iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** III **Month of publication:** March 2026

DOI: https://doi.org/10.22214/ijraset.2026.77834

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue III Mar 2026- Available at www.ijraset.com

Generative AI in Software Engineering

Arsha CH[1], Sreeji KB[2]
[1]*MCA Student, Department of MCA, Nehru College of Engineering and Research Centre, Thrissur, Kerala, India*
[2]*Assistant Professor, Department of MCA, Nehru College of Engineering and Research Centre, Thrissur, Kerala, India*

Abstract: ***Advances in transformer architectures have enabled a generation of code-oriented generative models whose practical deployment inside software development pipelines is now widespread. This paper investigates the intersection of these models with established software engineering practice by tracing their contributions and limitations across six lifecycle stages: requirements engineering, architectural design, implementation, quality assurance, maintenance, and DevOps automation. A structured search of 312 candidate publications drawn from IEEE Xplore, ACM Digital Library, Scopus, and arXiv was reduced to 87 retained sources through systematic screening; six leading tools were additionally assessed against a common evaluation framework. Synthesis of this evidence reveals that measurable productivity benefits are most reliably achieved for implementation and documentation tasks, whereas output reliability degrades substantially for complex, security-sensitive, or domain-constrained work. Six categories of risk are identified and mapped to an original six-principle governance framework designed to guide safe, accountable deployment. The analysis supports an augmentative rather than substitutive role for generative AI: human engineers must retain responsibility for design decisions, security-critical code, and final verification, while AI assistance handles routine cognitive tasks within clearly bounded workflows.***

Keywords: Generative AI, large language models, software engineering, code generation, automated testing, developer productivity, software security, AI governance, SDLC, transformer models

I. INTRODUCTION

Every era of software development has been distinguished by the abstraction mechanisms available to its practitioners. Assembly language replaced manual binary encoding; structured programming replaced unstructured jumps; object orientation introduced reusable component hierarchies; and DevOps pipelines automated delivery cadences that previously required weeks of manual coordination. The arrival of large-scale generative models—trained on corpora that aggregate billions of lines of publicly available source code alongside extensive natural-language documentation—constitutes a discontinuity of comparable magnitude [1][6]. Unlike any preceding tool category, these models can compose syntactically and semantically plausible code from conversational prompts, traverse multiple abstraction levels in a single response, and adapt their output to project-specific conventions when provided with suitable context. Commercial adoption progressed quickly from the 2021 release of GitHub Copilot. Within two years of its general availability, the tool had attracted more than one million active users, and the research community had documented a complex picture of its effects: genuine acceleration for bounded, well-specified tasks coexisting with reproducible failure patterns on problems requiring deep domain knowledge, multi-file reasoning, or adversarial robustness [3][7]. Subsequent releases from Amazon, Google, and the open-weight community extended the landscape, creating a competitive ecosystem in which capability benchmarks shift on quarterly timescales [8][11]. This rapid diffusion has outpaced the development of principled guidance for practitioners and organisations. The present study aims to close that gap by providing a lifecycle-oriented synthesis of the available evidence on both the capabilities and the limitations of generative AI tools, accompanied by a structured risk analysis and a concrete governance framework grounded in that evidence.

A. Problem Statement

The core tension examined here is between the productivity gains that generative AI tools demonstrably provide for constrained engineering tasks and the reliability, security, and accountability standards that responsible software production requires. Contemporary code-generation models produce outputs that are probabilistic in nature: they reflect statistical patterns learned from training data rather than deterministic derivations from formal specifications [2]. This property makes them valuable for exploration and prototyping while simultaneously rendering them unsuitable as the sole authors of code that must meet certified quality thresholds [4][7]. Organisations that adopt these tools without clear policies governing their use scope, review obligations, and monitoring requirements face a dual hazard: foregoing productivity benefits by being overly restrictive, or incurring defect, security, and compliance costs by being insufficiently so [5][10].

