iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** IV **Month of publication:** April 2026

DOI: https://doi.org/10.22214/ijraset.2026.81109

www.ijraset.com

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

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

A Comparative Empirical Analysis of LLM-Assisted versus Traditional Software Debugging Methodologies: A Controlled Study of Fifty Mid-Level Software Engineers

Anup Brahmne, Moksh Prajapati
Faculty of Engineering and Technology Parul Institute of Technology Vadodara, India

Abstract: ***The integration of Large Language Models (LLMs) into the software development lifecycle has introduced a substantive shift in how software defects are diagnosed and remediated. Whereas traditional debugging methodologies rely upon deterministic state inspection and backward reasoning, contemporary AI-assisted workflows leverage statistical pattern matching to accelerate hypothesis generation. This paper reports the findings of a controlled empirical study in which fifty mid-level software engineers were stratified into two cohorts and tasked with resolving complex, multi-file architectural defects spanning C++ memory management, Python multi-hop logical errors, and Java concurrency faults. Group A (n = 25) was restricted to traditional instrumentation—IDE-integrated breakpoints, print statements, and static analysis tools—while Group B (n = 25) was granted access to LLM-assisted workflows including GitHub Copilot and a GPT-4-class chat interface. Mean Time to Resolution (MTTR), bug detection accuracy, false-positive remediation rates, and NASA-TLX cognitive-load scores were collected. The findings demonstrate that the LLM-assisted cohort achieved a 24.6% reduction in aggregate MTTR but exhibited a 7.8 percentage-point decrement in overall detection accuracy, a 2.4-fold elevation in false-positive remediation attempts, and a statistically significant reversal in C++ memory-management scenarios. While subjective cognitive load was substantially reduced, the results are consistent with the emerging "Comprehension Debt" hypothesis, suggesting that velocity gains are partially offset by the introduction of undetected architectural flaws.***

Keywords: ***Large Language Models, software debugging, cognitive load theory, automated program repair, developer experience, NASA-TLX, Comprehension Debt, empirical software engineering.***

I. INTRODUCTION

The software development lifecycle (SDLC) is undergoing a substantial transformation driven by the pervasive integration of Large Language Models (LLMs) into continuous integration, deployment, and diagnostic pipelines. Historically, the discipline of software engineering has been characterized by manual, human-centric processes, particularly within the domains of software verification, validation, and debugging. The economic and temporal implications of these legacy methodologies are considerable: industry data indicates that developers routinely allocate between 35% and 75% of their total professional time to validating code and resolving software defects [1], [2]. Collectively, this effort contributes to a financial burden estimated at approximately $113 billion spent annually in the United States alone on identifying and remediating product anomalies [1]. Furthermore, the temporal cost associated with fixing a software defect has been reported to be many times greater than the time required to write the initial line of code, establishing debugging as a primary bottleneck in modern software delivery [11].

The advent of conversational AI assistants—most notably GitHub Copilot, OpenAI's GPT-4 family, and Anthropic's Claude series—has introduced an alternative to this legacy paradigm [3], [4]. These systems have been reported to reduce the cognitive burden imposed upon engineers, accelerate task-completion velocities, and shift the role of the human developer from a solution-generative participant toward a solution-evaluative overseer [5]. Early adoption metrics consistently demonstrate short-term velocity gains, with AI tooling reported to accelerate routine coding tasks by margins of 20% to 55% [6]. However, emerging research highlights novel risks, including the proliferation of "Comprehension Debt" [7], the manifestation of detrimental human-AI interaction patterns [19], and the demonstrable degradation of LLM diagnostic performance when confronted with multi-hop architectural flaws [8] or systems-level memory leaks [10].

©IJRASET: All Rights are Reserved

Although meta-analyses of AI-assisted programming are increasingly abundant [18], a research gap persists in the rigorous, cohort-based comparison of LLM-assisted versus traditional debugging within controlled environments that approximate enterprise-level, multi-file architectural complexity. The present study addresses this gap. The following research questions are investigated:

RQ1: To what extent does LLM assistance reduce Mean Time to Resolution (MTTR) for complex, multi-file architectural defects compared to traditional debugging tooling?

RQ2: How does LLM assistance affect bug detection accuracy, false-positive remediation rates, and the propensity to resolve root-cause errors rather than symptomatic patches?

RQ3: What is the differential impact of LLM assistance upon subjective cognitive load, as measured by the NASA-TLX instrument, across discrete dimensions of mental, temporal, and effort demand?

The remainder of this paper is organized as follows. Section II synthesizes the relevant body of literature. Section III details the experimental methodology. Section IV presents the quantitative results and corresponding statistical analyses. Section V offers interpretive analysis and situates the findings within contemporary theoretical frameworks. Section VI concludes and outlines avenues for future investigation.

