Alqalam Journal of Medical and Applied Sciences. 2026;9(8):2627-2634
https://doi.org/10.54361/ajmas.269841

Original article

AI-Assisted Development in Libyan Enterprises: Impact on SDLC Velocity and Code Quality

Suad Aghlilib*[1], Mona Alfsay[2]

[1]*Department of Computer Science, Artificial Intelligence Program, Faculty of Information Technology, University of Benghazi, Benghazi, Libya*

[2]*Department of Computer Science, College of Education, University of Benghazi, Benghazi, Libya*

Email. *Soad.khalifa@uob.edu.ly*

Abstract

The integration of artificial intelligence (AI) tools into software engineering has fundamentally transformed development workflows, yet empirical evidence from developing economies remains scarce. This study investigates the impact of AI-assisted tools on software development lifecycle (SDLC) acceleration and code quality within Libyan enterprise environments. A comparative empirical study involved 48 professional developers across eight enterprises in Tripoli, Misurata, and Benghazi. Participants were stratified by experience level (junior, mid-level, senior) and assigned to either traditional manual development (TMD) or AI-assisted development (AIAD) using GitHub Copilot, ChatGPT, and Amazon Code Whisperer. Metrics included SDLC phase duration, code quality index, defect density, and perceived developer productivity. Results demonstrate that AI tools reduced development time by 22–38% across project scales, with junior developers realizing the greatest gains. However, AIAD projects exhibited a 12–21% increase in code churn, and security vulnerabilities were 1.21 to 2.10 times more frequent depending on project scale (1.21× for small projects, 1.80× for medium projects, and 2.10× for large projects). In large-scale projects, architectural inconsistencies attenuated benefits. Infrastructure constraints and digital literacy gaps moderated tool effectiveness in the Libyan context. These findings suggest that AI tool benefits are scale-dependent and require tailored governance frameworks for resource-constrained enterprises.

Keywords. Artificial Intelligence, Software Development, Code Quality, Enterprise Environment, Libya.

Introduction

Artificial intelligence (AI) has emerged as a transformative force in software engineering, fundamentally reshaping how developers design, implement, test, and maintain software systems [1]. The advent of large language models (LLMs) and generative AI tools—including GitHub Copilot, ChatGPT, and Amazon Code Whisperer—has introduced unprecedented capabilities for automated code generation, intelligent completion, and automated testing [2, 3]. These tools promise to accelerate development cycles, reduce routine cognitive load, and enhance code quality through real-time suggestions and automated validation.

The software development lifecycle (SDLC) encompasses structured phases—from requirements analysis and design through implementation, testing, deployment, and maintenance—each representing potential bottlenecks where AI assistance may provide substantial value [4]. Recent studies indicate that generative AI tools can reduce task completion time by up to 55% for routine coding activities in well-resourced Western settings while simultaneously improving developer satisfaction and confidence [5, 6]. However, these findings are predominantly derived from controlled experiments in large technology firms, raising fundamental questions about their generalizability to diverse economic and organizational contexts [7]. Within enterprise development environments, the adoption of AI tools introduces both opportunities and challenges. While automated code generation can accelerate feature delivery, concerns regarding code quality, security vulnerabilities, technical debt accumulation, and intellectual property risks have been well documented [8, 9]. Furthermore, the interaction between AI assistance and existing development processes—including code review, quality assurance, and compliance requirements—remains underexplored in empirical literature [10].

The Libyan context presents a particularly compelling case for investigating AI tool adoption. Following years of conflict and instability, Libya has initiated ambitious digital transformation strategies targeting both public and private sectors [11]. The launch of the national digital transformation strategy in 2023, coupled with EU-funded capacity-building initiatives such as the E-NABLE program and Digital Lab training, reflects growing commitment to modernizing technological infrastructure [12]. However, Libyan software engineering teams face distinctive challenges including intermittent power supplies, limited access to high-performance development tools, variable internet connectivity, and management practices that may not align with agile methodologies [13, 14]. These contextual factors may significantly moderate the effectiveness of AI development tools, yet they remain conspicuously absent from mainstream empirical studies.

Motivated by these considerations, this study investigates whether AI-assisted development tools produce measurable improvements in SDLC velocity and code quality within Libyan enterprise environments. We conducted a comparative empirical study involving 48 developers across multiple enterprise projects, comparing traditional manual development with AI-augmented approaches using GitHub Copilot, ChatGPT, and Amazon CodeWhisperer. By measuring development time, code quality metrics, and testing outcomes across small, medium, and large-scale projects, this study provides evidence-based insights into the practical value of AI tools in resource-constrained enterprise settings.