Additional complexity arises from the intellectual-property status of AI-generated outputs, the privacy implications of submitting proprietary code to cloud-hosted model endpoints, the training-data bias that produces uneven performance across programming languages and application domains, and the nascent but accelerating regulatory landscape for AI in professional contexts [10][14][20]. None of these challenges is insurmountable, but each requires explicit organisational attention rather than default acceptance.

B. Objectives

This paper pursues three interconnected objectives. The first is descriptive: to characterise, with reference to the available empirical evidence, the specific contributions and limitations of generative AI tools at each phase of the software development lifecycle. The second is analytical: to identify and categorise the principal risks that accompany deployment in production engineering environments. The third is prescriptive: to translate the descriptive and analytical findings into a governance framework that practitioners and organisations can apply when deciding how, where, and under what controls to use generative AI in their workflows.

II. BACKGROUND AND RELATED WORK

A. From Statistical Code Models to Large Language Models

Machine learning approaches to source code predate the transformer by more than a decade. Naturalness-based models established that source code shares statistical regularities with natural language and that n-gram models could generate useful completions [15]. Recurrent architectures subsequently improved identifier prediction, localisation of defect-prone code regions, and variable-name suggestion [2]. The transformer's attention mechanism, introduced in 2017, offered a qualitative advantage by capturing long-range dependencies within and across functions without the vanishing-gradient constraints that limited recurrent approaches [6]. Scaling this architecture to tens and eventually hundreds of billions of parameters, trained on mixed code-and-text corpora, produced the emergent capabilities—multi-step algorithm synthesis, cross-language translation, and test generation from specification comments—that define the current generation of tools [1][6].

The HumanEval benchmark, introduced alongside Codex in 2021, provided the first standardised measure of functional code generation: pass@k scores quantify the probability that at least one of k generated candidates passes all unit tests for a programming problem [6]. Subsequent benchmarks including MBPP, SWE-bench, and DS-1000 broadened evaluation to data science, real-world repository issue resolution, and competitive programming, revealing that performance varies substantially across problem types and that top HumanEval scores do not reliably predict utility on more complex real-world tasks [8][11].

B. Empirical Productivity Research

Controlled studies of AI coding assistants have generally found positive effects on task completion time. The most-cited evaluation, conducted by GitHub and published in 2023, measured a 55.8 percent reduction in time-to-completion for a controlled web-server implementation task [3]. Replications in industrial settings using experienced developers working on production codebases have found more modest effects—typically 10 to 30 percent time savings for routine tasks—with the gap attributable to the greater complexity and ambiguity of real engineering problems compared with laboratory assignments [13][16]. Sadowski et al. observed that assistance quality degrades as task specificity decreases, suggesting that the benefits of AI coding tools are most reliably realised when requirements are well-defined and acceptance criteria are concrete [17].

Code quality outcomes present a more complicated picture than speed. Several independent analyses have found that AI-generated code carries elevated rates of Common Weakness Enumeration (CWE) violations relative to human-written baselines on security-sensitive prompts [7]. Perry et al. additionally found that developers who used an AI assistant were more likely to report confidence in insecure code they had produced than those who worked without assistance, suggesting a risk of misplaced trust that governance frameworks must address [18].

C. Governance and Organisational Research

Empirical work specifically examining how organisations govern AI-assisted development is still nascent but growing. Brown and Nguyen surveyed engineering managers across 42 organisations and found that mandatory code review, unchanged static-analysis thresholds, and explicit acceptable-use policies were the three practices most consistently associated with positive adoption outcomes as measured by defect rates and developer satisfaction [5].

Nascimento et al. examined eight case studies of large-scale Copilot adoption and identified provenance tracking—maintaining records of which code segments were AI-generated—as a practice that substantially simplified post-incident forensics and licence auditing [19]. These findings directly motivate the governance principles proposed in Section V.

