SACJ 38(2) July 2026
Research Article

Eliminating Human Intervention in the Software Development Lifecycle: A Novel Agentic Paradigm for Scalable Infrastructure Maintenance

Sowjanya Puligadda

Uber, USA

ABSTRACT
The persistent requirement for human intervention within software development and maintenance workflows has become the defining productivity bottleneck in modern platform engineering. Despite dramatic advances in AI-assisted code generation, the downstream lifecycle stages of build triage, code review, test authoring, and production incident response continue to impose a linear scaling constraint on engineering organizations: increasing software output requires proportional increases in human labor. This article presents a comprehensive Autonomous Software Engineering framework that deploys AI agents across all phases of the Software Development Lifecycle (SDLC), from requirement synthesis and autonomous code generation through self-healing build repair, AI-driven code review, generative test creation, and autonomous production monitoring. The framework leverages Model Context Protocol (MCP) interfaces to grant agents structured access to CI/CD pipelines, code repositories, test infrastructure, and operational telemetry, enabling end-to-end lifecycle automation without human intermediation for routine operations. Empirical evidence from large-scale deployments indicates that autonomous triage systems can reclaim 30 to 40 percent of engineering bandwidth currently consumed by environmental maintenance, while AI-driven code review reduces human review latency by up to 40 percent. The article examines the architectural requirements, operational risks, and organizational implications of this transition, with particular attention to the governance mechanisms required to maintain quality and safety in fully automated pipelines.

Keywords: Autonomous Software Engineering, Software Development Lifecycle, Agentic AI, CI/CD Automation, Self-Healing Systems, Model Context Protocol, Infrastructure Maintenance

Article history
Received: 30 May 2026
Accepted: 30 June 2026
Online: 12 July 2026

1. INTRODUCTION

The software engineering profession is undergoing a structural transformation driven by the convergence of large language model capabilities with the operational complexity of modern distributed platform infrastructure. While AI tools have dramatically accelerated the upstream phases of software creation, enabling engineers to produce production-quality code drafts in seconds, the downstream lifecycle remains largely unchanged from pre-AI-era practices. Build failures still require human log inspection. Missing test coverage still requires human authorship. Code reviews still require human peers. The result is a productivity paradox in which AI-assisted code generation has outpaced the human capacity to verify, integrate, and deploy that code, creating a new bottleneck at the exact point where velocity would otherwise compound [1].

This bottleneck manifests as what practitioners call the "outer loop" cost of software devel-

Puligadda, S. (2026). Eliminating Human Intervention in the Software Development Lifecycle: A Novel Agentic Paradigm for Scalable Infrastructure Maintenance. *South African Computer Journal,* 38(2), 12–21.

ISSN 1015-7999 (print) ISSN 2313-7835 (online)

opment: the cumulative time engineers spend not writing code but managing the infrastructure surrounding code delivery. Industry-wide benchmarks consistently indicate that engineers at large-scale technology organizations dedicate between 30 and 40 percent of their working hours to routine maintenance activities including build triage, flaky test repair, code review queuing, and environment debugging [2]. At the scale of thousands of engineers, this represents an enormous opportunity cost, talent allocated to sustaining existing systems rather than building new capabil-ities. The fundamental insight motivating autonomous software engineering is that many of these activities are not inherently human tasks but rather structured problem-solving processes that are well within the reasoning capability of current-generation AI agents, provided they are given the right environmental access and operational context.

The architecture that makes this insight actionable is the Model Context Protocol (MCP), which enables AI agents to interact with engineering infrastructure, CI/CD pipelines, code repositories, test runners, and operational telemetry, through standardized, tool-based interfaces. An agent equipped with MCP-mediated access to a failing build log, the code change that triggered the failure, and the repository history of similar failures can perform the triage, diagnosis, and remediation cycle autonomously, without a human ever opening the log viewer [3]. This article presents a comprehensive framework for deploying such agents across the full SDLC, analyzes the operational evidence for their efficacy, and examines the governance, risk, and organizational transformation challenges that the transition entails.