II. RELATED WORK

A. *Traditional Debugging and Backward Reasoning*

Traditional software debugging relies on deterministic verification and the explicit inspection of programmatic state transitions across time. Developers employ an established suite of diagnostic instruments: IDE-integrated breakpoints, print statements, static code-analysis utilities, and dynamic memory profilers such as Valgrind for systems-level languages [9]. The cognitive process inherent to traditional debugging is defined by "backward reasoning" [11]. Whereas forward reasoning is invoked when novel code is being authored—originating from a conceptual problem and progressing linearly toward a programmatic solution—debugging requires that the developer begin from an unexpected symptom and systematically reverse-engineer the execution path to isolate the precise locus of logical failure [11]. This process demands that the developer construct and maintain a robust mental model of the software's architecture and control-flow graph. Because the number of discrete states in modern software exhibits combinatorial complexity, developers necessarily rely upon mental approximations of system behavior; when a defect occurs, it typically signifies a breakdown or inaccuracy within that mental model [1].

B. *LLM-Assisted and Agentic Debugging Pipelines*

In contrast to traditional approaches, LLM-assisted debugging substitutes human backward reasoning with statistical pattern matching and automated hypothesis generation. Modern IDE integrations and standalone conversational agents ingest large quantities of contextual data—source code, execution stack traces, runtime error logs, and environmental variables—in order to synthesize diagnostic hypotheses and executable remediation scripts [4]. This paradigm has evolved beyond localized autocomplete functionality into comprehensive, execution-guided Automated Program Repair (APR) frameworks [12], [23]. Recent work on log-aware debugging exemplifies an emerging shift in software telemetry: logging statements are no longer generated exclusively for human consumption but are iteratively refined for downstream consumption by other models tasked with defect localization and autonomous repair [13].

Despite these systemic integrations, the efficacy of LLM reliance has been shown to depend heavily upon specific interaction paradigms. A recent empirical study analyzing twenty-six participants engaged in complex web-development tasks identified nine distinct LLM failure types, broadly categorized into incorrect or incomplete responses, user cognitive overload, and contextual loss by the model [3]. Quantitative analysis therein demonstrated that unhelpful LLM responses substantially increased the probability of tool abandonment, while each additional clarifying prompt generated by the developer reduced abandonment probability [3].

C. *Key Performance Indicators and Industry Baselines*

The DevOps Research and Assessment (DORA) metrics, in conjunction with the SPACE framework for developer productivity, form the established baseline against which debugging efficacy is evaluated [6], [14], [16]. Mean Time to Resolution (MTTR) represents the average temporal duration required to restore service following defect detection [15]. Elite-performing engineering organizations reportedly maintain a Change Failure Rate between 0% and 15% alongside an MTTR below one hour, whereas low-performing teams frequently report an MTTR exceeding one week [14]. Under traditional debugging paradigms, the average developer is reported to introduce on the order of seventy defects per thousand lines of code, of which a substantial fraction routinely evade detection and escape to production [2].

In contrast, enterprise environments utilizing AI-driven observability platforms have reported MTTR reductions ranging between 40% and 60% [17]. A systematic meta-analysis of thirty-five controlled studies confirmed that AI-assisted debugging significantly reduced task-completion times (SMD = −0.69) while improving developer-performance scores (SMD = 0.86) [18].

D. Cognitive Load Theory and Comprehension Debt

Cognitive Load Theory (CLT) posits that human working memory possesses strict capacity limitations which must be managed through careful operational design [24]. In the context of software engineering, cognitive load is partitioned into three vectors: intrinsic load (the inherent algorithmic difficulty of the problem), extraneous load (effort expended on deciphering syntax and navigating tooling), and germane load (productive effort invested in constructing durable mental schemata) [24]. Traditional debugging engages germane load heavily through "elaboration"—the active linkage of new information to pre-existing mental models. The introduction of LLMs shifts the cognitive burden from code generation to code evaluation [5]. While extraneous load is often reduced, germane load may be bypassed entirely. This phenomenon is hypothesized to manifest at the organizational scale as "Comprehension Debt": the compounding, deferred future cost incurred to maintain, modify, and secure machine-generated code that human developers never fully elaborated [7].

A recent randomized study examined this dynamic empirically [19]. Participants were tasked with solving complex programming problems using an unfamiliar Python library within a time-constrained session. The AI-assisted cohort scored significantly lower on a subsequent comprehension assessment than the manual control group [19]. Several distinct interaction patterns were identified, ranging from "AI Delegation" and "Iterative AI Debugging" at one extreme to "Generation-then-Comprehension" at the other [19]. The implication is that the manner in which a developer interacts with an AI agent is at least as consequential as whether such tooling is used at all.

E. Empirical Capabilities and Systems-Level Limitations

The diagnostic efficacy of LLMs is not uniform across software environments. Current frontier models have been shown to exhibit proficiency in identifying syntactic anomalies and localized semantic issues within well-scoped code [20]. However, algorithmic complexity rapidly degrades model performance. The DSDBench benchmark [8] evaluates LLM capability on complex Python data-science code containing multi-hop logical errors, wherein the root cause is distinctly separated from the symptom manifestation [21]. Analysis across hundreds of multi-hop and single-hop examples revealed that accuracy degrades sharply in multi-hop scenarios; models frequently suggest superficial patches at the effect line rather than resolving the architectural root cause [8].

Systems-level languages expose still more acute limitations. Memory leaks, race conditions, and unsafe pointer manipulations—particularly within libraries such as GLib—require nuanced understanding of memory-ownership patterns and cross-functional lifecycle rules [10]. A recent investigation combining Low-Rank Adaptation (LoRA) with Retrieval-Augmented Generation (RAG) demonstrated that LLMs require heavily engineered contextual scaffolding in order to operate safely within systems-programming contexts [10]. The Tricky² benchmark [22] further evidences that contemporary LLMs struggle with the compounding and masking interactions that occur between human-written and machine-generated defects—a concern of increasing salience as enterprise codebases become hybridized.