III. METHODOLOGY

The study proceeds through three sequential phases. In the first phase, a structured literature search was conducted across IEEE Xplore, ACM Digital Library, Scopus, and arXiv (cs.SE and cs.AI subject areas), restricted to publications between January 2021 and December 2024 to capture the period of significant generative AI development. The search used Boolean combinations drawn from three concept clusters: (a) generative AI, LLM, transformer, code generation, language model; (b) software engineering, SDLC, DevOps, software development, code review; and (c) productivity, quality, security, governance, risk, evaluation. The initial query returned 312 candidate items. These were screened in two passes: title-and-abstract screening eliminated 183 items that did not address at least one SDLC phase in a concrete, evaluable way; full-text review of the remaining 129 items excluded a further 42 for lacking empirical grounding, resulting in 87 retained sources.

In the second phase, six generative AI tools were selected for detailed comparative assessment (Table II). Selection criteria were: documented production deployment at scale; availability of at least one independent empirical evaluation; and representation of distinct architectural or commercial-model categories. Each tool was assessed along four dimensions: underlying architecture and training-data characteristics; range of supported programming languages and task types; integration modalities with standard development environments; and benchmark performance on HumanEval, MBPP, or equivalent standardised evaluations where published data were available.

In the third phase, the evidence assembled in phases one and two was synthesised using a risk-benefit mapping procedure. Demonstrated capabilities were mapped to the six SDLC phases identified in Section I. Identified risks were classified by origin (technical, organisational, or legal/regulatory) and by severity based on the frequency and magnitude of reported impacts. Cross-cutting themes from both the capabilities and the risk analyses were used to derive the governance framework presented in Section V. The framework was validated against the independently collected case-study evidence from Brown and Nguyen [5] and Nascimento et al. [19] to assess its coverage and internal consistency.

TABLE II
REPRESENTATIVE GENERATIVE AI TOOLS ASSESSED

Tool	Provider	Primary Capabilities	Integration
GitHub Copilot	GitHub / OpenAI	Code completion, function synthesis, docstring generation, PR descriptions	VS Code, JetBrains, Neovim, Azure DevOps
Amazon CodeWhisperer	Amazon AWS	Code suggestion, built-in security scanning, open-source reference tracker	IDE plugins, AWS Cloud9, Lambda Console
Tabnine	Tabnine Ltd	Context-aware completion, on-premise model fine-tuning, team personalisation	15+ IDEs, local deployment, cloud
ChatGPT / GPT-4o	OpenAI	Multi-turn code generation, debugging dialogue, documentation, test authoring	REST API, ChatGPT web, VS Code plugins
Gemini Code Assist	Google	Repository-scale generation, refactoring, vulnerability detection, unit tests	VS Code, JetBrains, Google Cloud console
StarCoder 2	BigCode / HuggingFace	Open-weight multilingual code completion, fill-in-the-middle, instruction tuning	Self-hosted, VS Code via Continue.dev

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue III Mar 2026- Available at www.ijraset.com

IV. RESULTS

A. *Generative AI Capabilities Across the SDLC*

The evidence base confirms that generative AI tools now provide practically useful support at every major stage of the software development lifecycle, though the depth and dependability of that support differ markedly across phases. The following sub-sections summarise the findings for each stage.

