failure, at what control point, and with what observation. That structure prevents a model-level metric from being silently reinterpreted as system-level assurance. Uneven pilot coverage exposes untested transitions. Insecure completion can motivate deployment controls without demonstrating their effectiveness. Patch generation does not establish safe prioritization, integration, release, or monitoring. Retrieval can reduce unsupported package recommendations while adding a new context source that must itself be trusted or validated. The pilot supports these as hypotheses and design concerns, not measured end-to-end effects.

Table 3: SDLC-stage and security-risk mapping. Source: authors' coding of S01-S13; CWE examples are retained only where supplied by a source.

SDLC stage	AI-assisted activity	Reported risk	Vulnerability / CWE where available	Supporting studies	Evidence type	Research status
Requirements	Security-requirement elicitation, misuse cases, or prompt requirements	No direct pilot observation	Not assigned	—	Targeted search required	<i>Not identified in the pilot</i>
Architecture/design	Threat modeling or architecture recommendation	No direct pilot observation	Not assigned	—	Targeted search required	<i>Not identified in the pilot</i>
Implementation	Generation, completion, assisted programming, and dependency recommendation	Unsafe output; poisoned model/context; package hallucination; indirect prompt injection; over-reliance	CWE-78/89 injection (S01); CWE-119/416/476/787 memory/pointer examples (S03); source-specific crypto misuse without inferred CWE (S07)	S01-S05, S07-S10, S12-S13	Controlled generation, human study, repository mining, benchmark, and attack demonstration	Pilot-supported; heterogeneous
Code review	AI-assisted source vulnerability detection	Missed defects, false assurance, misleading labels	Source labels and manually confirmed defects; no new crosswalk	S10-S11	Retrospective comparison and benchmark evaluation	Pilot-supported
Software testing	AI-generated security tests as the assisted activity	No direct pilot observation	Not assigned	—	Researcher-run tests are not coded as AI-assisted testing	<i>Not identified in the pilot</i>
Debugging	Distinct AI security-debugging workflow	No direct pilot observation	Not assigned	—	Patch generation coded as maintenance	<i>Not identified in the pilot</i>
Deployment	AI-generated configuration/IaC or release decision	Plausible dependency/configuration exposure, not directly observed	Not assigned	—	Targeted deployment-artifact evaluation required	<i>Not identified in the pilot</i>
Operations	AI-assisted runtime security monitoring of developed software	No direct pilot observation	Not assigned	—	Generic cybersecurity monitoring is outside scope	<i>Not identified in the pilot</i>
Maintenance	Corrective vulnerability repair	Residual weakness or functional regression after patching	Source-confirmed residual defects; no assumption that every failed patch creates a new CWE	S05, S06, S10	Repository and controlled repair comparisons	Pilot-supported

Table 4: Mitigation strategies and evidence. Source: compiled by the authors from the cited pilot studies. L1/L4 describe study design, not certainty or quality.

