Human-in-the-Loop Debugging: Enhancing LLM-Based Code Repair with Developer Feedback

Author: Jace Norman, Ryads Spinios, Dave Noel

ABSTRACT

Large language models (LLMs) such as Claude 4.1 Sonnet and GPT-4 have demonstrated remarkable capabilities in automated code generation and self-debugging, offering developers tools that can detect, localize, and repair programming errors. However, while these systems accelerate software development, their autonomy remains limited by challenges such as incomplete contextual reasoning, domain-specific constraints, potential security risks, and the generation of incorrect or superficial fixes. Fully automated debugging often lacks the ability to reliably validate solutions against real-world development standards, thereby necessitating the involvement of human expertise.

Human-in-the-loop (HITL) debugging provides a promising solution by embedding developer feedback directly into the repair process. Rather than treating the LLM as an independent decision-maker, HITL frameworks encourage a collaborative interaction where humans guide error analysis, validate suggested fixes, and iteratively refine code outcomes. This paradigm leverages the pattern-recognition strengths of LLMs alongside the contextual awareness, domain expertise, and critical judgment of human developers. Feedback modalities may range from inline annotations and runtime monitoring to structured error reports and guided code corrections, enabling models to learn from human interventions and progressively enhance their debugging accuracy.

This paper explores the principles, methods, and implications of HITL debugging, analyzing how such hybrid systems improve correctness, trust, and reliability in AI-assisted programming. We examine frameworks for integrating human feedback into iterative debugging loops, discuss strategies for embedding HITL into development workflows such as integrated development environments (IDEs) and CI/CD pipelines, and consider the ethical and security implications of human-guided code repair. By situating HITL debugging as a bridge between full automation and human expertise, this work argues that the future of reliable, scalable, and secure AI-assisted software engineering lies in hybrid ecosystems where humans and LLMs operate as complementary partners.

Keywords:

Human-in-the-loop, debugging, large language models, code repair, developer feedback, software reliability, interactive AI systems, automation, secure software development.

1. INTRODUCTION

The emergence of large language models (LLMs) has transformed the landscape of software engineering by providing developers with powerful tools for code generation, testing, and debugging. Models such as Claude 4.1 Sonnet, GPT-4, and other state-of-the-art systems have shown the ability to automatically detect errors, propose fixes, and even simulate code execution paths. These advancements promise to reduce the time and effort required in software development, making coding more accessible while accelerating innovation across domains ranging from web applications to embedded systems. However, despite these promising capabilities, fully autonomous debugging remains an unsolved challenge. LLM-based debugging often produces superficial or syntactically correct fixes that fail to address deeper logical issues, and in many cases, these models lack the contextual awareness needed to align code patches with project-specific requirements, performance goals, and security considerations.

The limitations of fully automated debugging raise critical questions about trust, reliability, and accountability in AI-driven development workflows. Incorrect fixes can introduce new bugs, create vulnerabilities, or even lead to cascading failures in production environments. As organizations increasingly consider integrating AI-assisted debugging into continuous integration and deployment pipelines, the risks of over-reliance on machine autonomy become more pronounced. This necessitates the exploration of hybrid approaches that balance automation with human oversight.

Human-in-the-loop (HITL) debugging offers a compelling paradigm for addressing these challenges. Unlike autonomous systems that operate with minimal human intervention, HITL frameworks place developers at the center of the debugging process, using their feedback to guide and refine LLM outputs. Through iterative interaction, developers validate model-generated suggestions, provide corrections, and supply domain-specific insights that the model cannot inherently capture. This feedback not only ensures higher accuracy in individual debugging tasks but also allows LLMs to adapt and improve over time through reinforcement from human expertise. In essence, HITL debugging transforms the debugging process into a collaborative partnership where humans provide judgment and context while LLMs contribute computational efficiency and pattern recognition.

