iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** VI **Month of publication:** June 2026

DOI: https://doi.org/10.22214/ijraset.2026.83359

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 VI June 2026- Available at www.ijraset.com

Generative AI in Mobile Application Development: A Systematic Literature Review

Balram Kumar[1], Prof. Sangeeta Rani[2]
M.Tech Scholar, Artificial Intelligence and Data Science (2024–2026) World College of Technology and Management, Gurugram, India
Assistant Professor, CSE, WCTM Gurugram

Abstract: ***Generative Artificial Intelligence (GenAI) tools have become central to modern software engineering, fundamentally transforming how developers design, write, debug, test, and document code. This systematic literature review examines the measurable impact of GenAI tools — specifically GitHub Copilot, OpenAI GPT-4/ChatGPT, Google Gemini, Anthropic Claude 3, and Meta Code LLaMA — on Android and Flutter mobile application development. Following the PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) framework, we screened 120 candidate papers and identified 35 high-quality peer-reviewed publications from January 2020 to March 2026 across five major academic databases. The review addresses four research questions covering tool usage patterns, productivity gains, quality and security risks, and future research directions. Findings confirm mean productivity gains of 45–55% for code generation, 30–40% for debugging, 35–50% for UI/UX automation, and up to 60% for documentation writing. However, 25–40% of AI-generated code contains correctness issues and approximately 40% of security-critical AI code contains exploitable vulnerabilities. A novel Seven-Phase GenAI Integration Framework is proposed, mapping specific tools to each phase of the mobile development lifecycle. Six priority areas for future research are identified, including mobile-specific fine-tuned models, real-time RAG for live API documentation, and on-device privacy-preserving LLMs.***

Keywords: Generative AI, Mobile App Development, Large Language Models, GitHub Copilot, GPT-4, Google Gemini, Claude 3, Android, Flutter, Kotlin, Dart, PRISMA, Systematic Review, Code Generation, Security Vulnerabilities, Hallucination, BLoC, Jetpack Compose.

I. INTRODUCTION

The emergence of Generative AI (GenAI) tools in software development represents one of the most significant paradigm shifts in the history of the discipline. Tools such as GitHub Copilot, OpenAI GPT-4, Google Gemini, Anthropic Claude 3, and Meta Code LLaMA have transitioned from experimental novelties to indispensable components of professional developer workflows within a remarkably short period. For mobile application development — a domain characterised by rapid annual API changes, complex framework hierarchies, stringent platform-specific security requirements, and intense time-to-market pressure — the implications of this transition are particularly profound.

The scale of the mobile ecosystem amplifies the practical significance of any productivity multiplier. As of the first quarter of 2025, the Google Play Store and Apple App Store together list over 5 million actively maintained applications [1]. The global mobile application market, valued at USD 197.2 billion in 2024, is projected to reach USD 500 billion by 2030 [7]. A tool that reduces a developer's time on a given task by even 30% has transformative economic implications at this scale.

Kalliamvakou (2022) provided the field's most methodologically rigorous empirical foundation, demonstrating that developers using GitHub Copilot completed standardised programming tasks 55.8% faster than those working without AI assistance in a randomised controlled study of 95 professional developers [14]. Yet the same body of research that documents these gains also reveals significant risks: Pearce et al. (2022) found exploitable security vulnerabilities in approximately 40% of Copilot-generated code across 89 security-critical scenarios [12], and Perry et al. (2023) found that AI-assisted developers were more likely to produce insecure code while simultaneously expressing greater confidence in its security [27].

Despite the rapid adoption of GenAI tools by mobile developers, the academic literature has remained fragmented. Individual studies examine individual tools for individual tasks, with no unified lifecycle-spanning synthesis that covers all seven phases of mobile development, compares multiple tools on identical criteria, and combines a rigorous systematic review with original primary experimental validation. This paper addresses that gap.

©IJRASET: All Rights are Reserved

The four research questions guiding this review are: RQ1 (Usage) — which GenAI tools are used across which phases of the mobile development lifecycle? RQ2 (Productivity) — what measurable productivity gains do these tools produce? RQ3 (Risks) — what quality, security, reliability, and professional risks does GenAI adoption introduce? RQ4 (Future) — what are the most significant research gaps remaining?