Mitigation	Problem addressed	Stage/activity	Study / setting	Evaluation / oracle	Metric and unit	Reported result	Limitations	Level
Security-aware prompts; recursive criticism and improvement	Generated code weaknesses	Implementation / generation	S12; LLMSecEval	Bandit, CodeQL for C, and manual task-validity review	Weakness count/rate and density under source denominators	Lower measured weaknesses in tested settings; effects depend on model and technique	No uniform significance, functional equivalence, or production benefit	L4
Learned security prefixes (SVEN)	Unsafe generation	Implementation / controlled generation	S04; CWE scenarios and HumanEval	CodeQL-oriented security assessment; separate functional benchmark	Security rate (% outputs); pass@k	CodeGen 2.7B security rate reported as 59.1% baseline and 92.3% with hardening	Security and functionality are not a single joint artifact-level endpoint	L4
Analyzer-informed repair prompts	Repository-snippet weaknesses	Implementation / maintenance	S05; Python/JS snippets	Static analyzers plus adjudication	Issues fixed / issues before repair	Source reports 19.3%, 31.8%, and 55.5% for /fix, basic, and enhanced prompts	Displayed enhanced counts 163/295 imply 55.25%; final artifact must reconcile the discrepancy	L4
Zero-shot patch generation	Known vulnerable code	Maintenance / repair	S06; synthetic, hand-crafted, historical cases	Compilation plus functional and security checks	Valid, functional, safe repair counts	Demonstrates feasible repairs and limitations on historical cases	Does not establish reliable unsupervised project-scale remediation	L4
RAG and self-refinement	Hallucinated packages	Implementation / dependency recommendation	S08; two-model mitigation experiment	Package-registry comparison under source pipeline	Hallucinated recommendations / all recommendations in each run (%)	Both methods reduce the source-reported rate; magnitude is model-dependent	Exact per-run counts are not reported in the article's Table 4; registry presence is not package safety	L4
Security-specific fine-tuning / ensemble	Hallucinated packages	Implementation / model adaptation	S08; DeepSeek Coder 6.7B, CodeLlama 7B	Registry comparison; separate HumanEval	Hallucination rate (%); HumanEval pass@1 over 164 tasks	Lowest reported hallucination rates, with separate functionality decreases after fine-tuning	No joint secure-and-functional artifact rate; ensemble functionality not reported	L4
LLM review/repair with rechecking	Developer-code defects	Code review / maintenance	S10; retrospective DevGPT corpus	Manual review plus four scanners	Detected or repaired defects / 56 confirmed defects	Detection/repair: GPT-4.1 46/42; GPT-5.4 44/44; Claude Opus 4.1 45/43	Retrospective evidence is not a prospective industrial trial	L4
Poisoning defenses; prompt-injection directions	Manipulated model or untrusted context	Implementation / trust boundary	S07/S13; controlled attacks	Attack-specific completion measures or qualitative demonstrations	Bait confidence / top-k; feasibility	S07 reports limited/negative defense evidence; S13 proposes directions without a general effect estimate	No robust general defense is established in the pilot	L4/L1

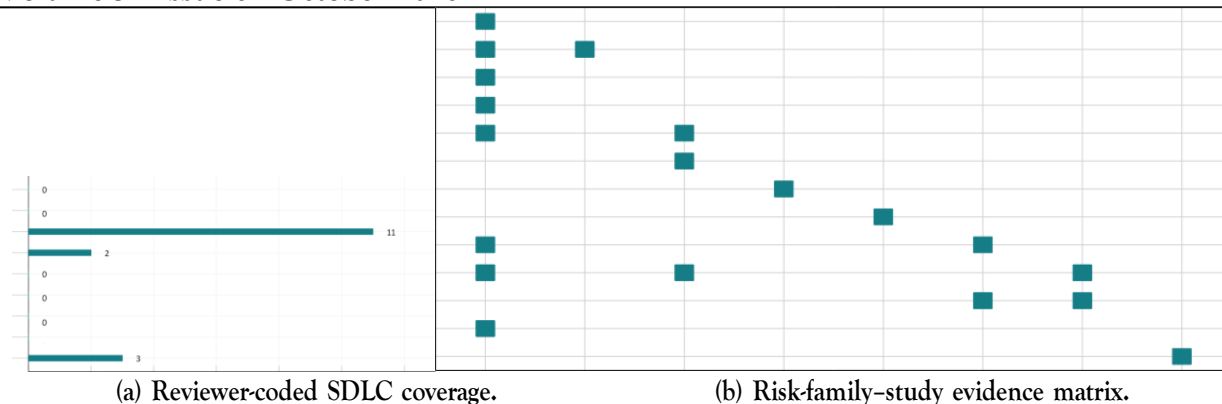


Figure 4: Pilot evidence map. Source: authors' analysis of the thirteen records in Table 2. Panel (a) permits multiple stage assignments, so counts sum to more than 13. Panel (b) is categorical and does not estimate prevalence or severity.

