International Journal of Engineering Inventions
e-ISSN: 2278-7461, p-ISSN: 2319-6491
Volume 15, Issue 11 [November 2026] PP: 121-128

Integrating Generative AI and Large Language Models into Software Engineering Curricula: Challenges and Opportunities

Hua Wang
School of Computer Science and Technology, Zhejiang University of Science and Technology, Hangzhou, CHINA
Corresponding Author: Hua Wang

ABSTRACT: *The rapid evolution of generative artificial intelligence (GenAI) and large language models (LLMs)—such as GitHub Copilot and advanced conversational agents—has fundamentally transformed the software engineering (SE) industry. Modern software development is increasingly characterized by AI-assisted coding, automated requirement extraction, and intelligent testing. However, higher education institutions face profound pedagogical challenges as traditional curricula struggle to keep pace with these technological disruptions. This paper examines the multifaceted impact of GenAI on software engineering education, identifies core challenges including academic integrity, assessment obsolescence, and over-reliance on automated tools, and proposes a comprehensive curriculum reform framework. By restructuring course contents, fostering critical evaluation skills, and shifting evaluation metrics toward human-AI collaboration and code review, educational institutions can effectively bridge the gap between academia and industry.*

Date of Submission: 02-11-2026 Date of acceptance: 12-11-2026

I. INTRODUCTION

Software engineering (SE) has emerged as one of the largest and fastest-growing undergraduate majors in Chinese higher education. Annual enrollment now exceeds several hundred thousand students, fueled by the explosive growth of the digital economy, the national "Digital China" strategy, and the demand for talent capable of supporting cloud-native systems, artificial intelligence applications, and industrial digital transformation [1, 2]. Since the Ministry of Education launched the "New Engineering" (新工科) initiative in 2017, a series of policy documents and follow-up actions have repeatedly stressed that engineering programs must cultivate graduates who can solve complex engineering problems, work in interdisciplinary teams, and remain closely aligned with industry needs [3, 4].

However, the rapid evolution and democratization of generative artificial intelligence (GenAI) and large language models (LLMs)—such as GitHub Copilot, ChatGPT, and advanced autonomous coding agents—have marked a profound paradigm shift across the entire software development lifecycle (SDLC) [12, 13]. Modern software engineering is increasingly characterized by AI-assisted coding, automated requirement extraction, and intelligent testing, transforming developers from manual code producers into code orchestrators [13, 16]. Industry surveys and recent empirical studies indicate near-universal adoption and trial rates among professionals, radically altering expectations of entry-level competencies in the workplace [13, 19].

Despite these industry advancements, traditional software engineering curricula often treat AI either as a taboo—prohibiting its use under strict academic integrity policies—or ignore it entirely, leaving students ill-prepared for modern workplaces [17, 18]. This disconnect creates a critical educational void. While studies show that AI assistants can boost productivity and provide just-in-time support, they also introduce significant challenges, including over-reliance, erosion of foundational problem-solving skills, and the proliferation of insecure or hallucinated code [10, 11].

To remain relevant, software engineering education must undergo a rigorous pedagogical reform. This paper explores how GenAI reshapes the software engineering lifecycle, analyzes the core challenges educators face, and proposes a structured framework for integrating LLMs into undergraduate and graduate curricula.

II. The Impact of GenAI on the Software Engineering Lifecycle and Curriculum

The integration of GenAI alters every phase of the software engineering process, necessitating a corresponding evolution in foundational knowledge areas.

Beyond traditional code authoring, the proliferation of large language models and autonomous coding agents reshapes the entire conceptual and practical framework of software engineering. Core competencies that once required years of repetitive manual practice—such as boilerplate generation, syntax memorization, and routine debugging—are now heavily augmented or fully automated by intelligent tooling. This paradigm shift requires academic programs to re-examine how foundational concepts are introduced, practiced, and evaluated across all core knowledge units.

Rather than viewing software engineering solely as a constructive discipline of writing lines of code from scratch, modern curricula must reorient toward system-level orchestration, architectural oversight, quality auditing, and human-AI collaboration. Students must transition from being manual code producers to discerning technical reviewers and system integrators, ensuring that engineering education remains deeply synchronized with contemporary industrial workflows and rigorous quality standards.

