Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

AI-Assisted Code Generation in Full-Stack JavaScript Development: A Structured Survey of Productivity, Correctness, and Security Evidence

Remona F. Denny
Department of Computer Engineering, Gyanmanjari Innovative University, Bhavnagar

Abstract **- Coding assistance and autonomous code generation agents now feature auto-completion, auto-generation, and auto-execution of code. JavaScript is mainly difficult because of its large browser and server runtimes, full-stack web setups, and large package ecosystems, and the evidence on JavaScript code is scattered across 52 papers (20212026) in the areas of AI programming productivity, functional correctness, and security. We define a search protocol, identify experiments' evaluation units and quality attributes, compare evaluation oracles, and build a taxonomy of the security risks reported. A search-sensitivity analysis indicates that 37 out of 49 primary studies (75.5%) were identified using the search strings, while 11 (22.4%) were found through snowballing. Three main findings are presented. First, productivity outcomes vary by context: field experiments indicate improvements, while a randomized trial involving experienced maintainers on mature repositories reports a 19% slowdown. Second, recurring security issues are identified across designs, like cross-site scripting, code injection, weak access control, hard-coded secrets, and hallucinated npm packages. Third, exploit-validated security evaluation exists for backend and repository-level tasks, but full-stack web benchmarks focus only on functionality, and application-level security evidence for JavaScript is limited to a few small studies, each examining no more than a dozen applications. The paper ends with a research agenda emphasizing exploit-validated, layer-attributed benchmarks for full-stack JavaScript applications.**

I. INTRODUCTION

A few years ago, an editor that could complete an entire function would have been a proof-of-concept for a research project. That is now the standard, and a growing share of programmers copy and paste entire features into a chat window or hand them to an autonomous coding agent. JavaScript is at the heart of this transformation: it runs in the browser, it runs on the server with Node.js, and its package ecosystem is large enough that the user interface, the API layer, and the database access code can all be expressed in one language.

But whether that is wise or foolish relies on evidence, which is patchy, inconsistently compared and incomparable. Some judge developer effort, some base their results on the number of failed tests, some analyse the generated code for vulnerabilities, some deploy the generated code and then carry out live operations on the service. They do not (often) all pose the same question or measure the same outcome as 'good' code.

This survey is putting that reconciliation on the table. 3 research questions are asked:

RQ1. What does the empirical literature say about the productivity and functional correctness of AI-generated code, and how much does this literature cover web development and JavaScript?

RQ2. What security issues are described in AI-generated code and what were they assessed?

RQ3. For each evaluation unit (snippet, repository, backend, full-stack) and language, what exploit-validated evaluation exists, and what is missing for full-stack JavaScript?

Contributions are: (i) search process described with counts from first screening and a sensitivity analysis of search-string design relative to a set of 52 recent papers (2021-2026) of exploit- driven benchmarks and API-level security literature; (ii) two-dimensional classification (unit of evaluation x quality attribute) and comparison of evaluation oracles; (iii) taxonomy of security risks for AI-generated web code based on exploits reported in primary studies; (iv) gap analysis of coverage gaps; and (v) research agenda.

The remainder of the paper is structured below. In Section II we present the background and our related surveys. In Section III, we present the survey way. In Section IV, we present the classification of studies by evaluation units and quality attributes, and the comparison of evaluation oracles. RQ1 and RQ2 are discussed in Sections V and VI, respectively. RQ3 and future research are discussed in Section VII. Implications and threats to validity are discussed in Section VIII. Conclusions are given in Section IX.

II. BACKGROUND AND RELATED SURVEYS

A. Full-Stack JavaScript

Within a single full-stack JavaScript app like for example, a React UI, a Node.js API server, and a database accessed directly or via an ORM, correctness spreads between layers (UI calls the API, the API enforces validity, the API matches the DB schema,) and security does too (XSS in the UI, injection at DB boundary, access control broken in the API, secrets in config, supply chain in npm dependency graph).

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

B. Evaluating Generated Code

Purposes of assessment Functional correctness has been assessed mainly by test execution, and pass@k [1], [2]. Small size of function-level test suites led to larger test suites [3], and multi-language versions [4]. Repository-level issue fixing [5] set the unit of evaluation in a later era.

C. Related Surveys

General surveys of LLMs for software engineering [6] and code generation [7] mention security only in passing. Security-oriented surveys [8][10] classify studies by vulnerability, mitigation, or task (detection, repair, or poisoning), and all also seem to be snippet-focused. To our knowledge, they do not classify reviews of the evidence by evaluation unit, describe full-stack applications, or aggregate the application-level benchmarks and case studies from 20252026 as we do here. They also do not appear to conduct sensitivity analyses of their searches to the included set (see Section III-E here).

III. SURVEY METHOD

The survey is conducted after the planning, execution, and reporting of the guidelines for systematic reviews [11], [12], and snowballing [13]. It is reported following the standards (PRISMA 2020 [14] and the reporting extension, PRISMA-S [15]) for this type of comprehensive structured survey with a documented protocol rather than a full systematic review, for reasons given in Section VIII-B. The open index searching was scripted, and the rules for de-duplication and prescreening were fixed before the search. Fig. 1 and Table A-I (Appendix A) show the amounts available; total hit counts for Scopus, IEEE Xplore and the ACM Digital Library are not available in this record, and are marked "hit count not available".

A. Sources and Search Strings

The searches were performed on the databases Scopus, IEEE Xplore, ACM Digital Library, arXiv, and OpenAlex in 2026. Table I shows the abstracted concept strings. Specific translations of queries to the interface, the date of the search for each database, and database exports are not reported in this paper, but for reproducibility, the number of records retrieved from each database during the open-index searches is given in Table A-II.

Two practical constraints limit what the search can claim, and these are reported rather than omitted. First, Web of Science was not examined (no subscription was available) and was so not searched. Second, non-interactive access to Scopus, IEEE Xplore, and the ACM Digital Library was refused; searches of these three databases were conducted through institutional interactive sessions, and their hit counts are not available in this record. Crossref was trialled as a counting source but rejected: its total-results value for a bibliographic search is a relevance-ranked hit count rather than a Boolean total hit count, so it cannot be used as a PRISMA identification number.

Since OpenAlex has a maximum of five Boolean operators, the string of each concept block was split down to bits so that the set union of the atomic queries forms each concept block (S1a-S1c, S2a-S2e, S3a-S3b, S4a). The effect of this split on recall (discussed in more detail in Section III-E) is that the two runs of published strings returned different numbers of studies, compared to this split change. Backward and forward snowballing [13] was run from the two secondary study sets and application-level benchmarks; the difference in the number of studies included when snowballing was used instead of a database string is presented in Section III-D.