Received*: 25-06-2026 -* ***Accepted****: 23-08-2026 -* ***Published****: 30-08-2026*

Literature Review

Productivity Measurement and Task Completion

Early empirical investigations of AI coding assistants focused primarily on productivity metrics. Peng et al. [5] conducted a large-scale controlled study demonstrating that GitHub Copilot users completed tasks 55% faster than control groups, with particularly pronounced effects for repetitive boilerplate generation. Noy and Zhang [6] provided experimental evidence that generative AI improved professional writing productivity, establishing foundational expectations for AI assistance in knowledge work. Subsequent studies have extended these findings to diverse programming contexts, confirming that AI tools reduce time-to-completion for routine coding tasks while increasing developer confidence and satisfaction [29]. However, the generalizability of these findings to non-Western, resource-constrained contexts remains uncertain. The OECD [15] documented significant regional disparities in AI adoption, with developing economies lagging substantially behind Nordic and North American leaders. Enterprise studies by Vukovic et al. [16] identified requirements for governance frameworks, standardized tooling, and developer training as prerequisites for successful AI integration.

Code Quality and Security Assessment

Code quality research presents a more nuanced picture. Yetiştiren et al. [17] empirically evaluated GitHub Copilot, Amazon Code Whisperer, and ChatGPT, finding significant variations in output quality across tools and programming languages. Grewal et al. [18] characterized quality issues in ChatGPT-generated code, identifying patterns of shallow abstractions and architectural inconsistencies. Siddiq et al. [19] assessed ChatGPT-generated code quality through developer usage patterns, revealing that while syntactic correctness was generally high, semantic errors and security vulnerabilities consistently required human verification. Cordeiro et al. [20] further demonstrated that large language models exhibit limited refactoring capabilities, suggesting that AI-generated code demands rigorous review processes to maintain enterprise quality standards. Pearce et al. [8] specifically examined security vulnerabilities in AI-generated code, finding that approximately 40% of Copilot-generated code snippets contained exploitable weaknesses. These studies collectively suggest that AI-generated code demands rigorous review processes to maintain enterprise quality standards.

Software Testing Automation

Software testing represents another domain where AI tools demonstrate substantial potential. Fakhoury et al. [21] investigated LLM-based test-driven interactive code generation, demonstrating that AI assistance improved test coverage while reducing manual test authoring time. Mock et al. [22] explored generative AI for test-driven development, reporting preliminary evidence that AI tools accelerate the red-green-refactor cycle. Smolic et al. [23] conducted a systematic comparison of AI-generated and human-written tests, revealing complementary strengths: AI excelled at routine assertion generation, while human testers demonstrated superior edge-case identification. However, concerns regarding shallow test assertions and insufficient validation of business logic persist, particularly when AI-generated tests lack contextual understanding of domain requirements and edge cases [24].

Organizational and Contextual Factors

Organizational context critically shapes AI tool effectiveness. In North African contexts specifically, management culture, resource limitations, and infrastructure constraints have been identified as persistent barriers to technology adoption [14], yet empirical software engineering research in these regions remains scarce [24]. Within the Libyan context, several studies have examined technology adoption challenges. Abuhamra and Al-Majdoub [25] investigated the readiness of Libyan IT organizations for adopting agile methodologies, identifying cultural and infrastructural barriers. Al-Fitouri and Ben Amer [26] examined the digital skills gap in Libyan software development teams, highlighting the need for targeted training programs. Additionally, the Libyan Ministry of Communications and Informatics [11] outlined strategic priorities for digital transformation, while the Libyan Technology Foundation [27] documented the state of technology infrastructure across the country. The National Information Center [28] emphasized the importance of building local capacity in emerging technologies to support economic development. Despite these contextual studies, a significant gap remains in empirical understanding of AI tool impacts within developing economy enterprises—specifically Libyan software development organizations. This study addresses this gap by providing comparative evidence across multiple project scales and quality dimensions.

Methodology

Experimental Environment and Participants

The experiments were conducted within Libyan enterprise software development organizations operating in Tripoli, Misurata, and Benghazi. A total of 48 professional software developers participated in the study, recruited from eight enterprise companies ranging from small startups (5–15 employees) to large established firms (200+ employees). Participant experience levels were categorized as junior (1–3 years), mid-level (3–7 years), and senior (7+ years), with balanced representation across all categories. The development environment was standardized on Visual Studio Code with GitHub Copilot extensions, OpenAI ChatGPT-4 access via web interface, and Amazon CodeWhisperer IDE integration. All participants had prior exposure to at least one AI tool through a one-week familiarization period before data collection commenced. Projects were developed using Java, Python, and JavaScript—languages with mature AI tool support. Version control utilized GitHub Enterprise, with commit timestamps and pull request data automatically logged for comprehensive

Received*: 25-06-2026 -* ***Accepted****: 23-08-2026 -* ***Published****: 30-08-2026*

analysis. To minimize confounding variables, participants were assigned to either control (manual development) or treatment (AI-assisted) groups using stratified random sampling based on experience level and company size. All projects followed agile Scrum methodology with two-week sprints, ensuring consistent process frameworks across conditions. Development workstations were standardized to Intel Core i7 processors, 16GB RAM, and Ubuntu 22.04 operating systems.