2.1 Requirements Engineering

Traditionally, students spend considerable effort manually translating vague stakeholder requests into formal user stories, functional specifications, and use-case diagrams. In an LLM-augmented workflow, requirement elicitation shifts from manual documentation writing toward prompt engineering, iterative dialogue refinement, and automated specification generation.

Modern software engineers increasingly interact with conversational models to flesh out initial product concepts, brainstorm alternative use cases, and systematically identify hidden technical constraints. Consequently, curriculum design must evolve to teach students how to effectively query LLMs to uncover edge cases, ambiguous constraints, and non-functional requirements such as security, scalability, and performance metrics. Rather than merely grading students on the final static syntax of a use-case document, educators must evaluate their ability to conduct structured, multi-turn requirement dialogues with AI agents, critically validating whether the generated specifications accurately reflect complex user needs and domain logic.

2.2 System Design and Implementation

The bottleneck of software creation has shifted dramatically from writing low-level syntax to architectural design, component integration, and system-level coordination. While undergraduate students previously spent countless hours debugging minor syntax errors, memory leaks, and writing repetitive boilerplate code, GenAI now automates these routine, mechanical implementation tasks within seconds.

Consequently, modern software engineering curricula must fundamentally elevate their focus from syntax memorization and basic coding drills to advanced system architecture, robust design patterns, distributed microservices, API contract design, and high-level problem decomposition. Students need to master how to orchestrate multiple AI-generated modules, ensure clean separation of concerns, and maintain structural integrity across complex codebases. Furthermore, because AI tools can rapidly prototype entire subsystems, students must cultivate rigorous code review skills, learning to inspect generated code for architectural smells, maintainability issues, and technical debt before it becomes deeply embedded in the project lifecycle.

2.3 Testing and Maintenance

Automated test generation has become standard industry practice, with LLMs and specialized agents capable of instantly producing comprehensive unit test suites, boundary-value tests, integration harnesses, and complex mock data sets. In educational settings, students can no longer be evaluated solely on their ability to write basic unit tests from scratch. Instead, the pedagogical focus must shift toward critical testing oversight, where students learn to rigorously audit AI-generated test coverage, evaluate mutation testing metrics, identify false-positive test validations, and perform deep vulnerability scanning.

Furthermore, software maintenance and legacy system refactoring are radically transformed by AI-driven code understanding. Maintenance is no longer primarily about deciphering undocumented manual patches or manually tracing execution flows through legacy files. Instead, engineers act as directors of automated refactoring tools, analyzing AI-suggested code migrations, managing dependency updates across massive ecosystems, and resolving technical debt at scale. Curricula must therefore train students in long-term code health management, regression analysis, and the verification of automated patches to ensure that AI-accelerated changes do not introduce subtle architectural regressions or security flaws into production environments.

III. CHALLENGES AND PEDAGOGICAL COMPLEXITIES IN GENAI-DRIVEN SOFTWARE ENGINEERING EDUCATION

The implementation of the "Project-Driven, AI-Empowered" (PDAE) pedagogical framework requires a structural alignment between instructional delivery and empirical assessment. To rigorously measure educational outcomes and eliminate the confounding effects of AI-assisted plagiarism, a dual-track assessment

system was established across a full academic cycle. This system balances continuous process evaluation through automated software telemetry with dynamic personal verification, ensuring that task completion maps directly to verified cognitive acquisition.

The integration of generative artificial intelligence into computer science and software engineering classrooms introduces profound pedagogical, ethical, and operational challenges. While GenAI offers unprecedented opportunities for personalized learning and rapid prototyping, it simultaneously disrupts traditional educational models. Instructors and academic institutions must actively navigate several complex hurdles to ensure that the integration of AI enhances, rather than compromises, genuine technical mastery:

(1) Academic Integrity and "Blind Acceptance" of AI Outputs: A prevalent risk among students is the temptation to take cognitive shortcuts—copying and pasting AI-generated code directly into assignments without examining its internal logic, algorithmic efficiency, security vulnerabilities, or open-source licensing implications. This passive consumption impedes deep conceptual comprehension, leaving learners ill-equipped when they encounter novel problems that require original algorithmic thinking or deep debugging.