TABLE I
SEARCH STRINGS

ID	String (abridged)
S1	("LLM" OR "large language model" OR Copilot OR "AI assistant" OR "code generation") AND (security OR vulnerab* OR CWE)
S2	S1 concept group AND (JavaScript OR TypeScript OR "Node.js" OR npm OR "web application" OR "full-stack")
S3	(benchmark OR evaluation) AND (backend OR frontend OR "full-stack" OR "website generation") AND LLM
S4	(productivity OR developer) AND (Copilot OR "coding assistant") AND ("controlled experiment" OR "field experiment" OR randomized)
S5	("code generation" OR "LLM-generated code" OR "generated code") AND (vulnerability OR security OR insecure OR CWE OR "supply chain")
S6	(benchmark OR "evaluation suite" OR "repository-level" OR "issue resolution" OR HumanEval OR MBPP OR "pass@k" OR SWE-bench) AND ("code generation" OR code model OR LLM OR agent)

S5 and S6 were added after S1S4 searches had been performed, to the sensitivity check presented in Section III-E. The methods of S5 and S6 retrieval were not tested against the original search records.

B. Inclusion and Exclusion Criteria

We included studies that (I1) assessed AI- or LLM-based code generation (including completion, chat, and agentic approaches), (I2) presented empirical findings on productivity, correctness, or security, or were secondary analyses of such findings, (I3) were published between 2021 and October 2026, and (I4) were written in English. Evidence from studies not focused on JavaScript was retained as context and is presented in Table II. We excluded unpublished manuscripts lacking empirical evidence; duplicates (counting a journal article and its preprint as one); studies on vulnerability detection or repair that did not evaluate generated code; and vendor marketing materials. One industrial report [16] was included as grey literature due to its influence and scale.

E1E7date (prescreen language), AI / code-generation, concept domain outcome empirical / review content and genre exclusion prescreen codes were used. Usage of the E4 domain rule is not clear; non-JavaScript studies appear to conflict with the rule. As a prescreening log was not provided in this record, it is not clear whether the prescreening rule and any exceptions were followed, and if so, retrospectively.

C. Selection and Extraction

Exports were de-duplicated by DOI title normalized title, and version; title/abstract-screened; accessible full-text-screened. Prescreen required AI code-generation domain outcome, empirical/review terms; its domain condition is incompatible with the addition of the mixed-period context studies above (Section III-B). These are required to audit the process; exports from the varying engines, exact interface-specific queries, date-stamped search logs, prescreen code,

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

exclusion decision, and full-texts. Extraction logged evaluation unit, attribute, language, model, oracle, results, and publication.

Selection, extraction, and classification were performed by a single reviewer. No independent screening or classification of a sample was carried out, so no inter-rater agreement is reported.

D. Corpus

The corpus consists of 52 studies (49 primary and 3 secondary studies). 15 studies mention and examine JavaScript/TypeScript/Node.js and were good enough to meet the low end of the simple coding rule. The corpus of studies (and the list of studies) is provided in Table II (and the list of studies in Table V). The 49 primary studies include 11 snowballed studies and 1 grey literature study. The search and screening process is shown in Fig. 1 and Table A-I. The corpus is a purposive sample.

TABLE II
CORPUS OVERVIEW

Dimension	n	Note
Primary studies	49	Experiments, benchmarks, measurements
Secondary studies	3	Systematic reviews
Peer-reviewed (as retrieved)	33	As retrieved; see Table V
Preprint or report	18	As retrieved; see Table V
Grey literature	1	Industry report
JavaScript named explicitly	15	Lower bound
Published 2021–2022	7	
Published 2023–2024	20	
Published 2025–2026	25	Fast-moving; see Sec. VIII

E. Search Sensitivity

A corpus-based sensitivity check on the 49 primary studies was then performed. With this search, 37/49 (75.5%) of the studies were produced by the search strings used, 11/49 (22.4%) by snowballing, and one is a grey-literature reference outside bibliographic indexes. Title only matching lists 16/49 (32.7%). These totals are for this corpus; the study per-study search and screening histories are needed to confirm the denominators, sources, and percentages in isolation. Number attributable to each query is displayed in Table A-II.

The S5 and S6 searchers followed the S1S4 sensitivity test. Since the sensitivity check revealed that 11 of the studies could not be retrieved by any of the published strings because the titles of the studies lack the terms security, productivity, or JavaScript, we didn't try to search for them by S5 and S6 (several concepts). You can't search for them without the last strings, the search field, and the per-study files.

Most (48/49; 98.0%) of the primary studies are included in an open index. The other primary study, which we could not identify for certain as being the grey-literature report [16], would not be included in bibliographic indexes, but we have not checked the indexes to confirm this. Database coverage and search-string sensitivity are separate issues.

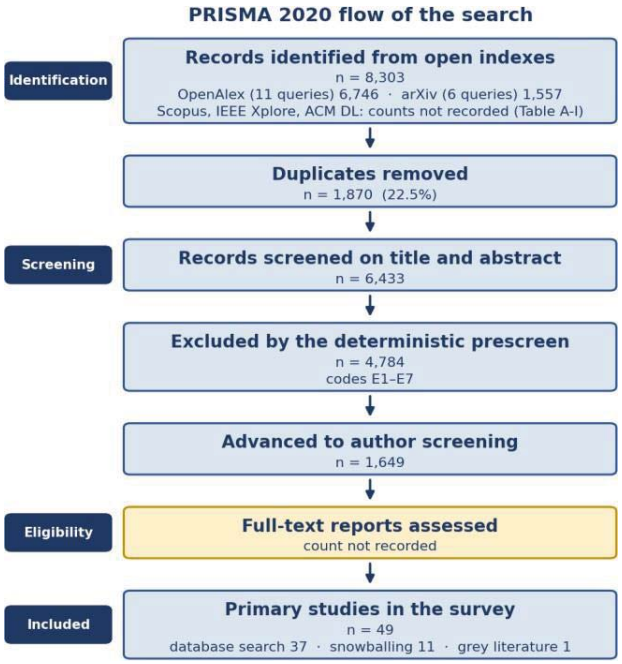


Figure 1. Search and screening flow (PRISMA 2020). #s indicate the number of results for the open-index searches (OpenAlex and arXiv); in this record the OpenAlex and arXiv search results are reported in the main text and the hit results for Scopus, IEEE Xplore and ACM Digital Library, as well as the full-text screening results, are not available (see Table A-I). Of the 49 primary studies, 37 were identified from the database search, 11 from snowballing, and 1 from a grey-literature report.

IV. CLASSIFICATION

Each main study is classified along two axes. The evaluation unit is what the study evaluates or produces: a human workflow (H), a function or code fragment (F), a repository or issue (R), a front-end page (W), a back-end service (B), or a full application (A). The quality property investigated is productivity (P), functional correctness (C), or security (S): some studies consider more than one property. The matrix of axes and their number in each axe is given in Table III and in Fig. 2.

