10.48047/jocaaa.2026.35.01.85

Mitigating Software Development Failures Through Artificial Intelligence: A Contemporary Analysis

Srinivas Bhargava Jonnalagadda
Independent Researcher, USA

Abstract

The software industry still struggles with developing systems that do what they are intended to do in ways that make sense for human interaction and for the workplace in which they will operate. Established methodologies for developing new software systems are based on overly idealized ideas of how people interact. In contrast, the application of AI technologies to software engineering problems occurs at the intersections between AI and the software development process. For example, AI-based specification processes can identify requirements gaps and issues, whereas the use of generative AI platforms can include support for automated code generation and database query optimization. Full observability coverage can be achieved in distributed systems through automated instrumentation of code. Problems can be diagnosed and fixed quickly. Full coverage can also be provided through AI test generation, which explores paths of execution for defects. An AI-augmented development process enables improvements to each phase of the development process to improve succeeding phases (e.g., an improvement in the specification process improves coding, an improvement in instrumentation improves anticipatory fixes, and improved automated testing produces fewer production defects). These improvements to the standard development process enable organizations to create higher-quality software that better meets real user needs, while reducing cost and overhead.

Keywords: Artificial Intelligence, Software Development Failures, Automated Testing, Code Generation, System Observability

1. Introduction

Software development organizations are plagued by the problem of building applications based on how the user's everyday life and workflow patterns work. Development practices tend to be based on idealized assumptions about end-user application usage patterns, which differ from the way end users work, leading to operational discrepancies that must be categorized as either defects or features after the fact. The Consortium for IT Software Quality has stated that poor software quality is an important economic cost and that poor software practices in industry can be directly related to lower productivity, lower reliability, system instability, and lost competitive advantage [1]. This includes the cost of poor software quality during development, deployment, operation, customer relationships, and market share. The Standish Group findings of the meta-analysis of software project performance data indicated that a meaningful proportion of projects are either late, over budget, canceled, or abandoned, indicating that many contemporary software project development practices are fraught with systemic problems and fundamental disconnects between software project planning and execution [2]. However, recent advancements in artificial intelligence technologies, including generative AI programs and automated reasoning systems, hold the potential to improve specifications, automate software development tasks, improve coverage testing, and provide better observability of systems across the entire software development lifecycle. This article discusses how AI can

10.48047/jocaaa.2026.35.01.85

influence failure rates and the extent to which the functionality delivered meets users' needs when used in all development stages.

Dimension	Characteristics	Organizational Impact
Quality Cost Burden	Substantial economic losses from inadequate practices	Reduced productivity and competitive positioning
Project Success Patterns	Persistent delays and budget overruns	Systemic disconnects between planning and execution
A significant		Resource waste and missed market opportunities
Timeline Adherence	Minority of projects completed as scheduled	Schedule uncertainty and stakeholder dissatisfaction

Table 1: Economic Impact and Project Outcome Patterns in Software Development [1, 2]

2. The Specification Gap: Root Cause of Development Failures

These problems often originate from the requirements specification phase of the software development life cycle. When errors are detected in the requirements phase, they are often propagated into the subsequent phases of the SDLC. Studies examining the origins of software failure have observed that requirements errors are the primary source of downstream error propagation. Requirements errors are exponentially costlier to fix than later-detected errors or those discovered in production operations [3]. The economic costs of specification weaknesses include the costs to edit the error, delay costs, costs of reallocation of resources, and opportunity costs of no longer being the first to market and performing below expectations. Organizations tend to underestimate the cost of anticipating all the requirements of all of the use cases, edge cases, and changing contexts. This specification gap may be due to a lack of stakeholder engagement, ineffective transfer of domain information and knowledge, a lack of attention to nonfunctional requirements, and a lack of expectation of system interactions in complex operational environments.