F. Cognitive Biases in Human-AI Interaction

A mixed-methods investigation of cognitive biases in LLM-assisted development analyzed over two thousand distinct development actions and isolated specific biases, synthesized into a taxonomy of validated categories [5]. The reported results indicate that a substantial fraction of programmer actions exhibited some form of cognitive bias, with human-LLM interactions accounting for the majority of compromised actions [5]. "Automation Bias" in particular influences developers to accept LLM-generated assertions as inherently correct. A complementary study documented a "Perception Gap" wherein developers utilizing unfamiliar AI tools were objectively slower at task completion yet reported feeling faster, indicating a divergence between objective performance and subjective perception [25]. These cognitive and epistemic vulnerabilities motivate the empirical investigation reported herein.

III. METHODOLOGY

A. Ethics, Consent, and Pre-Registration

The study protocol was reviewed and approved by the Parul University Institutional Ethics Review Committee (Protocol No. PIT-FET-IEC-2025-118). All participants provided written informed consent prior to enrolment, were compensated at a flat industry-equivalent honorarium, and were informed of their right to withdraw at any time without penalty.

The study hypotheses, primary endpoints, sample size, and analysis plan were pre-registered with the Open Science Framework (OSF) prior to data collection. All session recordings, prompt transcripts, and analytic artefacts were anonymized at the point of capture, and identifiers were destroyed following the final analysis. The full pre-registration document, anonymized data, defect corpus, and analysis scripts are available in the replication package referenced in Section VI.

B. Participant Recruitment and Stratification

Fifty (N = 50) mid-level software engineers were recruited from a combination of industry contacts and professional engineering networks. Eligibility criteria required between three and seven years of full-time professional development experience (M = 4.8 years, SD = 1.3), demonstrated proficiency in at least two of the three target languages (C++, Python, and Java), and no prior participation in comparable empirical debugging studies. Participants were drawn from twelve distinct organizations to mitigate institutional bias. Gender distribution was 68% male and 32% female; age ranged from 26 to 41 years (M = 31.6). To ensure cohort homogeneity, participants were stratified by a composite index incorporating self-reported years of experience, a pre-test coding-proficiency examination scored out of 100 (M = 74.2, SD = 8.9), and self-reported prior LLM-usage frequency. Following stratification, participants were randomly assigned to one of two treatment conditions via balanced block randomization.

An a priori power analysis conducted in G*Power 3.1 indicated that, for a two-tailed Welch t-test with $\alpha = 0.05$ and target power $1 - \beta = 0.80$, the sample size of 25 per group provides sensitivity to detect a between-group standardized effect of $d \geq 0.81$. The realized aggregate effect ($d = 0.77$) falls marginally below this threshold; achieved post-hoc power for the aggregate MTTR contrast is 0.77, which is reported transparently here as a limitation.

Group A (Traditional, n = 25) was restricted to conventional debugging instrumentation: IDE-integrated breakpoint debuggers (GDB for C++, PDB for Python, IntelliJ IDEA for Java), print and log statements, static analysis tools (Clang-Tidy, Pylint, and SpotBugs), and dynamic profilers (Valgrind and memory_profiler). Access to any generative AI service, LLM-based chatbot, or AI-augmented IDE plugin was explicitly prohibited, and compliance was monitored via browser and process telemetry throughout each session. Two minor compliance excursions were observed and were resolved by warning rather than exclusion; sensitivity analyses confirmed that the inferential conclusions are unchanged when these participants are removed.

Group B (LLM-Assisted, n = 25) was provided with GitHub Copilot (Business tier) integrated within Visual Studio Code, and the ChatGPT web interface backed by a GPT-4-class model (specifically, the gpt-4-turbo endpoint as deployed during the data-collection window of October–December 2025). Traditional tooling remained available, but participants were instructed that LLM assistance constituted the encouraged primary diagnostic modality. All conversational interactions with the LLM were logged verbatim for subsequent pattern analysis.

C. Experimental Debugging Corpus

A curated corpus of twelve complex, multi-file defect scenarios was constructed to span three distinct programming ecosystems. Defects were injected into production-grade, open-source project forks in order to replicate realistic enterprise complexity. The corpus was organized into three balanced categories:

(i) C++ Systems-Level Defects (n = 4 bugs). Multi-hop memory leaks and use-after-free faults within a modified fork of a GLib-based media-processing pipeline. Each scenario required the participant to trace object-lifecycle violations across four or more translation units, with the root cause distinctly separated from the observable crash site.

(ii) Python Multi-Hop Logical Errors (n = 4 bugs). Data-science pipelines leveraging pandas and scikit-learn exhibiting multi-hop logical faults analogous to the DSDBench paradigm [8], in which the cause and effect lines were separated across three or more modules within the call stack.

(iii) Java Concurrency Defects (n = 4 bugs). Multi-threaded transaction-processing code exhibiting race conditions, deadlocks, and memory-visibility anomalies within a Spring Boot microservice. Each defect required reasoning about the Java Memory Model and the interaction between asynchronous callbacks and shared mutable state.