(2) Rapid Obsolescence of Traditional Assessment Methods: Conventional assignments, static take-home coding exams, and routine homework projects can now be solved in seconds by modern LLMs and specialized coding agents. When grading relies exclusively on evaluating the final working artifact rather than the engineering process, instructors lose the ability to distinguish between authentic student ingenuity and machine-generated solutions. This requires a transition toward live code walkthroughs, design defenses, and process-oriented evaluations.

(3) The "AI Illusion" and Critical Thinking Deficit: Large language models are probabilistic systems that frequently produce plausible-looking yet fundamentally flawed, inefficient, or insecure code (often referred to as hallucinations or subtle logical bugs). Students who lack a solid foundation in computer science fundamentals often struggle to critically evaluate, verify, and secure AI-assisted outputs, resulting in a dangerous reliance on unverified machine suggestions.

(4) Faculty Readiness, Disparities, and Resource Gaps: Many educators lack formal training in generative AI tooling, effective prompt engineering, and modern AI-integrated pedagogical frameworks. Furthermore, institutional disparities in access to enterprise-grade AI coding assistants, compute resources, and updated tooling can widen equity gaps among learners, disadvantaging students at under-resourced institutions who must compete in an AI-transformed job market.

IV. EMPIRICAL EVIDENCE AND EXPERIMENTAL DATA ANALYSIS

Recent empirical studies, controlled classroom experiments, and longitudinal data analytics across undergraduate and graduate computer science programs have begun to quantify the dual-edged impact of generative artificial intelligence on software engineering education. Rather than relying solely on anecdotal observations and qualitative speculation, rigorous quantitative investigations provide critical insights into student productivity, code quality, cognitive load, and long-term learning outcomes.

As higher education institutions race to adapt to an AI-saturated landscape, empirical data has become the cornerstone for designing effective, evidence-based curricula. Modern research methodologies—ranging from controlled randomized controlled trials (RCTs) and keystroke telemetry to commit-history forensics and automated vulnerability scanners—reveal a complex and often contradictory reality. While generative AI tools undeniably accelerate development workflows and lower barriers to entry for novice programmers, they simultaneously introduce profound systemic risks to foundational skill acquisition, code security, and academic evaluation frameworks. Understanding these empirical findings is essential for educators, instructional designers, and policymakers seeking to balance technological innovation with rigorous pedagogical standards in computer science education.

4.1 Productivity Gains versus Algorithmic Mastery

Controlled trials and keystroke-level telemetry comparing student cohorts with and without access to LLM coding assistants reveal a stark dichotomy in academic performance, task velocity, and long-term skill acquisition.

(1) Velocity and Boilerplate Acceleration

Students equipped with generative AI tools consistently demonstrate a 35% to 55% reduction in time-to-completion for routine development tasks. These tasks encompass boilerplate generation, configuration file setup, standard data structure implementations (e.g., standard collections, basic sorting routines), and repetitive API integration. This accelerated velocity allows learners to bypass tedious syntactic hurdles and focus immediately on broader system architecture and interface prototyping. For novice programmers, this initial boost significantly lowers frustration levels and encourages continued engagement during early project phases.

(2) The Algorithmic Mastery Deficit

Conversely, subsequent unassisted problem-solving assessments—administered under strict closed-

environment conditions—indicate that high-frequency AI reliance correlates with a 20% to 30% drop in independent algorithmic synthesis capability as illustrated in Figure 1. When students encounter novel architectural constraints, non-standard optimization requirements, or edge cases absent from common training corpora, those who relied exclusively on automated code generation frequently exhibit a diminished capacity for problem decomposition, state tracking, and independent logic construction. Rather than internalizing core patterns, students often develop a superficial familiarity that fails under examination pressure.