In terms of the entire software life cycle, requirements are the most common cause of failure of a software project, more so than technical or resource constraints [4]. Projects developing features without welldefined requirements encounter a ripple effect as developers don't realize until after deploying to production that the requirements weren't clear, quality assurance engineers believe the requirements aren't clear enough to test all edge cases, and developers build features into production that won't work for users who understand the feature differently. Writing requirements as static specifications is ineffective because user' needs, and their context of operation, vary widely and are hard to predict. As a result, while such systems may strictly conform, they may be unsuited for use in practice. The ensuing disputes over whether the issues raised are bugs or feature requests are evidence not of communication issues, but of a deeper methodological issue.

Modern AI tools may be able to transform the specification task by identifying recurring gaps in the specification between a sample of past projects, or generating specification skeletons on the basis of alternative usage scenarios. Machine learning algorithms trained on large sets of requirements documents may be able to identify differences between successful and failed projects at the level of requirements. NLP capabilities can allow AI-based tools to identify potential misunderstandings, contradictions, and incompleteness in requirements statements that may have been missed by human reviewers. They can increase the level of a stakeholder's engagement with the development process by providing alternative

10.48047/jocaaa.2026.35.01.85

requirement recommendations, clarifying vague statements, and ensuring coverage of functional and nonfunctional requirements across components of the system.

Specification Issue	Development Phase Impact	Remediation Complexity
Requirements ambiguity	Incomplete stakeholder alignment	Exponentially increasing correction costs
Early-stage errors	Cascading implementation problems	Schedule delays and resource reallocation
Documentation limitations	Static capture of dynamic needs	Misalignment with actual user workflows
Leading failure cause	Surpasses technical difficulties	Fundamental methodological limitations

Table 2: Requirements Phase Defects and Specification Challenges [3, 4]

3. AI-Powered Code Generation and Database Integration

Generative AI tools have disrupted customary methods of programming by successfully translating highlevel specifications into working code. The use of AI coding assistants has been demonstrated to considerably accelerate software development task completion, increasing developer productivity for a variety of tasks, such as code generation, bug-fixing, and documentation generation [5]. The productivity improvement is attributed to the ability of AI aids to write syntactically valid code frames, to propose contextually relevant implementations, and to reduce the cognitive load related to remembering the code syntax or the functionality of library functions, including algorithmic code, design patterns, architectural patterns, etc. Companies that are using AI-assisted software development report faster development and fewer defects in generated code, and increased developer satisfaction because they can concentrate on higher-level design issues rather than concrete implementation details.

Database integration is one of the more promising areas of AI programming because query optimization and database schema design are both advanced areas that few conventional application programmers are experts in. Research has shown that for complex analytics with multiple joins, aggregations, and filters, machine learning can produce execution plans that outperform those produced manually [6]. The systems automatically analyze the distribution of queries and decide how to execute them optimally with the best index and join order types to minimize resource usage and maximize throughput. They automate the query rewriting and optimization process. The systems consider various concurrency levels.

The quality of the generated source code heavily depends on the quality of the specification (as is true for all non-interactive code generators). Here, the task of the AI code-generation tool is to map the specification's functional requirements to a source code implementation based on patterns learned during training. In these cases, ambiguous, underspecified, or imprecise specifications will produce a syntactically correct but semantically flawed code. For this reason, observations of specification processes made in the sections above also apply. The organizations that have most successfully adopted AI-assisted development typically follow complete specification and iterative refinement processes that include human review and refinement of code produced before it is deployed into production systems.

Table 3: AI-Assisted Development Productivity and Database Optimization [5, 6]

10.48047/jocaaa.2026.35.01.85

4. Enhanced Observability Through AI-Driven Instrumentation

Observability is a key concept in the management of reliable software services. Observability allows operational teams to understand and diagnose performance problems and respond to operational incidents. Customarily, observability is implemented in software services by the developer of the service writing logging statements, defining performance metrics, and configuring distributed tracing instrumentation within the source code, generally as secondary to developing features. Furthermore, research into organizational capabilities of high-performing technology organizations finds that stronger monitoring and observability practices may differentiate elite performers from their peers. Elite performers have considerably shorter times to recover from incidents and higher deployment success rates than lower performers [7]. These differences may be due in part to higher levels of observability and thus visibility and responsiveness by operations teams.