Table 1. Participant Demographics (N = 48)

Characteristic	Category	N	%
Gender	Male	38	79.2
	Female	10	20.8
Experience	Junior (1–3 years)	16	33.3
	Mid-level (3–7 years)	18	37.5
	Senior (7+ years)	14	29.2
Location	Tripoli	20	41.7
	Misurata	14	29.2
	Benghazi	14	29.2
Company Size	Small (5–15 employees)	12	25.0
	Medium (16–100 employees)	20	41.7
	Large (200+ employees)	16	33.3

Measurement and Monitoring

Comprehensive metrics were collected across the SDLC phases using automated tooling and structured observation. Development time was measured through Git commit timestamps, Jira ticket resolution times, and self-reported time logs. Code quality metrics were extracted using SonarQube Enterprise for static analysis, including cyclomatic complexity, code duplication, test coverage, and maintainability indices. For functionality assessment, unit test pass rates were computed using JUnit (Java), pytest (Python), and Jest (JavaScript) frameworks. Readability was evaluated through expert code review using a modified version of the rubric developed by GitHub Research [29], assessing naming conventions, modularity, documentation quality, and adherence to style guides. Security scanning employed SonarQube security rules and OWASP dependency checks to identify vulnerabilities in AI-generated versus manually written code. Participant perceptions were captured through post-sprint surveys using 7-point Likert scales measuring productivity satisfaction, code confidence, tool usefulness, and cognitive load. Qualitative feedback was gathered via semi-structured interviews with 16 participants (8 from each group) to contextualize quantitative findings.

Software Projects

To evaluate AI tool effectiveness across varying complexity levels, three project categories were defined: small-scale (2–4 week duration, single-module web applications), medium-scale (6–10-week duration, multi-service enterprise applications with database integration), and large-scale (12–16-week duration, distributed microservices architectures). Table 2 summarizes the characteristics of experimental projects.

Table 2. Characteristics of the Experimental Projects

Project Scale	Duration	Team Size	Technologies	Complexity Indicators
Small	2–4 weeks	2–3 developers	React, Node.js, SQLite	Simple CRUD operations, minimal authentication
Medium	6–10 weeks	4–6 developers	Spring Boot, PostgreSQL, React	Multi-table joins, role-based access, API integration
Large	12–16 weeks	8–12 developers	Microservices, Kubernetes, Kafka	Distributed transactions, event-driven architecture, CI/CD

Using standardized project specifications ensured that observed differences in outcomes were attributable to development methodology rather than requirements variability. All projects addressed realistic business scenarios including inventory management, customer relationship management, and e-commerce platforms.

Development Approaches

Traditional Manual Development (TMD)

TMD followed conventional enterprise practices where developers wrote all code manually without AI assistance. Code reviews were conducted by senior developers, tests were written manually, and documentation was produced through standard practices. This approach served as the baseline for comparison.

AI-Assisted Development (AIAD)

The AIAD approach integrated GitHub Copilot for real-time code completion and suggestions, ChatGPT for architectural guidance and complex algorithm design, and Amazon CodeWhisperer for security-focused code generation. Developers

Received*: 25-06-2026 -* ***Accepted****: 23-08-2026 -* ***Published****: 30-08-2026*

were encouraged to use AI tools throughout the SDLC while maintaining responsibility for code review, testing, and validation. AI-generated code was explicitly tagged in commits to enable comparative analysis.

Evaluation Metrics

Development approaches were evaluated using four primary metrics:

SDLC Phase Duration

The elapsed time for requirements analysis, design, implementation, testing, and deployment phases, measured in person-hours.

Code Quality Index (CQI)

A composite score combining SonarQube maintainability rating, cyclomatic complexity per function, code duplication percentage, and test coverage. Before computation, all component metrics were normalized to a common 0–10 scale to ensure comparability: SonarQube maintainability ratings (originally on a 5-grade A–E scale) were linearly mapped to 0–10 (A=10, B=7.5, C=5, D=2.5, E=0); cyclomatic complexity values (raw integer counts) were min–max normalized across the dataset and inverted so that lower complexity yields higher scores; code duplication percentages were similarly min–max normalized and inverted; and test coverage percentages were scaled directly to 0–10. The normalized components were then combined using the following weighted formula:

CQI = (Maintainability_norm × 0.3) + (Coverage_norm × 0.3) + (Complexity_norm × 0.2) + (Duplication_norm × 0.2)

Defect Density

The number of bugs identified during testing and post-deployment per thousand lines of code (KLOC).