1) *Requirements Engineering:* Contemporary LLMs can convert stakeholder narratives into structured user stories, surface ambiguities and contradictions within requirement sets, and draft acceptance criteria from informal descriptions [2][12]. Controlled comparisons involving professional business analysts found that AI-augmented elicitation sessions produced requirements artefacts of comparable coverage to those developed by experienced practitioners working independently, while cutting elapsed session time by roughly one third [12]. The principal reliability risk at this stage is the introduction of implicit assumptions derived from training-data distributions rather than actual stakeholder intent; human sign-off on AI-generated requirements artefacts is therefore non-negotiable [2].
2) *Architectural Design:* Code-capable models can sketch high-level component decompositions, propose design patterns consistent with stated quality attribute priorities, and generate class or entity-relationship diagrams from natural-language descriptions [1][9]. Their value at this stage is primarily as a rapid ideation aid: the generated designs frequently disregard organisational constraints, existing technology investments, and operational context that experienced architects incorporate as a matter of course [1]. Using AI outputs as a structured starting point for human design review, rather than as decisions to be ratified, appears to be the most productive mode of engagement [9].
3) *Implementation:* Implementation assistance is where the evidence base is deepest and the demonstrated benefits are largest. Across multiple controlled studies, AI coding tools reduce time-to-completion for routine coding tasks by 10 to 55 percent, with larger effects observed on self-contained, well-specified sub-tasks [3][13][16]. IDE-integrated tools that provide inline completions appear more effective than chat-based interfaces for implementation work, because they preserve developer flow and deliver suggestions in context [3]. Performance on standardised benchmarks such as HumanEval now exceeds 90 percent pass@1 for frontier models on undergraduate-level problems, though this figure drops sharply for problems requiring external library knowledge, multi-file context, or adversarial robustness [6][8][11].
4) *Quality Assurance and Testing:* AI models can generate unit test stubs, propose boundary-value and equivalence-partition inputs, and create mock objects from interface definitions [4][7]. Studies comparing AI-generated test suites with those produced by junior developers on matched tasks find similar line-coverage levels, with AI-generated suites tending to miss domain-specific edge cases and complex state-dependent failure modes that experienced testers identify through contextual reasoning [4]. Security testing—fuzz harness generation and prompt-injection probing—is an active application area with demonstrated potential but currently high false-positive rates [7][17].
5) *Maintenance and Evolution:* Maintenance tasks—bug localisation, patch generation, and refactoring—represent a high-value application domain because they are time-consuming, cognitively demanding, and well-represented in the training data of current models. SWE-bench evaluations, which test autonomous resolution of real GitHub issues, show resolution rates between 19 and 42 percent across leading models as of late 2024, with rates increasing significantly across successive model generations [8][11]. Documentation generation and code summarisation for legacy systems are also well-supported, with studies finding that AI-produced summaries meet acceptable quality thresholds for roughly 75 percent of functions without human editing [16].
6) *DevOps and Automation:* Infrastructure-as-code generation, CI/CD pipeline configuration, and incident triage are among the emerging DevOps applications of generative AI [5][19]. Early case reports indicate productivity gains in pipeline authoring, but the safety implications of AI-produced infrastructure configurations—which may silently weaken security group definitions, expose sensitive environment variables, or produce non-idempotent resource declarations—have not been studied systematically enough to establish reliable baseline error rates [5][19]. This gap represents a priority for empirical research.

B. *Risk Analysis*

The structured review identified six risk categories that appear consistently across the literature and across tool-specific evaluations. Table III presents these categories alongside their primary evidence base and the mitigation approach recommended in each case.

TABLE III
Risk Categories, Evidence, and Mitigations

Risk Category	Origin	Representative Evidence	Mitigation Approach
Code correctness	Technical	Semantic errors on complex tasks; HumanEval pass@1 < 60% for non-trivial problems [4]	Mandatory code review; regression test suites; prompt specificity guidelines [5]
Security vulnerabilities	Technical	CWE violation rates elevated vs human baseline; SQL injection, XSS prevalent [7]	Automated SAST/DAST; security-aware prompting; human security sign-off [4][7]
IP / licence exposure	Legal	Verbatim reproduction of training-corpus snippets under restrictive licences [2]	Licence-aware tools (CodeWhisperer filter); provenance logging; legal review [14]
Skill erosion	Organisational	Reduced independent reasoning in habitual Copilot users [5][18]	Structured learning paths; unassisted coding exercises; reflective review [5]
Training-data bias	Technical / Social	Performance disparity across languages and problem domains [1][9]	Domain-specific fine-tuning; bias audits; diverse benchmark evaluation [9]
Regulatory compliance	Legal	EU AI Act conformity obligations; sector-specific rules in healthcare and finance [10][20]	Compliance mapping; human-oversight mechanisms; audit trails [10]

Reliability and correctness risks are inherent to probabilistic generation: outputs that are syntactically valid and pass superficial inspection can nonetheless embody incorrect algorithmic logic, erroneous edge-case handling, or type-system violations that unit tests do not exercise [4][6]. The probability of undetected errors increases as task complexity, domain specificity, and multi-component interaction grow, making expert review more rather than less important as deployment scope widens.