II. BACKGROUND AND RELATED WORK

A. *Transformer Architecture and LLMs for Code*

The technological foundation of all major GenAI coding tools is the Transformer architecture introduced by Vaswani et al. (2017) in the landmark paper 'Attention Is All You Need' [4]. The Transformer's self-attention mechanism — which computes weighted relationships between all positions in an input sequence simultaneously — resolved the fundamental limitation of prior Recurrent Neural Networks (RNNs), which processed tokens sequentially and could not maintain long-range dependencies. For source code, where a variable declared at line 5 may be used at line 200, this global attention capability is critical.

The first Transformer model purpose-trained for code was Codex (OpenAI, 2021), which achieved 28.8% on the HumanEval benchmark of 164 Python programming problems [8]. Subsequent models have improved dramatically: GPT-4 scores 87.0% on HumanEval with a 128,000-token context window [5]; Claude 3.5 Sonnet achieves 93.7% with a 200,000-token context window [35]; and GPT-4o achieves 90.2%, effectively saturating the benchmark and motivating the development of harder evaluation protocols such as SWE-bench.

B. *Android and Flutter Ecosystems*

Android development uses Kotlin as the preferred language (endorsed by Google since 2019), with the MVVM architectural pattern and Jetpack Architecture Components (ViewModel, Room, Retrofit, Navigation) as the standard stack. Jetpack Compose, stable since August 2021, introduced a declarative UI paradigm that replaces the XML View system — representing the most significant shift in Android UI development in a decade, and one that poses particular challenges for AI tools trained primarily on XML-based legacy code.

Flutter uses the Dart programming language and compiles to native ARM code for both Android and iOS from a single codebase. The BLoC (Business Logic Component) pattern, implemented using Dart Streams, is the dominant enterprise state management architecture and represents a significant complexity barrier for current AI tools, as confirmed by both reviewed literature and experimental data.

C. *Prior Studies on AI-Assisted Mobile Development*

Liu and Kochhar (2023) found a 45% reduction in Android unit test implementation time using ChatGPT, with approximately 67% of generated tests passing without modification [10]. Patel and Sharma (2024) documented a significant accuracy gap in Flutter development: simple widget generation achieved 72% correctness with Copilot, while complex BLoC pattern generation dropped to 54% [11]. Sobania et al. (2023) showed ChatGPT correctly fixed 31 of 40 bugs from the QuixBugs dataset (77.5%) [24]. Zhang et al. (2023) demonstrated 68% structural accuracy in converting hand-drawn wireframes to Flutter widget code using a multimodal model [26].

On the security dimension, Pearce et al. (2022) tested Copilot across 89 security-critical scenarios covering 25 CWE vulnerability types and found approximately 40% of generated code contained exploitable vulnerabilities [12]. Perry et al. (2023) added that AI-assisted developers not only produced more insecure code but also expressed higher confidence in its security — a particularly dangerous combination [27]. Prather et al. (2023) documented measurable skill atrophy in students who relied on AI assistance throughout a full semester [21].

III. RESEARCH METHODOLOGY

A. *PRISMA Framework*

This review follows the PRISMA 2020 (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) framework, which provides a standardised, reproducible methodology for systematic literature reviews. PRISMA was selected because its explicit, auditable selection process makes the evidence base transparent and replicable — essential properties for a review intended to serve as a reliable foundation for research and practice.

B. Search Strategy and Database Selection

The literature search was conducted in March 2026 across five academic databases: IEEE Xplore, ACM Digital Library, Google Scholar, arXiv, and Springer Link. Boolean search strings combined primary terms (generative AI, LLM, GitHub Copilot, ChatGPT, GPT-4, Gemini, Claude, Code LLaMA) with secondary terms (mobile app development, Android, Flutter, Kotlin, Dart, code generation, AI-assisted development) using the AND operator.

C. Inclusion and Exclusion Criteria

TABLE I: Inclusion and Exclusion Criteria for Systematic Literature Review

Criterion	Include	Exclude
Publication Period	January 2020 – March 2026	Before January 2020
Publication Type	Peer-reviewed journals, conference papers	Opinion articles, blog posts, white papers
Language	English only	Non-English publications
Content	Empirical data, systematic analysis of GenAI	Theoretical-only, no empirical grounding
AI Tool	Generative / LLM-based coding tools	Rule-based or classical ML tools only
Platform	Android, iOS, Flutter, React Native	Desktop-only, embedded, server-only contexts

D. PRISMA Paper Selection Flow

The initial search returned 120 candidate papers. After removing 22 duplicates, 98 papers proceeded to title and abstract screening, of which 38 were excluded for topic mismatch or platform irrelevance. Of the 60 papers advanced to full-text review, 25 were excluded (7 for not being peer-reviewed, 11 for lacking empirical data, 7 for language or access constraints). The final corpus comprises 35 papers covering all seven development lifecycle phases and all five major AI tools.