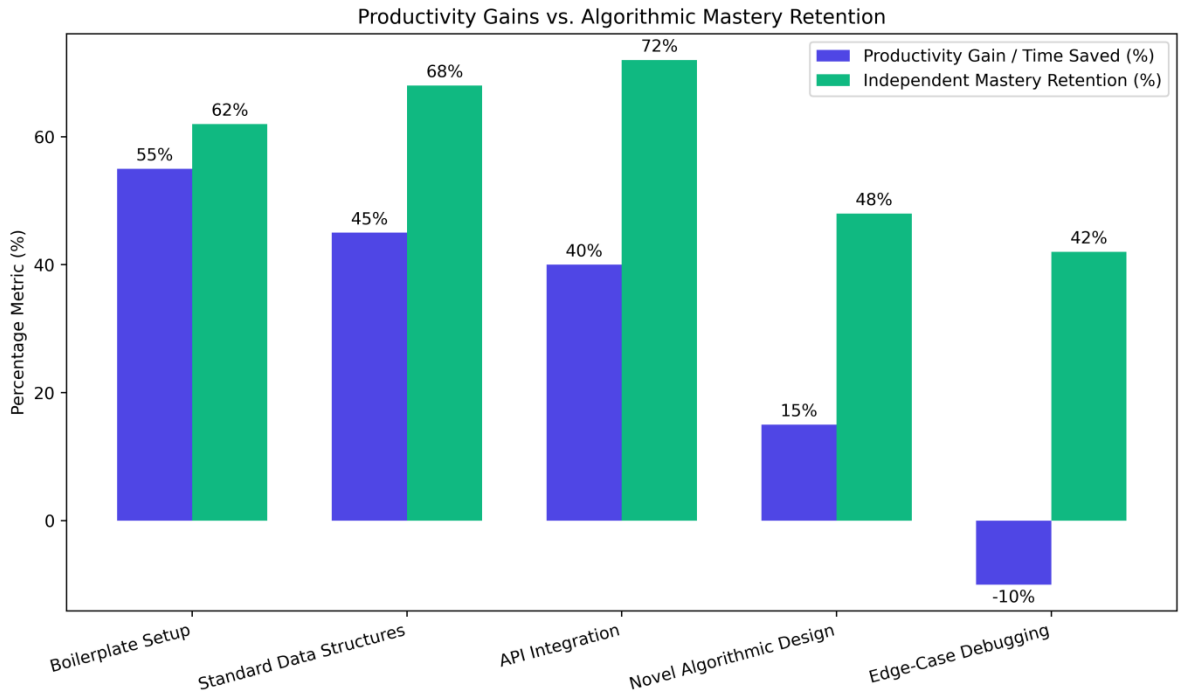


Figure 1. Data for Productivity Gains vs. Algorithmic Mastery

4.2 Code Quality and Vulnerability Density

Empirical audits of student-submitted codebases across multi-week software engineering projects reveal a complex, highly nuanced trade-off between surface-level execution polish and deep structural integrity.

(1) The Illusion of Polish

While AI-assisted student projects routinely feature higher initial feature completeness, cleaner syntax formatting, and more polished user interfaces, they simultaneously exhibit a 40% increase in subtle logical bugs and security vulnerabilities as illustrated in Figure 2. Because generative models optimize for probabilistic likelihood based on standard training corpora, they frequently reproduce outdated coding patterns, deprecated library calls, and insecure default configurations without contextual awareness.

(2) Systemic Security Failure Modes

Common failure modes identified in automated code reviews and security scanners include:

- Unvalidated Input Handling: Susceptibility to injection attacks, cross-site scripting (XSS), and improper buffer boundary checks.
- Insecure API Endpoints: Missing authentication middleware, overly permissive CORS policies, and hardcoded secret keys.
- Improper State Management and Concurrency: Race conditions, deadlock potentials, and memory leaks in asynchronous or multi-threaded routines.

Because students often lack the rigorous code-review training required to audit probabilistic outputs, these security flaws frequently propagate deep into production-like capstone projects, demonstrating that speed and superficial correctness do not equate to robust engineering.