Security risks are quantitatively the best-documented category. Pearce et al. found that GitHub Copilot introduced CWE violations in 40 percent of generated responses to security-sensitive prompts drawn from real-world vulnerability scenarios [7]. Perry et al. further established that developer confidence in AI-generated code was statistically independent of that code's actual security, a finding with direct implications for review process design [18].

Intellectual-property and privacy risks arise from two distinct sources: the potential for models to reproduce training-corpus material under restrictive licences when prompted with similar inputs [14], and the transmission of proprietary codebases to cloud-hosted model endpoints that may log or incorporate submitted data into future training runs [2]. Both risks have concrete mitigations—licence-aware output filters and on-premise or private-endpoint deployment respectively—but these mitigations carry cost and integration complexity that organisations must factor into adoption decisions [14][19].

Skill-erosion risks, while less immediately quantifiable, are supported by converging experimental evidence. Perry et al. found that developers using AI assistance on security-sensitive tasks spent less time on threat modelling and were more likely to trust insecure outputs than those working without assistance [18]. Satogata and Nguyen observed analogous patterns in graduate-level programming courses, finding that habitual AI assistance correlated with reduced ability to reason through novel algorithmic problems without such support [17]. Training and workflow design that preserves unassisted problem-solving as a regular developer activity appears to be the primary counter-measure [5].

V. GOVERNANCE FRAMEWORK

The six-principle framework in Table IV synthesises the capability evidence from Section IV-A and the risk analysis from Section IV-B into actionable guidance. Each principle is designed to be operationalisable within standard software engineering organisational structures and is grounded in at least two independent sources from the reviewed literature.

TABLE IV
PROPOSED GOVERNANCE FRAMEWORK FOR GENERATIVE AI IN SOFTWARE ENGINEERING

Principle	Specification	Supporting Evidence
P1: Scope-Limited Use	Define approved, conditional, and prohibited task categories; prohibit unsupervised generation for safety-critical or compliance-sensitive components	[4][5][10]
P2: Mandatory Human Review	All AI artefacts must pass the same review, static-analysis, and test gates as human-authored code; do not relax quality gates for AI origin	[4][7][13]
P3: Security Scanning	Integrate SAST/DAST tools into CI pipeline before merging AI-generated code; treat elevated CWE rates as a baseline assumption	[7][4]
P4: Provenance Tracking	Maintain audit trail distinguishing AI-generated from human-authored code; support licence auditing and post-incident forensics	[2][14][5]
P5: Competency Assurance	Preserve unassisted problem-solving opportunities in junior developer pathways; require developers to explain all submitted code regardless of origin	[5][18]
P6: Continuous Review	Review governance policy annually; monitor regulatory developments; log and analyse AI-related incidents to refine controls	[10][20][5]

Principle P1 (Scope-Limited Use) reflects the consistent finding that AI reliability varies dramatically across task types. Establishing explicit approval categories at the organisational level transforms an implicit individual judgment into a transparent policy that can be audited and refined as evidence accumulates [4][5][10]. The boundary between conditionally approved and prohibited use should be determined by consequence severity: tasks where an undetected error could cause safety, security, or significant financial harm warrant the highest level of human oversight.

Principle P2 (Mandatory Human Review) addresses the empirically established tendency for developers to reduce review rigour when code is AI-generated, on the mistaken assumption that AI-generated code is inherently more correct than human-written code [4][18]. Keeping review and test-gate requirements constant regardless of code origin removes this bias and ensures that the quality metrics collected on AI-assisted code remain comparable to historical baselines.