TABLE III
PRIMARY STUDIES BY EVALUATION UNIT AND ATTRIBUTE

Unit	Productivity	Correctness	Security
Human / workflow	[17]–[21]	—	[22], [23]
Function / snippet	—	[1]–[4], [24]–[30]	[16], [24], [30]–[44]
Repository / issue	—	[5], [45]–[48]	[45], [48]
Front-end page	—	[49], [50]	—
Backend service	—	[51], [52]	[51]
Full-stack app	—	[53]–[55]	[56]–[59]

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

Secondary studies (reviews) are excluded from the grid. A study appears under each attribute it evaluates.

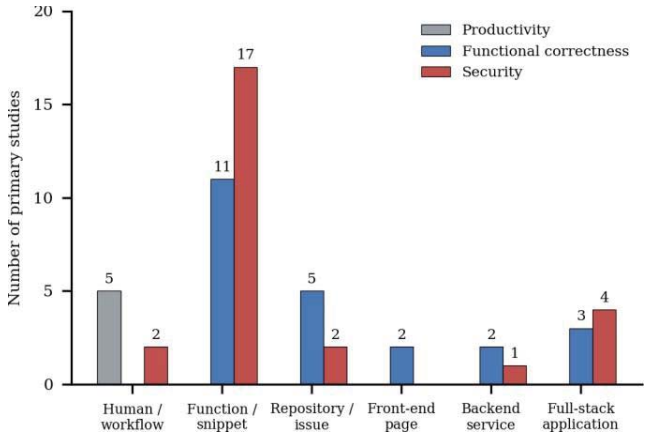


Fig. 2. Primary studies by evaluation unit and quality attribute. Security evidence is concentrated at function level; application-level evidence is sparse.

There are two patterns. First, while evidence at the function and snippet levels exists for security, it is usually focused on functions and snippets, whereas productivity evidence is focused on the human end-user. There is little evidence at the application level for either function or security, and these areas often do not overlap (see Section VII).

A. Evaluation Oracles

The oraclesmethods by which a study verifies the correctness of generated codevary between grids, making direct comparison challenging. Table IV lists the oracles used. Test-based oracles are most commonly applied to assess functional correctness. Security oracles range from static pattern matching, which is inexpensive but less effective for logical errors and susceptible to false positives, to exploit execution, which is more costly and tailored to specific exploited vulnerabilities.

TABLE IV
EVALUATION ORACLES AND REPRESENTATIVE STUDIES

Oracle	Strength / weakness	Examples
Unit or E2E tests	Objective; blind to security	[1], [3], [53]
GUI-agent tests	Scales to UIs; agent error	[54], [55]
Static analysis (SAST)	Cheap; false positives and misses	[31], [32], [34]
LLM judge	Catches semantic flaws; unverifiable	[32]
Fuzzing / PoC input	Precise; needs real vulnerabilities	[45], [48]
End-to-end exploit	Most realistic; costly per CWE	[51]
Hybrid SAST+DAST+manual	Complementary; small scale	[56], [59]
User study	Captures behavior; small samples	[19], [22], [23]

TABLE V
INCLUDED STUDIES AND EXTRACTED DATA (STATUS: P PEER-REVIEWED, X PREPRINT OR REPORT, G GREY LITERATURE)

Ref.	Study	Year	Status	Unit / attr.	Design and scale (as reported)	Headline result (as reported)
[17]	Peng et al.	2023	X	H / P	Controlled experiment; JavaScript HTTP server task	Treated group 55.8% faster (95% CI 21–89%)
[18]	Cui et al.	2025	P	H / P	Three field RCTs, 4,867 developers	+26.08% completed tasks; larger for less experienced
[19]	Becker et al.	2025	X	H / P	RCT, 16 developers, 246 tasks, mature repositories	AI allowed: 19% longer; forecast was 24% faster
[20]	Vaithilingam et al.	2022	P	H / P	Within-subjects usability study	No significant gain in speed or success rate
[21]	Ziegler et al.	2022	P	H / P	Telemetry plus developer survey	Acceptance rate tracks perceived productivity
[22]	Perry et al.	2023	P	H / S	User study, assistant vs. no assistant	Less secure code and higher confidence with assistant
[23]	Sandoval et al.	2023	P	H / S	User study, low-level C task	Only a small security impact measured
[33]	Asare et al.	2023	P	F / S	Copilot vs. human on known vulnerabilities	Reintroduced known flaws in about a third of cases
[1]	Chen et al.	2021	X	F / C	HumanEval; pass@k	Introduced the standard function-level benchmark
[2]	Austin et al.	2021	X	F / C	MBPP; entry-level Python tasks	Second standard function-level benchmark
[3]	Liu et al.	2023	P	F / C	EvalPlus; HumanEval with ~80x tests; 26 LLMs	pass@k drops by up to 19.3–28.9%
[4]	Cassano et al.	2023	P	F / C	MultiPL-E; translated benchmarks, many languages	Cross-language function-level evaluation incl. JavaScript
[25]	Nguyen and Nadi	2022	P	F / C	33 LeetCode problems, 4 languages	Correctness: Java 57%, JavaScript 27% (lowest)
[26]	Mo et al.	2025	P	F / C	2,033 LeetCode problems, 4 languages	54.1% of problems had a correct JavaScript suggestion