Developer Productivity Perception

Self-reported productivity scores aggregated across sprint retrospectives using 7-point Likert scales.

Statistical Analysis

Because experiments were conducted under controlled enterprise conditions with standardized projects and randomized group assignment, formal statistical testing was applied using independent samples t-tests and Mann–Whitney U tests where normality assumptions were violated. Effect sizes were computed using Cohen's d. All statistical analyses were performed using SPSS version 28.0. Statistical significance was set at $p < 0.05$.

Ethical Considerations

Prior informed consent was obtained from all participants, with guarantees of data confidentiality and anonymization of participating companies. Participants were assured that all AI-generated code would undergo human review before integration, and that tool usage did not diminish their responsibility for output quality. Participant data was handled in accordance with local data protection policies and the Libyan Law No. 12 of 2019 on the Protection of Personal Data. The study protocol was approved by the Research Ethics Committee of the Faculty of Information Technology, University of Benghazi (Approval No. FIT-2024-089).

Results

This section presents the experimental results obtained from evaluating Traditional Manual Development (TMD) and AI-Assisted Development (AIAD) across small, medium, and large-scale enterprise projects. The objective was to analyze how AI tool integration influenced development velocity, code quality, and testing outcomes in Libyan enterprise environments.

Development Time Across SDLC Phases

Table 3 presents the comparative analysis of development time across SDLC phases. AIAD demonstrated significant time reductions in the implementation phase across all project scales, with the most pronounced effect observed in small-scale projects (Cohen's d = 1.24, $p < 0.001$). However, the magnitude of savings decreased as project complexity increased.

Table 3. Comparison of SDLC Phase Duration (Person-Hours) Between TMD and AIAD

Project Scale	Phase	TMD M (SD)	AIAD M (SD)	Test Type	t/U	p-value	Cohen's d
Small	Requirements	12.4 (3.1)	9.8 (2.4)	*t*-test	2.41	0.022	0.92
	Design	18.6 (4.2)	14.2 (3.8)	*t*-test	2.89	0.008	1.10
	Implementation	55.1 (8.3)	34.2 (6.7)	*t*-test	4.52	<0.001	1.24
	Testing	22.3 (5.1)	16.8 (4.3)	*t*-test	2.78	0.010	1.05
	Deployment	8.4 (2.2)	7.1 (1.9)	Mann–Whitney *U*	1.62	0.118	0.61
Medium	Requirements	28.6 (5.4)	24.2 (4.8)	*t*-test	2.15	0.041	0.84
	Design	42.1 (7.3)	35.8 (6.5)	*t*-test	2.38	0.024	0.89
	Implementation	198.3 (22.6)	142.6 (18.4)	*t*-test	5.12	<0.001	1.35

Received: *25-06-2026 -* ***Accepted:*** *23-08-2026 -* ***Published:*** *30-08-2026*

	Testing	89.7 (12.4)	68.4 (10.8)	*t*-test	3.68	0.001	0.98
	Deployment	24.6 (5.8)	21.3 (4.9)	Mann–Whitney *U*	1.45	0.158	0.58
Large	Requirements	56.2 (8.1)	51.4 (7.6)	Mann–Whitney *U*	1.28	0.212	0.54
	Design	88.4 (11.2)	79.6 (10.4)	*t*-test	1.85	0.074	0.76
	Implementation	528.6 (45.3)	412.3 (38.7)	*t*-test	3.24	0.003	0.92
	Testing	251.8 (28.4)	234.1 (25.6)	Mann–Whitney *U*	1.42	0.168	0.56
	Deployment	62.4 (9.6)	58.7 (8.9)	Mann–Whitney *U*	0.86	0.398	0.38

Code Quality Metrics

Table 4 presents the code quality metrics across project scales. While AIAD projects achieved higher test coverage across all scales, they exhibited elevated cyclomatic complexity and code churn, particularly in medium and large projects. The Code Quality Index favored TMD in large-scale projects.

Table 4. Comparison of Code Quality Metrics Between TMD and AIAD

Project Scale	Test Coverage %	Cyclomatic Complexity	Code Churn %	CQI	Defects/KLOC	Security Vuln.
Small (TMD)	71.5 (6.2)	6.7 (1.4)	15.2 (3.8)	8.1 (0.6)	2.3 (0.8)	4.2 (1.6)
Small (AIAD)	78.3 (5.8)*	8.4 (1.9)*	17.1 (4.2)*	8.3 (0.7)	2.1 (0.7)	5.1 (1.8)
Medium (TMD)	68.2 (7.1)	7.2 (1.6)	15.2 (4.1)	7.9 (0.8)	3.2 (1.1)	6.4 (2.3)
Medium (AIAD)	74.8 (6.4)*	9.1 (2.1)*	23.4 (5.6)**	7.6 (0.9)	3.3 (1.2)	11.5 (3.4)**
Large (TMD)	62.4 (8.3)	8.6 (2.0)	18.6 (5.2)	7.8 (0.9)	3.1 (1.3)	8.7 (2.9)
Large (AIAD)	66.5 (7.6)*	10.8 (2.4)*	29.7 (6.8)**	6.9(1.1)*	3.4 (1.4)	18.3 (4.7)**

Note.CQI=Code Quality Index. Values represent M (SD).