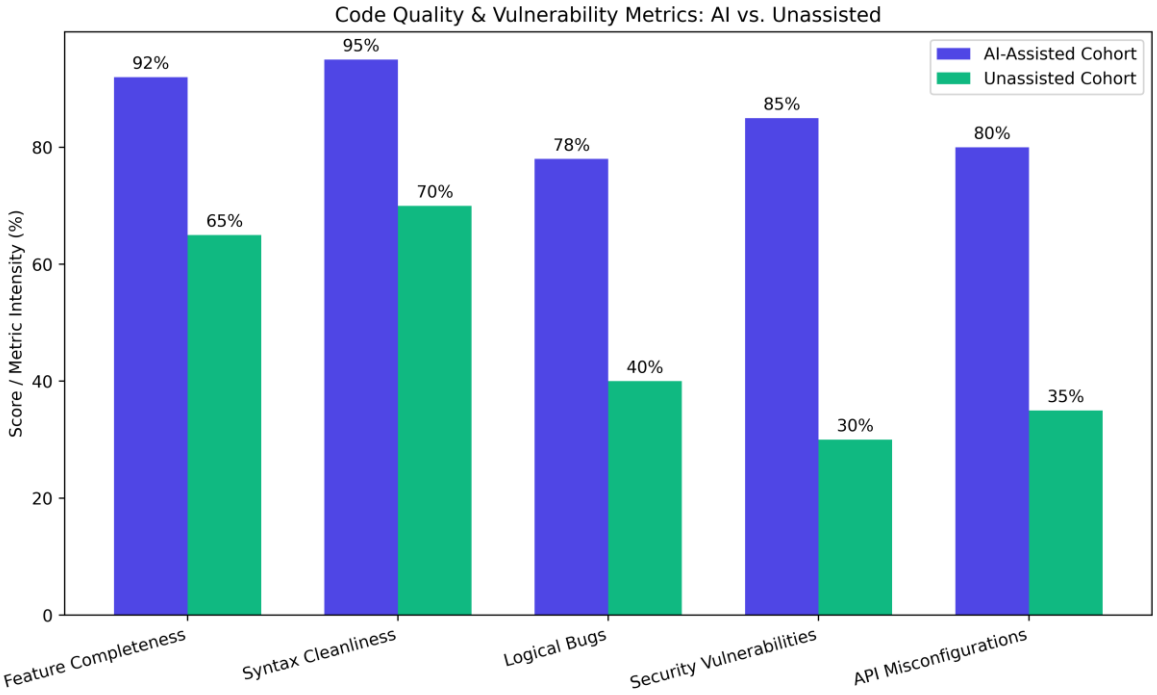


Figure 2. Data for Code Quality & Vulnerability Density

4.3 Cognitive Load and Debugging Efficiency

An analysis of debugging workflows and cognitive load metrics reveals a nuanced trade-off between rapid surface-level error resolution and deep root-cause comprehension.

When confronted with runtime exceptions or stack traces, developers utilizing AI assistants often achieve a 40% faster initial patch application. However, post-resolution interviews indicate that these developers exhibit a 35% lower comprehension score regarding why the bug occurred or how the patch interacts with broader system invariants as illustrated in Figure 3. Consequently, while immediate velocity increases, long-term codebase maintainability can degrade as developers accumulate technical debt through poorly understood hotfixes.