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

Ref.	Study	Year	Status	Unit / attr.	Design and scale (as reported)	Headline result (as reported)
[27]	Dakhel et al.	2023	P	F / C	Copilot vs. human solutions, basic algorithms	Compared Copilot with human programmers
[28]	Schäfer et al.	2024	P	F / C	LLM unit-test generation on JavaScript npm packages	First large-scale JavaScript test-generation evaluation
[5]	Jimenez et al.	2024	P	R / C	SWE-bench; 2,294 GitHub issues, Python repos	Standard repository-level issue benchmark
[46]	Yang et al.	2025	P	R / C	SWE-bench Multimodal; visual JavaScript domains	Issue resolving with visual context
[47]	Zan et al.	2025	X	R / C	Multi-SWE-bench; multilingual issues incl. JS/TS	Issue resolving beyond Python
[49]	Si et al.	2024	P	W / C	Design2Code; screenshot to front-end code	Visual-fidelity evaluation of generated pages
[50]	Zhu et al.	2025	X	W / C	FrontendBench; 148 tasks; Jest and Puppeteer	Interactive front-end tests across five levels
[54]	Lu et al.	2025	P	A / C	WebGen-Bench; 647 test cases; websites from scratch	Best off-the-shelf 27.8%; fine-tuned 38.2%
[53]	Xu et al.	2025	X	A / C	Web-Bench; 50 projects x 20 dependent tasks	Best Pass@1 25.1% (Claude 3.7 Sonnet)
[55]	Lu et al.	2026	P	A / C	FullStack-Agent; frontend, backend, database tests	Best system 64.7 / 77.8 / 77.9% (FE / BE / DB)
[29]	ByteDance Seed team	2024	X	F / C	FullStack Bench; 3,374 problems, 16 languages, 11 domains	Unit-test tasks; HumanEval does not predict ranking
[52]	Guo et al.	2026	X	B / C	BackendForge; 56 OpenAPI-contract backend tasks	Best 55.4% (base oracle) vs. 28.6% (final oracle)
[51]	Vero et al.	2025	P	B / C/S	BaxBench; 392 backend tasks; exploit-based security	Best 62% correct; ~half of correct programs exploitable
[48]	Zhao et al.	2026	P	R / C/S	SusVibes; 186 repository tasks; coding agents (Python)	Best agent 57% correct, 11.8% correct and secure
[45]	Dilgren et al.	2025	X	R / C/S	SecRepoBench; 318 C/C++ tasks, 27 repositories	Best LLM 39.3% secure-pass@1
[31]	Pearce et al.	2022	P	F / S	89 scenarios, 1,689 programs; CodeQL plus review	About 40% of completions vulnerable
[34]	Fu et al.	2025	P	F / S	733 snippets from GitHub projects; Python and JavaScript	Weaknesses in 29.5% (Py) and 24.2% (JS) of snippets
[35]	Spracklen et al.	2025	P	F / S	16 LLMs, 576,000 samples; Python and JavaScript	Hallucinated packages: JS 21.3%, Python 15.8%
[36]	Siddiq and Santos	2022	P	F / S	SecurityEval; 130 Python prompts, 75 CWEs	Semi-automated static-analysis screening
[37]	Khoury et al.	2023	P	F / S	21 ChatGPT-generated programs	Only 5 of 21 programs secure on first attempt
[38]	Hajipour et al.	2024	P	F / S	CodeLMSec; 280 prompts, 13 CWEs, 2 languages	Black-box search for vulnerability-inducing prompts
[24]	Peng et al.	2025	X	F / C/S	CWEval; 119 tasks, 31 CWEs, 5 languages	Joint functionality-and-security tests; many correct yet insecure
[30]	Yang / Nie et al.	2024	P	F / C/S	SecCodePLT; 2024 v1, revised 2025 peer-reviewed version	Insecure-code and cyberattack-helpfulness evaluation
[39]	Bhatt et al.	2024	X	F / S	CyberSecEval 2; behavioral security suite	Prompt injection, interpreter abuse, insecure code
[32]	Li et al.	2025	X	F / S	SafeGenBench; 558 cases, 44 CWEs, 12 languages	Zero-shot mean 37.44% secure; +20 points with safety prompt
[40]	Tihanyi et al.	2024	X	F / S	FormAI-v2; 331,000 C programs, 9 LLMs	At least 62.07% of programs vulnerable
[41]	Kharma et al.	2025	X	F / S	Multi-language, multi-model security and quality analysis	Security effectiveness varies by language
[42]	He and Vechev	2023	P	F / S	SVEN; security hardening and adversarial testing	Controls security of generated code
[43]	Tony et al.	2025	P	F / S	Prompting techniques for secure code generation	Weaknesses reduced; iterative self-critique most consistent
[44]	Shukla et al.	2025	P	F / S	Controlled iterative-refinement experiment	+37.6% critical vulnerabilities after five rounds
[16]	Veracode	2025	G	F / S	Industry report; 100+ models, 80 tasks	About 45% of samples carry an OWASP Top 10 weakness
[57]	Tóth et al.	2024	X	A / S	2,500 PHP websites generated by GPT-4	About 26% had at least one web-exploitable flaw
[58]	Dora	2025	X	A / S	Five LLMs vs. a web-security checklist	Significant non-compliance in authentication and sessions

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

Ref.	Study	Year	Status	Unit / attr.	Design and scale (as reported)	Headline result (as reported)
[59]	Heidaripour et al.	2026	P	A / S	10 full-stack apps, 5 stacks incl. Node.js/Express	275 findings; misconfiguration and auth/session dominate
[56]	Andročec	2026	P	A / S	12 apps (6 tasks x 2 prompts), Python and Express	75 confirmed findings; security prompt: 24 vs. 51
[8]	Negri-Ribalta et al.	2024	P	S / S	Systematic literature review	Risks of AI-generated code for security-critical software
[10]	Basic and Giaretta	2026	X	S / S	Systematic literature review	Ten categories of vulnerabilities in LLM-generated code
[9]	Criollo Ramírez et al.	2024	P	S / S	Systematic review (Kitchenham guidelines)	7 vulnerability kinds, 6 mitigations, 4 tools

Metadata: unit: H (human/workflow), F (function/snippet), R (repository/issue), W (front-end page), B (back-end service), A (whole application), S (secondary study). Attribute: P (productivity), C (correctness), S (security). Values are copied directly from the sources cited and have not been recalculated. Publication-status tags mark the version of the resource that we obtained as we quoted the publication.

V. Productivity and Functional Correctness (RQ1)

A. Does It Make Developers Faster?

Mixed, time to complete was reduced by 55.8% (95% CI 2189%) in an RCT where developers were asked to implement an HTTP server in JavaScript in a controlled setting [17]. An experiment done with 4867 developers over three companies demonstrated a 26.08% increase in completed tasks, with the largest effects for less experienced subjects [18]. A usability study reported no effect on either speed or success but that users appreciated the tool [20]. Telemetry work demonstrates suggestion acceptance rate is related to perceived productivity [21].

Most alarming was a randomized trial of 16 experienced open-source developers, who attempted 246 tasks in mature repositories (average age of five years). Use of the AIs in early 2025 set everything back by 19%, where developers would have expected a 24% boost and still gave the AI credit for being helpful [19]. This is "one data point across one environment, " note the authors.

These findings are not contradictory. They address separate groups of developers, different tasks, and distinct measures of benefit. Collectively, they suggest that benefit is highest for less experienced developers and for isolated issues, and lowest or potentially negative for experts working with large codebases. No study in the corpus investigates the impact on full-stack work involving a front end, an API, and a database.

B. Is the Code Correct?

HumanEval evaluates correctness at the function level [1]. For example, up to 19.3-28.9% lower bootstrapped pass@k than the original pass@k across 26 models when tests were bootstrapped with roughly 80X the original number of tests. Small test sets can bring Really overinflated estimates of correctness [3].

Most JavaScript-specific results are even more limited and inconsistent. A report based on 33 problems on LeetCode found JavaScript-based suggestions worst of four languages was 27% (Java, 57%) [25], while a report based on 2,033 problems published accuracy was 54.1% of problems have at least one suggestion for JavaScript [26]. But, the two studies were based on different problem sets, metrics, and tool versions, and cannot be directly compared. Yet, both consistently show the problem-specific nature of "JavaScript accuracy". Work has been done on units tests generation from JavaScript npm packages [28].