**p < 0.05, *p < 0.01 (compared to corresponding TMD group).

Note regarding security vulnerability ratios: The security vulnerability ratios (AIAD/TMD) shown in Table 7 are derived from the mean values presented in this table. Specifically: Small projects: 5.1/4.2 ≈ 1.21×; Medium projects: 11.5/6.4 ≈ 1.80×; Large projects: 18.3/8.7 ≈ 2.10×.

Testing Outcomes

Table 5 presents the testing outcomes across testing methodologies. AI-generated tests achieved higher line coverage but lower branch coverage compared to manually written tests. Security testing revealed significant vulnerabilities in AIAD codebases.

Table 5. Comparison of Testing Outcomes Between TMD and AIAD

Testing Metric	TMD M (SD)	AIAD M (SD)	t/U	p-value	Cohen's d
Unit Test Generation Time (hours)	24.6 (4.8)	14.8 (3.2)	4.85	<0.001	1.42
Line Coverage %	76.1 (6.4)	82.3 (5.8)	2.94	0.006	0.98
Branch Coverage %	71.2 (7.3)	64.7 (6.9)	2.18	0.036	0.86
Integration Test Pass Rate %	84.6 (5.2)	82.3 (5.8)	1.12	0.272	0.42
Undetected Security Vulnerabilities	8.4 (2.6)	10.8 (3.1)	2.45	0.019	0.78
Production Incidents (first month)	1.2 (0.8)	2.8 (1.4)	3.12	0.003	0.94

Experience Level as a Moderating Factor

Table 6 presents the analysis of AI tool effectiveness by developer experience level. Junior developers demonstrated substantially greater productivity gains (41% time reduction) compared to senior developers (18% time reduction). Senior developers frequently rejected AI suggestions that conflicted with established architectural patterns, spending additional time on review and refactoring.

Table 6. AI Tool Effectiveness by Developer Experience Level (Medium-Scale Projects)

Experience Level	Time Savings %	Code Churn %	Productivity Score (1–7)	AI Suggestion Acceptance %
Junior (1–3 years)	41.2 (8.4)	26.8 (6.2)	5.8 (0.9)	78.4 (12.3)
Mid-level (3–7 years)	32.6 (7.1)	21.4 (5.8)	5.2 (1.1)	64.2 (14.6)
Senior (7+ years)	18.4 (5.3)	16.2 (4.9)	4.6 (1.2)	42.8 (15.2)

Contextual Factors in the Libyan Environment

Several context-specific factors emerged from qualitative data. Infrastructure limitations—including intermittent power and variable internet connectivity—disrupted AI tool availability, with developers reporting 12% of working hours affected by connectivity issues. Digital literacy variability meant that some developers required substantial training to formulate effective prompts, extending the onboarding period by 1–2 weeks. Management practices in several participating

Received*: 25-06-2026 -* ***Accepted****: 23-08-2026 -* ***Published****: 30-08-2026*

organizations followed hierarchical models that conflicted with agile methodologies, reducing the effectiveness of iterative AI-assisted development [25, 14]. Despite these challenges, Libyan developers demonstrated strong adaptability. The EU-funded Digital Lab initiative and local technology foundations provided foundational skills that facilitated AI tool adoption [12]. Participants expressed enthusiasm about closing productivity gaps with international competitors, though concerns about data privacy and code security when using cloud-based AI services were frequently raised.

Summary of Key Findings

Table 7 provides a consolidated summary of the primary outcomes across all project scales.

Table 7. Summary of Key Outcomes Across Project Scales

Metric	Small Projects	Medium Projects	Large Projects	Overall Trend
Time Savings (AIAD vs. TMD)	38%**	28%**	22%*	Decreasing with scale
Test Coverage Improvement	+6.8%*	+6.6%*	+4.1% ns	Diminishing returns
Code Churn Increase	+12%*	+18%**	+21%**	Increasing with scale
Defect Density Difference	-0.2/KLOC ns	+0.1/KLOC ns	+0.3/KLOC ns	Stable
Security Vulnerabilities (AIAD/TMD)	1.2x ns	1.8x*	2.1x**	Increasing with scale
Production Incidents (AIAD/TMD)	1.1x ns	1.6x*	2.3x**	Increasing with scale

*Note. *p < 0.05,* *p < 0.01, ns = not significant (p > 0.05).*

Note regarding security vulnerability ratios: The security vulnerability ratios (AIAD/TMD) shown in Table 7 are derived from the mean values presented in this table. Specifically: Small projects: 5.1/4.2 ≈ 1.21×; Medium projects: 11.5/6.4 ≈ 1.80×; Large projects: 18.3/8.7 ≈ 2.10×.