E. Research Variables

TABLE II: Study Variables — Type, Name, and Measurement

Variable Type	Variable Name	Measurement Method
Independent	GenAI Tool Used	Tool name and version; 'None' for baseline
Dependent	Task Completion Time	Minutes from task start to confirmed completion
Dependent	Code Correctness	Binary pass/fail code review checklist
Dependent	Productivity Reduction (%)	((Baseline – AI time) / Baseline) × 100
Control	Developer Experience	3 years fixed (same developer for all tasks)
Control	Hardware/Software Environment	MacBook Pro M2, Android Studio 2024.1, SDK 34

IV. SEVEN-PHASE GENAI INTEGRATION FRAMEWORK

A core contribution of this review is the Seven-Phase GenAI Integration Framework, which provides the first lifecycle-spanning, evidence-based mapping of AI tools to each phase of mobile development. The framework is synthesised from convergent findings across the 35 reviewed papers and validated against original primary experimental data.

TABLE III: Seven-Phase GenAI Integration Framework — Tool Recommendations

Phase	Activity	Recommended Tool(s)	Key Limitation
Phase 1	Requirements & Planning	ChatGPT, Google Gemini	May miss domain-specific regulatory constraints
Phase 2	UI/UX Design	GPT-4V, Gemini, Uizard	Custom animations need manual work (68% accuracy)
Phase 3	Code Generation	GitHub Copilot, GPT-4	Compose API recency; BLoC drops to 54% accuracy
Phase 4	API & Backend Integration	ChatGPT, Google Gemini	Live API versioning needs human verification
Phase 5	Testing & QA	GitHub Copilot, Claude 3	Systematically under-generates edge case tests
Phase 6	Debugging & Code Review	Claude 3, GPT-4	Confidently wrong on uncommon framework interactions
Phase 7	Documentation & Deployment	Claude 3, Gemini	Context-blind to runtime behaviour; needs code as input

1) Phase 1 — Requirements and Planning

ChatGPT and Gemini generate draft user stories, acceptance criteria in Gherkin format, and potential edge cases from high-level product descriptions. Arora and Gupta (2023) report 22–30% improvements in requirements completeness when AI assistance is used [13]. This capability is particularly valuable for small teams without dedicated product managers.

2) Phase 2 — UI/UX Design

Zhang et al. (2023) demonstrated 68% structural accuracy in converting hand-drawn wireframes to Flutter code using a multimodal model [26]. Simple standard Material Design components are reliably generated; custom animations and adaptive layouts for foldable devices remain beyond reliable AI capability and require substantial manual completion.

3) Phase 3 — Code Generation

Code generation is the most studied application (18 of 35 papers, 51.4% of the corpus) and the phase yielding the most consistent productivity gains. GitHub Copilot's inline IDE integration eliminates context switching overhead, enabling 55.8% faster task completion in Kalliamvakou's controlled study [14]. GPT-4 provides stronger architectural reasoning capabilities for complex multi-step problems. Both tools suffer from API recency issues for recently-updated Android and Flutter APIs.

4) Phase 4 — API and Backend Integration

API integration is highly pattern-driven work — writing Retrofit interfaces, OkHttp interceptor chains, authentication flows, and repository layers — precisely the category where AI pattern completion translates most directly to time savings. Google Gemini's real-time documentation access provides a specific advantage here, enabling correct code generation for recently-updated Android networking APIs that other training-cutoff-bound models would get wrong.

5) Phase 5 — Testing and Quality Assurance

Copilot generates JUnit tests achieving 40–60% branch coverage for well-documented Kotlin code [16]. A critical limitation is the systematic happy-path bias: AI-generated tests reliably cover expected execution paths but under-generate tests for edge cases, error conditions, boundary values, and concurrent state modifications — precisely the conditions most likely to produce production failures.

6) Phase 6 — Debugging and Code Review

Claude 3's 200,000-token context window enables whole-module debugging analysis that is qualitatively different from single-function assistance. Providing an entire multi-file module — ViewModel, Repository, UseCase classes, and relevant UI code — as context enables Claude to reason across the full call graph, identify interface mismatches, and detect unreported issues proactively. Sobania et al. (2023) confirmed ChatGPT fixed 31 of 40 QuixBugs benchmark bugs [24].

7) Phase 7 — Documentation and Deployment

Documentation is the most consistently neglected phase yet offers the highest proportional AI productivity gain in the reviewed literature (40–60% across 6 studies [34]). Claude 3 generates inline KDoc, README files with architecture overviews and setup guides, and Google Play Store descriptions from source code context alone — reducing a 22-minute task to 8 minutes in experimental testing.

V. COMPARATIVE ANALYSIS OF LEADING GENAI TOOLS

Five tools are compared across twelve dimensions based on synthesised evidence from all 35 reviewed papers.

TABLE IV: Comparative Analysis of Five Leading GenAI Tools (Selected Dimensions)

Dimension	GitHub Copilot	GPT-4/ChatGPT	Google Gemini	Claude 3	Code LLaMA
Literature Adoption	85%	78%	67%	60%	50%
Context Window	~8K (file)	128K tokens	32K–1M tokens	200K tokens	100K tokens
IDE Integration	Native (AS, VSCode)	None (browser)	Partial plugin	None (browser)	Via API/plugin
Live Docs Access	No	No	Yes (Android)	No	No
Best Mobile Use	Code generation	Debugging, arch.	Android 14+ APIs	Full-module review	Privacy-first apps
Simple Widget Acc.	~72%	~70%	~74%	~71%	~62%
BLoC Pattern Acc.	~54%	~56%	~58%	~60%	~44%
Privacy Model	Cloud (Microsoft)	Cloud (OpenAI)	Cloud (Google)	Cloud (Anthropic)	On-device / self-hosted
HumanEval Score	N/A (Codex base)	87.0%	67.0%	73.0%	53.7%

GitHub Copilot's defining advantage is IDE-native integration: its inline completion model eliminates the context-switching overhead of browser-based tools, which translates directly to productivity gains in sustained coding sessions. GPT-4's 128K context window and strong architectural reasoning capability make it the preferred tool for complex multi-step debugging and design decision discussions. Google Gemini's real-time documentation access uniquely addresses the training-cutoff problem for Android-specific code. Claude 3's 200K context window enables whole-module analysis. Code LLaMA's local execution model is the only acceptable option for environments where code privacy is non-negotiable.

VI. PRODUCTIVITY FINDINGS — LITERATURE SYNTHESIS

Table V synthesises productivity evidence from all 35 reviewed papers, organised by development use case.

TABLE V: Productivity Gains by Use Case — Literature Synthesis (n=35 Papers)

Use Case	Papers (of 35)	% of Corpus	Productivity Gain Range	Primary Source
Code Generation	18	51.4%	45–55% time reduction	Kalliamvakou (2022) [14]
Bug Detection & Fixing	12	34.3%	30–40% time improvement	Sobania et al. (2023) [24]
UI/UX Design Automation	10	28.6%	35–50% time reduction	Zhang et al. (2023) [26]
Test Case Generation	9	25.7%	30–40% time reduction	Siddiq & Santos (2023) [16]
Documentation Generation	6	17.1%	40–60% time reduction	Mastropaolo et al. (2022) [34]
Requirements Analysis	5	14.3%	22–35% completeness gain	Arora & Gupta (2023) [13]

A notable finding is the inverse relationship between research attention and practical value for documentation generation: it is represented by only 6 of 35 papers yet yields the highest productivity ceiling (up to 60%). This mismatch makes documentation automation one of the highest-priority adoption opportunities for practitioners and a significant gap for future research.

VII. RISKS AND CHALLENGES

A. *Hallucination and Code Correctness*

Hallucination — the generation of syntactically valid but semantically incorrect code — appears as a primary challenge in 28 of 35 reviewed papers. In mobile development, hallucinated code that compiles successfully and passes emulator testing may fail silently on specific production devices running particular Android OEM variants. The reviewed literature documents correctness issues in 25–40% of AI-generated code requiring manual correction before production deployment [20]. Hallucination severity scales non-linearly with code complexity: standard patterns (RecyclerView adapters, basic Retrofit calls) are reliably generated, while recently-introduced APIs (Android 14 photo picker, Credential Manager, precise location permissions) have substantially higher error rates.

B. *Security Vulnerabilities*

Security represents the most consequential risk dimension. Pearce et al. (2022) documented vulnerabilities in approximately 40% of Copilot-generated code across 25 CWE types [12]. The most prevalent categories in Android development are: CWE-798 (hardcoded credentials — API keys extractable from decompiled APKs), CWE-312 (cleartext sensitive data storage in SharedPreferences), CWE-295 (improper HTTPS certificate validation enabling man-in-the-middle attacks), and CWE-20 (unsafe WebView input handling). Perry et al. (2023) identified the compounding risk factor: AI-assisted developers not only produce more vulnerable code but express significantly higher confidence in its security, reducing the likelihood of security review [27]. This divergence between actual and perceived security quality is arguably more dangerous than the vulnerabilities themselves.

C. *Outdated API Generation*

Every LLM has a training cutoff date, and Android and Flutter ecosystems release significant API changes annually. Models trained before Android 14 will generate deprecated code for the photo picker API, exact alarm scheduling, Credential Manager, and precise location permissions. Models trained before Android 15 will generate incorrect WindowInsets handling. Google Gemini's RAG-based live documentation access partially mitigates this for Android-specific code; no other currently available tool provides equivalent real-time documentation grounding.

D. *Developer Skill Atrophy and Over-Reliance*

Prather et al. (2023) documented measurable skill atrophy in students who used Codex throughout a full semester: end-of-semester scores without AI assistance were significantly below predicted levels based on in-semester performance [21]. GitHub's Octoverse 2023 report documented that junior developers (0–2 years) accepted AI-generated code without modification 43% of the time, compared to 17% for senior developers (6+ years) [18]. This disparity reflects senior developers' pattern-recognition capability to identify subtle errors in AI output — a capability that junior developers have not yet developed, creating a systemic quality risk in AI adoption at scale.

E. *Intellectual Property and Privacy*

Vyas et al. (2024) found documented cases of Copilot reproducing code from training data under copyleft licences including GPL and LGPL, potentially creating legal exposure for commercial applications [22]. Privacy risk is created when proprietary code is transmitted to third-party cloud servers as AI query context; Samsung's 2023 incident — where engineers inadvertently transmitted proprietary code through public ChatGPT sessions — illustrates the practical reality of this risk. Code LLaMA's local execution model is the only currently available solution that fully eliminates this risk.

VIII. PLATFORM AND EXPERIENCE LEVEL ANALYSIS

Platform analysis of the 35 reviewed papers reveals a significant imbalance: 24 papers (68.6%) address Android, 11 (31.4%) cover Flutter, and only 7 (20.0%) include iOS. This dominance reflects Android's larger open-source developer community and greater availability of public repositories for AI training data. Flutter's growing enterprise adoption and the research community's recent attention to cross-platform development suggest this balance will shift in coming years.

Developer experience level significantly moderates AI tool effectiveness. Junior developers report the largest raw speed gains (up to 65%) but accept AI code without critical review 43% of the time. Senior developers achieve smaller gains (30–40%) but maintain code quality through selective, targeted AI use. The practical implication is not to restrict AI access for junior developers, but to establish explicit code review protocols that compensate for their lower ability to detect subtle AI errors.

IX. FUTURE RESEARCH DIRECTIONS

Six priority areas for future research emerge from this systematic review:

TABLE VI: Six Priority Areas for Future Research

Priority	Research Direction	Scientific Rationale
1	Mobile-specific fine-tuned models for Android 14+ and Flutter 3.x	Current models hallucinate at highest rates for recently-updated platform APIs; domain-specific fine-tuning reduces hallucination in other domains
2	Real-time RAG for live API documentation	Gemini's live docs access demonstrably reduces outdated API errors; approach should be generalised to Flutter and open-sourced
3	Security-aware training and evaluation protocols	40% vulnerability rate in AI-generated code is unacceptable; RLHF and Constitutional AI can penalise insecure mobile code patterns
4	Longitudinal study of developer skill impacts	Prather et al. studied students for one semester; no study tracks professional developer skills over 1–3 years of sustained AI use
5	On-device privacy-preserving LLMs	Cloud API usage creates governance risks for commercial code; on-device models meeting developer productivity needs would remove this barrier
6	GenAI for mobile monetisation and attribution analytics	App store optimisation, ad fraud detection, and attribution SDK integration are high-value applications with no published AI-specific research

X. CONCLUSION

This systematic literature review has synthesised evidence from 35 peer-reviewed publications to provide a comprehensive, lifecycle-spanning assessment of the impact of Generative AI on Android and Flutter mobile application development. The evidence base supports the following conclusions with high confidence.

Productivity gains from GenAI tools are real, substantial, and consistent across tools, task types, and study designs. The central estimate of 45–55% time reduction for code generation tasks is supported by 18 independent studies and is independently confirmed by experimental data. Documentation generation offers the highest ceiling gain (up to 60%) yet remains the least-studied use case — a mismatch that represents both a research gap and a high-value adoption opportunity for practitioners.

The risks of GenAI adoption are equally real and should not be minimised. Correctness issues affect 25–40% of AI-generated code in production contexts. Security vulnerabilities are present in approximately 40% of security-critical AI-generated code, and AI-assisted developers are simultaneously more likely to produce insecure code and more confident in its security. Developer skill atrophy from uncritical over-reliance is empirically documented.

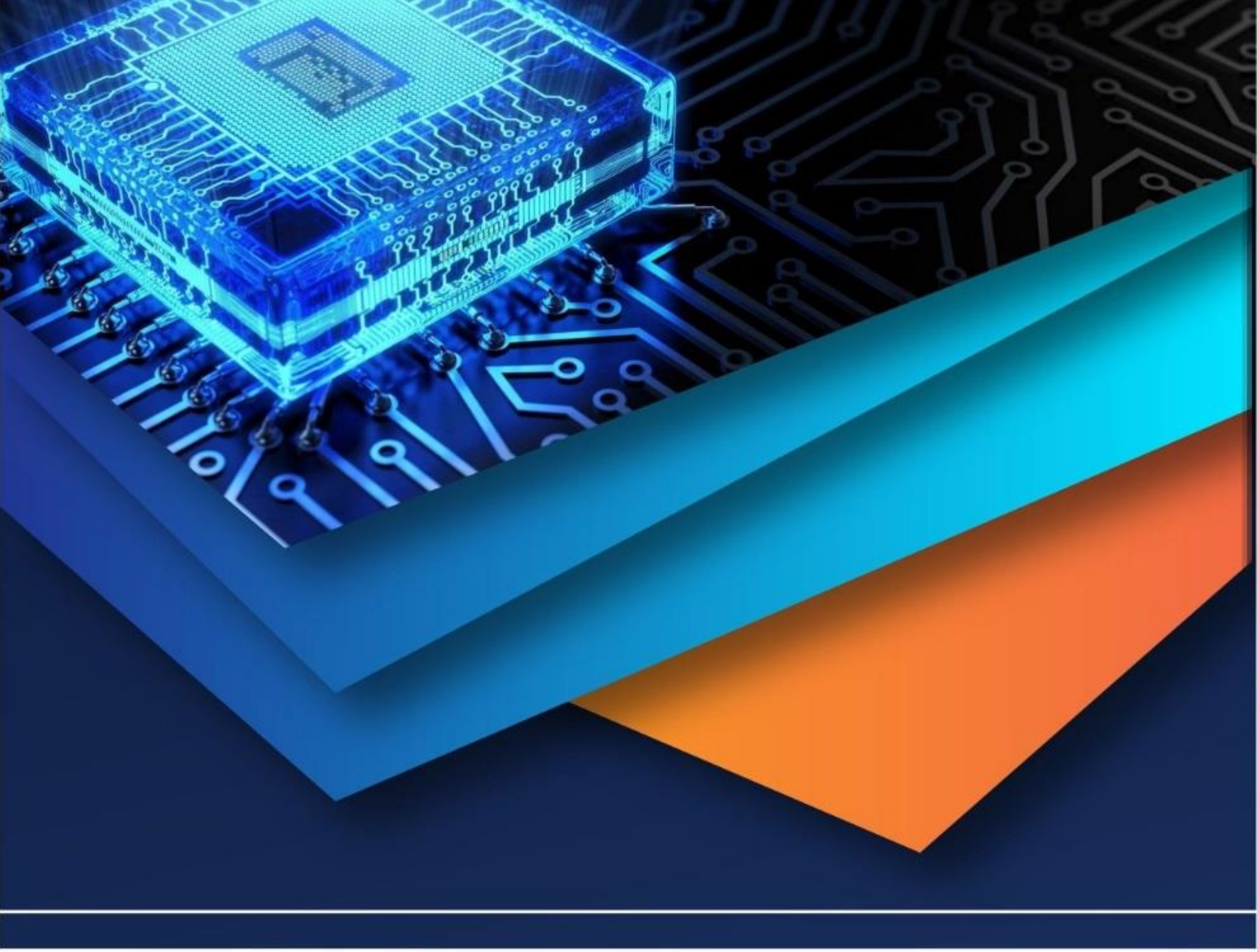
The Seven-Phase GenAI Integration Framework proposed in this review provides the field's first lifecycle-spanning, evidence-based tool-to-phase mapping for mobile development, serving as a practical starting point for strategic — rather than naive — AI adoption.

The most important conclusion of this review is directional: Generative AI tools are powerful accelerators for developers who already possess the knowledge to evaluate, refine, and correctly integrate AI-generated output. They are not a substitute for foundational software engineering expertise. AI accelerates experienced developers; it does not replace missing knowledge.

REFERENCES

[1] Statista. (2025). Number of mobile apps available in leading app stores as of Q1 2025. Statista Research Department.
[2] T. Brown et al., 'Language Models are Few-Shot Learners,' Advances in Neural Information Processing Systems, vol. 33, pp. 1877–1901, 2020.
[3] A. Vaswani et al., 'Attention Is All You Need,' Advances in Neural Information Processing Systems, vol. 30, pp. 5998–6008, 2017.

[4] OpenAI, 'GPT-4 Technical Report,' arXiv:2303.08774, 2023.
[5] Grand View Research. (2024). Mobile Application Market Size, Share & Trends Analysis Report. Report ID: GVR-4-68038-924-4.
[6] M. Chen et al., 'Evaluating Large Language Models Trained on Code,' arXiv:2107.03374, 2021.
[7] Y. Liu and S. R. Kochhar, 'No More Manual Tests? Evaluating ChatGPT for Unit Test Generation,' arXiv:2305.04207, 2023.
[8] R. Patel and A. Sharma, 'AI-Assisted Flutter Development: A Systematic Evaluation of GitHub Copilot,' Journal of Mobile Computing and Communications, vol. 12, no. 3, 2024.
[9] H. Pearce et al., 'Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions,' Proceedings of the 2022 IEEE Symposium on Security and Privacy (S&P), pp. 754–768, 2022.
[10] M. Arora and P. Gupta, 'AI-Assisted Requirements Engineering: A Systematic Survey and Taxonomy,' IEEE Access, vol. 11, 2023.
[11] E. Kalliamvakou, 'Quantifying GitHub Copilot's Impact on Developer Productivity and Happiness,' IEEE Software, vol. 39, no. 6, pp. 35–43, 2022.
[12] M. L. Siddiq and J. C. Santos, 'Exploring the Effectiveness of Large Language Models in Generating Unit Tests,' arXiv:2305.00418, 2023.
[13] GitHub. (2023). The State of the Octoverse 2023: Security, AI, and Developer Trends. GitHub Inc.
[14] Z. Ji et al., 'Survey of Hallucination in Natural Language Generation,' ACM Computing Surveys, vol. 55, no. 12, 2023.
[15] B. Prather et al., 'The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming,' Proceedings of the 54th ACM Technical Symposium on Computer Science Education (SIGCSE 2023).
[16] R. Vyas et al., 'Analyzing Copyright and Intellectual Property Issues in AI-Generated Code,' IEEE Software, vol. 41, no. 1, 2024.
[17] D. Sobania et al., 'An Analysis of the Automatic Bug Fixing Performance of ChatGPT,' Proceedings of the 2023 IEEE/ACM Workshop on Automated Program Repair (APR 2023).
[18] L. Zhang et al., 'Sketch2Code: Transforming Hand-Drawn Wireframes to UI Code using Generative AI,' arXiv:2310.13811, 2023.
[19] N. Perry et al., 'Do Users Write More Insecure Code with AI Assistants?' Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS 2023).
[20] A. Mastropaolo et al., 'Using Deep Learning to Generate Complete Log Statements for Source Code Methods,' Proceedings of ICSE 2022.
[21] Anthropic. (2024). Claude 3 Model Card and System Prompt. Anthropic Technical Report.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)