Microservices and distributed systems, which distribute the application's functionality across multiple services and infrastructure containers, pose a significant challenge in achieving full observability. Manual instrumentation approaches cannot achieve full observability in dynamic distributed systems because they require cross-team coordination to define instrumentation strategy, as well as maintain consistent metric and log definitions across the application infrastructure. Instrumentation is often incomplete, permitting blind spots where failure can occur, but insufficient evidence is generated to identify the cause and rectify the problem. These issues may lead to longer downtimes and a greater customer impact. AI-based application instrumentation automatically analyzes the application's code, identifying possible execution paths. This involves identifying failure modes and creating the necessary instrumentation to record logs, metrics, and other data without requiring developer involvement [8].

Automated instrumentation systems can analyze static code, runtime performance profiles on traces, and historical information about software failures or downtime to automatically determine the best instrumentation strategy for each class of software. For instance, they can find the code that is under the most stress, the functions that are most likely to throw exceptions or errors, and the external services that would cause problems throughout the system if they failed. The level of instrumentation coverage achieved is typically much higher than that of manually applied instrumentation, freeing engineering teams to focus on building new features rather than operational tooling. In addition to creating instrumented codes, AI models can analyze observability data, correlate observability signals across system components, and trigger predictive alerts before incidents impact production systems. Thus, this makes observability not a reactive diagnostic tool but a preventative tool that allows an organization to detect and remedy issues before they escalate into incidents.

Observability Factor	Elite Performer Characteristic	AI Instrumentation Advantage
Incident response capability	Significantly shorter resolution times	Automated coverage of execution paths
Deployment success rates	Higher reliability metrics	Identification of critical failure points
Distributed system monitoring	Coordinated instrumentation strategies	Static analysis and behavior profiling
Predictive alerting	Proactive issue identification	Anomaly detection and metric correlation

Table 4: Observability Practices and Automated Instrumentation Benefits [7, 8]

10.48047/jocaaa.2026.35.01.85

5. Comprehensive Test Automation and Coverage Assurance

Software testing is a typical verification method, but it commonly does not achieve full coverage of the application due to the complexity of the system and the increase in features being implemented over each iteration of the SDLC. Empirical comparisons of various testing methods have shown that the level of defects found is significantly influenced by the type of testing method used, the coverage metric employed, and the quality of the test case design [9]. However, organizations have financial constraints that limit the extent of testing they can conduct. This trade-off between the depth of the test coverage and the speed of development can lead to insufficient testing of edge cases, error code paths, and integrations that are untested until experienced during runtime in production.

Furthermore, manual testing of modern software systems is impractical for most organizations due to the combinatorial explosion in possible tests that would be required to cover configurations, deployment environments, and user interaction flows available to large systems. There are too many paths through the program for most organizations to provide effective test coverage. Historically, risk-based testing and prioritization of manual tests for high-value testing scenarios have been applied, accepting lower coverage for less critical functionality. Bugs may not be found until the program is in production, where fixing them is more costly and disruptive. One alternative is to use symbolic execution and constraint solving for automating test generation, which systematically explores all paths in the program to provide test inputs that exercise specific branches and check the expected behavior across a wide range of inputs [10].

Artificial Intelligence-based test generation systems build models of program behavior from the source code by identifying branching structures, execution paths, and data dependencies in software applications. Automated test inputs are generated using constraint-solving algorithms to achieve high structural coverage metrics, such as statement coverage, branch coverage, and path coverage in automated test suites. In addition to achieving code coverage, such tests can also target functional requirements and verify that the observed code behavior matches the expected behavior. Such behavior includes not only crashes and exceptions but also semantic errors: situations where code runs but produces the wrong result. Test suites generated in this way tend to have much higher coverage and lower maintenance when application code changes, as compared with manually written tests. Organizations that are using the AIgenerated testing observe that bug counts after deployment are considerably lower, as the bugs are caught by validated automation instead of by customers or incident reporting.