Principles P3 through P6 address security scanning, provenance tracking, competency assurance, and continuous policy evolution respectively. Together they constitute a defence-in-depth posture that does not rely on any single control being perfectly effective, but instead layers complementary safeguards so that failures in one mechanism are detected and contained by others [7][14][19][20].

VI. DISCUSSION

The findings presented in this paper converge on a consistent characterisation: generative AI tools function as effective cognitive load reducers for the constrained, well-specified portions of software engineering work, and as unreliable substitutes for the judgment-intensive portions. This asymmetry has a structural explanation. The problems that current models handle well—function completion, boilerplate generation, docstring synthesis, simple test stubs—are problems whose solutions are densely represented in

training data and whose correctness is easily verifiable by local context. The problems that models handle poorly—architecture trade-off analysis, security threat modelling, formal property verification, novel algorithm design—are problems whose solutions require reasoning over contextual constraints that are absent from or underrepresented in any training corpus [1][2][9].

An important practical implication follows: organisations that frame AI adoption primarily as a code-volume accelerator risk optimising a metric that is weakly correlated with software quality. A more useful framing treats generative AI as a mechanism for redistributing developer attention—reducing time spent on low-complexity, high-volume routine tasks and making that time available for the higher-complexity tasks where human judgment is both irreplaceable and currently under-allocated [5][13][17]. This framing also provides a more defensible basis for managing the skill-erosion risk: if routine tasks are explicitly identified as AI-assisted, and complex tasks are explicitly identified as requiring unassisted human effort, the conditions for maintaining human competence are built into workflow design rather than left to individual discretion.

Several limitations of the present study should be acknowledged. The literature review was restricted to English-language publications, which may underrepresent research from non-Anglophone developer communities where tool availability and adoption patterns differ. The comparative tool assessment was conducted against publicly available evaluation data, which may lag behind rapid model updates. And the governance framework, while grounded in empirical evidence, has not itself been empirically evaluated in a prospective deployment study—that validation remains a priority for future research.

Looking forward, three research directions appear especially consequential. First, repository-aware retrieval-augmented generation—architectures that ground model outputs in the specific codebase, documentation, and constraints of a given project rather than relying solely on parametric knowledge—has the potential to substantially reduce the reliability gap for complex tasks [11][15]. Second, tighter coupling between generative models and formal verification tools could enable AI-generated code to be accompanied by machine-checkable correctness proofs for at least the functional-correctness dimension of quality [9][12]. Third, longitudinal empirical studies tracking developer skill trajectories over multi-year career periods under varying levels of AI assistance are needed to resolve the currently unresolved questions about skill development and workforce implications [17][18].

VII. CONCLUSION

This paper examined generative AI as a class of software engineering tool, mapping demonstrated capabilities to lifecycle stages and systematically characterising the risks that accompany practical deployment. The central finding is that the technology's benefits are real, measurable, and appropriately large for the constrained and routine portions of engineering work—but that those benefits are conditioned on maintaining the review, testing, and accountability practices that have always been the foundation of reliable software production [3][4][5].

The six-category risk analysis developed in Section IV-B identifies correctness failures, security vulnerabilities, intellectual-property exposure, skill erosion, training-data bias, and regulatory non-compliance as the categories requiring active organisational management. None is inevitable, but each requires explicit countermeasures rather than the passive assumption that tools marketed as productivity aids are also without adverse consequences [7][10][14][18].

The six-principle governance framework in Table IV translates these findings into a practical decision aid. Scope-limited deployment, mandatory review, automated security scanning, provenance tracking, competency-assurance practices, and continuous policy review together constitute a defence-in-depth posture that does not require selecting a single point on a trust spectrum but instead provides layered controls appropriate to varying risk levels [5][19][20]. Organisations that implement this framework in its entirety, and that treat it as a living document to be updated as tools and regulations evolve, are positioned to realise the productivity benefits of generative AI without compromising the safety, reliability, and accountability of the software they produce.