C. From Functions to Applications

TABLE VI
Application-oriented benchmarks: unit and oracles

Benchmark	Unit	Functional oracle	Security oracle
WebGen-Bench [54]	Website	647 GUI-agent test cases	—
Web-Bench [53]	50 projects × 20 tasks	Playwright E2E tests	—
FullStack-Agent (Lu et al.) [55]	Full-stack site	Frontend, backend, DB tests	—
FrontendBench [50]	Front-end page	Jest / Puppeteer scripts	—
BackendForge [52]	Backend service	HTTP tests vs. OpenAPI	—
BaxBench [51]	Backend app	Functional API tests	End-to-end exploits
SusVibes [48]	Repo feature task	Unit tests	Security unit tests
SecRepoBench [45]	C/C++ completion	Developer unit tests	Fuzzer PoC inputs

"-": no security oracle is introduced in the material we read.

Table VI -application-oriented benchmark results with web-based versus software-based tests- application generation has not yet achieved complete application generation but, because benchmarks, agents, models, and dates are not directly comparable, the benchmarks listed below are distinct. For WebGen-Bench, the off-the-shelf best was 27.8% accuracy and the open model fine-tuned best was 38.2%, which was claimed to beat the best commercial model [54]. The best Web-Bench percentage is 25.1% Pass@1 [53]. FullStack-Agent achieved 64.7%, 77.8%, and 77.9% of the frontend, backend, and database tests respectively [55]. The improvement to the oracle variable helped BackendForge: the best model pass rate was 55.4% for the base oracle and 28.6% for the final oracle [52]. Because these percentages do not correspond to the same tasks and were performed by different agents across different timeframes, their values are not ranked or even compared with

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

issue-resolution percentages; but, they indicate progress is still being made and the test oracle is clearly a factor.

Another question is scope. Unit-style benchmarks, like a multilingual, full-stack benchmark [29], shows that models that perform well on a function-level task, like HumanEval, do not work on practical tasks.

VI. SECURITY RISKS (RQ2)

A. Insecure Suggestions at Snippet Level

Security-related completion cases show that a plausible suggestion can be syntactically correct but still insecure. Pearce et al. assessed GitHub Copilot on 89 security-related scenarios, examining the 1,689 programs it generated [31]. Based on the experiment's set-up, this represents 40% of all the programs it produced (and not all the final programs).

What else should we consider? Benchmark design. SecCodePLT finds untrusted code through rule-based security analysis and dynamic testing, and exposes failures in categories of weaknesses [30]. Results of static finding assessments and dynamic test failures give different responses, and should not be taken as a binary answer of exploitability in production. The choice of model prompt model version, and security oracle shape the meaning of what is found by each benchmark, and the transfer of information from isolated snippets about individual programs to exploitability of full-stack JavaScript applications. We examined another repo-based attack model survey in which 76 of 314 (24.2%) JavaScript snippets contained a weakness found by static analyzers, including JavaScript program vulnerabilities like path traversal and code injection [34]. This sample was exclusively verifiably AI-generated JavaScript snippets, not AI-generated JavaScript in general, and not in-production applications, so cannot be compared directly with other data.

B. Real-World Code

Fu et al. examined 733 code snippets produced by Copilot, CodeWhisperer, and Codeium, which programmers had stored in GitHub repositories. Deficiencies affected 29.5% of Python and 24.2% of JavaScript snippets across 43 CWE categories, including weak randomness (CWE-330), code injection (CWE-94), and cross-site scripting (CWE-79); Copilot Chat was able to fix up to 55.5% with analyzer warnings [34]. It is the strongest JavaScript field evidence in this corpus.

C. Human Factors

On security, assisted users generated more insecure code and thought it was secure [22], a user study of a low level C task found weak security related effects [23], and Copilot was not consistently worse than a human programmer at re-adding established security bugs [33]. The risk is that the model makes an error and it goes unnoticed.

D. Dependencies and the Supply Chain

Models suggested packages that do not exist for 21.3% (of 576,000) of JavaScript and 15.8% (of 576,000) of Python samples over 16 LLMs [35]. An attacker could register such packages and wait for a victim to install them. Although the average is higher for JS than for Python, this is a language-agnostic problem and since npm has a large average number of dependencies, it further worsens the risk.

E. Backends, Repositories and Agents

BaxBench replaced checks with entire backends: 392 tasks (28 scenarios and 14 setups for 6 languages (including the Node.js Express, Fastify, Koa, and NestJS ones)) with functional tests and end-to-end exploits. The best model was 62% correct, with exploits working against about half of each models' correct programs, less popular setups being more difficult [51]. At the repo level, SecRepoBench (C/C++) shows the best LLM had a 39.3% secure-pass@1 [45], and SusVibes shows the best coding agent was 57% functionally correct and 11.8% correct and secure, with 79.3% of its correct solutions insecure, and vulnerability hints not able to fix this [48].

F. Whole Web Applications

Few studies have evaluated fully generated applications. One review of 2,500 PHP websites produced with GPT-4 found that approximately 26% had at least one vulnerability accessible through web interaction [57]. A checklist-based assessment of five LLMs identified significant shortcomings in authentication, session management, and input validation [58]. A cross-language analysis of ten full-stack applications across five technology stacks, including Node.js/Express, found 275 issues, most involving security misconfiguration and authentication or session management [59]. A twin-prompt study of 12 applications in Python and Express identified 75 vulnerabilities, with 48 categorized as security misconfigurations; adding a security section to the prompt reduced confirmed vulnerabilities from 51 to 24 and eliminated all critical and high-severity issues, with each application generated once [56]. That research also noted that SAST, DAST, and manual testing were complementary, and that the most severe issuea hard-coded JWT secretwas detected only through manual review.

G. Mitigation

Prompt-level mitigation offers some improvement but remains inadequate. A systematic comparison showed that prompting methods can lessen weaknesses, with iterative self-critique proving the most reliably effective [43]; adding a safety instruction raised the secure rate by about 20 points on SafeGenBench [32]. Still, brief vulnerability hints did not benefit coding agents [48], and in one controlled study, five rounds of AI-assisted refinement led to a 37.6% rise in critical vulnerabilities [44]. Hardening during training is another possible strategy [42].

H. Taxonomy of Reported Risks

In Table VII, the main drawbacks pointed out by the primary studies are listed about web application categories, per identifiers found in the CWE [60] and OWASP Top 10 [61] reports. These references are ours, the papers listed provide the evidence.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

TABLE VII
TAXONOMY OF RISKS IN AI-GENERATED WEB CODE

Category	Typical CWE / OWASP	Reported evidence
Injection	CWE-89, 78, 94; A03	Code injection among top JS weaknesses [34]; present in generated apps [56]
Cross-site scripting	CWE-79	Prominent in real-project snippets [34]
Access control and authentication	CWE-862, 306, 287; A01, A07	Lowest category score in SafeGenBench zero-shot (24.07%) [32]; forgeable tokens and sessions [56]
Secrets and credentials	CWE-798, 259	Hard-coded JWT secret, found only manually [56]
Weak randomness / crypto	CWE-330, 327	CWE-330 prominent in real-project snippets [34]
Misconfiguration	CWE-16; A05	Missing headers, CSRF and cookie flags [59]; 48 of 75 findings [56]
Supply chain	CWE-1104	Hallucinated packages, JS 21.3% [35]
Resource exhaustion	CWE-400	Tested exploit class in BaxBench [51]
Human factors	—	Overconfidence in insecure output [22]

VII. GAPS AND RESEARCH AGENDA (RQ3)

A. What Is and Is Not Covered

Here is a precise way to say that, for the application layer, the corpus has gained: BaxBench on exploits on full generated backends [51]; SusVibes on agent repository-level feature tasks [48]; four studies of full generated web apps [56][59]. What is missing from the corpus? A benchmark covering functional correctness, exploit-confirmed security, in full-stack JavaScript apps (frontend backend database) at scale. For the 52 studies we surveyed; it has not been missing from the literature [see III-E for how we determined our negative conclusion].