The remainder of this article is structured as follows. Section 2 characterizes the human-in-the-loop friction that defines current SDLC practice and quantifies its cost. Section 3 presents the autonomous SDLC framework phase by phase, from requirement intelligence through production self-healing. The operational impacts of these large-scale deployments will be assessed in Section 4 while the limitations in capability and governance requirements on autonomous operation will be discussed in Section 5. Section 6 examines the long-term organizational and economic implications of the framework, and Section 7 concludes.

2 THE HUMAN-IN-THE-LOOP FRICTION: CHARACTERIZING THE BOTTLE-NECK

Current software development infrastructure is, by design and historical accumulation, optimized for human consumption. The log formats exhibit raw textual traces that are specifically intended for humans to detect patterns of behavior. CI dashboards provides summaries of failure that have been developed to be consumed by engineers. Code review workflows stimulate the process of consciously evaluating the risk and appropriateness of a proposed change by a reviewer (ie. Engineered Solution) based on the judgement of the human reviewer who takes into account every detail necessary to form a precise analysis of that potential change. Thus at the time of forming logic for these communications, the design philosophies that were applied were based on the availability of humans to use these signals. In 2026, however, AI agents with the reasoning capability to parse logs, evaluate code diffs, and apply heuristic rules to classify failure types represent a qualitatively different class of consumer, one that is available continuously, scales without additional cost, and applies consistent logic without the cognitive fatigue that degrades human judgment over long review sessions [4]

The economic cost of Human-in-the-Loop (HITL) friction is substantial and compounding. At large platform organizations maintaining monorepos with millions of lines of code and CI pipelines processing thousands of builds per day, even modest reductions in per-incident human triage time translate into thousands of recovered engineer-hours monthly. More critically, the latency introduced by human triage creates feedback loop delays that cascade through the development process: a build that fails at 2 AM and waits until 9 AM for a human to investigate represents seven hours of blocked integration time for every engineer who depends on that code path [5]. Autonomous triage systems that respond to failure signals within seconds, rather than

Puligadda, S. (2026). Eliminating Human Intervention in the Software Development Lifecycle: A Novel Agentic Paradigm for Scalable Infrastructure Maintenance. *South African Computer Journal,* 38(2), 12–21.

ISSN 1015-7999 (print) ISSN 2313-7835 (online)

hours, compress this feedback loop dramatically, enabling the continuous-flow development cadence that high-velocity platform engineering demands.

The HITL dependency also creates knowledge concentration risk. Failures on specialized infrastructure, distributed consensus failures, cross-region replication delays, platform-specific runtime anomalies, typically require engineers with deep, organization-specific knowledge to resolve. When incidents pile up, that expertise funnels through a small number of senior engineers who become the bottleneck for the entire organization. Agentic systems trained on the organization's incident history, runbook library, and infrastructure topology can distribute this specialized diagnostic capability broadly, enabling junior engineers and automated systems alike to resolve issues that would previously have required escalation to the most stretched members of the team [6].

3 THE AUTONOMOUS SDLC FRAMEWORK

3.1 Phase 1: Requirement intelligence and planning

The autonomous lifecycle starts in the requirements phase with agents using Retrieval-Augmented Generation (RAG) methods to produce structured Engineering Requirement Documents based upon unstructured data inputs in natural language. Instead of creating effort for engineers to create requirements from scratch, the RAG-enabled planning agent scans existing source code, architectural decision record, and historical requirement documents as source data for identifying examples, identifying duplicate risk, and creating structured requirement models that are consistent with the company standards for producing requirements [7]. This phase also includes automated risk identification: the agent analyzes proposed changes against the dependency graph of the codebase, identifies services that will be affected, and generates a pre-implementation impact assessment that surfaces potential regression risks before a single line of code is written. This front-loaded risk analysis represents a direct operationalization of the shift-left quality principle at the requirement level

Table 1: SDLC automation: Manual vs. AI-assisted vs. fully autonomous [6]