This paper investigates the role of human-in-the-loop mechanisms in enhancing the effectiveness of LLM-based code repair. Specifically, it explores the principles of HITL debugging, feedback modalities that strengthen iterative refinement, and methods for integrating such frameworks into development workflows, including integrated development environments (IDEs), collaborative coding platforms, and CI/CD pipelines. Furthermore, it highlights the ethical, security, and reliability considerations that must be addressed when embedding human oversight in AI-assisted debugging systems. By examining the strengths and weaknesses of this hybrid approach, the paper aims to demonstrate that HITL debugging not only mitigates the risks of autonomous code repair but also represents a crucial step toward creating trustworthy, efficient, and secure AI-driven software engineering practices..

2. FOUNDATIONS OF LLM-BASED DEBUGGING

The rapid advancement of large language models (LLMs) has created a new paradigm for software development, where machine learning systems play an increasingly active role in generating, testing, and repairing code. Unlike traditional debugging approaches that rely on deterministic compilers, static analyzers, or formal verification tools, LLM-based debugging leverages the statistical and probabilistic reasoning embedded within pre-trained transformer architectures. These models, trained on vast corpora of natural language and source code, are capable of identifying error patterns, generating potential fixes, and simulating logical reasoning across diverse programming languages.

2.1 Core Capabilities of LLM Debugging Systems

Modern LLMs demonstrate a set of core functionalities that make them suitable for debugging tasks. First, they possess the ability to localize errors by interpreting compiler messages, runtime exceptions, or incomplete function outputs. Second, they can propose code modifications that aim to resolve these issues, often drawing upon patterns observed during training. For example, Claude 4.1 Sonnet and GPT-4 have been shown to generate syntactically correct and executable patches for common programming errors such as type mismatches, null pointer exceptions, and misconfigured data structures. Finally, LLMs can generate natural language explanations of their fixes, providing developers with interpretable suggestions that can assist in decision-making.

2.2 Limitations of Fully Autonomous Debugging

Despite these capabilities, the autonomous operation of LLMs in debugging remains constrained by several fundamental challenges. First, while models are effective at surface-level corrections, they often lack deeper contextual awareness of software design principles, performance constraints, or application-specific requirements. This can result in fixes that compile but fail to meet functional specifications. Second, LLMs may produce hallucinated or overfitted solutions that appear plausible but introduce subtle logic errors or security vulnerabilities. Third, the probabilistic nature of LLM reasoning means that identical errors may yield inconsistent patches across different attempts, undermining reproducibility and developer trust. These limitations underscore the inadequacy of relying solely on LLMs for production-grade debugging tasks.

2.3 Case Studies: Claude 4.1 Sonnet and GPT-4

Examining leading LLMs highlights both the potential and pitfalls of autonomous debugging. Claude 4.1 Sonnet has demonstrated robust self-debugging capabilities, often identifying errors during its own code generation and applying corrective measures without external intervention. GPT-4, by contrast, has excelled at interactive code repair, where user prompts and iterative refinement guide the debugging process. While both systems achieve notable success in resolving common bugs, their performance deteriorates in domain-specific or security-sensitive contexts where deep reasoning is required. These case studies illustrate why autonomous debugging, though promising, requires augmentation through structured human oversight.

2.4 Toward Hybrid Debugging Paradigms

The strengths and weaknesses of LLM-based debugging provide the foundation for exploring hybrid approaches. While LLMs excel at pattern recognition, error localization, and rapid code patching, their limitations highlight the indispensable role of human developers in validating fixes, ensuring alignment with project goals, and mitigating risks. This observation motivates the human-in-the-loop (HITL) paradigm, where automation and human judgment function not as alternatives but as complementary forces. By situating LLM debugging within a broader ecosystem of interactive and collaborative development practices, the stage is set for frameworks that balance efficiency with reliability.