The evidence is:. The only large exploit-driven benchmarks focus on backends, and list 6 languages, with only one being JavaScript [51]. The repository-level agent benchmarks focus on Python [48], or C/C++ [45] systems. The three full-stack functional benchmarks [53][55] specify no security oracle in their sources. For the 4 application-level security papers, three are small (up to 12 applications each), use bespoke tasks and single assistant or multiple chat interfaces, and almost all use static or manual analysis; only one explicitly includes Express applications with Python [56], and one also includes Node.js as one of 5 stacks [59]. Only 5 primary studies evaluate both functionality and security [24], [30], [45], [48], [51], and none specialise in full-stack JavaScript applications.

B. Thin and Uneven JavaScript Evidence

Only 15/52 papers explicitly talk about JavaScript, TypeScript, or Node.js. Many security findings are derived from C, Python, and PHP papers [40], [45], [48], [57] but would also apply to JavaScript: injection, access control, and misconfiguration arelanguage independent. It is an assumption, not yet backed up by data, that these would be equally common in JavaScript - that's how far the JavaScript-specific risks of prototype pollution, client-side XSS via escape hatches, or npm dependencies, have been analyzed.

C. Non-Comparable Evaluation Methods

Results are grouped by pass@k, GUI agent, CodeQL alerts fuzzers exploits, task timing, and expert review (Table IV). Results from one approach cannot be mapped onto those of others, as models, agents, and dates vary by approach. For example, the percentages (25.1%, 27.8%) characterize the benchmark at a specific point in time, and are not equivalent to "AI."

D. Maintainability and Long-Term Effects

The only work in our corpus that mentions the long term of generated full-stack applications is the productivity study of mature repositories [19] which measure the time to completion. We failed to find a work measuring maintainability, change-proneness, or incremental security debt of generated full-stack applications. Iterative degradation [44] demonstrates that this is a relevant risk.

E. Research Agenda

We identify seven directions for future work.

1) Exploit-validated full-stack benchmark. Specify product requirements; demand a deployable React or similar UI, a Node.js API, and a database; evaluate functionality with end-to-end tests, and score security based on exploits per category (XSS CSRF injection, access control, secrets).

2) Layer attributed reporting. Report problems by layer and trust boundary to allow that to be combined across studies.

3) Live dependency checks. For each candidate package, check with the registry and check for known vulnerabilities, and increase the hallucination measurements from [35].

4) Variance and version control. Repeat each task multiple times, document the variation, and keep track of model and tool versions, to overcome the single-run constraint observed in small-application research [56].

5) Joint indicators. Use application level indicators that demand function and security together e.g secure-pass@1 [45].

6) Full application user study. Determine if reviewers are able to detect application errors in generated applications in a multi-task setting, unlike the single task setting used in [22].

7) Mitigation in the wild. Under the expected operational rate, evaluate security recommendations, agent self-assessment, and SAST/DAST in the loop on full applications, taking into account their functional cost [43], [48].

VIII. DISCUSSION AND THREATS TO VALIDITY

A. Implications

The state of the picture is patchy. Defensive evidence is rapidly moving from functions into applications, while security evidence is thick at the snippet, backend, and repository levels, but weak at JavaScript application level. For the practitioners, the message is easily summarized: treat generated code as untrusted: review authentication, session, and configuration code, check all suggested dependencies, run static, dynamic,

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

and manual tests, since each finds a different set of bugs [56], and don't rely solely on prompt text [44], [48].

B. Threats to Validity

Searches and inclusions. The searches were conducted on Scopus, IEEE Xplore, the ACM Digital Library, arXiv, and OpenAlex. They were not conducted on and have not been conducted on Web of Science; studies only available from there may be absent from this review. Hit numbers for the 3 subscription resources and the full-text screening counts are not available on this record (see Table A-I and Fig. 1). The searches, full-text exclusions, and sensitivity numbers should be checked against the source records. The first numbers 37/49 and 11/49 are for primary studies only and do not include the 52 total number of studies.

Screening. Scripted de-duplication, prescreening, and counting are available, but reproducibility cannot be shown without access to scripts and exports. Screening and classification were performed by a single reviewer, which may introduce bias in both selection and coding. Reporting inter-rater reliability from an independent sample would help address this concern.

Evidence quality. The publishing status reflects the version date of retrieval from the source. A full audit of the source versioning was not performed, and many items had previously been published as preprints. The studies are also too diverse in the task language model, oracle, and experimental design to compare directly, apply to full-stack JavaScript or aggregate for a meaningful pooled meta-analysis.

Temporality. The results are only relevant for the models or tools that generated them and a 2025 result may be far out of date with the advent of a new model, as we saw with the difference between [25] and [26]. Think of the numbers as timestamps.

Not possible to construct (not existent) a claim that a benchmark or study does not exist is a claim about a group of sources that have been searched but for which nothing was found. Such claims can be supported by reproductions with a formal protocol, exported from a database, and a second reviewer.

IX. CONCLUSION

AI code generation already has proven value and known defined risks. For full-stack JavaScript, there is more and more proven capability and strong evidence of security at the snippet, backend, and repository levels without a single benchmark bringing together exploit-validated security and layered functional testing at the frontend, backend, and database level. Closing that gap with layer-attributed reporting, live dependency tracking, and repeated, versioned generation is the surest way to make the evidence on AI assisted full-stack JavaScript cumulative.

ACKNOWLEDGMENT

Preparation of papers was aided by computers while preparing this manuscript (literature searching, search strategies, data extraction, double entry and duplicate deletion, language editing). The author personally validated and cross checked every statement of evidence and every reference, fact, and analysis of information, and takes sole responsibility for the contents of this article.

REFERENCES

[1] M. Chen *et al.*, “Evaluating large language models trained on code,” arXiv:2107.03374, 2021.

[2] J. Austin *et al.*, “Program synthesis with large language models,” arXiv:2108.07732, 2021.

[3] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation,” in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2023, arXiv:2305.01210.

[4] F. Cassano *et al.*, “MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation,” *IEEE Trans. Softw. Eng.*, vol. 49, no. 7, pp. 3675–3691, Jul. 2023.

[5] C. E. Jimenez *et al.*, “SWE-bench: Can language models resolve real-world GitHub issues?,” in *Proc. 12th Int. Conf. Learn. Represent. (ICLR)*, 2024, arXiv:2310.06770.

[6] X. Hou *et al.*, “Large language models for software engineering: A systematic literature review,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, Art. no. 220, 2024.

[7] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, “A survey on large language models for code generation,” arXiv:2406.00515, 2024.

[8] 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,” *Front. Big Data*, vol. 7, Art. no. 1386720, 2024, doi: 10.3389/fdata.2024.1386720.

[9] L. Criollo Ramírez, X. Limón, Á. 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 *Proc. 12th Int. Conf. Softw. Eng. Res. Innov. (CONISOFT)*, Puerto Escondido, Mexico, 2024, pp. 331–339, doi: 10.1109/CONISOFT63288.2024.00050.

[10] E. Basic and A. Giaretta, “From vulnerabilities to remediation: A systematic literature review of LLMs in code security,” arXiv:2412.15004v4, Mar. 2026.

[11] B. Kitchenham and S. Charters, “Guidelines for performing systematic literature reviews in software engineering,” Keele Univ., Keele, U.K., Tech. Rep. EBSE-2007-01, 2007.

[12] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, “Systematic mapping studies in software engineering,” in *Proc. 12th Int. Conf. Eval. Assess. Softw. Eng. (EASE)*, 2008, pp. 68–77.

[13] C. Wohlin, “Guidelines for snowballing in systematic literature studies and a replication in software engineering,” in *Proc. 18th Int. Conf. Eval. Assess. Softw. Eng. (EASE)*, 2014, Art. no. 38.

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

[15] M. J. Rethlefsen et al., “PRISMA-S: An extension to the PRISMA statement for reporting literature searches in systematic reviews,” Syst. Rev., vol. 10, Art. no. 39, 2021, doi: 10.1186/s13643-020-01542-z.

[16] Veracode, “2025 GenAI code security report,” Veracode, Burlington, MA, USA, 2025.

[17] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, “The impact of AI on developer productivity: Evidence from GitHub Copilot,” arXiv:2302.06590, 2023.

[18] Z. Cui, M. Demirer, S. Jaffe, L. Musolff, S. Peng, and T. Salz, “The effects of generative AI on high-skilled work: Evidence from three field experiments with software developers,” *Manage. Sci.*, 2025, doi: 10.1287/mnsc.2025.00535.

[19] J. Becker, N. Rush, B. Barnes, and D. Rein, “Measuring the impact of early-2025 AI on experienced open-source developer productivity,” arXiv:2507.09089, 2025.

[20] P. Vaithilingam, T. Zhang, and E. L. Glassman, “Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models,” in *Ext. Abstr. CHI Conf. Human Factors Comput. Syst.*, 2022, Art. no. 332, doi: 10.1145/3491101.3519665.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

[21] A. Ziegler *et al.*, "Productivity assessment of neural code completion," in *Proc. 6th ACM SIGPLAN Int. Symp. Mach. Program. (MAPS)*, 2022, pp. 21–29, doi: 10.1145/3520312.3534864.

[22] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, Copenhagen, Denmark, 2023, pp. 2785–2799, doi: 10.1145/3576915.3623157.

[23] 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 Secur. Symp.*, 2023.

[24] J. Peng, L. Cui, K. Huang, J. Yang, and B. Ray, "CWEval: Outcome-driven evaluation on functionality and security of LLM code generation," arXiv:2501.08200, 2025.

[25] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub Copilot's code suggestions," in *Proc. 19th Int. Conf. Mining Softw. Repositories (MSR)*, 2022, pp. 1–5, doi: 10.1145/3524842.3528470.

[26] R. Mo, D. Wang, W. Zhan, Y. Jiang, Y. Wang, Y. Zhao, Z. Li, and Y. Ma, "Assessing and analyzing the correctness of GitHub Copilot's code suggestions," *ACM Trans. Softw. Eng. Methodol.*, 2025, doi: 10.1145/3715108.

[27] A. M. Dakhel *et al.*, "GitHub Copilot AI pair programmer: Asset or liability?," *J. Syst. Softw.*, vol. 203, Art. no. 111734, 2023.

[28] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Trans. Softw. Eng.*, vol. 50, no. 1, pp. 85–105, Jan. 2024.

[29] ByteDance Seed Foundation Code Team, Y. Cheng *et al.*, "FullStack Bench: Evaluating LLMs as full stack coders," arXiv:2412.00535, 2024.

[30] Y. Yang et al., "SecCodePLT: A unified platform for evaluating the security of code GenAI," arXiv:2410.11096v1, 2024; revised version: Y. Nie et al., "SECODEPLT: A unified benchmark for evaluating the security risks and capabilities of code GenAI," NeurIPS Datasets and Benchmarks, 2025, arXiv:2410.11096v2.

[31] 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 Symp. Secur. Privacy (SP)*, 2022, pp. 754–768, doi: 10.1109/SP46214.2022.9833571.

[32] X. Li, J. Ding, C. Peng, B. Zhao, X. Gao, H. Gao, and X. Gu, "SafeGenBench: A benchmark framework for security vulnerability detection in LLM-generated code," arXiv:2506.05692, 2025.

[33] O. Asare, M. Nagappan, and N. Asokan, "Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code?," *Empir. Softw. Eng.*, vol. 28, no. 6, Art. no. 129, 2023, doi: 10.1007/s10664-023-10380-1.

[34] 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 Trans. Softw. Eng. Methodol.*, vol. 34, no. 8, pp. 1–34, 2025, doi: 10.1145/3716848.

[35] 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 *Proc. 34th USENIX Secur. Symp.*, Seattle, WA, USA, 2025, pp. 3687–3706.

[36] M. L. Siddiq and J. C. S. Santos, "SecurityEval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques," in *Proc. 1st Int. Workshop Mining Softw. Repositories Appl. Privacy Secur. (MSR4P&S)*, 2022, pp. 29–33, doi: 10.1145/3549035.3561184.

[37] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How secure is code generated by ChatGPT?," in *Proc. IEEE Int. Conf. Syst., Man, Cybern. (SMC)*, 2023, pp. 2445–2451, doi: 10.1109/SMC53992.2023.10394237.

[38] H. Hajipour, K. Hassler, T. Holz, L. Schönherr, and M. Fritz, "CodeLMSec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models," in *Proc. IEEE Conf. Secure Trustworthy Mach. Learn. (SaTML)*, 2024, pp. 684–709.

[39] M. Bhatt *et al.*, "CyberSecEval 2: A wide-ranging cybersecurity evaluation suite for large language models," arXiv:2404.13161, 2024.

[40] N. Tihanyi, T. Bisztray, M. A. Ferrag, R. Jain, and L. C. Cordeiro, "How secure is AI-generated code: A large-scale comparison of large language models," arXiv:2404.18353, 2024.

[41] M. F. 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.

[42] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, 2023, pp. 1865–1879.