SDLC Phase	**Manual (Human-in-Loop)**	**AI-Assisted Hybrid**	**Fully Autonomous**
Requirement Analysis	Manual drafting; workshop-driven	RAG-assisted synthesis; human validation	Automated generation from intent; human review only
Code Generation	Manual implementation	LLM-assisted drafting; human refinement	Autonomous generation; self-healing on failure
Build Triage	Manual log inspection; 20-40 min/incident	Automated alerting; human remediation	Autonomous diagnosis, fix, and retry loop
Code Review	Human peer review; high latency	AI suggestions; human decision	Risk-based routing; auto-approve low-risk
Test Authoring	Manual test case design	Template-assisted generation; human curation	Autonomous generation with mutation validation

Puligadda, S. (2026). Eliminating Human Intervention in the Software Development Lifecycle: A Novel Agentic Paradigm for Scalable Infrastructure Maintenance. *South African Computer Journal,* 38(2), 12–21.

ISSN 1015-7999 (print) ISSN 2313-7835 (online)

Deployment	Manual release coordination	Automated deployment with manual approval gates	Autonomous rollout with automated rollback on metric regression
Incident Response	Manual diagnosis and mitigation; pages at night	Automated alerting; human-driven resolution	Autonomous detection, diagnosis, and remediation

The planning agent also performs boundary condition modeling, enumerating the data states, system configurations, and external dependency behaviors that the proposed implementation must handle correctly. This modeling output serves as the specification input for the autonomous test planning phase, creating a continuous chain of traceability from requirement to test that eliminates the coverage gaps that arise when testing is treated as an afterthought. The integration of RAG with organizational knowledge bases means that the planning agent improves continuously as the organization accumulates more architectural decisions, incident postmortems, and design patterns, creating a compounding knowledge asset that grows more valuable with each engineering cycle [8].

3.2 Phase 2: Autonomous code generation and self-healing build repair

The second phase of Autonomous Code Generation and Self-Healing Build Repair consists of producing implementation drafts from structured requirement documents, which are aligned to an organization's conventions, dependencies, and interfaces. Critically, the value of autonomous code generation is amplified by its integration with self-healing build repair. When the initial build of a generated implementation fails, the repair agent does not alert a human. When there is a build failure, a remediation loop occurs where the agent parses the failure log/trace, classifies it according to a predefined error taxonomy, selects a remediation strategy from the runbook, applies the remediation to the build definition and re-runs the build. The remediation loop continues with multiple iterations until the build is successful or the failure is classified as requiring human escalation. [3]

The self-healing loop represents a qualitative departure from both traditional manual triage and first-generation AI coding assistants. Traditional triage requires a human to navigate to the failure, interpret the log, and apply their knowledge to identify a fix, a process that typically consumes 20 to 40 minutes per incident. First-generation AI coding assistants can suggest fixes when a human presents them with the error, but they do not independently detect, retrieve, and act on failures. The autonomous repair agent combines the detection capability of monitoring systems, the diagnostic capability of large language models, and the action capability of MCP-mediated tool use into a single, continuously operating process that eliminates human intermediation entirely for the large class of build failures that match established patterns [9].

3.3 Phase 3: AI-driven code review and autonomous test generation

Low-risk changes, dependency upgrades, documentation updates, formatting corrections, and routine bug fixes with clear precedent, clear the automated review

ISSN 1015-7999 (print) ISSN 2313-7835 (online)

pathway in seconds. The practical effect is significant: human reviewers are no longer buried under a backlog of trivially safe changes competing for attention alongside genuinely consequential ones [10].

Running in parallel, the Test Architect agent takes on validation coverage without waiting for engineers to schedule it. It works directly from the abstract syntax tree of the generated implementation, mapping functions, tracing code paths, and identifying the boundary conditions that edge-case failures hide behind. Unit test suites are generated using whichever framework the target language demands. Where the code touches external services, the agent pulls API interface definitions and produces contract-based integration tests that verify the implementation actually honors those interfaces, not just that it compiles cleanly. For anything with a user-facing surface, it generates structured interaction sequences, authentication flows, navigation journeys, transaction completions, formatted as direct inputs to the agentic end-to-end testing layer. Mutation testing serves as the final check: intentional faults injected into the code confirm that the generated tests will catch real failures, not merely execute without error [11].