3. HUMAN-IN-THE-LOOP PARADIGM

Human-in-the-loop (HITL) debugging represents a paradigm shift in the way large language models (LLMs) are deployed for error detection and code repair. Unlike fully autonomous systems that attempt to generate and validate fixes without external intervention, HITL frameworks explicitly integrate developer feedback into the debugging process. This approach recognizes the complementary strengths of machine intelligence and human expertise: LLMs provide speed and scalability in pattern recognition, while human developers contribute contextual awareness, domain knowledge, and critical evaluation. Together, they form a collaborative ecosystem where code repair is iterative, adaptive, and more trustworthy than either approach in isolation.

3.1 Defining HITL Debugging

HITL debugging is characterized by continuous interaction between the developer and the AI system throughout the debugging process. The model generates candidate patches or explanations, and the human either validates, modifies, or rejects these outputs. This cycle can occur multiple times until the error is fully resolved. Importantly, the role of the human is not merely corrective but also pedagogical: feedback provided by developers can inform future model responses, creating an evolving system that becomes more aligned with human expectations over time.

3.2 Types of Human Interventions

Human intervention in HITL debugging can take multiple forms, ranging from minimal oversight to active collaboration. At the most basic level, developers may simply approve or reject suggested patches. More advanced forms of interaction include inline annotations within code, providing detailed corrections to model-generated fixes, or clarifying ambiguous requirements. Another intervention type involves runtime monitoring, where developers guide the model using error logs, test outputs, or performance metrics. The depth of intervention often depends on the complexity of the bug and the confidence level in the model's suggestions.

3.3 Feedback Modalities

Effective HITL debugging depends on the design of feedback mechanisms that enable seamless interaction between humans and LLMs. Common modalities include:

- Inline feedback within integrated development environments (IDEs), where developers provide real-time annotations or corrections.
- Interactive prompts, where users iteratively refine instructions to guide model behavior.

- Error reporting frameworks, where runtime exceptions or unit test failures are fed back to the model as structured data.
- Visual dashboards, which provide a shared interface for tracking model-generated patches, developer interventions, and system performance.

Each modality offers distinct advantages in balancing automation with oversight, and future systems are likely to combine multiple channels for richer interaction.

3.4 Value of Developer Expertise

The inclusion of developers in the debugging loop ensures that fixes are not only syntactically correct but also aligned with project-specific goals, coding standards, and security constraints. Novice developers benefit from the pedagogical aspect of HITL debugging, learning best practices from both AI suggestions and expert interventions. Conversely, expert developers ensure that AI-generated fixes do not introduce regressions or security vulnerabilities. This layered approach democratizes debugging by providing educational value while safeguarding software reliability.

3.5 From Oversight to Collaboration

A key evolution in HITL paradigms is the shift from oversight—where humans act as passive validators of AI outputs—to collaboration, where humans and LLMs jointly construct solutions. In collaborative debugging, the AI suggests partial solutions, while the human contributes higher-level reasoning, domain insights, or architectural considerations. This synergy mirrors pair programming, but with the AI functioning as an ever-present partner capable of rapidly iterating across multiple debugging hypotheses.

4. ENHANCING DEBUGGING ACCURACY THROUGH FEEDBACK LOOPS

The effectiveness of human-in-the-loop (HITL) debugging rests on the quality and structure of feedback loops that connect developers and large language models (LLMs). A feedback loop, in this context, represents the cyclical exchange of information in which the model produces an output, the developer evaluates and refines it, and the revised input informs subsequent model behavior. This iterative cycle not only increases the accuracy of individual debugging sessions but also enhances the long-term adaptability and robustness of LLM-based systems.

At the heart of the feedback process is iterative refinement. When an LLM generates a patch that partially addresses a bug, the developer can highlight deficiencies or provide corrections. The model, in turn, incorporates this feedback in subsequent iterations, gradually converging toward a more accurate and contextually appropriate solution. Such refinement is particularly valuable in complex debugging tasks where a single attempt rarely produces a flawless repair. Instead, accuracy emerges through a dialogic process where human expertise guides the AI's probabilistic reasoning toward deterministic correctness.