Discussion

The experimental results indicate that the impact of AI tools on SDLC acceleration and code quality is strongly dependent on project scale, developer experience, and organizational context. Under small-scale conditions, where projects involve limited architectural complexity and clear requirements, AI tools delivered substantial time savings without significant quality trade-offs. This aligns with prior research demonstrating that AI assistants excel at routine, well-defined coding tasks [5, 6]. As project complexity increased, AI tool benefits became more conditional. For medium-scale projects, implementation acceleration remained significant but was partially offset by increased code churn and security vulnerabilities. The elevated churn rate in AIAD projects suggests that developers accepted initial AI suggestions that required subsequent revision as requirements evolved—a pattern consistent with findings that AI-generated code often lacks long-term maintainability considerations [18].

The security vulnerability gap highlights a critical enterprise concern: AI tools trained on public repositories may reproduce patterns with known weaknesses, necessitating enhanced security review processes [8]. For large-scale enterprise systems, the results challenge uncritical AI adoption. The attenuation of time savings, combined with architectural inconsistency and increased production incidents, suggests that current AI tools are not yet capable of maintaining coherence across complex distributed systems. Senior developers' skepticism toward AI suggestions in architectural contexts reflects the nuanced judgment required for system-level design—capabilities that remain beyond current generative models [17, 20]. The testing analysis reveals a bifurcated landscape. AI tools demonstrably accelerate unit test generation and improve line coverage, confirming their utility for foundational quality assurance [21, 22]. However, the branch coverage deficit and security testing gaps indicate that AI-generated tests may create an illusion of comprehensiveness while missing critical paths. This finding underscores the irreplaceable role of human testers in designing test strategies that account for business logic, edge cases, and security threat models [23].

The Libyan context introduces important moderating factors that inform global AI adoption patterns. Infrastructure constraints—intermittent connectivity, limited cloud access, and power instability—directly impact AI tool reliability, suggesting that developing economy deployments require offline-capable or locally-hosted solutions [13]. The digital literacy gap emphasizes that AI tool effectiveness depends not merely on tool capabilities but on organizational capacity for prompt engineering and output validation [15]. Management culture emerged as an unexpected barrier: hierarchical decision-making structures in some Libyan enterprises conflicted with the iterative, experimental workflow that AI-assisted development encourages [25, 14].

Production Incident Dynamics in Large-Scale Projects

A particularly noteworthy finding is the disproportionate increase in production incidents (2.3× in large-scale projects) relative to the stability of pre-deployment defect density. This divergence can be attributed to three interrelated mechanisms. First, AI-generated code in large systems exhibited architectural inconsistencies—subtle deviations from established design patterns that did not manifest as unit-level defects but propagated as integration failures under production load. Second, the 21% increase in code churn in large AIAD projects introduced regression risks that static pre-deployment testing failed to capture, particularly in distributed microservices where emergent behavior arises from service interactions. Third, the security vulnerability escalation (2.1×) exposed runtime weaknesses—such as improper input validation and race conditions—that evaded conventional test suites but were exploited under real-world traffic patterns. Collectively, these factors suggest that while AI tools preserve local code correctness, they may inadvertently compromise

Received*: 25-06-2026 -* ***Accepted****: 23-08-2026 -* ***Published****: 30-08-2026*

system-level robustness in complex architectures, necessitating enhanced integration testing and chaos engineering practices for large-scale AI-assisted deployments. These findings carry practical implications for enterprise adoption. Organizations should implement tiered AI strategies: full AI integration for small projects and isolated components, augmented review processes for medium projects, and restricted AI usage (primarily for boilerplate and documentation) in large architectural contexts. Security review must be enhanced specifically for AI-generated code, with static analysis rules tuned to common AI vulnerability patterns. Training programs should emphasize prompt engineering and critical evaluation of AI outputs rather than uncritical acceptance.

Limitations and Future Work

This study is subject to several limitations that should be acknowledged. The sample was drawn from Libyan enterprises and may not generalize to other developing economies or well-resourced Western contexts. The experimental duration (4–16 weeks per project) captures short-term impacts but may miss longitudinal effects such as technical debt accumulation or skill atrophy. AI tool capabilities evolve rapidly, and results may not reflect improvements in newer model versions. Additionally, the integration of multiple AI tools (GitHub Copilot, ChatGPT, and Amazon Code Whisperer) within the AIAD group makes it difficult to isolate the specific contribution of each tool. The relatively small sample size per project category limits the statistical power of subgroup analyses.