4 OPERATIONAL IMPACT: EVIDENCE FROM LARGE-SCALE DEPLOYMENTS

The evidence for autonomous SDLC impact is sharpest in organizations that have already pushed these systems into production at genuine scale, environments where thousands of builds run daily, engineering teams number in the hundreds, and the cost of friction is large enough to measure precisely. The LogSage framework, validated industrially across large CI/CD pipelines, demonstrates the feasibility of large language model (LLM)-powered root cause analysis and automated remediation for CI failure patterns, reporting substantial reductions in mean time to resolution for the subset of failures addressed by the system [12]. Analogous results emerge from deployments of agentic CI/CD pipeline managers that autonomously handle build configuration, deployment sequencing, and failure recovery without human coordination, reducing deployment lead times and eliminating the queue buildup that occurs when human operations teams are the rate-limiting factor in the release pipeline.

Table 2: Autonomous SDLC framework: Phase-by-phase architecture [11]

Phase	Agent Type	Input Sources	Output / Action	Escalation Trigger
Requirement Intelligence	Planning Agent	Natural language input, codebase, ADRs	Structured requirements; risk assessment	Ambiguity detection
Code Generation	Codegen Agent	Requirements, dependency graph, conventions	Implementation draft; PR ready	Novel architecture pattern
Self-Healing Build Repair	Repair Agent	Build logs, source code, runbooks, history	Fixed code; retry loop; escalation	Unclassified error after 3 iterations
Code Review Triage	Review Agent	Code diff, risk taxonomy, PR context	Approval routing; human queue assignment	High-risk or novel pattern

ISSN 1015-7999 (print) ISSN 2313-7835 (online)

Test Generation	Test Architect	AST, API specs, boundary conditions, framework	Complete test suite; mutation validation	Coverage gap or unsupported framework
Production Self-Healing	Incident Agent	Metrics, logs, traces, runbooks, topology	Automated remediation; rollback on failure	Business logic requires human judgment

On code review throughput, the pattern holds across industry verticals. Fintech and enterprise software organizations deploying AI-driven review systems have documented average review latency reductions of 30 to 40 percent, with the bulk of that improvement coming from fast-track routing of low-risk changes [10]. The downstream benefit matters as much as the headline number: when routine changes stop competing with architecturally sensitive ones for reviewer time, the quality of review for the changes that actually need it improves, more focused attention, less approval fatigue.

Test maintenance tells a similar story. In high-velocity mobile and web environments, keeping a test suite functional consumes a disproportionate share of QA engineering time, locators break, new features ship without coverage, and the backlog of repair work grows faster than engineers can clear it. Agentic test generation systems that synthesize and update suites in step with code changes attack this problem at its root. Some deployments have moved close enough to the zero-maintenance threshold that dedicated test upkeep has effectively ceased to exist as a scheduled activity [13].

Table 3: Measured outcomes by automation maturity [13]

Metric	**Manual**	**AI-Assisted**	**Autonomous**
Avg. Build Triage Time	25-40 minutes	10-15 minutes	15-45 seconds
Review Latency Reduction	Baseline (4-8 hours)	20-30% improvement	40-50% improvement
Engineering Hours on Maintenance	30-40% of total	20-25% of total	5-10% of total
Test Coverage (Auto-generated)	~50-60% of code	~70-75% of code	85-95% of code paths
Mean Time to Production	2-3 days	1-2 days	4-8 hours
Deployment Failure Rate	2-5% of releases	1-2% of releases	0.2-0.5% of releases

5 RISKS, LIMITATIONS, AND GOVERNANCE

Where autonomous systems stumble is at the edge of what they have been trained to recognize. Failures involving cross-system dependencies, cross-region replication behavior, or application logic that requires organizational context to interpret correctly sit outside the pattern coverage that current agent architectures handle reliably [14]. The danger is not that the agent fails outright; it is that the agent continues iterating through remediation attempts that are wrong in ways that are not immediately obvious, potentially pushing incorrect fixes further into the codebase before anyone notices. Escalation criteria that are precisely defined and strictly enforced are the difference between an autonomous system that is trustworthy and one that is merely fast.