[43] C. Tony, N. E. Díaz Ferreyra, M. Mutas, S. Dhiff, and R. Scandariato, "Prompting techniques for secure code generation: A systematic investigation," *ACM Trans. Softw. Eng. Methodol.*, 2025, doi: 10.1145/3722108.

[44] S. Shukla, H. Joshi, and R. Syed, "Security degradation in iterative AI code generation: A systematic analysis of the paradox," in *Proc. IEEE Int. Symp. Technol. Soc. (ISTAS)*, 2025, arXiv:2506.11022.

[45] C. Dilgren, P. Chiniya, L. Griffith, Y. Ding, and Y. Chen, "SecRepoBench: Benchmarking LLMs for secure code generation in real-world repositories," arXiv:2504.21205, 2025.

[46] J. Yang *et al.*, "SWE-bench Multimodal: Do AI systems generalize to visual software domains?," in *Proc. 13th Int. Conf. Learn. Represent. (ICLR)*, 2025, arXiv:2410.03859.

[47] D. Zan *et al.*, "Multi-SWE-bench: A multilingual benchmark for issue resolving," arXiv:2504.02605, 2025.

[48] S. Zhao, D. Wang, K. Zhang, J. Luo, Z. Li, and L. Li, "Is vibe coding safe? Benchmarking vulnerability of agent-generated code in real-world tasks," in *Proc. 43rd Int. Conf. Mach. Learn. (ICML)*, ser. PMLR, vol. 306, 2026, arXiv:2512.03262.

[49] C. Si *et al.*, "Design2Code: How far are we from automating front-end engineering?," arXiv:2403.03163, 2024.

[50] H. Zhu, Y. Zhang, B. Zhao, J. Ding, S. Liu, T. Liu, D. Wang, Y. Liu, and Z. Li, "FrontendBench: A benchmark for evaluating LLMs on front-end development via automatic evaluation," arXiv:2506.13832, 2025.

[51] M. Vero, N. Mündler, V. Chibotaru, V. Raychev, M. Baader, N. Jovanović, J. He, and M. Vechev, "BaxBench: Can LLMs generate correct and secure backends?," in *Proc. 42nd Int. Conf. Mach. Learn. (ICML)*, ser. PMLR, vol. 267, 2025, pp. 61344–61390.

[52] Y. Guo, M. Wu, Y. Cao, J. Wei, D. Ran, W. Yang, and T. Xie, "BackendForge: Benchmarking agentic end-to-end code generation with backend services," arXiv:2607.11042, 2026.

[53] K. Xu, Y. Mao, X. Guan, and Z. Feng, "Web-Bench: A LLM code benchmark based on web standards and frameworks," arXiv:2505.07473, 2025.

[54] Z. Lu, Y. Yang, H. Ren, H. Hou, H. Xiao, K. Wang, W. Shi, A. Zhou, M. Zhan, and H. Li, "WebGen-Bench: Evaluating LLMs on generating interactive and functional websites from scratch," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2025, arXiv:2505.03733.

[55] Z. Lu, H. Ren, Y. Yang, K. Wang, Z. Zong, M. Zhan, and H. Li, "FullStack-Agent: Enhancing agentic full-stack web coding via development-oriented testing and repository back-translation," in Proc. 43rd Int. Conf. Mach. Learn. (ICML), 2026, arXiv:2602.03798.

[56] D. Andročec, "Vibe coding and web application security: A twin-prompt study," accepted for presentation at the *37th Central Eur. Conf. Inf. Intell. Syst. (CECIIS)*, Varaždin, Croatia, 2026, arXiv:2608.20963.

[57] R. Tóth, T. Bisztray, and L. Erdődi, "LLMs in web development: Evaluating LLM-generated PHP code unveiling vulnerabilities and limitations," arXiv:2404.14459, 2024.

[58] S. Dora, "The hidden risks of LLM-generated web application code: A security-centric evaluation of code generation capabilities in large language models," arXiv:2504.20612, 2025.

[59] M. Heidaripour *et al.*, "How secure are full-stack web applications generated by large language models? A multi-language security study," Springer, Aug. 2026, doi: 10.1007/978-3-032-35576-8_4.

[60] MITRE Corporation, "Common Weakness Enumeration (CWE)." [Online]. Available: https://cwe.mitre.org/ (accessed Nov. 2, 2026).

[61] OWASP Foundation, "OWASP Top 10:2021." [Online]. Available: https://owasp.org/Top10/2021/ (accessed Nov. 2, 2026).

APPENDIX A. COUNTING RECORD

Table A-I shows the number of records we have for the search and screening stages; in Table A-II we show how many records

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

each separate search and screen query produced as a proportion of the 6433 total available records. Missing counts for available records are indicated as 'Not recorded'; no attempt has been made to estimate the missing counts. Full text counts are for the screening record, not search engine hits.

TABLE A-I
COUNTING RECORD FOR THE SEARCH AND SCREENING STAGES

Stage	Source / rule	n
Records identified	Scopus, S1–S6	Not recorded
Records identified	IEEE Xplore, S1–S6	Not recorded
Records identified	ACM Digital Library, S1–S6	Not recorded
Records identified	OpenAlex, 11 atomic queries	6,746
Records identified	arXiv, 6 full-record queries	1,557
Records identified	arXiv, 4 count-only queries	312 (counts only)
Records identified, total	OpenAlex and arXiv full-record queries	8,303
Records from other methods	backward/forward snowballing	11 included studies
Duplicates removed	DOI, then normalized title, then version merge	1,870 (22.5%)
Records screened	title and abstract	6,433
Excluded at prescreen	deterministic rules, codes E1–E7	4,784
Advanced to author screening		1,649
Full-text reports assessed		Not recorded
Reports excluded at full text		Not recorded
Studies included	primary studies (37 database search, 11 snowballing, 1 grey literature)	49
Executed string recall	against the included set	37/49 = 75.5%
Open-index coverage	of the included set	48/49 = 98.0%

Note: The identification and screening counts are internally consistent: 6,746 + 1,557 = 8,303 records identified; 8,303 − 1,870 = 6,433 records screened; 6,433 − 4,784 = 1,649 records advanced to author screening. The 312 count-only arXiv records are excluded from these totals. Hit counts for the subscription databases, the full-text counts, and any additional studies identified through them are not recorded and have not been estimated.

TABLE A-II
PER-QUERY CONTRIBUTION (RECORDS RETURNED BY THAT QUERY ALONE)

Query	Records unique to this query	% of unique records
S1a	1	0.02
S1b	52	0.81
S1c	33	0.51
S2a	57	0.89
S2b	531	8.25
S2c	821	12.76
S2d	2	0.03
S2e	577	8.97
S3a	796	12.37
S3b	23	0.36
S4a	1272	19.77
V2a	101	1.57
V2b	127	1.97
V2c	218	3.39
V2d	3	0.05
V2e	241	3.75
V2f	287	4.46

(This work is licensed under a Creative Commons Attribution 4.0 International License.)