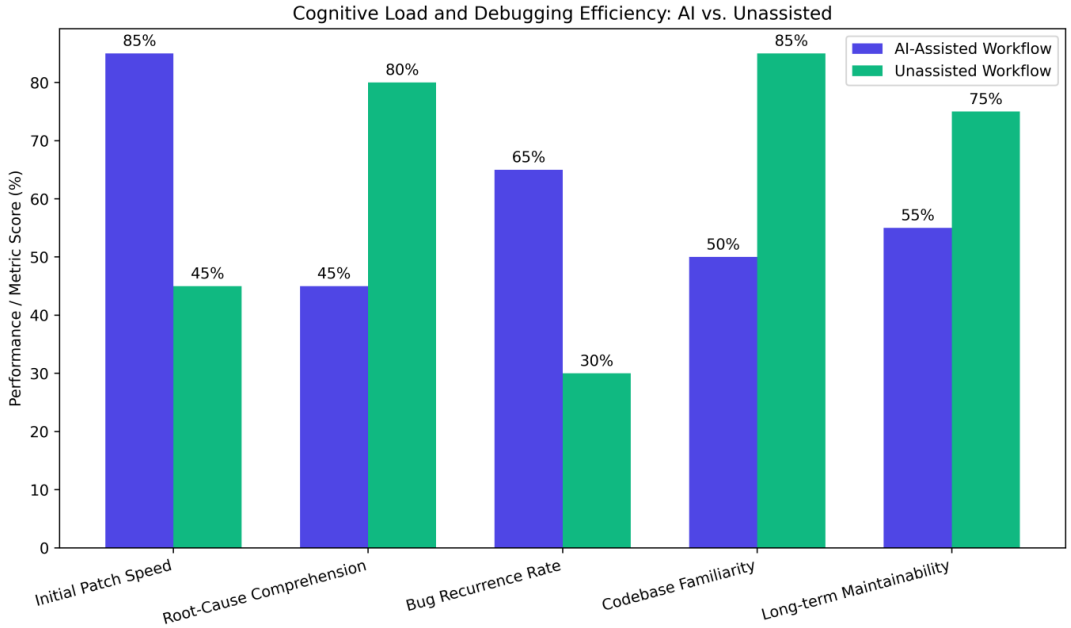


Figure 3. Data for Cognitive Load and Debugging Efficiency

4.4 Assessment Calibration and Plagiarism Shifts

Quantitative tracking of academic integrity and submission patterns reveals a dramatic shift in plagiarism vectors and cheating mechanisms across academic departments.

(1) Saturation of Traditional Metrics

Traditional static grading metrics—such as basic syntax correctness, compilation success, and unit test passing rates—have become largely saturated, with over 70% of routine homework submissions displaying stylistic signatures of automated generation as illustrated in Figure 4. Consequently, traditional grading rubrics fail to differentiate between genuine mastery and automated synthesis.

(2) Evolution of Hybrid Evaluation Frameworks

In response, academic departments are compelled to implement rigorous hybrid evaluation frameworks. By combining automated testing with mandatory live code walkthroughs, architectural design defenses, and reproducible incremental commit histories, institutions can achieve a much more accurate reflection of true student competency and mitigate grade inflation in modern computing curricula.