ISSN 1015-7999 (print) ISSN 2313-7835 (online)

Observability is the second structural requirement, and it does not come for free. Human engineers leave traces by default, tickets, review comments, Slack threads, and postmortem notes. None of that happens automatically when an agent makes a decision. Without deliberate instrumentation, autonomous systems produce no audit trail, which means no accountability when something goes wrong and no learning signal for improving what the system does next. Agent telemetry infrastructure that captures reasoning traces and decision logs at sufficient resolution for retrospective analysis is not optional, it is the governance foundation that everything else rests on [15]. Metrics such as autonomous resolution rate, escalation frequency, and fix accuracy provide the signal needed to continuously improve agent capabilities and identify the failure categories where human involvement remains essential.

Legacy tooling creates the third friction point. CI platforms, code review systems, and deployment pipelines were built for humans who navigate UIs and read notifications. Autonomous agents need none of that, and making those systems agent-accessible typically requires MCP wrappers or custom integration work that carries real cost, particularly for organizations with years of accumulated toolchain debt. Incremental adoption, starting with failure categories that are well-understood and low-stakes, is the practical path through this constraint.

6 ORGANIZATIONAL AND ECONOMIC IMPLICATIONS

What changes fundamentally in an autonomous SDLC is the unit economics of software delivery. The traditional assumption, that shipping more software requires hiring more engineers, breaks when the routine maintenance work that consumes 30 to 40 percent of engineering capacity gets absorbed by autonomous systems. Smaller teams can sustain larger, more complex codebases. Release velocity becomes a function of engineering judgment and architectural quality rather than headcount. The competitive moat in software shifts from who has the most people to who has built the most capable organizational knowledge base, the runbooks, incident histories, and architectural records that power agent behavior [6].

The environmental dimension is less discussed but worth naming. Running redundant CI pipelines at scale as a workaround for flaky tests and unstable environments generates substantial computational waste. Autonomous systems that resolve failures on first attempt, rather than retrying indefinitely, cut that waste directly. As AI infrastructure scales to support broader autonomous engineering adoption, the energy efficiency advantage of high first-pass resolution becomes a non-trivial consideration in platform design. The marginal cost of software delivery declines in step with the autonomous resolution rate, which means that the organizations investing earliest in high-quality agent knowledge bases are also the ones that will see the steepest efficiency curves.

The framing that matters most, though, is what this shift does to the engineering role itself. In an organization where routine maintenance, triage, and test authoring are handled autonomously, the human engineer's value contribution shifts entirely toward higher-order activities: architectural decision-making, requirement precision, system design, and the curation of the knowledge bases and rules that govern agent behavior. Delegating execution to autonomous systems does not diminish human agency, it concentrates it where it has always mattered most. Engineers who are no longer pulling locator repair and build triage shifts are free to make the architectural decisions, define the requirements, and curate the knowledge systems that determine what autonomous agents do and how well they do it [6].

Puligadda, S. (2026). Eliminating Human Intervention in the Software Development Lifecycle: A Novel Agentic Paradigm for Scalable Infrastructure Maintenance. *South African Computer Journal,* 38(2), 12–21.

ISSN 1015-7999 (print) ISSN 2313-7835 (online)

Table 4: Key contributions and organizational implications

Contribution	Technical Mechanism	Organizational Benefit
Build Triage Automation	Pattern-based failure classification with LLM analysis of logs and code diffs	Reclaim 30-40% of engineering hours; compress feedback loop from hours to seconds
RAG-Based Planning	Retrieval from codebase, ADRs, incident histories; semantic matching against precedents	Front-load risk analysis; democratize architectural knowledge; improve requirement precision
Classification-Based Code Review	Risk taxonomy + agent routing; fast-track approval for low-risk changes via policy	Reduce review latency 40-50%; allow human reviewers to concentrate on high-value decisions
AST-Driven Test Generation	Abstract syntax tree parsing; boundary condition enumeration; mutation testing for validation	Achieve 85-95% code path coverage; reduce test maintenance burden; ensure tests detect faults
Production Self-Healing	Real-time metric monitoring + automated remediation rules; safe rollback on metric regression	Eliminate on-call pages for routine incidents; reduce MTTR; improve customer experience
Knowledge Democratization	Agentic systems trained on org-specific runbooks, topology, and incident history	Distribute specialized expertise across org; enable junior engineers to resolve complex incidents