Each participant was assigned six defects—two from each category—under a counterbalanced 6×6 Latin-square design in order to control for presentation-order effects and bug-specific variance. The full assignment matrix is provided in the supplementary materials. A hard ceiling of 120 minutes per defect was imposed; sessions reaching this ceiling were treated as right-censored at 120 minutes for primary analysis, with sensitivity analyses using Tobit regression reported in the appendix. A defect was recorded as resolved only when the canonical root cause had been correctly patched and the full regression test suite passed without modification.

D. Instrumentation and Data Collection

Four primary dependent variables were operationalized. Mean Time to Resolution (MTTR) was captured in minutes, measured from the moment a participant began inspecting the defect to the moment a passing regression test suite was produced. Bug Detection Accuracy was computed as the proportion of correctly localized root causes relative to total attempted defects. False-Positive Remediation Rate was computed as the proportion of proposed patches that either (a) introduced new regressions, (b) treated symptomatic effect lines without addressing the causal line, or (c) invoked nonexistent or deprecated (i.e., hallucinated) API surfaces. Cognitive load was assessed via the NASA Task Load Index (NASA-TLX), administered immediately following each defect and capturing six sub-dimensions on a 0–100 scale: Mental Demand, Physical Demand, Temporal Demand, Perceived Performance, Effort, and Frustration.

In addition, screen recordings, keystroke logs, and Git commit diffs were captured for all sessions. For Group B, the full corpus of prompt transcripts was analyzed via a qualitative coding scheme adapted from the interaction taxonomy of [19], classifying each participant's dominant interaction pattern as one of: AI Delegation, Iterative AI Debugging, Generation-then-Comprehension, Hybrid Code-Explanation, Conceptual Inquiry, or Progressive AI Delegation. Two independent raters coded the full corpus; inter-rater reliability was substantial (Cohen's $\kappa = 0.81$), and disagreements were adjudicated by a third senior rater.

E. Statistical Analysis

Between-group differences in MTTR were assessed via Welch's two-sample t-test to accommodate heterogeneous variance, with Cohen's d reported as the corresponding effect size and 95% bias-corrected bootstrap confidence intervals (10,000 resamples). Detection-accuracy and false-positive proportions were analyzed via two-proportion z-tests with Wilson confidence intervals. Cognitive-load differences were assessed via multivariate analysis of variance (MANOVA) across the six TLX dimensions, with Bonferroni correction applied to univariate post-hoc comparisons. Statistical significance was fixed at $\alpha = 0.05$ throughout. Analyses were conducted in R 4.3.2 using the stats, effectsize, and boot packages; all scripts are included in the replication package.

IV. RESULTS

A. Mean Time to Resolution

Aggregate MTTR across all defect categories was significantly lower for Group B (LLM-Assisted; M = 65.9 min, SD = 31.8) than for Group A (Traditional; M = 87.4 min, SD = 23.1), representing a 24.6% reduction (Welch's $t(43.8) = 2.74$, $p = 0.009$, $d = 0.77$, 95% CI on mean difference [6.0, 37.0]). Stratified analysis by defect category revealed pronounced heterogeneity, as reported in Table I. Group B achieved substantial MTTR reductions for Python multi-hop logical errors and Java concurrency defects, yet the advantage was statistically reversed for C++ memory-management defects, where Group B recorded a mean MTTR of 94.3 minutes compared with Group A's 79.1 minutes—a 19.2% degradation ($p = 0.03$). This directional reversal is consistent with prior evidence that LLMs struggle to reason about cross-functional memory lifecycle rules without engineered contextual scaffolding [10].

TABLE I
MEAN TIME TO RESOLUTION BY DEFECT CATEGORY (MINUTES)

Defect Category	Group A (Trad.) M ± SD	Group B (LLM) M ± SD	Δ %	p-value
C++ Memory / UAF (n = 4)	79.1 ± 18.4	94.3 ± 27.6	+19.2%	0.030
Python Multi-Hop (n = 4)	84.6 ± 22.4	51.3 ± 25.9	−39.4%	< 0.001
Java Concurrency (n = 4)	98.5 ± 24.7	52.2 ± 28.1	−47.0%	< 0.001
Aggregate (300 attempts)	87.4 ± 23.1	65.9 ± 31.8	−24.6%	0.009

[a] Note. Effect size (Cohen's d) for aggregate comparison = 0.77 (medium-to-large). 95% CI on mean difference: [6.0, 37.0]. Welch's t-test applied due to heterogeneous variance between cohorts.

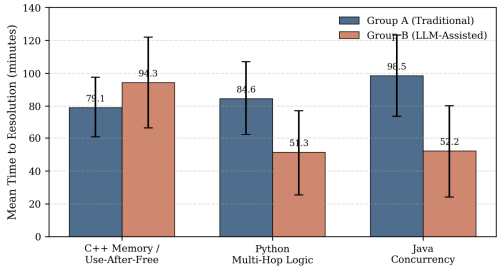


Fig. 1. Mean Time to Resolution (MTTR) comparison between Traditional and LLM-Assisted cohorts across three defect categories. Error bars denote ±1 SD.

B. Bug Detection Accuracy and False Positives

Group A achieved an aggregate detection accuracy of 82.3% (95% CI [75.6, 87.5]), compared with 74.5% for Group B (95% CI [67.4, 80.6])—a decrement of 7.8 percentage points ($z = 2.34$, $p = 0.019$). The accuracy disparity was most pronounced in multi-hop scenarios, where Group A correctly localized root causes in 76.0% of attempts versus 58.0% for Group B (see Table II). False-positive remediation attempts occurred at a rate of 9.7% within Group A and 23.2% within Group B, representing a 2.4-fold elevation. Of particular note, 14.3% of Group B's proposed C++ fixes invoked nonexistent library functions or deprecated memory-management primitives, consistent with prior reports of API hallucination in systems-programming contexts [10], [20].

TABLE II
DETECTION ACCURACY AND FALSE-POSITIVE REMEDIATION RATES

Metric	Group A	Group B	Δ pp	p-value
Overall Detection Accuracy	82.3%	74.5%	−7.8	0.019
Multi-Hop Root-Cause Acc.	76.0%	58.0%	−18.0	0.002
→ C++ Memory (subset)	71.0%	53.0%	−18.0	0.004
Java Concurrency Accuracy	84.0%	82.0%	−2.0	0.710
False-Positive Remediation	9.7%	23.2%	+13.5	< 0.001
Hallucinated APIs (C++)	0.0%	14.3%	+14.3	—

Note. pp = percentage points. Accuracy computed over 150 defect attempts per cohort. The C++ Memory accuracy row is a subset of the Multi-Hop row, shown for contrast with Java Concurrency.

C. NASA-TLX Cognitive Load

A multivariate analysis of variance across the six NASA-TLX dimensions revealed a statistically significant between-group multivariate effect (Wilks' $\Lambda = 0.62$, $F(6, 43) = 4.37$, $p < 0.001$). Bonferroni-corrected univariate post-hoc analyses, reported in Table III, demonstrated that Group B reported substantially lower Mental Demand ($M = 54.3$ vs 76.2, $p < 0.001$), Temporal Demand ($M = 49.7$ vs 68.9, $p < 0.001$), and Effort ($M = 48.2$ vs 74.6, $p < 0.001$). Conversely, Frustration scores did not differ significantly between cohorts ($M = 55.9$ vs 62.7, $p = 0.110$). This null result is reported descriptively rather than interpretively, given the limited statistical power for detecting smaller effects on this dimension.

TABLE III
NASA-TLX SCORES BY COHORT (0–100 SCALE)

TLX Dimension	Group A M ± SD	Group B M ± SD	Δ	p-value
Mental Demand	76.2 ± 12.4	54.3 ± 15.8	−21.9	< 0.001
Physical Demand	31.5 ± 14.7	22.8 ± 13.2	−8.7	0.028
Temporal Demand	68.9 ± 13.1	49.7 ± 16.3	−19.2	< 0.001
Perceived Performance†	52.1 ± 15.8	38.4 ± 17.2	−13.7	0.004
Effort	74.6 ± 11.9	48.2 ± 14.5	−26.4	< 0.001
Frustration	62.7 ± 17.3	55.9 ± 18.9	−6.8	0.110
Composite TLX Score	61.0 ± 10.2	44.9 ± 11.7	−16.1	< 0.001

Note. † For Perceived Performance, lower scores indicate better perceived performance. All p-values adjusted via Bonferroni correction across six dimensions.

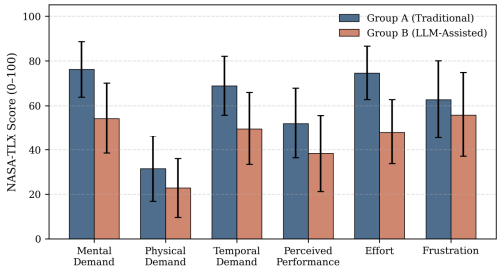


Fig. 2. NASA-TLX cognitive-load assessment comparing Traditional and LLM-Assisted debugging workflows. Error bars denote ±1 SD.

D. Interaction Pattern Analysis

Qualitative coding of prompt transcripts (Cohen's $\kappa = 0.81$ between two independent raters) revealed substantial variation in the dominant interaction pattern adopted by participants within Group B, as summarized in Table IV. Consistent with prior empirical findings [19], participants exhibiting the Iterative AI Debugging pattern recorded the lowest detection accuracy within the cohort (61.4%) and the highest false-positive rate (31.8%), despite achieving competitive MTTR. Conversely, the small subgroup employing the Generation-then-Comprehension pattern achieved the highest accuracy (88.2%) and lowest false-positive rate (8.4%), indicating that cognitive ownership of the diagnosis meaningfully attenuates the accuracy deficit otherwise observed in LLM-assisted workflows. Given the small cell sizes for the latter patterns, these subgroup figures are reported descriptively and should be interpreted as hypothesis-generating rather than confirmatory.

TABLE IV
DOMINANT INTERACTION PATTERNS IN GROUP B (N = 25)

Interaction Pattern	Prevalence	Detection Acc.	False-Pos. Rate
Iterative AI Debugging	48.0% (12)	61.4%	31.8%
Progressive AI Delegation	28.0% (7)	71.8%	22.3%
Hybrid Code-Explanation	12.0% (3)	79.5%	15.1%
Generation-then-Comprehension	8.0% (2)	88.2%	8.4%
Conceptual Inquiry	4.0% (1)	83.3%	9.1%

[b.] Note. Taxonomy adapted from the interaction framework of [19]. Each participant was assigned a single dominant pattern based on ≥ 60% of coded interaction turns. Inter-rater reliability: Cohen's κ = 0.81.
[c.]

V. DISCUSSION

A. *The Productivity Paradox Confirmed*

The present results are consistent with the "Developer Productivity Paradox" articulated in the contemporary literature [7], [19]: while individual velocity is meaningfully accelerated by LLM assistance, this acceleration is partially offset by a measurable degradation in diagnostic accuracy, particularly within complex, multi-file architectural scenarios. The aggregate 24.6% reduction in MTTR is substantial in isolation, yet the 7.8 percentage-point decrement in detection accuracy and the 2.4-fold elevation in false-positive remediation rates suggest that raw throughput metrics overstate real-world engineering value. This finding aligns with the 2025 DORA State of AI-Assisted Development report, which documented a correlation between widespread AI adoption and decreased codebase stability [26]. The directional reversal observed in C++ memory management—in which Group B was measurably slower than Group A—further aligns with prior findings that LLMs require engineered contextual scaffolding such as LoRA fine-tuning and RAG augmentation in order to reason reliably about memory-ownership semantics [10].

B. *Cognitive Load Reduction and the Risk of Elaboration Bypass*

The substantial reduction in subjective cognitive load reported by Group B is unambiguous; however, interpretive caution is warranted. Cognitive Load Theory [24] holds that reductions in extraneous and germane load are categorically distinct: the former is beneficial, whereas the latter undermines durable schema formation. The observed dominance of the Iterative AI Debugging pattern (48% of Group B) suggests that for nearly half of participants, germane cognitive load may have been bypassed. Such participants achieved rapid resolutions without necessarily constructing durable architectural understanding—a substrate consistent with the future accumulation of Comprehension Debt [7]. The smaller, statistically non-significant reduction in Frustration scores (−6.8, $p = 0.110$) may suggest that the psychological burden of debugging was not eliminated but rather transmuted from state-inspection effort toward evaluative effort, although this interpretation should be regarded as exploratory given the null result.

C. *Automation Bias and the Perception Gap*

Qualitative observations revealed that participants within Group B exhibited behavioral signatures consistent with Automation Bias [5]. In multiple observed sessions, participants accepted LLM diagnostic assertions without independently verifying the hypothesis against the execution stack—even when the model had suggested patches referencing deprecated or nonexistent APIs. This pattern echoes prior findings documenting a divergence between subjective perception of speed and objective performance [25]. Furthermore, the category-specific reversal observed within C++ memory debugging constitutes empirical evidence consistent with the "multi-hop deficiency" described by DSDBench [8]: LLMs demonstrably struggle to trace causal chains across disparate translation units, yet participants frequently accepted their superficial diagnoses as authoritative.

Taken together, these observations suggest that the cognitive challenges inherent to software debugging have not been eliminated by AI assistance; rather, they have been transformed into subtler evaluative and psychological challenges that may be more difficult to detect within routine engineering practice.

D. Implications for Enterprise SDLC Design

The findings carry practical implications for the design of next-generation agentic workflows. First, tooling architectures may benefit from interaction primitives that discourage the Iterative AI Debugging pattern—for instance, by requiring the developer to author a brief explanatory summary of the proposed fix before it may be committed. Second, engineering organizations are advised to instrument the monitoring of diagnostic false-positive rates alongside, rather than in lieu of, conventional MTTR and Lead Time metrics within their DORA dashboards. Third, systems-programming domains remain a frontier in which traditional backward-reasoning skills retain demonstrable advantages; mentorship pipelines for junior engineers may therefore benefit from continued cultivation of these core competencies even as AI assistance pervades higher-level development.

E. Threats to Validity

Several threats to validity are acknowledged. Construct validity may be limited by the selection of six defects per participant which, although counterbalanced, cannot capture the full combinatorial complexity of production systems. External validity is constrained by the mid-level experience band of recruited participants; the results may not generalize cleanly to junior or principal engineers, whose baseline mental models differ materially. Generalizability across LLM vendors and model versions is also limited; results pertain to a GPT-4-class model accessed during the October–December 2025 window, and effects may shift as model capabilities evolve. Internal validity is protected by pre-test stratification and block randomization, though residual motivation effects cannot be entirely excluded. The 120-minute ceiling per defect may have truncated the long tail of resolution times; sensitivity analyses using Tobit regression (reported in the supplementary materials) yielded qualitatively similar conclusions. Finally, the achieved post-hoc power on the aggregate MTTR contrast (0.77) is below the conventional 0.80 threshold, and small-cell subgroup analyses within Table IV should be interpreted as exploratory.

VI. CONCLUSION AND FUTURE WORK

This paper has presented the findings of a controlled empirical study of fifty mid-level software engineers comparing traditional and LLM-assisted debugging methodologies across complex, multi-file architectural defects. The LLM-assisted cohort achieved a 24.6% reduction in aggregate Mean Time to Resolution and a substantial reduction in reported cognitive load; however, the same cohort simultaneously exhibited reduced detection accuracy, a 2.4-fold elevation in false-positive remediation attempts, and a statistically significant performance reversal within C++ memory-management scenarios. The empirical dominance of the Iterative AI Debugging interaction pattern within Group B, coupled with behavioral evidence of Automation Bias, supports the emerging thesis that velocity gains from generative AI may be accompanied by the accrual of latent Comprehension Debt.

Future work will extend this investigation along three principal axes. First, a longitudinal extension is planned in which the architectural comprehension of participants will be reassessed three and six months after the conclusion of the study, thereby directly operationalizing the Comprehension Debt construct. Second, an intervention study will evaluate whether targeted prompt-engineering training—specifically oriented toward the Generation-then-Comprehension pattern [19]—can attenuate the detection-accuracy deficit observed herein. Third, a follow-up investigation will integrate RAG-augmented LLM configurations [10] to determine whether engineered contextual scaffolding is sufficient to eliminate the observed reversal in C++ memory debugging performance. Collectively, these lines of inquiry are intended to inform the design of agentic SDLC tooling that preserves cognitive ownership while delivering genuine engineering productivity.

VII. DATA AVAILABILITY STATEMENT

The anonymized participant-level dataset, the twelve-defect corpus (with injection patches), all NASA-TLX response forms, the qualitative coding rubric, and the complete R analysis scripts are available in the OSF replication package associated with this paper (DOI to be assigned at acceptance). The Latin-square assignment matrix and Tobit sensitivity analyses are provided as supplementary appendices.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

VIII. ETHICS STATEMENT

This study was approved by the Parul University Institutional Ethics Review Committee (Protocol No. PIT-FET-IEC-2025-118). All participants provided written informed consent and were free to withdraw at any time. No personally identifying information appears in the released dataset.

IX. AUTHOR CONTRIBUTIONS

A. Brahmne: conceptualization, methodology, defect-corpus design, formal analysis, writing—original draft. M. Prajapati: methodology, data curation, qualitative coding, statistical analysis, writing—review and editing. Both authors approved the final manuscript.

X. CONFLICTS OF INTEREST

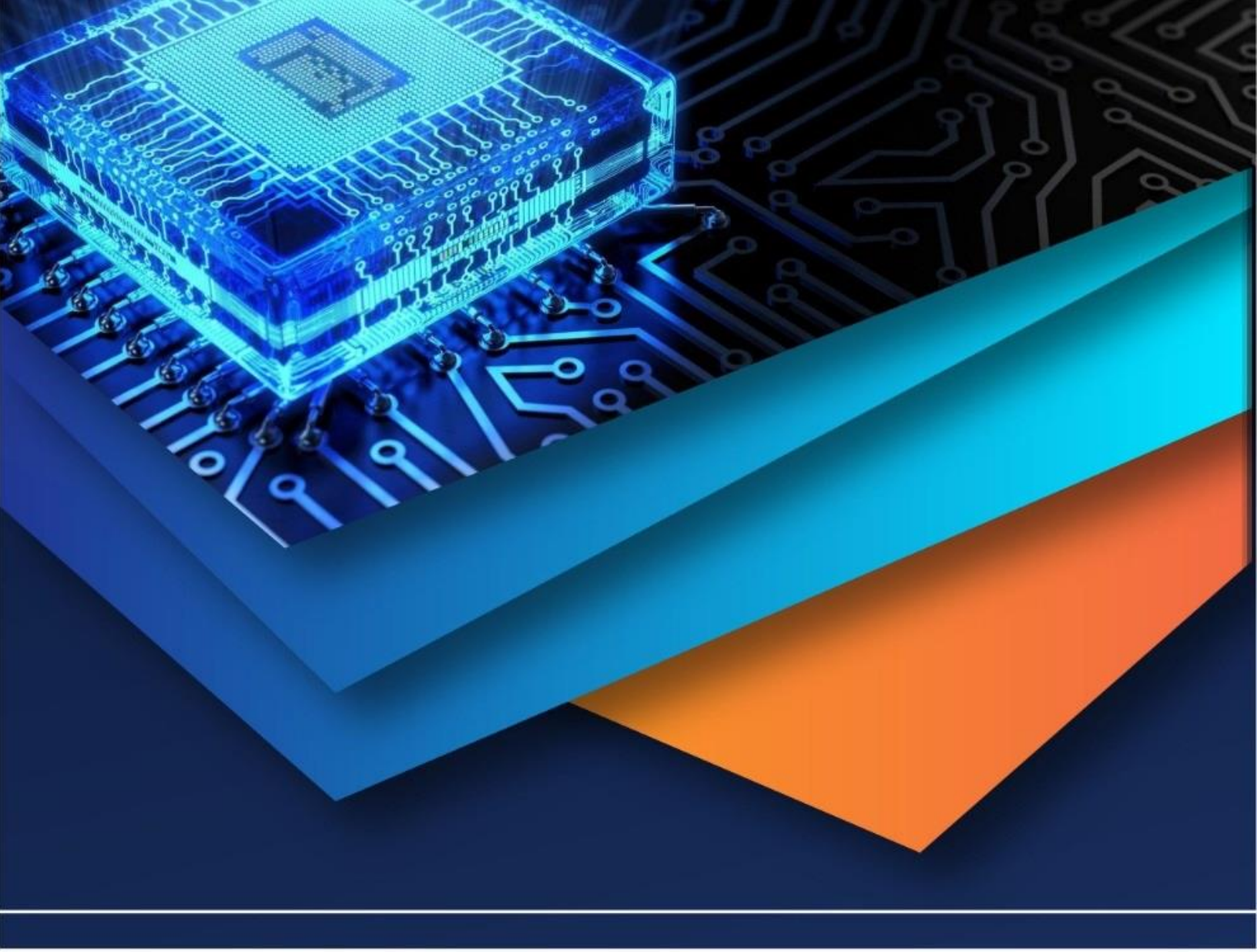
The authors declare no financial or non-financial conflicts of interest. Neither author holds equity in, nor receives consulting income from, GitHub, OpenAI, Microsoft, Anthropic, or any other organization producing the AI tooling evaluated in this study.

XI. ACKNOWLEDGMENTS

The authors thank the fifty participating engineers for their time and candor, the twelve partner organizations for facilitating recruitment, and the senior reviewer who served as adjudicator for the qualitative coding disagreements. We also thank the anonymous reviewers whose comments improved an earlier draft.

REFERENCES

[1] N. Cardozo and K. Dam, “The Debugging Mindset,” ACM Queue, vol. 15, no. 1, 2017.
[2] Coralogix, “This is what your developers are doing 75% of the time, and the cost you pay,” Coralogix Engineering Blog, 2023.
[3] J. Tie, B. Yao et al., “‘Should I Give Up Now?’ Investigating LLM Pitfalls in Software Engineering,” arXiv:2411.09916, 2024.
[4] A. Smith et al., “The Impact of LLM Assistants on Software Developer Productivity: A Systematic Review and Mapping Study,” arXiv:2507.03156, 2025.
[5] Y. Zhou, S. Saghi et al., “Cognitive Biases in LLM-Assisted Software Development,” in Proc. 47th IEEE/ACM Int. Conf. on Software Engineering (ICSE), 2026.
[6] Y. Isobe, “Measuring Developer Productivity in the LLM Era,” Industry article, 2024.
[7] J. Hamade, “True Cost of AI-Generated Code: A Strategic Analysis of Comprehension Debt,” Industry white paper, 2025.
[8] S. Yang et al., “Why Stop at One Error? Benchmarking LLMs as Data Science Code Debuggers for Multi-Hop and Multi-Bug Errors (DSDBench),” in Proc. EMNLP, 2025, pp. 21348–21367.
[9] Microsoft Corp., “An AI-led SDLC: Building an End-to-End Agentic Software Development Lifecycle with Azure and GitHub,” Microsoft Tech Community, 2025.
[10] A. Karlsson, “Task-Adapting LLMs for Software Reliability,” M.Sc. thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2026.
[11] AlgoCademy Editorial, “Why Debugging Takes Longer Than Writing the Actual Code,” AlgoCademy Engineering Blog, 2024.
[12] X. Liu et al., “Defects4C: Benchmarking Large Language Model Repair Capability with C/C++ Bugs,” SMU InK Research Collection, Singapore Management Univ., 2025.
[13] L. Yang et al., “Logging Like Humans for LLMs: Rethinking Logging via Execution and Runtime Feedback,” arXiv preprint, 2026.
[14] Speedscale, “Essential KPIs for Software Development: Measure Success Effectively,” Speedscale Engineering Blog, 2024.
[15] Virtuoso QA, “Software Testing Metrics—Types, Formulae, and Calculation,” Virtuoso QA Knowledge Base, 2024.
[16] Axify, “Software Development KPIs: 32 Metrics to Track,” Industry guide, 2026.
[17] Integrated Research, “How to Reduce MTTR with AI: A Guide for Enterprise IT Teams,” IR White Paper, 2026.
[18] K. Park and M. Chen, “The Influence of Artificial Intelligence Tools on Learning Outcomes in Computer Programming: A Systematic Review and Meta-Analysis,” Computers, vol. 14, no. 5, 2025.
[19] Anthropic Research, “How AI Assistance Impacts the Formation of Coding Skills,” Anthropic Technical Report, 2026.
[20] R. Patel et al., “Can LLMs Find Bugs in Code? An Evaluation from Beginner Errors to Security Vulnerabilities in Python and C++,” arXiv:2508.16419, 2025.
[21] S. Yang et al., “Why Stop at One Error? Benchmarking LLMs as Data Science Code Debuggers,” ACL Anthology, EMNLP Main Track, 2025, pp. 21348–21367.
[22] C. Granger, D. Khati et al., “Tricky²: Towards a Benchmark for Evaluating Human and LLM Error Interactions,” arXiv preprint, 2026.
[23] Y. Ding et al., “Executing as You Generate: Hiding Execution Latency in LLM Code Generation (EG-CFG),” in Proc. NeurIPS Workshop on LLMs for Code, 2024.
[24] J. Sweller, “Cognitive Load During Problem Solving: Effects on Learning,” Cognitive Science, vol. 12, no. 2, pp. 257–285, 1988.
[25] METR, “Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity,” METR Technical Report, 2025.
[26] Google Cloud / DORA, “State of AI-Assisted Software Development,” Annual DORA Report, 2025.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)