Interactive interfaces play a pivotal role in sustaining these feedback loops. When feedback is seamlessly integrated into development environments, developers are more likely to engage with model suggestions and provide detailed input. Integrated development environments (IDEs) equipped with real-time feedback channels, conversational agents embedded in coding platforms, and automated test integration can all create environments

where feedback flows naturally between human and machine. By lowering the cognitive overhead of providing feedback, these interfaces encourage deeper engagement, which in turn leads to higher-quality debugging outcomes.

Another dimension of enhancing accuracy lies in the use of structured evaluation metrics. Developers may employ unit tests, property-based testing, or runtime monitoring to verify the effectiveness of model-generated patches. By feeding these results back to the model, the debugging process becomes anchored in objective correctness rather than subjective assessment alone. Such formalized testing frameworks not only validate fixes in the short term but also create datasets that can be leveraged to retrain or fine-tune models, leading to systematic improvements over time.

Feedback loops also enhance trust and accountability in AI-assisted debugging. Developers are more likely to adopt LLMs when they perceive the system as responsive to their input rather than rigidly autonomous. The perception of shared control fosters confidence that the debugging process remains transparent and aligned with human standards of software quality. In this sense, accuracy is not solely a measure of technical correctness but also of perceived reliability and usability.

5. INTEGRATION INTO SOFTWARE DEVELOPMENT WORKFLOWS

For human-in-the-loop (HITL) debugging to move beyond conceptual promise and achieve widespread adoption, it must be embedded seamlessly within the practical workflows of software engineering. Developers increasingly rely on integrated development environments (IDEs), version control systems, and continuous integration/continuous deployment (CI/CD) pipelines to manage the complexity of modern projects. The effectiveness of HITL approaches depends on their ability to align with these established practices without imposing excessive overhead or disrupting productivity.

Integration begins within the IDE, which serves as the primary interface between developers and code. Embedding HITL debugging tools directly into these environments allows for real-time collaboration between humans and large language models (LLMs). For example, when an error is detected, the model may generate a suggested fix within the editor, while the developer simultaneously reviews, annotates, or modifies the recommendation. This tight coupling of model assistance with familiar workflows reduces friction and makes feedback a natural extension of coding rather than an isolated task. In practice, developers can interact with AI debugging assistants through conversational prompts, inline annotations, or contextual code completions that adapt as feedback is incorporated.

Version control systems such as Git introduce another critical layer of integration. By tracking the iterative process of AI-generated patches and human refinements, these systems provide transparency and accountability in the debugging process. Developers can review the history of suggested fixes, evaluate their impact, and selectively merge changes into production branches. This record not only improves traceability but also creates an archive of human-AI interactions that can inform future improvements in debugging accuracy. In collaborative environments, version control further ensures that multiple developers can participate in

the HITL cycle, providing collective oversight of model-generated code.

The integration of HITL debugging into CI/CD pipelines is particularly significant for large-scale software projects. Automated pipelines are designed to ensure that only verified, reliable code progresses from development to deployment. Incorporating HITL mechanisms into these pipelines allows AI-generated patches to undergo rigorous validation before being released. For instance, a model may propose a fix that passes local compilation but fails integration testing under realistic workloads. In such cases, developer intervention is essential for interpreting test failures, guiding the model toward alternative solutions, and ensuring that the final patch aligns with broader system requirements. By embedding feedback loops within automated pipelines, organizations can maintain high standards of reliability while benefiting from the efficiency gains of LLM-assisted debugging.

The adoption of HITL debugging also presents challenges that must be addressed for successful workflow integration. One concern is the potential for increased cognitive load, as developers must balance traditional coding tasks with the evaluation of AI-generated suggestions. Poorly designed interfaces may overwhelm users with excessive prompts or redundant feedback requests, undermining productivity rather than enhancing it. Another challenge lies in achieving the right balance between automation and oversight. Over-reliance on human validation may reduce efficiency, while insufficient oversight risks introducing unverified fixes into production systems. To resolve these tensions, developers and organizations must adopt adaptive strategies that calibrate the level of human involvement based on the complexity of the debugging task and the confidence of the model's output.