Future research should investigate long-term maintenance costs of AI-generated codebases, develop architectural patterns that improve AI consistency across microservices, and design culturally adapted AI adoption frameworks for developing economy enterprises. Comparative studies across multiple developing economies would help identify universal versus context-specific factors in AI tool adoption.

Conclusion

This study examined whether AI-assisted development tools accelerate the software development lifecycle and improve code quality testing in enterprise environments, with particular attention to the Libyan context. The experimental results demonstrate that AI tools provide measurable SDLC acceleration—ranging from 38% for small projects to 22% for large systems—with quality outcomes that vary by project scale and testing methodology. For small and medium projects, AI assistance improved implementation velocity and test coverage while introducing manageable increases in code churn and security review requirements. For large-scale enterprise systems, benefits were attenuated by architectural incoherence and integration challenges. The findings reveal that AI tool effectiveness is context-dependent, with developer experience, project complexity, and organizational readiness serving as critical moderating factors. In the Libyan enterprise context, infrastructure limitations, digital literacy variability, and hierarchical management culture emerged as significant barriers that require tailored adoption strategies rather than direct transplantation of Western best practices. Despite these challenges, the study demonstrates that when deployed with appropriate governance, security review, and developer training, AI tools can meaningfully accelerate development while maintaining quality standards. This research contributes to the growing body of empirical evidence on AI adoption in software engineering, particularly within the understudied context of developing economies. It provides practical guidance for enterprise practitioners, highlights the need for scale-dependent AI strategies, and identifies critical success factors for AI tool integration in resource-constrained environments. Future work should focus on architectural frameworks for large-scale AI integration, culturally adapted adoption models, and longitudinal studies of AI-assisted development outcomes.

Acknowledgments

The authors gratefully acknowledge the support of the participating software development enterprises in Tripoli, Misurata, and Benghazi. We extend our sincere appreciation to the EU-funded E-NABLE program and the Digital Lab initiative for providing foundational training resources that facilitated this research. We also thank the Faculty of Information Technology at the University of Benghazi for administrative support and access to research facilities. Special thanks to all 48 developers who dedicated their time and expertise to this study.

Conflicts of Interest

The authors declare no conflicts of interest. No financial support was received from any of the AI tool vendors (GitHub, OpenAI, Amazon) or their affiliates for this study.

Artificial Intelligence (AI) Usage Disclosure

During the preparation of this manuscript, the authors used Grammarly and Microsoft Editor for grammar checking and language refinement. No generative AI tools were used to draft the original content, analyze data, or generate figures. All data analysis was conducted using SPSS version 28.0. The authors take full responsibility for the content of this manuscript.

References

1. Nguyen-Duc A, Cabrero-Daniel B, Abrahamsson P, Arpteg A, Bäck A, Brinne B, et al. Generative artificial intelligence for software engineering—A research agenda. Softw Pract Exp. 2025;55(1):1-25. https://doi.org/10.1002/spe.3312
2. Hou X, Zhao Y, Liu Y, Yang Z, Wang K, Li L, et al. Large language models for software engineering: A systematic literature review. ACM Trans Softw Eng Methodol. 2024;33(8):1-79. https://doi.org/10.1145/3652102

Received*: 25-06-2026 -* ***Accepted****: 23-08-2026 -* ***Published****: 30-08-2026*