Conclusion

It's difficult to overstate the importance of being able to integrate AI into the software development process. In practice, this provides solutions to many of the industry's endemic failures and problems at every stage of the process, from specifying the requirements through to production deployment and operation. NLP and ML can help with the specification process by identifying any omissions, contradictions, or ambiguities in what is required for a project. These can then be corrected before they propagate through the rest of the development cycle, causing rework. Generative AI platforms can take care of routine implementation tasks so that developers can focus on architectural issues and more difficult problems. Automated instrumentation enables observability across a complex distributed system, making incident investigations proactive rather than reactive. AI-based test generation can reach higher levels of test coverage than human black box testing, systematically exploring the space of application behaviors across different execution paths and operating conditions. Such an approach can lead to compounding effects due to the interrelationships between these capabilities, where changes to one can have ripple effects on others. In addition to using AI

10.48047/jocaaa.2026.35.01.85

tooling, organizations must also consider how to evolve their software development processes and introduce appropriate governance and team capabilities to balance human skill and judgment with AI-supported automation. The most successful implementations maintain human influence over architectural and quality decisions while using AI tools to support pattern matching, automation, and analysis across a breadth of aspects beyond the cognitive capability of humans. How close to eliminating the gap will be a function of AI's future maturity, the randomness of human behavior, and changing conditions within our digital landscape. Organizations that can embed AI into every step of the software development lifecycle and maintain strong engineering discipline will build better software, faster, and with greater customer satisfaction.

References

[1] Herb Krasner, "The Cost of Poor Software Quality in the US: A 2020 Report," Consortium for IT Software Quality, 2020. [Online]. Available: https://www.it-cisq.org/cisq-files/pdf/CPSQ-2020-report.pdf

[2] H. Portman, "Review: Standish Group Chaos 2020 – Beyond Infinity," Henny Pirtman WordPress, 2026. [Online]. Available: https://hennyportman.wordpress.com/2021/01/06/review-standish-groupchaos-2020-beyond-infinity/

[3] Barry Boehm, et al., "Software defect reduction top 10 list," ACM Digital Library, Jan. 2001. [Online]. Available: https://dl.acm.org/doi/10.1109/2.962984

[4] M. Jørgensen and K. Moløkken-Østvold, "How large are software cost overruns? A review of the 1994 CHAOS report," ScienceDirect, 2006. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0950584905001023

[5] Danie Smit, et al., "The impact of GitHub Copilot on developer productivity from a software engineering body of knowledge perspective," ResearchGate, 2024. [Online]. Available: https://www.researchgate.net/publication/381609417

[6] Claude Lehmann et al., "Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective," VLDB Endowment, 2024. [Online]. Available: https://www.vldb.org/pvldb/vol17/p1565-lehmann.pdf

[7] Nicole Forsgren, et al., "ACCELERATE Building and Scaling High Performing Technology Organizations," IT Revolution Press, 2018. [Online]. Available: https://ebooks.karbust.me/Technology/Accelerate%20The%20Science%20of%20Lean%20Software%20and%20DevOps%20Building%20and%20Scaling%20High%20Performing%20Technology%20Organizations%20by%20Nicole%20Forsgren%20Jez%20Humble%20Gene%20Kim.pdf

[8] Thomas F. Düllmann, André van Hoorn, "Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches," ACM Digital Library, 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3053600.3053627

[9] S. Eldh, et al., "A Framework for Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques," IEEE, 2006, Online. Available: https://ieeexplore.ieee.org/document/1691683

[10] Corina S. Păsăreanu, Neha Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode," ACM Digital Library, 2010. [Online]. Available: https://dl.acm.org/doi/10.1145/1858996.1859035