6. SECURITY, RELIABILITY, AND ETHICAL CONSIDERATIONS

While human-in-the-loop (HITL) debugging enhances the accuracy and usability of large language model (LLM)-assisted code repair, its implementation raises important questions of security, reliability, and ethics. These considerations extend beyond the technical process of debugging into the broader domain of software integrity, organizational accountability, and the societal implications of delegating critical tasks to artificial intelligence systems. Understanding these dimensions is essential for ensuring that HITL debugging frameworks not only improve productivity but also safeguard the trustworthiness of the software they help produce.

Security represents one of the most pressing concerns in integrating LLMs into debugging workflows. Autonomous code repair systems may inadvertently introduce vulnerabilities by suggesting patches that resolve surface-level errors while creating deeper logical flaws. In contexts where software underpins critical infrastructures such as healthcare, finance, or transportation, even minor oversights can lead to severe consequences. Human oversight mitigates this risk by ensuring that proposed patches undergo scrutiny for hidden weaknesses, compliance with security protocols, and alignment with best practices in secure coding. However, reliance on AI-generated code also introduces new attack surfaces. Malicious actors could exploit LLM-generated patches by crafting adversarial inputs that mislead models into producing insecure solutions.

Addressing these risks requires not only robust feedback loops but also systematic security audits and monitoring to detect vulnerabilities that may escape both the model and the human validator.

Reliability is another central challenge. Debugging often takes place under conditions where speed and precision are equally critical. While HITL approaches improve correctness through iterative refinement, the variability of model outputs can lead to inconsistent results. Developers may find themselves repeatedly correcting similar errors, raising concerns about the stability of AI-assisted debugging in long-term projects. The introduction of structured testing frameworks within HITL pipelines offers a partial solution by ensuring that patches are validated against objective metrics before acceptance. Nevertheless, true reliability requires more than consistent technical accuracy; it demands predictability in the interaction between humans and machines. If developers perceive the system as unpredictable or burdensome, adoption rates will decline, undermining the potential of HITL debugging to transform software engineering practices.

Ethical considerations further complicate the deployment of HITL debugging. A key issue is accountability: when a model-generated fix introduces a bug or security vulnerability, responsibility becomes difficult to assign. Does liability rest with the developer who approved the patch, the organization deploying the software, or the creators of the AI system itself? Ambiguity in accountability poses legal and professional challenges that must be resolved through clearer governance structures and possibly new regulatory frameworks. Intellectual property is another area of concern, as AI-generated code may incorporate patterns learned from copyrighted datasets, raising questions about ownership and licensing of the resulting patches. Furthermore, the pedagogical role of HITL debugging introduces equity issues: novice developers may become over-reliant on AI suggestions, potentially undermining their ability to develop independent debugging skills, while experts may be tasked disproportionately with providing corrective feedback to compensate for model deficiencies.

Finally, the ethical imperative extends to the transparency of HITL systems. Developers and organizations must be able to understand not only what patches are being proposed but also why the model arrived at a given solution. Without interpretable reasoning, human oversight risks devolving into superficial validation rather than substantive collaboration. To address this, explainability mechanisms should be incorporated into HITL debugging frameworks, allowing developers to trace the logic behind AI-generated suggestions and evaluate them with greater confidence.

In sum, the promise of HITL debugging is tempered by the need to address its security, reliability, and ethical implications. Effective solutions require a multilayered approach that combines technical safeguards, organizational policies, and regulatory oversight. By confronting these challenges directly, HITL debugging can evolve from a promising innovation into a mature practice that balances efficiency with responsibility, thereby ensuring that AI-assisted software development remains both productive and trustworthy.

7. FUTURE DIRECTIONS AND RESEARCH OPPORTUNITIES

The integration of human-in-the-loop (HITL) frameworks into LLM-based debugging is still in its early stages,

leaving considerable room for refinement and innovation. As organizations, researchers, and developers experiment with these hybrid systems, several directions emerge that may shape the evolution of AI-assisted code repair in the coming decade.