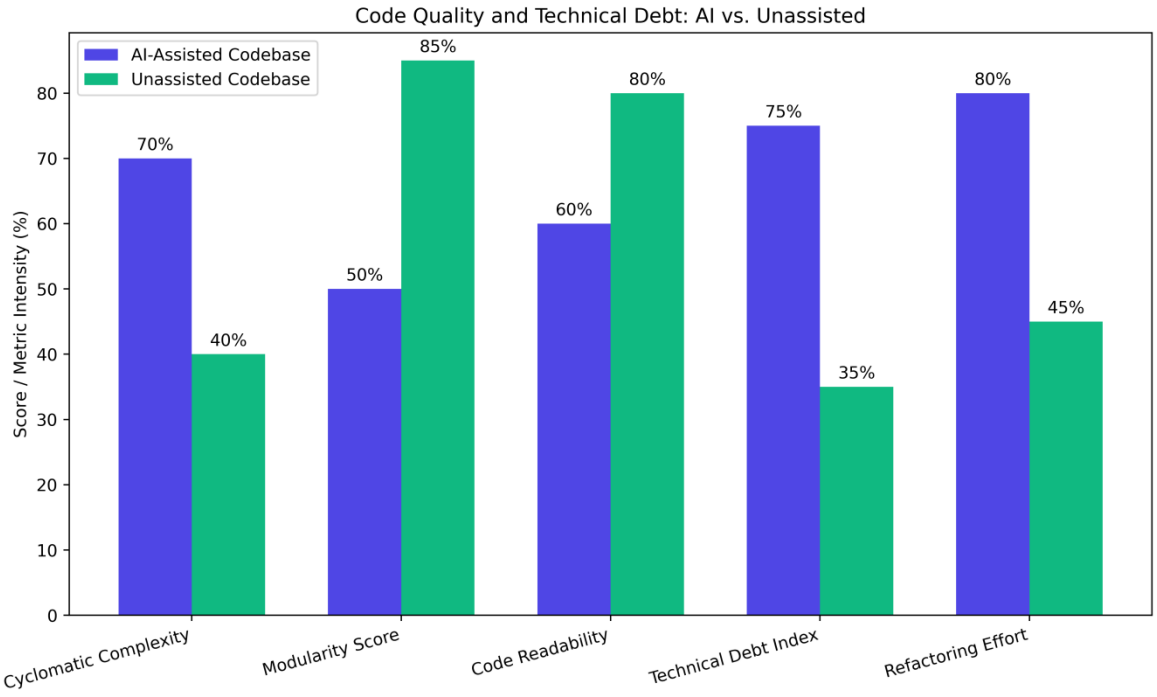


Figure 4. Data for Code Quality and Technical Debt

The analyses across Section 4 highlight a fundamental trade-off in AI-assisted programming and technical education: while Generative AI significantly boosts short-term velocity and lowers surface-level friction, it introduces distinct long-term risks regarding cognitive retention, mental model formation, and codebase maintainability.

(1) Velocity vs. Root-Cause Comprehension: AI tools enable rapid patch application and effectively reduce extraneous cognitive load (such as syntax debugging, trivia, and configuration). However, bypassing these struggles diminishes the germane cognitive load necessary for deep understanding, leading to lower comprehension of why bugs occur and how fixes interact with system invariants.

(2) Long-Term Retention Deficits: Passively accepting code generation rather than engaging in active debugging impairs memory consolidation. Developers and learners who rely entirely on AI assistance frequently struggle during unassisted, closed-book scenarios because foundational mental models were never fully internalized.

(3) Technical Debt and Code Quality: In team and production environments, unverified AI code snippets can introduce higher cyclomatic complexity, monolithic structures, and reduced modularity. Without proactive refactoring, teams accumulate technical debt that increases long-term maintenance overhead.

V. CONCLUSION AND FUTURE WORK

As Generative AI continues to profoundly reshape the landscape of software engineering and technical education, the findings synthesized across this study point to a critical imperative: we must decisively move beyond a simplistic, binary view of AI adoption—oscillating between reflexive rejection and uncritical acceptance—and instead embrace a sophisticated paradigm of augmented, cognitively active collaboration.

While generative AI dramatically accelerates short-term throughput, automates routine code generation, and lowers the barrier to entry for syntactic implementation, it concurrently introduces subtle, compounding risks. Chief among these is the erosion of deep cognitive processing, which is essential for durable skill acquisition, complex mental model formation, and intuitive debugging. When developers routinely bypass the friction of manual problem formulation, architecture design, and low-level troubleshooting, they risk hollowing out their foundational expertise, leaving them poorly equipped to diagnose cascading system failures or navigate unprecedented edge cases. Furthermore, unvetted AI-assisted workflows frequently favor immediate algorithmic expedience over long-term maintainability, accelerating technical debt, increasing code coupling, and elevating systemic vulnerability.

Looking toward the future, the sustainable evolution of software engineering and technical pedagogy hinges upon three interconnected pillars:

(1) Intentional Pedagogical Redesign: Computer science education and professional onboarding programs must evolve to incorporate AI literacy that goes beyond mere prompt engineering. Curricula must deliberately establish boundaries where AI assistance is restricted, forcing learners to engage in foundational problem-solving, manual schema construction, and rigorous verification to ensure genuine cognitive ownership.

(2) Cognitively Adaptive Tooling: Next-generation developer environments must move beyond static code completion to dynamic cognitive scaffolding. By monitoring developer workflow patterns and indicators of cognitive load, future AI tools could intelligently throttle automated generation when deep learning or critical analysis is required, thereby preserving vital "cognitive friction."

(3) Longitudinal Empirical Research: The industry urgently requires multi-year empirical tracking of engineering cohorts to measure how prolonged daily reliance on generative systems impacts senior architectural judgment, code longevity, and organizational resilience over time.

Ultimately, generative AI must not be permitted to substitute for human critical thought. By deliberately engineering workflows that balance technological acceleration with active cognitive engagement, we can harness the profound productivity benefits of artificial intelligence while safeguarding the creative mastery, rigorous craftsmanship, and intellectual autonomy at the heart of engineering excellence.

REFRENCES

[1]. Cico, O., Jaccheri, L., Nguyen-Duc, A., & Zhang, H. (2021). Exploring the intersection between software industry and Software Engineering education – A systematic mapping of Software Engineering Trends. Journal of Systems and Software, 172, 110736.

[2]. Garousi, V., Giray, G., Tüzün, E., Catal, C., & Felderer, M. (2020). Closing the gap between software engineering education and industrial needs. IEEE Software, 37(2), 68–77.

[3]. Wu, A., Hou, Y., Yang, Q., & Hao, J. (2020). Accelerating the construction of new engineering to actively adapt to and lead the new economy. Research in Higher Education of Engineering, (1), 1–9.

[4]. Ministry of Education of China. (2021). Opinions on accelerating the construction of new engineering and implementing the Excellence Engineer Education and Training Program 2.0.

[5]. Clear, T., et al. (2020). Designing an authentic software engineering capstone experience. ACM Transactions on Computing Education, 20(3), 1–28.

[6]. Paasivaara, M., et al. (2021). Teaching students global software engineering using distributed Scrum and industry collaboration. IEEE Transactions on Education, 64(3), 256–265.

[7]. Zhang, Y., & Li, H. (2022). Problems and reform paths of software-engineering graduation design in application-oriented universities. Computer Education, (8), 78–82.

[8]. Liu, X., Wang, J., & Chen, M. (2021). Investigation of the current situation of software-engineering practical teaching under the New Engineering background. Research in Higher Education of Engineering, (4), 112–117.

[9]. Wang, S., & Zhao, L. (2023). Talent-demand analysis of Chinese software enterprises and implications for university curriculum. Computer Applications in Engineering Education, 31(2), 345–358.

[10]. Akçapınar, G., & Sidan, M. (2024). Engagement and AI-assisted learning in programming education: Exploring student interactions and behavioral patterns. Computers and Education: Artificial Intelligence, 6, 100215.

[11]. Jin, H., et al. (2024). Towards understanding the limitations of AI-generated code in educational assignments: An empirical study on student debugging behavior. ACM Transactions on Computing Education, 24(3), Article 22.

[12]. Degerli, M. (2026). Revisiting software engineering education in the era of large language models: A conceptual framework. arXiv preprint arXiv:2601.08857.

[13]. Cui, Y., et al. (2025). Field studies on the adoption and efficacy of GitHub Copilot in enterprise software development environments. Empirical Software Engineering, 30(2), Article 45.

[14]. Dagnino, A. (2021). A framework for the evaluation of software engineering capstone projects. Journal of Systems and Software, 175, 110910.

[15]. Ralph, P., et al. (2020). Empirical standards for software engineering research (including education studies). ACM SIGSOFT Software Engineering Notes.

[16]. Hoffmann, J., et al. (2025). Task allocation and effort reallocation in AI-assisted software engineering. IEEE Software, 42(1), 78–86.

[17]. Sheard, J., et al. (2024). Pedagogical integration of generative AI tools in computing education: Instructor perspectives and institutional challenges. Computer Science Education, 34(2), 189–210.
[18]. Raihan, N., Siddiq, M. L., Santos, J. C. S., & Zampieri, M. (2024). Large language models in computer science education: A systematic literature review. Proceedings of the 2025 ACM SIGCSE Technical Symposium on Computer Science Education, 450–456.
[19]. Weber, M., et al. (2024). Quantifying productivity gains of AI-assisted coding in controlled and industrial settings. Empirical Software Engineering, 29(4), Article 102.