Controls are also not automatically additive. Static analysis can be both an intervention component and an evaluation instrument. If the same analyzer guides repair and supplies the success criterion, improvement may partly reflect the selected oracle. Stronger evidence combines independent security checks with functional testing, transparent configurations, fair comparator budgets, and an environment that matches the intended claim [6].

Implications

For researchers. Record attack mechanism, artifact weakness, assurance failure, and human acceptance as distinct constructs. Report both the intervention mechanism and the strength and setting of its evidence. Preserve denominators and co-report functionality whenever a security claim concerns usable software.

For practitioners. Treat assistant output as an artifact requiring verification, not as an assurance decision. Preserve provenance, explicit security requirements, and review responsibility. If a scanner guides generation or repair, retain independent checks and functional tests. These are cautious implications of the pilot and foundational protection principles, not a quantified guarantee [2, 6, 19].

For industry and security teams. Adoption evidence should match the intended languages, projects, threat model, and workflow. Repository

benchmarks, student studies, retrospective developer interactions, and prospective industrial trials answer different questions. Neither high completion accuracy nor a low warning count alone establishes release readiness.

Threats to Validity and Limitations

Selection and search validity. The pilot is purposive and web-discovered. It lacks complete native database exports and a verified screening population; publication, ranking, coverage, and language bias cannot be quantified. The LLM-heavy queries may underrepresent non-generative AI/ML security tools. No claim of saturation or completeness is warranted.

Extraction and construct validity. Source reports and identifiable sections support the presented claims, but full claim-level extraction, final-version reconciliation, and independent coding remain incomplete. SDLC assignments are reviewer interpretations. Alternative assignments could change Figure 4(a), especially for source inspection and corrective patching.

Evidence and internal validity. The appraisal worksheet is not scored. Evidence labels are provisional design classifications, not validated quality grades. Mitigation percentages use heterogeneous denominators and oracles; no raw experimental reproduction or statistical pooling was performed. The S05 rate/count discrepancy is retained as an audit issue rather than silently resolved.

External and temporal validity. Historical models and bounded tasks limit transfer to current assistants and production workflows. Inclusion of

2026 publications does not make the reported experiments measurements of products as of 17 September 2026. Public repositories and developer interactions are not representative industrial samples, and repeated experiments do not automatically constitute longitudinal

evidence.

Bibliographic validity. DOI and venue details are tied to publisher or proceedings records where available. Retraction/Crossmark checks were not completed systematically.

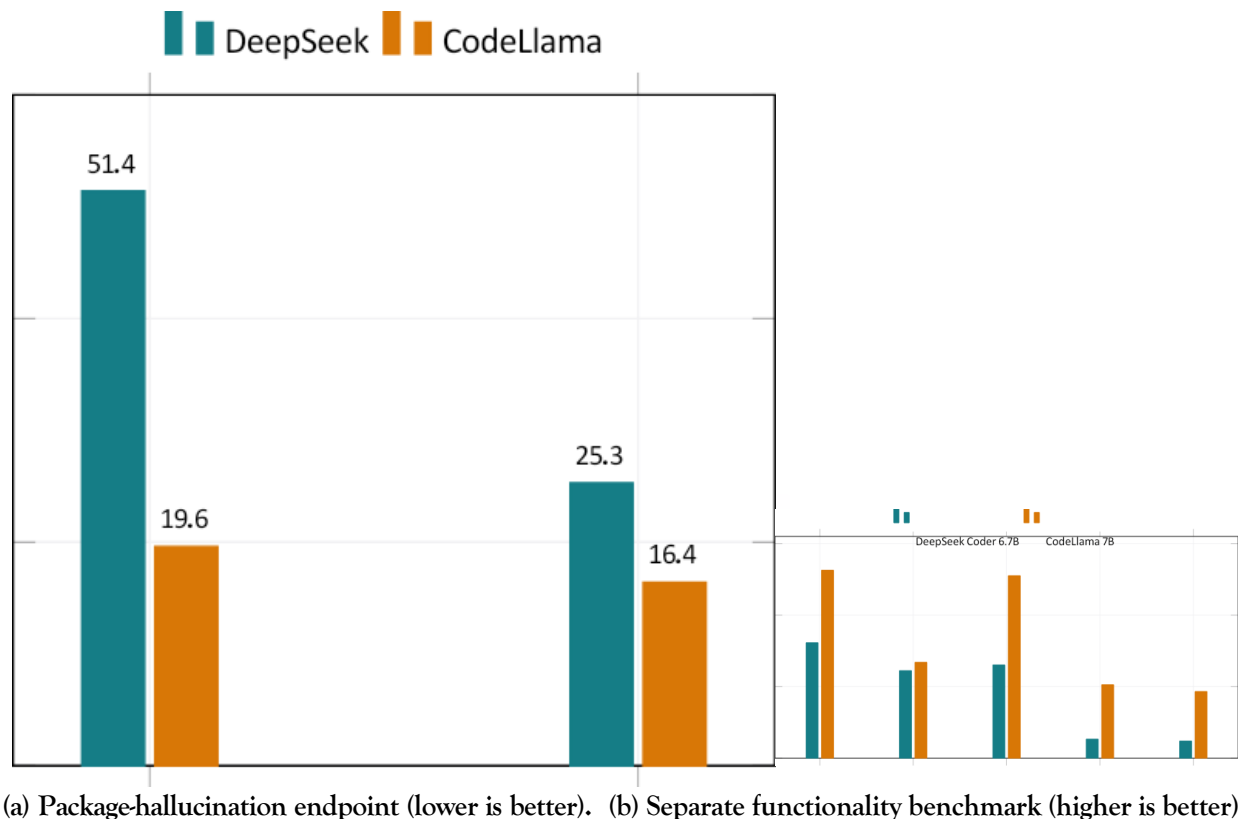


Figure 5: Package-hallucination mitigation and separate functionality outcomes in S08. Source: authors' visualization of exact values reported by Spracklen et al. [8], Tables 4–5. Panel (a) uses source-defined package-recommendation runs; exact counts are not printed in the table. Panel (b) uses 164 HumanEval tasks. The panels are not combined, and no confidence interval, pooled effect, or unreported ensemble functionality result is inferred.

Author identity, affiliation, funding, conflict-of-interest, contribution, data-license, and venue-specific disclosure statements must be finalized by the human research team.

Conclusion

The pilot identifies distinct security problems in

AI-assisted development: unsafe output, poisoned or injected assistance, package hallucination, unreliable detection and repair, and overconfident acceptance. It also identifies comparative mitigation evidence, but that evidence is conditional on the task, model, oracle, denominator, and functional endpoint. Conflicting human-study findings and security-utility trade-offs make a universal effectiveness claim inappropriate.

The proposed contribution is an auditable claim-level connection among the assisted activity, SDLC stage, risk origin/control point, reported weakness, mitigation, evaluation design, security and functional outcomes, and residual uncertainty. The accessible comparison supports

this narrower framing while demonstrating substantial overlap

with prior generated-code, CodeLM, prompting, and life-cycle reviews. Complete searches, independent screening, final-version reconciliation, appraisal, and corpus-wide extraction remain necessary. At present, this manuscript is a documented protocol and evidence-led pilot, not a completed systematic review.

Author Contributions

Muhammad Afzal: Proposed responsibility for synthesizing the security risks and software vulnerabilities associated with AI-generated and AI-assisted code, including insecure outputs, injection weaknesses, memory-safety defects, cryptographic misuse, and insecure handling of data or resources. This allocation also covers interpretation of the implementation-stage evidence while preserving the distinctions among source-reported vulnerabilities, analyzer findings, and assurance limitations.

Fareed Hussain: Proposed responsibility for mapping AI-assisted development activities and security risks across the software development lifecycle, from requirements and architecture through implementation, review, testing, deployment, operations, and maintenance. This allocation includes distinguishing pilot-supported stages from targeted research gaps and maintaining the stage-activity-risk relationships used in the evidence map.

Hadi Bux: Proposed responsibility for the study conceptualization, research design, formulation of the research questions, systematic-mapping approach, and development of the claim-level security-risk framework. This allocation includes structuring the evidence chain that connects the assisted activity and SDLC stage to the risk origin, reported weakness, mitigation, evaluation design, outcomes, and residual uncertainty.

Allah Dino: Proposed responsibility for investigating AI-assisted vulnerability detection, code review, missed defects, misleading labels, and false-assurance risks. This allocation also covers the analysis of detection and repair evidence under scanner-based, manually confirmed, benchmark,

and rechecking oracles, without treating the absence of a detected finding as proof of security.

Meer Gul: Proposed responsibility for analyzing mitigation strategies and security controls, including secure prompting, learned security prefixes, analyzer-informed repair, zero-shot patching, retrieval-augmented approaches, self-refinement, security-oriented fine-tuning, and combined controls. This allocation includes comparison of mitigation effectiveness under the reported tasks, models, denominators, security oracles, and separately measured functional outcomes.

Ameer Hussain: Proposed responsibility for analyzing AI-related threat mechanisms and trust-boundary risks, including model poisoning, indirect prompt injection, package hallucination, over-reliance on assistant output, and manipulated or untrusted context. This allocation also covers residual security risk, uncertainty, transfer limitations, and the conditions that prevent model-level or benchmark improvements from being interpreted as end-to-end software assurance.

Because the original manuscript does not explicitly document individual author contributions, these assignments represent a proposed thematic allocation based on the research content and should be confirmed by all authors before publication.

Data and Research Integrity Statement

No external figure is copied and no decorative or AI-generated image is used. Figures 1–3 are original conceptual/status diagrams with source attribution. Figure 4 is reproducible from the study and stage/risk mappings in Tables 2–3. Figure 5 contains only values reported in S08. No PRISMA total, field-wide percentage, publication trend, vulnerability frequency, confidence interval, statistical significance, or effectiveness ranking is invented.

REFERENCES

- 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. doi:10.1109/SP46214.2022.9833571.
- N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do Users Write More Insecure Code with AI Assistants?" in Proc. ACM SIGSAC Conference on Computer and Communications Security, 2023. doi:10.1145/3576915.3623157.
- Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen, "Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study," ACM Transactions on Software Engineering and Methodology, 2025. doi:10.1145/3716848.
- J. He and M. Vechev, "Large Language Models for Code: Security Hardening and Adversarial Testing," in Proc. ACM SIGSAC Conference on Computer and Communications Security, 2023. doi:10.1145/3576915.3623175.
- 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 32nd USENIX Security Symposium, 2023. <https://www.usenix.org/conference/usenixsecurity23/presentation/sandoval>.
- S.-C. Dai, J. Xu, and G. Tao, "Rethinking the Evaluation of Secure Code Generation," in 48th IEEE/ACM International Conference on Software Engineering, 2026. doi:10.1145/3744916.3773217.
- R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion," in 30th USENIX Security Symposium, 2021. <https://www.usenix.org/conference/usenixsecurity21/presentation/schuster>.
- 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," in 34th USENIX Security Symposium, pp. 3687–3706, 2025. <https://www.usenix.org/conference/usenixsecurity25/presentation/spracklen>.
- K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection," in Proc. 16th ACM Workshop on Artificial Intelligence and Security, 2023. doi:10.1145/3605764.3623985.
- 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, 2024. doi:10.3389/fdata.2024.1386720.
- L. Criollo Ramírez, X. Limón, A. J. Sánchez-García, and J. C. Pérez-Arriaga, "State of the Art of the Security of Code Generated by LLMs: A Systematic Literature Review," in 2024 12th International Conference in Software Engineering Research and Innovation, 2024. doi:10.1109/CONISOFT63288.2024.00050.
- L. Criollo Ramírez, X. Limón, A. J. Sánchez-García, and J. C. Pérez-Arriaga, "State of the Art of the Security of Code Generated by LLMs: A Multivocal Literature Review," Programming and Computer Software, vol. 51, pp. 587–604, 2025. doi:10.1134/S0361768825700446.
- Y. Chen, W. Sun, C. Fang, Z. Chen, Y. Ge, T. Han, Q. Zhang, Y. Liu, Z. Chen, and B. Xu, "Security of Language Models for Code: A Systematic Literature Review," ACM Transactions on Software Engineering and Methodology, vol. 35, no. 7, pp. 1–65, 2026. doi:10.1145/3735554.
- C. Tony, N. E. Díaz Ferreyra, M. Mutas, S. Dhif, and