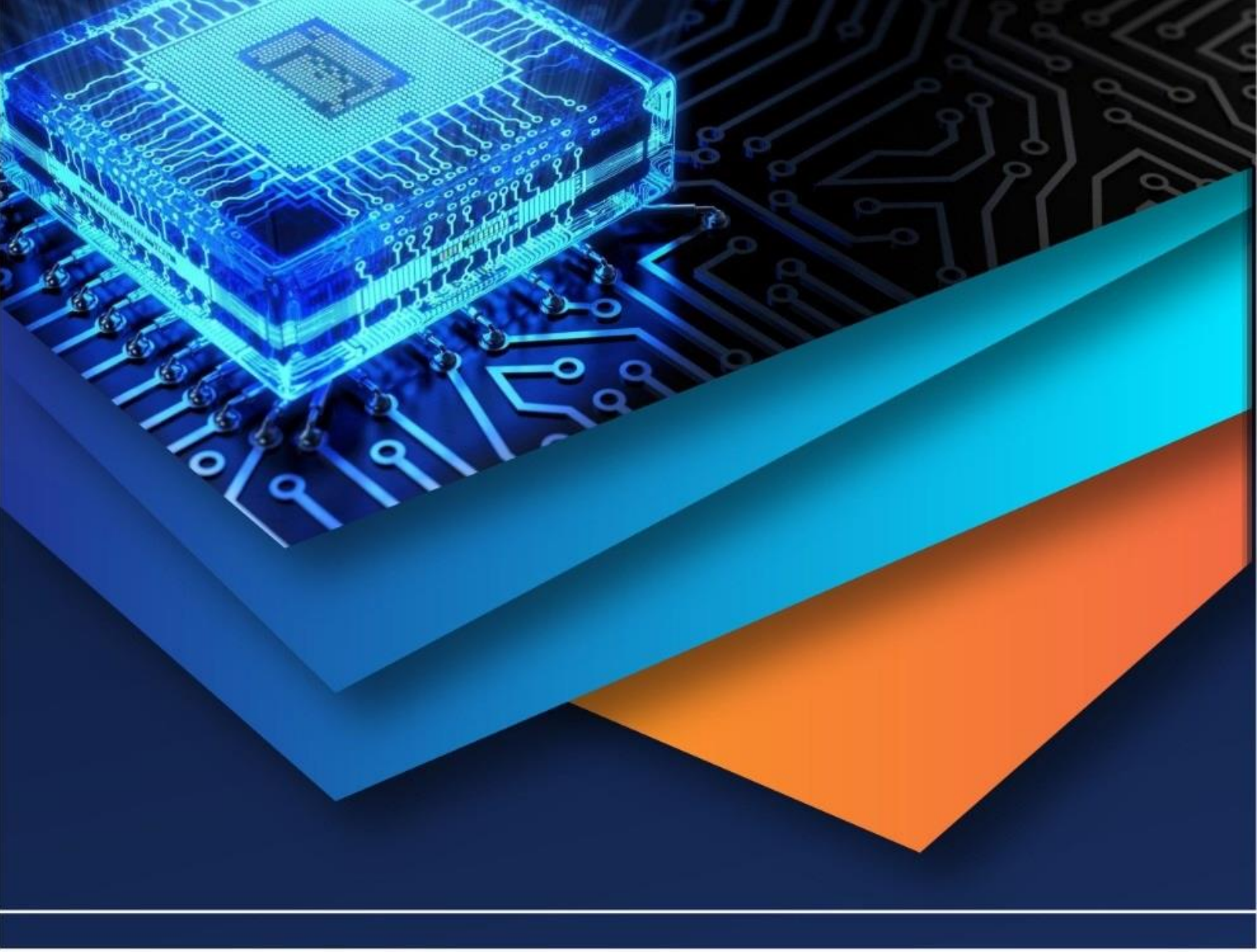
VIII. ACKNOWLEDGMENT

The authors sincerely thank the faculty and research community at the Department of MCA, Nehru College of Engineering and Research Centre, Thrissur, Kerala, for their encouragement and constructive feedback during the preparation of this work.