3. Ebert C, Louridas P. Generative AI for software practitioners. IEEE Softw. 2023;40(4):31-37. https://doi.org/10.1109/MS.2023.3262346
4. Pressman RS, Maxim BR. Software engineering: A practitioner's approach. 9th ed. New York: McGraw-Hill Education; 2019.
5. Peng S, Kalliamvakou E, Cihon P, Demirer M. The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv. 2023. https://doi.org/10.48550/arXiv.2302.06590
6. Noy S, Zhang W. Experimental evidence on the productivity effects of generative artificial intelligence. Science. 2023;381(6654):187-192. https://doi.org/10.1126/science.adh2586
7. Vaithilingam P, Xie W, Anderson K. Expectations vs. reality: Understanding developer experience with AI-powered code generation tools. In: Proceedings of the CHI Conference on Human Factors in Computing Systems. New York: ACM; 2022. p. 1-15. https://doi.org/10.1145/3491102.3517925
8. Pearce H, Ahmad B, Tan B, Dolan-Gavitt B, Karri R. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. Commun ACM. 2025;68(2):96-105. https://doi.org/10.1145/3696114
9. Garousi V, Felderer M, Kuhrmann M, Herzwurm G. Software engineering for AI-based systems: A survey. ACM Comput Surv. 2025;58(1):1-38. https://doi.org/10.1145/3698580
10. Rasheed Z, Sami MA, Waseem M, Barenkamp M, Rebstadt J, Jabeen F. AI-powered code review with LLMs: Early results. arXiv. 2024. https://doi.org/10.48550/arXiv.2404.18496
11. Libyan Ministry of Communications and Informatics. National digital transformation strategy 2023-2028. Tripoli: Government of Libya; 2023.
12. Oxford Business Group. Misrata targets digital transformation, literacy to improve service provision [Internet]. In: Libya 2024 report. 2024 [cited 2026 Aug 12]. Available from: https://oxfordbusinessgroup.com/reports/libya/2024-report/ict-innovation/transformation-and-literacy-analysis/
13. Tribesquare. Challenges African software engineering teams face and how to overcome them [Internet]. 2024 Mar 12 [cited 2026 Aug 12]. Available from: https://www.tribesquare.co/blog/challenges-african-software-engineering-teams-face-and-how-to-overcome-them
14. Maatalla A. The hidden struggle of software developers in North Africa [Internet]. Medium. 2025 Jan 15 [cited 2026 Aug 12]. Available from: https://medium.com/@abedmaatalla/the-hidden-struggle-of-software-developers-in-north-africa-a-management-crisis-51504b1be194
15. Kergroach S. Emerging divides in the transition to artificial intelligence. OECD Science, Technology and Industry Working Papers. 2025/06. Paris: OECD Publishing; 2025. https://doi.org/10.1787/2e9b2e2a-en
16. Vukovic M, Pan R, Ho T, Krishna R, Pavuluri R, Sinha V, et al. Usage, effects and requirements for AI coding assistants in the enterprise: An empirical study. arXiv. 2026. https://doi.org/10.48550/arXiv.2601.20112
17. 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. 2023. https://doi.org/10.48550/arXiv.2304.10778
18. Grewal B, Lu W, Nadi S, Bezemer CP. Refining ChatGPT-generated code: Characterizing and mitigating code quality issues. ACM Trans Softw Eng Methodol. 2024;33(5):1-26. https://doi.org/10.1145/3636501
19. Siddiq ML, Roney L, Zhang J, Santos JC. Quality assessment of ChatGPT generated code and their use by developers. In: Proceedings of the 21st International Conference on Mining Software Repositories. New York: ACM; 2024. p. 152-156. https://doi.org/10.1145/3643991.3644877
20. Cordeiro J, Noei S, Zou Y. An empirical study on the code refactoring capability of large language models. ACM Trans Softw Eng Methodol. 2024;33(6):1-45. https://doi.org/10.1145/3643676
21. Fakhoury S, Naik A, Sakkas G, Chakraborty S, Lahiri SK. LLM-based test-driven interactive code generation: User study and empirical evaluation. IEEE Trans Softw Eng. 2024;50(8):1-18. https://doi.org/10.1109/TSE.2024.3424956
22. Mock M, Melegati J, Russo B. Generative AI for test driven development: Preliminary results. In: Stray A, Stray V, Moe NB, editors. Agile processes in software engineering and extreme programming. Cham: Springer; 2024. p. 45-60. https://doi.org/10.1007/978-3-031-68340-1_4
23. Smolic S, Grbic D, Huljenic D, Car Z. Systematic comparison of AI-generated and human-written tests: Complementary strengths and quality assessment. Empir Softw Eng. 2026;31(2):1-35. https://doi.org/10.1007/s10664-025-10532-3
24. Damyanov I, Tsankov N, Nedyalkov I. Applications of generative artificial intelligence in the software industry. TEM J. 2024;13(2):123-135. https://doi.org/10.18421/TEM132-14
25. Abuhamra A, Al-Majdoub M. Readiness of Libyan IT organizations for adopting agile development methodologies. J Tech Sci. 2023;19(2):78-102.
26. Al-Fitouri A, Ben Amer M. The digital skills gap in Libyan software development teams: Challenges and proposed solutions. Libyan J Eng Inf Technol. 2024;11(1):45-71.
27. Libyan Technology Foundation. Annual report for 2024. Tripoli: Libyan Technology Foundation; 2024.
28. National Information Center. Building local capacity in emerging technologies: A strategic vision for economic development. Tripoli: National Information Center; 2025.
29. GitHub Research. Does GitHub Copilot improve code quality? Here's what the data says [Internet]. GitHub Blog. 2024 Feb 27 [cited 2026 Aug 12]. Available from: https://github.blog/news-insights/research/does-github-copilot-improve-code-quality-heres-what-the-data-says/

Received*: 25-06-2026 -* ***Accepted****: 23-08-2026 -* ***Published****: 30-08-2026*