- R. Scandariato, "Prompting Techniques for Secure Code Generation: A Systematic Investigation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 8, pp. 1–53, 2025. doi:10.1145/3722108.
- H. A. Al-Hashimi, R. A. Khan, H. S. Alwageed, A. M. Al-garni, S. Ayouni, and A. O. Almagrabi, "Exploring the Role of Generative AI in Enhancing Cybersecurity in Software Development Life Cycle," *Array*, vol. 28, art. 100509, 2025. doi:10.1016/j.array.2025.100509.
- V. H. G. Moia, I. J. Sanz, G. A. F. Rebello, R. D. de Meneses,
- B. Hitaj, and U. Lindqvist, "LLM in the Middle: A Systematic Review of Threats and Mitigations to Real-World LLM-Based Systems," *Computer Science Review*, vol. 61, art. 100916, 2026. doi:10.1016/j.cosrev.2026.100916.
- 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*, 2024. doi:10.1016/j.hcc.2024.100211.
- J. Zhang, H. Bu, H. Wen, Y. Liu, H. Fei, R. Xi, L. Li, Y. Yang,
- H. Zhu, and D. Meng, "When LLMs Meet Cybersecurity: A Systematic Literature Review," *Cybersecurity*, 2025. doi:10.1186/s42400-025-00361-w.
- J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, 1975. doi:10.1109/PROC.1975.9939.
- 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. doi:10.1109/SP46215.2023.10179324.
- X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo,
- D. Lo, J. Grundy, and H. Wang, "Large Language Models for Software Engineering: A Systematic Literature Review," *ACM Transactions on Software Engineering and Methodology*, 2024. doi:10.1145/3695988.
- B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner,
- J. Bailey, and S. Linkman, "Systematic Literature Reviews in Software Engineering—A Systematic Literature Review," *Information and Software Technology*, vol. 51, no. 1, pp. 7–15, 2009. doi:10.1016/j.infsof.2008.09.009.
- 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. doi:10.1016/j.infsof.2015.03.007.
- C. Wohlin, "Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering," in *Proc. 18th International Conference on Evaluation and Assessment in Software Engineering*, 2014. doi:10.1145/2601248.2601268.
- M. J. Page, J. E. McKenzie, P. M. Bossuyt, et al., "The PRISMA 2020 Statement: An Updated Guideline for Reporting Systematic Reviews," *BMJ*, vol. 372, n71, 2021. doi:10.1136/bmj.n71.
- V. Belozarov, P. J. Barclay, and A. Sami, "Secure Coding with AI—From Detection to Repair," *Empirical Software Engineering*, vol. 31, art. 93, 2026. doi:10.1007/s10664-026-10812-8.
- Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alo-mair, D. Wagner, B. Ray, and Y. Chen, "Vulnerability Detection with Code Language Models: How Far Are We?" in *47th IEEE/ACM International Conference on Software Engineering*, 2025. doi:10.1109/ICSE55347.2025.00038.