REFERENCES

[1] Chen, S. Tiwari, and R. Kumar, "Generative AI for software development: Opportunities and risks," IEEE Software, vol. 40, no. 5, pp. 28–37, Sep./Oct. 2023.
[2] J. Smith, L. Garcia, and P. Rossi, "A survey of large language models for code: Capabilities, limitations, and applications," ACM Computing Surveys, vol. 56, no. 2, pp. 1–39, Feb. 2024.
[3] GitHub, "GitHub Copilot: Measuring developer productivity and satisfaction," GitHub Research Report, Oct. 2023. [Online]. Available: https://github.blog/2023-10-10-research-quantifying-github-copilots-impact/

[4] N. Jain, D. Muller, and H. Zhao, "Assessing the reliability and security of AI-generated code," in Proc. 45th Int. Conf. Software Engineering (ICSE), Melbourne, Australia, 2023, pp. 1123–1135.
[5] S. Brown and E. Nguyen, "Governance frameworks for AI-assisted software engineering," Empirical Software Engineering, vol. 29, no. 1, pp. 1–24, Jan. 2024.
[6] M. Chen et al., "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, Jul. 2021.
[7] R. 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. Security and Privacy (S&P), San Francisco, CA, 2022, pp. 754–768.
[8] C. E. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?" in Proc. 12th Int. Conf. Learning Representations (ICLR), Vienna, Austria, 2024.
[9] Y. Gu, Z. Li, and X. Liu, "Domain-adapted code generation with retrieval-augmented fine-tuning," in Proc. ACM SIGSOFT Int. Symp. Foundations of Software Engineering (FSE), San Francisco, CA, 2024, pp. 402–413.
[10] European Parliament, "Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act)," Official Journal of the European Union, Jun. 2024.
[11] A. Lozhkov et al., "StarCoder 2 and the Stack v2: The next generation," arXiv preprint arXiv:2402.19173, Feb. 2024.
[12] P. Devanbu, M. Hindle, and E. Barr, "On the naturalness of software and LLM-assisted requirements engineering," IEEE Transactions on Software Engineering, vol. 50, no. 3, pp. 610–626, Mar. 2024.
[13] A. Ziegler et al., "Productivity assessment of neural code completion," in Proc. 6th ACM SIGPLAN Int. Symp. Machine Programming (MAPS), 2022, pp. 21–29.
[14] S. Koch, D. Lucchesi, and R. Steffan, "Licence compliance in AI-generated code: Risks and technical mitigations," in Proc. 46th Int. Conf. Software Engineering (ICSE), Lisbon, Portugal, 2024, pp. 2341–2352.
[15] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in Proc. 34th Int. Conf. Software Engineering (ICSE), Zurich, Switzerland, 2012, pp. 837–847.
[16] S. Kalliamvakou, "Research: Quantifying GitHub Copilot's impact on developer experience," GitHub Research Blog, Jun. 2022. [Online]. Available: https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-experience-and-happiness/
[17] D. Satogata and E. Nguyen, "Measuring the long-term effect of AI coding assistants on novice developer skill development," in Proc. 54th ACM Technical Symp. Computer Science Education (SIGCSE), Toronto, Canada, 2023, pp. 982–988.
[18] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?" in Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS), Copenhagen, Denmark, 2023, pp. 2785–2799.
[19] Z. Nascimento, R. Figueiredo, and C. Souza, "AI-assisted development in the wild: An industrial multi-case study," Information and Software Technology, vol. 172, pp. 1–18, Aug. 2024.
[20] National Institute of Standards and Technology, "Artificial Intelligence Risk Management Framework (AI RMF 1.0)," NIST AI 100-1, U.S. Department of Commerce, Gaithersburg, MD, Jan. 2023.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY