Universal Library of Innovative Research and Studies Research Article

ISSN: 3065-0003 | Volume 3, Issue 1

Open Access | PP: 65-74

DOI: https://doi.org/10.70315/uloap.ulirs.2026.0301009

Accelerating Mobile Application Development and Testing with Artificial Intelligence: A Systematic Literature Review

Maksym Yurko

Delivery Director, Mobile CoE, Brentwood, USA.

Abstract

The article presents a systematic literature review examining how artificial intelligence methods (LLM/GenAI, computer vision, deep learning, and multi-agent architectures) accelerate mobile application development and testing within the mobile SDLC. The objective is to address a deficit of domain-specific systematisation for mobile engineering and to answer three classes of questions: which AI approaches are applied across development and QA stages, which acceleration and efficiency metrics are empirically substantiated, and which quality/security risks accompany the adoption of generative tools. The relevance is driven by the growing complexity of mobile ecosystems and the limited scalability of manual testing and script-based automation, particularly due to brittleness under GUI changes. The novelty of the review lies in synthesising evidence from 28 selected studies from 2019 to 2025, with an explicit focus on mobile-specific constraints, and in juxtaposing speed gains against a trust contour. The principal findings are as follows: AI assistants demonstrate a substantial acceleration of routine tasks (up to ~55%) and a reduction in timelines for large-scale migrations. In testing, a transition toward LLM agents is observed, enabling high coverage at both scenario and element levels, as well as resilience to UI evolution. Concurrently, a trust crisis is documented due to a significant share of vulnerabilities in generated code, dependency hallucinations, and increased technical debt, necessitating the institutionalisation of verification practices and secure-by-default principles. The article is intended to be beneficial to software engineering researchers and to mobile development/QA practitioners integrating GenAI into workflows.

Keywords: *Mobile Development, Generative AI, Large Language Models, PRISMA.*

INTRODUCTION

Over the past decade, the mobile app ecosystem has evolved from the simple development of native apps to advanced ecosystems based on cloud computing, the Internet of Things, and embedded AI. Analysts expect revenue from AI-enabled mobile apps to increase to $26.3 billion by 2030, growing at a compound annual growth rate (CAGR) of 38.7% over this period [1]. An exponential increase in complexity accompanies this growth: developers must maintain functional parity between Android and iOS platforms, adapt interfaces to hundreds of screen resolutions, and ensure data security under continuously evolving threats [2].

Conventional development and quality assurance (QA) methodologies are approaching the limits of their scalability. Manual testing, which still constitutes a substantial share of processes in the mobile industry, is labour-intensive, slow, and susceptible to human error [3]. Script-based automated testing (e.g., using Appium or Espresso) suffers from the brittleness problem: minor changes to the graphical user interface (GUI) can cause test failures, requiring significant resources for maintenance and updates [4]. In development, a shortage of qualified personnel exacerbates technical debt, compelling teams to automate routine cognitive tasks.

The emergence and rapid development of large language models (LLMs) and generative artificial intelligence (GenAI) in 2022–2025 marked the onset of a new era in software engineering. Tools such as GitHub Copilot, Amazon CodeWhisperer, and specialised testing agents provide not only automation of repetitive actions but also partial autonomy in decision-making.

Unlike traditional static code analysers or record-and-replay tools, contemporary AI systems exhibit the capacity for semantic contextual understanding [5]. They can also be used to generate syntactically correct code from natural-language descriptions (Text-to-Code), recognise and understand screenshots, including interface components, as a human would (Vision-based Testing), and autonomously plan and execute complex user workflows, including robustness to unexpected states and behaviours of the tested application.

Citation: Maksym Yurko, "Accelerating Mobile Application Development and Testing with Artificial Intelligence: A Systematic Literature Review", Universal Library of Innovative Research and Studies, 2026; 3(1): 65-74. DOI: https://doi.org/10.70315/uloap.ulirs.2026.0301009.

It is expected to solve the mobile development trilemma of speed, quality, and cost simultaneously. However, practice indicates that AI adoption is not risk-free. The phenomena of hallucinations and the production of vulnerable code introduce new challenges for the industry, requiring rigorous scientific analysis [6].

Despite the active growth of publications on AI applications in software engineering, a clear deficit remains in systematic studies focused specifically on the mobile domain. Most existing reviews address either general issues of LLM adoption in SE or narrow aspects of GUI testing [2]. Mobile development specificity, device resource constraints, touch interaction characteristics, stringent UX requirements, and security constraints necessitate dedicated analysis.

The present article aims to fill this gap by providing a comprehensive systematic literature review (SLR) that consolidates fragmented empirical evidence on acceleration metrics, the effectiveness of new testing architectures, and real-world security risks arising from the use of AI in the mobile SDLC.

To conduct this review in an organised manner and achieve its objectives, the following research questions were posed:

RQ1 (Taxonomy and Architecture): Which artificial intelligence approaches, models, also architectures (LLM, CV, RL) do they use in the different steps during mobile app development and test?

RQ2 (Effectiveness Metrics): In the literature, for use to determine how effective AI is at improving development speed, test coverage, defect detection, and team productivity, what quantitative empirical measures are reported?

RQ3 (Quality and Risks): What specific quality issues appear for reliability or maintainability, and what security issues surface for vulnerabilities or hallucinations when someone integrates AI-generated code and tests in the mobile ecosystem?

MATERIALS AND METHODS

This study follows the PRISMA 2020 guidelines, a systematic checklist, which ensures that the methods for selecting and analysing the literature are transparent, reproducible, and rigorous.

The scope and inclusion criteria were guided by the PICO framework, as detailed in Table 1.

Table 1. PICO criteria

Component	Description and Detail
Population	Mobile applications (Native Android/iOS, cross-platform Flutter/React Native), mobile development ecosystems, and QA processes in the mobile domain.
Intervention	Artificial intelligence methods: large language models (LLMs), generative AI (GenAI), computer vision, deep learning, reinforcement learning (RL), and multi-agent systems.
Comparison	Traditional methods: manual coding, manual testing, scripted automation (Appium, Selenium, Espresso), heuristic algorithms (Monkey testing), and existing non-AI practices.
Outcome	Quantitative and qualitative metrics: reduced development time (time reduction), increased code coverage, bug detection rate, code quality, and security vulnerabilities.

The search for primary studies was conducted in the following electronic databases from January 1, 2019, to 2025, to capture the latest advances in generative AI: IEEE Xplore, ACM Digital Library, Scopus, MDPI, as well as preprint repositories, given the rapid pace of industrial evolution. The search query was constructed using logical operators and adapted to the syntax of each database. The baseline query structure was:

(mobile application OR Android OR iOS OR app testing OR mobile development) AND (artificial intelligence OR AI OR machine learning OR deep learning OR LLM OR Large Language Model OR generative AI OR GitHub Copilot OR ChatGPT) AND (testing OR development OR acceleration OR productivity OR GUI testing OR test generation OR code generation)

Strict criteria were applied to filter results:

- **IC1:** Studies published in English in 2019–2025.
- **IC2:** Works containing empirical data (experiments, case studies, practitioner surveys) on AI use in mobile development or testing.
- **IC3:** Articles proposing new AI-agent architectures or methodologies (e.g., ScenGen, AutoQALLMs).
- **IC4:** Systematic literature reviews and mapping studies on adjacent topics for contextualization.
- The exclusion criteria were as follows:
- **EC1:** Articles unrelated to the mobile domain (web-only or desktop-only), except foundational code-generation works applicable across domains.
- **EC2:** Marketing materials and white papers without methodological descriptions, short abstracts (less than 3 pages).
- **EC3:** Duplicate texts (e.g., preprint and subsequently published journal article, only the latest version retained).
- **EC4:** Studies in which an AI (e.g., recommender system)

is embedded into an application (the AI is an application feature, not a development/testing tool)

Identification and full-text screening stages of the selection process are detailed and illustrated in Figure 1.

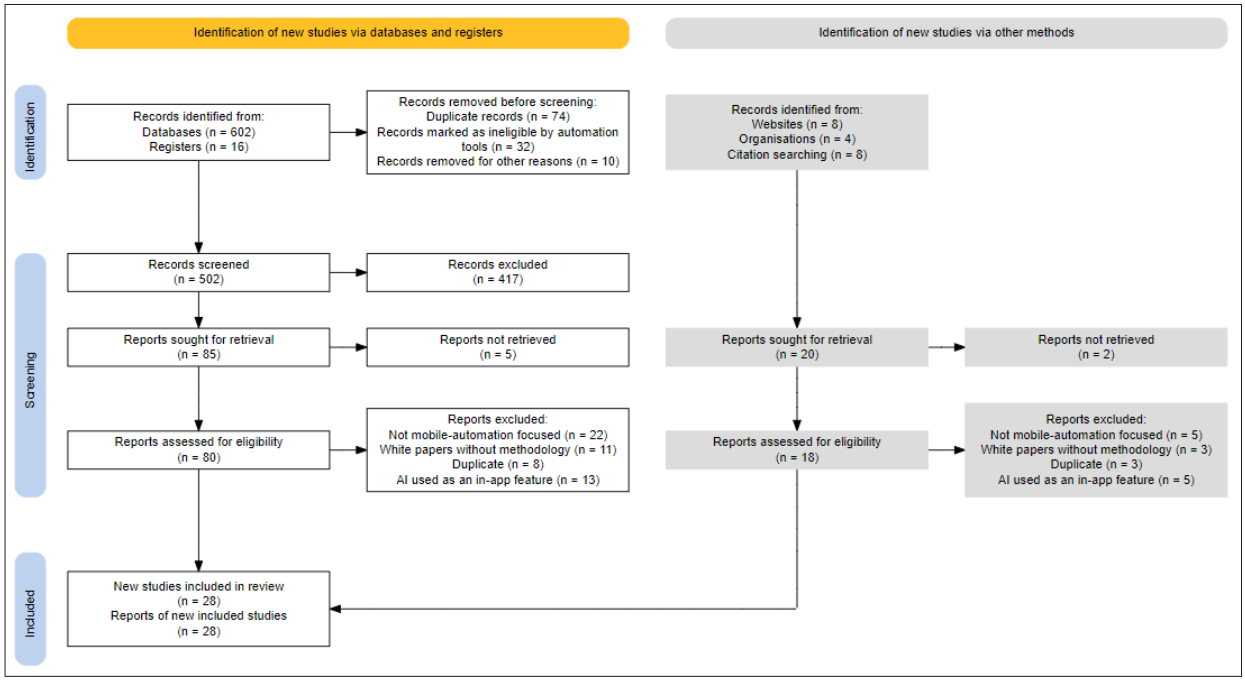

Figure 1. PRISMA Flow Diagram

To assess the credibility of selected studies, an adapted checklist based on the recommendations in [7] was used. The checklist included the following criteria:

- **QA1 (Goals):** Are the study goals and research questions clearly formulated?
- **QA2 (Context):** Is the context adequately described (AI models used, programming language versions, test applications)?
- **QA3 (Design):** Is the design appropriate to address RQs (e.g., sample size, control group)?

Each study was rated on a 0-1 scale (0 = does not meet, 0.5 = partially meets, 1 = fully meets) in 5 categories, and a score of 3 out of 5 was used as the cut-off to ensure high evidentiary quality of the included studies.

RESULTS

The analysis of 28 selected studies revealed fundamental changes in approaches to developing mobile software. Results are grouped in accordance with the research questions.

RQ1: Taxonomy and Architecture of AI in the mobile SDLC

The contemporary landscape of AI tools in mobile development can be classified along two dimensions: task type (Code Generation vs. Testing) and agent architecture (Reactive vs. Cognitive).

Within the category of generative assistants, large language models (LLMs) trained on massive code corpora dominate. The principal paradigm is Text-to-Code. Table 2 provides a comparative analysis of code generation tools for mobile platforms. Table 2 illustrates a comparative analysis of code generation tools for mobile platforms.

Table 2. Comparative analysis of code generation tools for mobile platforms

Tool	Base Model	Core Capabilities	Mobile-Specific Notes (Findings)	Source
GitHub Copilot	OpenAI Codex / GPT-4	Line completion, function generation from comments, and chat interface.	High effectiveness for Kotlin/Java and React Native. Lower accuracy for Swift/SwiftUI due to Apple's closed ecosystem and a smaller volume of open-source training data.	[8]
ChatGPT (OpenAI)	GPT-3.5 / GPT-4o	Solving algorithmic tasks, documentation generation, refactoring, and code explanation.	Higher code correctness (65.2%) vs. Copilot (46.3%) on isolated tasks, but requires context switching (copy-paste).	[9]

Amazon CodeWhisperer	Proprietary LLM	Security-oriented, AWS integration.	Shows lower technical debt in generated code compared with competitors.	[9]
AlphaCode / Gemini	DeepMind models	Solving complex logic tasks.	Strong potential for generating complex algorithmic structures, but currently less integrated into IDE workflows for mobile development.	[10]

In testing, an evolution is observed from simple Monkey testers (random taps) to sophisticated multi-agent systems. According to the taxonomy proposed in [11], three generations of agents are distinguished.

The first generation of AI approaches in mobile GUI testing and automation is typically described as single-agent systems. Such solutions include, for example, Humanoid and GPTDroid. Their architectural concept is that a single model, generally built on deep learning methods or early versions of large language models, receives the current GUI state as input, in the form of a screenshot or an XML representation, and predicts exactly one following action. However, this scheme has fundamental limitations: the agent lacks a stable memory of previous steps, thereby performing poorly in multi-step scenarios such as adding an item to the cart and paying, and exhibiting a tendency to loop, repeatedly executing similar actions without progressing toward the goal.

The second generation is associated with multi-agent systems in which cognitive functions are partitioned among specialised agents. Examples include AutoQALLMs [12] and ScenGen [13]; the multi-agent paradigm is regarded as the most promising in the current literature. In the ScenGen architecture, the central role is played by the Observer, responsible for interface perception. It utilises multimodal models, such as GPT-4V, to transform the application's visual state into a semantic description. Importantly, this concerns not pixel recognition per se, but the construction of an interpretation at the level of this is a login screen; the button is inactive until a password is entered, which reduces the system's domain knowledge dependency [13]. Based on this description, the Decider performs planning: it aligns the current state provided by the Observer with the test goal, i.e., the test scenario, and applies a chain of reasoning (Chain-of-Thought) to select the next step. The Executor then translates a high-level command, such as press the 'Log in' button, into low-level driver code, i.e., a concrete action implemented via coordinates or element identifiers. The Supervisor performs result validation: if the screen does not change after the button press or an error occurs, it informs the Decider that the plan must be adjusted [13]. Finally, the Recorder maintains an action log in contextual memory, enabling the system to adapt within a session through in-context learning.

The third generation is commonly associated with plan-then-act architectures, where an agent first constructs a complete, high-level plan to achieve its goal and then executes it, dynamically correcting steps as deviations occur. AppAgent can be cited as an example. The key rationale is that this approach reduces the number of calls to the large language model at each step, thereby lowering token consumption and accelerating execution, as a substantial portion of reasoning is shifted to the pre-planning phase, making execution more communication-efficient with the model.

Beyond test generation, AI methods are applied to narrow but critical automation tasks that directly affect test-run stability and the operational efficiency of testing processes.

PopSweeper is a computer-vision-based tool designed to detect and close blocking pop-up windows such as advertising overlays, app-rating prompts, and system permission dialogues [14]. The solution is based on an object detection approach that identifies closing controls, for example, an X button or close. Detection accuracy is reported to reach 91.7%, and the tool thereby eliminates one of the most common causes of instability in automated tests, where unexpected pop-ups interrupt the scenario and lead to false failures.

EncrePrior is a system oriented toward prioritising bug reports originating from crowdsourced testing [15]. It analyses screenshots associated with defects, clusters them by visual similarity, and extracts unique issues, reducing the share of repetitive or duplicate reports. As a result, triage efficiency increases; the original description reports a 15.6% improvement in efficiency.

RQ2: Acceleration and Effectiveness Metrics

Empirical data collected in 2023–2025 studies enable a quantitative characterisation of the effect of AI adoption, including acceleration of development and code migration. The influence of AI assistants on coding speed is documented in both controlled experiments and industrial cases, yielding conclusions that are practically comparable.

According to GitHub research, developers using Copilot complete typical tasks, such as creating an HTTP server, 55% faster [16]. The same study notes an overall acceleration in product delivery, with the Time-to-Market metric improving by 30% [16]. A reduction in routine work is additionally emphasised: AI can generate up to 30–40% of code volume for experienced developers [16]. The most pronounced effect is observed where template-driven activity predominates, including generation of boilerplate code, tests, and documentation; in these categories, acceleration reaches 50% [17].

A separate highlight is the Airbnb case associated with code migration [18], which is frequently cited as a demonstration of the effectiveness of LLMs in modernising an existing

codebase. In this example, 3500 test files were migrated from the Enzyme framework to React Testing Library. The traditional engineering effort estimate was 1.5 years, whereas the AI-based solution relied on an LLM pipeline and iterative prompt refinement. The task was completed in 6 weeks, with 97% of files migrated automatically and only 3% requiring manual intervention [18]. This case illustrates AI's potential not only for accelerating greenfield development but also for large-scale transformation of existing software.

AI transforms testing from script checking into behaviour exploration. Table 3 shows the evolution from random and brittle testing (Monkey and Appium scripts) toward more intelligent approaches (DL and LLM) that improve coverage and the ability to discover unique crashes: Stoat yields +17–31% over baseline coverage and 3× more unique failures, while LLM solutions (ScenGen, AutoQALLMs) provide high scenario/element coverage and better detect logical errors. At the same time, maximum robustness to UI changes is achieved by LLM approaches due to semantic adaptation and test repair mechanisms (e.g., RegEx + LLM), whereas scripted scenarios remain most vulnerable. Table 3 illustrates the comparative effectiveness of testing methods.

Table 3. Comparative effectiveness of testing methods

Method / Tool	Code Coverage	Unique Crash Detection	Robustness to UI Changes	Source
Monkey (Random)	Baseline level	Low (gets stuck in simple loops)	High (independent of UI structure)	[19]
Scripted (Appium)	High (for predefined scenarios)	Low (only expected failures)	Low (fragile)	[13]
Deep Learning (Stoat)	+17–31% vs. baseline	3× more than Monkey/Sapienz	Medium	[19]
LLM-Based (ScenGen)	High scenario coverage	High (logical errors)	Very high (semantic adaptation)	[13]
AutoQALLMs	96% of UI elements	Comparable to manual (98%)	High (RegEx + LLM repair)	[12]

Scenario accuracy is one of the key indicators for evaluating whether an AI system can correctly execute logical test scenarios, i.e., make decisions based on interface state and the target test intent. In study [11], ScenGen demonstrated high Logical Decision-Making Accuracy across several applications with differing interface complexity. For the Music application, characterised by a simple interface, accuracy was 100%, and an analogous 100% result was obtained on the Photo application. For the Email application, where scenarios are complicated by forms, attachments, and a larger number of contextual states, accuracy was 97.44%. On the Note application, featuring hidden menus and less obvious transitions, the metric decreased to 90.14% [11]. Collectively, these values indicate that multi-agent systems can address tasks requiring the interpretation of business logic and contextual conditions, approaching a level of scenario understanding that was previously only practical through manual testing.

At the same time, empirical evidence indicates that AI effectiveness is uneven depending on the development platform [10]. For Android, where Kotlin and Java are the primary languages, high generation accuracy is reported, attributed to a larger volume of training data and publicly available examples. For cross-platform stacks such as Flutter and React Native, very high effectiveness is documented, since the declarative nature of UI code, especially in Flutter, aligns well with how LLMs structure output code. It is additionally stated that frame times and overall performance of Flutter applications generated with AI remain competitive relative to native solutions [20]. For iOS, where Swift and SwiftUI are used, reduced accuracy is described: large language models more frequently hallucinate by suggesting nonexistent SwiftUI modifiers or using outdated syntax, thereby increasing the need for careful review and amplifying the role of iOS developers' oversight [10].

RQ3: Quality Issues and Security Risks

Against the backdrop of improved speed metrics, the literature analysis documents a concerning shift in the quality and security of AI-generated code. This concerns not merely isolated implementation defects, but a systemic problem appropriately denoted a trust crisis: increased productivity is accompanied by erosion of guarantees of correctness, verifiability, and predictability.

First, studies show that AI in code generation often prioritises functional correctness at the expense of security. In particular, according to [21], 51.42% of LLM-generated code contains known vulnerabilities classified via CWE. This estimate does not appear to be an isolated anomaly: across CWE scenarios, the share of vulnerable programs reaches approximately 40–44% depending on the selected slice, and for CWE-79 (Cross-Site Scripting) the authors separately report that among Copilot's top-scoring variants no vulnerabilities are observed (0%), whereas in the overall set of variants vulnerability is present in 19% of cases [22]. Such heterogeneity underscores that a model's best answer and its typical behaviour can diverge statistically, complicating safe deployment practice: stable guarantees that a successful example reflects the norm are not provided.

Second, the literature indicates persistent security blind spots arising from the induction of frequent patterns from training data. AI may propose weak hashing algorithms,

such as MD5, or hardcode API keys, as these patterns are common in example corpora and educational materials [23]. As a result, the model reproduces recognisable solutions that appear workable in the short term, but in fact create an attack surface and degrade risk controllability.

Third, a particular form of technical debt emerges: even when code appears correct, it can be redundant, bloated, and poorly maintainable. Developers report that AI-generated code often complicates maintenance, and in [24], respondents explicitly indicated the need to modify generated fragments prior to use. This should be interpreted not as a cosmetic adjustment, but as a symptom: generation accelerates artefact production while simultaneously increasing future work volume for validation, refactoring, and conformance to quality standards.

A distinct category of risks is formed by hallucinations in the coding context. Here, the problem is not novice mistakes, but the generation of syntactically valid yet functionally nonexistent constructs that appear plausible and may therefore pass initial attention-based checks. A particularly dangerous manifestation is dependency hallucination (Hallucinated Packages, Slopsquatting): the model may recommend importing libraries with realistic names that do not exist in the ecosystem, such as fast-async-auth-helper. Adversaries may monitor such hallucinations, register packages with corresponding names, and embed malicious code, converting a generation error into a supply-chain compromise channel. Study [25] shows that 21.7% of generated dependencies are hallucinations. This indicates that the risk is not peripheral, but statistically significant, where isolated cases can escalate into large-scale events under operational scaling.

Additionally, LLMs exhibit a tendency to create synthetic abstractions: instead of utilising standard libraries and accepted practices, they invent proprietary mini-security frameworks, creating an illusion of protection without verifiable guarantees [26]. Such pseudo-standardisation is hazardous precisely because it produces an appearance of solution completeness: the code looks structured and engineered, yet its properties are not confirmed by industry audits and do not rely on a mature ecosystem.

Finally, mobile platforms intensify specific risks. API obsolescence is particularly acute: models trained on data up to 2023 often generate code using deprecated Android APIs, resulting in compatibility issues with newer OS versions [27]. In this context, a generation error becomes not only a vulnerability or a bug, but also a source of operational costs, since it requires additional adaptation to current SDKs and platform policies.

In parallel, the limitations of On-Device AI are documented. Attempts to run testing LLM agents directly on devices, driven by concerns for privacy and autonomy, face hardware constraints. Small models exhibit a significant drop in reasoning capability compared to large-scale cloud models, rendering them currently unsuitable for complex testing scenarios [28]. Table 4 illustrates a comparative analysis of key risks in LLM-based code generation for mobile platforms.

Table 4. Comparative analysis of key risks in LLM-based code generation for mobile platforms

Phenomenon / Risk Area	What it looks like in AI-generated code	Quantitative / Empirical signal	Mobile-specific notes	Source
Known vulnerabilities (CWE) in generated code	Functional correctness is often prioritised over security, leading to insecure implementations.	51.42% of LLM-generated code contained known CWE vulnerabilities.	General (not platform-specific in this finding).	[21]
Hallucinated packages / Slopsquatting	Suggested dependencies may not exist; attackers can register those names and inject malware into the supply chain.	21.7% of generated dependencies were hallucinations.	High relevance to mobile build/dependency ecosystems; no platform split reported in text.	[25]
Deprecated Android APIs (API obsolescence)	Generation relies on deprecated Android APIs, creating compatibility issues on newer OS versions.	Qualitative finding (models trained on data up to 2023 often output deprecated APIs).	Android-specific compatibility risk with newer OS versions.	[27]
On-device LLM limitations for testing agents	Small on-device models lack the reasoning capacity of cloud models, which limits their application in complex testing scenarios.	Qualitative finding: significant reasoning drop; not suitable yet for complex tests.	Mobile/on-device constraints are driven by privacy/offline goals, but are limited by hardware.	[28]

Thus, the mobile contour faces dual pressure: on one hand, the demand for autonomy and privacy; on the other, insufficient computational resources to sustain the required level of reasoning in quality and security assurance tasks.

DISCUSSION

The conducted review indicates a fundamental shift in the profession. Whereas previously the developer's primary skill was knowledge of syntax and algorithms for writing code from scratch, in the GenAI era, the key competence becomes verification. The developer is transformed into an architect who formulates the task (prompt), and an auditor who critically evaluates the output. The degree of topic maturity is considered below.

First, the review results should be interpreted in the context of the research field's maturity and the limitations of the current evidence base. The tertiary study by Zein et al. shows that mobile engineering has accumulated a substantial number of SLR/mapping works; however, they are frequently fragmented by subdomain (QA, architecture, UX, security) and heterogeneous in methodological rigour, complicating direct comparison of effects across studies and the transferability of conclusions to industry [2]. In parallel, survey studies on factors impeding automated testing adoption report that barriers are not only technical but also organisational (maintenance cost of computerised tests, competence deficits, resistance to process change) [3], and a systematic review of mobile automation framework applicability confirms brittleness and high script-test maintenance costs under UI evolution [4]. Collectively, in baseline mobile automation and adoption challenges, the literature suffices to assert structural limitations of traditional approaches, against which AI appears as a mechanism for reducing cognitive and operational costs. At the same time, gaps remain: there are insufficient comparable, multi-site industrial studies that simultaneously account for application type, CI/CD maturity, team composition, and real support costs (TCO). Consequently, AI effects at the organisational or mobile product portfolio level are described more weakly than at the level of individual tools and cases.

Second, the literature on AI-assisted programming supports the productivity-growth thesis, while simultaneously indicating conditions under which this effect is stable and economically meaningful. A review of NLG/NLU big code systems systematises why LLMs perform best in template generation, transformations, and context explanation [5]. Empirical Copilot assessments show that the applicability of suggestions and time savings are heterogeneous and depend on the task context [8]. Comparative studies of code quality across Copilot/CodeWhisperer/ChatGPT complement the picture by showing that acceleration must be evaluated jointly with qualitative artifact characteristics (readability, standards compliance, maintainability), since these determine total cost of ownership at subsequent stages [9]; moreover, comparative evaluation for Swift generation emphasizes platform asymmetry and elevated error risks in less data-rich ecosystems [10]. With respect to AI impact on micro-productivity (task completion speed) and quality variability, the literature base is sufficient for substantiated conclusions; however, notable gaps remain specifically for the mobile domain: limited numbers of studies measure LLM impact on mobile-specific concerns (architectural patterns, performance/energy consumption, lifecycle correctness, accessibility, store-policy compliance), and long-term longitudinal data are lacking on how AI-assisted code affects technical debt and defectiveness across multiple releases rather than only at the moment of generation.

Third, the discussion of security and robustness risks demonstrates that speed increases without strengthening trust boundaries can produce systemic debt, particularly when sensitive to the mobile attack surface. Large-scale measurements of LLM generation security and evaluations of Copilot's contribution to vulnerabilities show that the probability of insecure implementations is statistically material and requires process-level institutionalisation of secure-by-default (SAST/DAST, secret scanning, dependency policies) [21, 22, 23]. The hallucinated packages risk expands the threat model to the supply chain, where a generation error can be monetised via the registration of dependency lookalikes [25]. Studies of user adaptation to hallucinations and privacy challenges emphasise that robustness becomes an organisational practice rather than a tool property [6]. Survey data on the applicability of AI assistants indicate that a significant share of code requires refinement before industrial use, shifting the economics of gains toward verification costs [24, 26]. On mobile platforms, the Android API evolution and compatibility problem is additionally critical [27], and edge-efficient LLM limitations support the conclusion of the necessity of hybrid architectures (on-device execution and cloud reasoning) [28]. Regarding general AI-code risks and supply-chain concerns, the contemporary literature provides sufficient basis to justify mandatory strengthened controls; however, mobile-ecosystem-specific gaps persist: comparatively few studies address Android/iOS-characteristic vulnerabilities (IPC, deep links, WebView, keychain/keystore, certificate pinning), privacy under transmission of UI data (screenshots/accessibility trees) to cloud LLMs, and practices for secure prompting and data governance in mobile teams, where regulatory and store requirements are particularly strict.

The data indicate that the largest productivity gains accrue to senior developers who can quickly recognise an error or vulnerability in AI suggestions. For juniors, AI may become a trap of confident incompetence, where syntactically correct but logically flawed code is accepted uncritically. Industry faces a classic trade-off. AI adoption can radically reduce Time-to-Market, which is critical in a competitive mobile environment. However, this gain may be negated by deferred penalties in the form of security incidents and the accumulation of technical debt. Organisations adopting AI must revise CI/CD processes. Human-only code review is insufficient, since the volume of generated code increases. Integration of automated SAST/DAST tools is required,

specifically configured to detect patterns characteristic of AI errors (hallucinated packages, hardcoded secrets).

Current advances in multi-agent systems (ScenGen) demonstrate that automated testing has approached the level of human understanding in the sense that it can interpret interface states and consistently execute meaningful verification actions guided by context and expected system behaviour. At the same time, such progress does not imply that corresponding approaches have become technologically inexpensive or seamlessly applicable at scale: key constraints remain token costs and latency arising from transmitting screenshots to cloud LLMs, hindering pervasive deployment.

Against this background, the most plausible development direction appears to be hybrid architectures in which computational roles and temporal requirements are stratified across levels. The cloud contour can be viewed as a heavy reasoning layer: a powerful model forms the testing strategy and assumes analysis of complex failures, solving tasks where generalisation, hypothesis construction, and interpretation of non-standard situations are essential.

At the device level, by contrast, fast execution is naturally described: a lightweight local model performs navigation and simple checks in real-time without requiring network connectivity. In such a distribution, emphasis shifts toward the autonomous execution of frequently occurring and fast actions, while rare, expensive, and cognitively intensive operations are delegated to the cloud, where a more powerful reasoning apparatus is available.

CONCLUSION

The conducted systematic literature review demonstrates that, between 2019 and 2025, mobile development and testing underwent a qualitative shift toward the generative AI paradigm and multi-agent architectures. The selection of 28 studies from 638 identified records, along with quality assessment using an adapted Sjøberg checklist, establishes an evidentiary framework for the review's conclusions. This is not a set of disparate technological observations, but a synthesis of empirical data comparable in terms of acceleration, effectiveness, and risk metrics.

Regarding development acceleration, the SLR results document a stable effect from adopting AI assistants, primarily manifesting in template code generation, refactoring, and artefact transformation. The literature provides quantitative estimates of up to 55% acceleration in coding for typical tasks, and Time-to-Market improvements on the order of 30%, while noting that a substantial share of routine code volume can be generated by AI. Particularly illustrative is the industrial legacy migration case: the transition of 3,500 test files from Enzyme to React Testing Library, which, under traditional estimates, could have taken up to 1.5 years, was completed in 6 weeks with the automated migration of 97% of files. At the same time, the review emphasises platform and language heterogeneity: the highest performance is observed for Kotlin/Java and cross-platform stacks (Flutter/React Native), whereas Swift/SwiftUI generation quality is reduced, associated with training-data scarcity and ecosystem closure.

In testing, the systematic synthesis indicates a transition from deterministic, brittle scripted scenarios to agent-based approaches capable of semantic interface perception and planning. Multi-agent systems such as ScenGen and AutoQALLMs are characterised as the most promising architectural generation, wherein GUI perception, action selection, execution, and result revision are divided into specialised roles, and context is maintained via action memory. Empirically, this is expressed in code-coverage increases of 20–30% compared to traditional tools, and in high accuracy for executing complex scenarios. For ScenGen, Logical Decision-Making Accuracy values range from 90.14% to 100%, depending on the application. An important maturity element is also provided by visual self-healing mechanisms, as well as specialised solutions addressing operational causes of run instability. These solutions include the closure of blocking pop-up windows with 91.7% detection accuracy and a 15.6% increase in the efficiency of crowdsourced bug-report processing via visual clustering.

The key contribution of this SLR is not only the documentation of acceleration, but also the systematic fixation of quality and security risks that scale alongside generation. The literature describes a trust crisis in which productivity growth is accompanied by erosion of correctness and verifiability guarantees. The most stringent signal concerns security: according to included studies, up to 51.42% of LLM-generated code contains known CWE-classified vulnerabilities, and aggregate scenario-set estimates indicate a range of 40–44% vulnerable implementations. An additional, fundamentally novel threat class is formed by dependency hallucinations (hallucinated packages / slopsquatting): the share of nonexistent dependencies in generated recommendations reaches 21.7%, turning a generation error into a practical supply-chain attack vector. The review also notes reproduction of unsafe frequent patterns (e.g., weak hashes or hardcoded secrets), growth of specific technical debt due to bloated and poorly maintainable fragments, and, for the mobile domain, an additional API obsolescence problem, whereby models trained on data up to 2023 tend to propose deprecated Android APIs, increasing operational costs and compatibility risks.

Collectively, the systematic literature review substantiates the conclusion that the developer's role is transforming from primary code production toward verification, architectural oversight, and security auditing, as well as management of the probabilistic nature of generative outputs. Empirical evidence confirms that AI accelerates the mobile SDLC and increases the intellectual capacity of testing; however, the same body of studies indicates that adoption cannot be treated as a neutral

tool substitution: it requires institutionalization of trust contours, strengthening of security protocols, and revision of quality assurance approaches, where deterministic checks are complemented by practices designed for statistical variability, hallucinations, and synthetic vulnerabilities.

REFERENCES

1. Shah J. AI App Development: Build Smarter Apps in 2026 [Internet]. Openxcell. 2025 [cited 2025 Dec 17]. Available from: https://www.openxcell.com/ai-application-development/
2. Zein S, Salleh N, Grundy J. Systematic reviews in mobile app software engineering: A tertiary study. Information & Software Technology. 2023 Dec 1;164:107323.
3. Murazvu G, Parkinson S, Khan S, Liu N, Allen G. A Survey on Factors Preventing the Adoption of Automated Software Testing: A Principal Component Analysis Approach. Software. 2024 Jan 2;3(1):1–27.
4. Berihun NG, Dongmo C, Van der Poll JAV der. The Applicability of Automated Testing Frameworks for Mobile Application Testing: A Systematic Literature Review. Computers. 2023 May 3;12(5):97.
5. Wong MF, Guo S, Hang CN, Ho SW, Tan CW. Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review. Entropy. 2023 Jun 1;25(6):888.
6. Lee C, Kim J, Lim JS, Shin D. Generative AI Risks and Resilience: How Users Adapt to Hallucination and Privacy Challenges. Telematics and Informatics Reports. 2025 Jun 13;19:100221.
7. Molléri JS, Petersen K, Mendes E. An empirically evaluated checklist for surveys in software engineering. Information and Software Technology. 2020 Mar 1;119:106240.
8. Nguyen N, Nadi S. An empirical evaluation of GitHub copilot's code suggestions. Proceedings of the 19th International Conference on Mining Software Repositories. 2022 May 23;
9. Yetiştiren B, Özsoy I, Ayerdem M, Tüzün E. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. arXiv preprints. 2023 Apr 21;
10. Wan B. Evaluating Large Language Models for Code Generation: A Comparative Study on Python, Java, and Swift. Applied and Computational Engineering. 2025 Apr 21;146:109–26.
11. Liu W, Liu L, Guo Y, Xiao H, Lin W, Chai Y, et al. LLM-Powered GUI Agents in Phone Automation: Surveying Progress and Prospects. Preprints org. 2025 Jan 6;
12. Mallipeddi S, Yaqoob M, Khan JA, Mehmood T, Mylonas A, Pitropakis N. AutoQALLMs: Automating Web Application Testing Using Large Language Models (LLMs) and Selenium. Computers. 2025 Nov 18;14(11):501.
13. Yu S, Ling Y, Fang C, Zhou Q, Zhao Y, Chen C, et al. LLM-Guided Scenario-based GUI Testing. arXiv preprints. 2025 Jun 5;
14. Guo L, Liu W, Heng YW, Chen TH, Wang Y. PopSweeper: Automatically Detecting and Resolving App-Blocking Pop-Ups to Assist Automated Mobile GUI Testing. arXiv preprints. 2024 Dec 4;
15. Fang C, Yu S, Zhang Q, Li X, Liu Y, Chen Z. Enhanced Crowdsourced Test Report Prioritization via Image-and-Text Semantic Understanding and Feature Integration. IEEE Transactions on Software Engineering. 2024 Dec 12;51:283–304.
16. Peng S, Kalliamvakou E, Cihon P, Demirer M. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. arXiv preprints. 2023 Feb 13;
17. Pandey R, Singh P, Wei R, Shankar S. Transforming Software Development: Evaluating the Efficiency and Challenges of GitHub Copilot in Real-World Projects. arXiv. 2024 Jun 25;
18. Covey-Brandt C. Accelerating Large-Scale Test Migration with LLMs [Internet]. Airbnb. 2020 [cited 2025 Nov 17]. Available from: https://airbnb.tech/uncategorized/accelerating-large-scale-test-migration-with-llms/
19. Li Y, Yang Z, Guo Y, Chen X. Humanoid: A Deep Learning-based Approach to Automated Black-box Android App Testing. arXiv preprints. 2019;
20. Zhou T, Zhao Y, Hou X, Sun X, Chen K, Wang H. Bridging Design and Development with Automated Declarative UI Code Generation. arXiv preprints. 2024 Sep 18;
21. Tihanyi N, Bisztray T, Ferrag MA, Jain R, Cordeiro LC. How secure is AI-generated code: a large-scale comparison of large language models. Empirical Software Engineering. 2024 Dec 21;30(2).
22. Pearce H, Ahmad B, Tan B, Dolan-Gavitt B, Karri R. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. Communications of the ACM. 2025 Jan 22;68(2):96–105.
23. Huang Y, Li Y, Wu W, Zhang J, Lyu MR. Your Code Secret Belongs to Me: Neural Code Completion Tools Can Memorize Hard-Coded Credentials. Proceedings of the ACM on Software Engineering. 2024 Jul 12;1(FSE):2515–37.
24. Liang JT, Yang C, Myers BA. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. arXiv preprints. 2024 Feb 6;

25. Spracklen J, Wijewickrama R, Nazmus S, Maiti A, Jadliwala M. We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs. arXiv preprints. 2024 Jun 11;

26. Khoury R, Avila AR, Brunelle J, Camara BM. How Secure is Code Generated by ChatGPT? arXiv preprints. 2023 Apr 19;

27. Mahmud T, Che M, Yang G. An empirical study on compatibility issues in Android API field evolution. Information and Software Technology. 2024 Jul 20;175:107530.

28. Wang R, Gao Z, Zhang L, Yue S, Gao Z. Empowering large language models to edge intelligence: A survey of edge efficient LLMs and techniques. Computer Science Review. 2025 Apr 9;57:100755.