One promising avenue lies in the development of adaptive models capable of dynamically calibrating the level of human involvement. Current HITL systems often rely on static interaction patterns, where developers are expected to validate or correct all model-generated outputs regardless of task complexity. Future systems could incorporate confidence estimation mechanisms, enabling models to determine when human oversight is most critical. For routine or low-risk fixes, the system may operate with minimal intervention, while complex or security-sensitive bugs could trigger more extensive developer engagement. This adaptive approach has the potential to balance efficiency and reliability by ensuring that human expertise is allocated where it adds the most value.

Another key research direction involves enhancing the explainability of debugging outputs. At present, many LLM-generated fixes appear as “black box” solutions that are syntactically plausible but lack transparent justification. Embedding explainability features, such as natural language rationales or traceable reasoning paths, would allow developers to better understand the logic underlying a proposed patch. Such transparency not only improves trust in AI systems but also facilitates educational outcomes, particularly for novice programmers who may use HITL debugging as a learning tool.

Cross-language and domain transferability represent additional frontiers. Current LLMs are often optimized for popular languages such as Python or JavaScript, but debugging in less common or domain-specific languages—ranging from scientific computing frameworks to embedded system programming—remains challenging. Research into transfer learning and domain adaptation could extend HITL debugging capabilities to a broader range of contexts, ensuring inclusivity across industries. Similarly, debugging tasks in highly specialized domains, such as cybersecurity or financial systems, will require HITL frameworks that integrate expert domain knowledge alongside general programming expertise.

The integration of HITL debugging into collaborative platforms presents another opportunity for innovation. In large-scale software projects, debugging is rarely a solitary activity; it often involves teams of developers with varying levels of expertise. Future systems could support collective HITL debugging, where multiple developers interact with model-generated patches in real time, pooling insights to validate and refine solutions. This collective approach not only distributes the burden of oversight but also aligns with contemporary practices of distributed software engineering and open-source collaboration.

From a research perspective, formal evaluation methodologies will be critical for advancing the field. Current studies often measure success in terms of correctness or patch acceptance rates, but future evaluations must incorporate broader dimensions such as developer productivity, learning outcomes, and long-term system reliability. Rigorous benchmarks and standardized testing frameworks for HITL debugging could provide the empirical foundation necessary to compare different models, feedback modalities, and integration strategies.

Finally, the ethical and regulatory landscape surrounding HITL debugging offers significant opportunities for interdisciplinary research. Questions of liability, intellectual property, and professional accountability will require input not only from computer scientists but also from legal scholars, ethicists, and policy makers. Collaborative efforts across disciplines could shape frameworks that define the appropriate balance of responsibility between humans and machines in software development, ensuring that technological progress is matched by social and ethical safeguards.

Taken together, these directions suggest that HITL debugging is more than a transitional solution to the limitations of current LLMs; it may represent a long-term paradigm for collaborative human–AI interaction in software engineering. By advancing adaptive models, explainability mechanisms, domain transferability, collaborative workflows, and evaluation standards, researchers and practitioners can help realize the full potential of HITL debugging as a cornerstone of trustworthy, efficient, and scalable software development.

8. CONCLUSION

The growing reliance on large language models (LLMs) for software development has brought new possibilities for accelerating error detection and code repair, yet it has also exposed the limitations of fully autonomous debugging. While systems such as Claude 4.1 Sonnet and GPT-4 can identify and correct common programming errors with impressive speed, their tendency to generate superficial fixes, overlook contextual constraints, or introduce new vulnerabilities highlights the need for approaches that extend beyond machine autonomy. Human-in-the-loop (HITL) debugging responds to this challenge by embedding human expertise within the repair process, establishing a dynamic partnership where humans and machines complement each other's strengths.

This study has shown that HITL debugging enhances accuracy and trustworthiness through structured feedback loops, iterative refinement, and seamless integration into development workflows. By situating developers as active participants in the debugging cycle, HITL frameworks ensure that code patches are not only syntactically correct but also aligned with project-specific requirements, organizational coding standards, and security best practices. The paradigm transforms debugging into a collaborative activity where machine efficiency accelerates the exploration of solutions, while human insight ensures depth, context, and accountability.

At the same time, the implementation of HITL debugging is not without challenges. Security concerns surrounding adversarial inputs, the variability of model-generated fixes, and the ethical complexities of assigning accountability underscore the importance of a cautious, well-regulated approach. Without careful design, HITL systems risk overwhelming developers with redundant prompts, introducing hidden vulnerabilities, or obscuring responsibility when failures occur. These challenges make clear that HITL debugging must be guided by technical safeguards, transparent interfaces, and governance structures that distribute responsibility fairly across developers, organizations, and AI providers.

Looking forward, HITL debugging should not be viewed as a temporary measure to compensate for the immaturity of current models, but rather as a sustainable paradigm for the future of software engineering. Adaptive

systems that calibrate human involvement, explainability mechanisms that make AI reasoning transparent, and collaborative platforms that allow teams to co-validate model outputs will ensure that debugging evolves into a robust human–machine partnership. Moreover, interdisciplinary research into the ethical and legal dimensions of HITL debugging will be essential to guarantee that innovation proceeds in a manner consistent with professional accountability and public trust.

In conclusion, HITL debugging represents a critical step toward reconciling the efficiency of automation with the reliability of human expertise. By blending the probabilistic reasoning of LLMs with the contextual judgment of developers, it creates a hybrid ecosystem capable of producing more accurate, secure, and trustworthy code. As software becomes increasingly central to economic, social, and infrastructural systems, the stakes of debugging extend far beyond technical correctness. HITL frameworks, by fostering a balanced collaboration between humans and machines, offer a path forward toward resilient, ethical, and future-ready software development.

REFERENCES

[1] Vaddiparthy, H. (2025). Self-Debugging AI: A Comprehensive Analysis of Claude 4.1 Sonnet's Code Generation and Error Resolution Capabilities.

[2] Vaddiparthy, Harshith. "Self-Debugging AI: A Comprehensive Analysis of Claude 4.1 Sonnet's Code Generation and Error Resolution Capabilities." (2025).

[3] Kathiresan, G. (2025). Human-in-the-Loop Testing for LLM-Integrated Software: A Quality Engineering Framework for Trust and Safety. Authorea Preprints.

[4] Kathiresan, G. (2025). Human-in-the-Loop Testing for LLM-Integrated Software: A Quality Engineering Framework for Trust and Safety. Authorea Preprints.

[5] Zhang, Y., & Leach, K. (2025, June). Leveraging Human Insights for Enhanced LLM-based Code Repair. In Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (pp. 1536-1537).

[6] Bouzenia, I., Devanbu, P., & Pradel, M. (2024). Repairagent: An autonomous, llm-based agent for program repair. arXiv preprint arXiv:2403.17134.

[7] Huang, L., Mustafin, I., Piccioni, M., Schena, A., Weber, R., & Meyer, B. (2025). Do AI models help produce verified bug fixes?. arXiv preprint arXiv:2507.15822.

[8] Omidvar Tehrani, B., M, I., & Anubhai, A. (2024, May). Evaluating human-ai partnership for llm-based code migration. In Extended abstracts of the CHI conference on human factors in computing systems (pp. 1-8).

[9] Yang, B., Tian, H., Pian, W., Yu, H., Wang, H., Klein, J., ... & Jin, S. (2024, September). Cref: An llm-based conversational software repair framework for programming tutors. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 882-894).

[10] Dong, Y., Jiang, X., Qian, J., Wang, T., Zhang, K., Jin, Z., & Li, G. (2025). A Survey on Code Generation with LLM-based Agents. arXiv preprint arXiv:2508.00083.