7 CONCLUSION

This article has presented a comprehensive framework for Autonomous Software Engineering that deploys AI agents across all phases of the SDLC to eliminate the human-in-the-loop friction that constrains engineering velocity at modern platform organizations. The evidence from large-scale deployments is compelling: autonomous triage systems, self-healing build repair agents, AI-driven code review bots, and generative test creation frameworks each deliver measurable productivity improvements that compound into transformative organizational impact when deployed together as an integrated autonomous lifecycle. The structural problem they collectively solve, the linear scaling constraint imposed by HITL dependency, is one of the most significant productivity bottlenecks in the software industry, and its resolution enables a fundamentally new economic model for software development.

The challenges associated with this transition are real but tractable. Governance frameworks that define clear escalation criteria, robust telemetry infrastructure that makes autonomous decision-making observable and accountable, and incremental adoption strategies that begin with low-risk failure categories and expand as agent capability is demonstrated, these are the architectural requirements for safe and effective autonomous SDLC deployment. Organizations that invest in these foundational capabilities now will build the institutional knowledge and tooling infrastructure necessary to operate at the frontier of autonomous software engineering as the technology continues to mature.

The long-term vision is a software development ecosystem in which autonomous agents handle the full spectrum of routine SDLC operations, leaving human engineers free to concentrate entirely on

ISSN 1015-7999 (print) ISSN 2313-7835 (online)

the work where their judgment is irreplaceable: defining what systems should do, evaluating whether they are doing it correctly, and ensuring that the values embedded in their design reflect the needs of the people they serve. That is an expansion of strategic influence, not a reduction of it, and it is the most compelling case for moving deliberately toward autonomous software engineering now

References

1. S. Karuppuchamy. Ai-augmented software engineering for rapid feature delivery and operations automation. In *Proc. 2026 IEEE 16th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 1–8, 2026. URL https://ieeexplore.ieee.org/document/11393761.
2. Mohammadamin Madani et al. Towards the integration of large language models into the software development life cycle: A systematic literature review. In *Proc. 2025 3rd International Conference on Foundation and Large Language Models (FLLM)*, pages 1–8, 2025. URL https://ieeexplore.ieee.org/document/11390990.
3. R. Nagvekar. Agentic ai-driven ci/cd pipelines for autonomous software delivery. In *Proc. 2025 IEEE 5th International Conference on ICT in Business Industry & Government (ICTBIG)*, pages 1–7, 2025. URL https://ieeexplore.ieee.org/document/11323919.
4. Praneesh Roshan P et al. Sdlc autopilot ai: Agentic automation of software development life cycle. In *Proc. 2025 International Conference on Intelligent Computing, Information and Control Systems (ICOIICS)*, pages 1–7, 2025. URL https://ieeexplore.ieee.org/document/ 11390299.
5. Dina Omar Salem et al. Ai-driven continuous integration: Automating code review and deployment with llms. In *Proc. 2025 10th International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 1–6, 2025. URL https://ieeexplore.ieee.org/document/11119365.
6. Mohanakrishnan Hariharan. Reinforcement learning integrated agentic rag for software test cases authoring. 2026 IEEE 5th International Conference on AI in Cybersecurity (ICAIC), 2026. URL https://ieeexplore.ieee.org/document/11395683.
7. M. Usman et al. Multi-agent systems in software testing: From planning to reporting. In *Proc. 2025 27th International Multitopic Conference (INMIC)*, pages 1–7, 2025. URL https://ieeexplore.ieee.org/document/11348399.
8. Mohammed Akour and Mamdouh Alenezi. Agentic ai in software development and devops: A systematic review of frameworks, tools, and future directions. *IEEE Access*, 13:1–20, 2025. URL https://ieeexplore.ieee.org/document/11452134.
9. M. H. Hariharan. Reinforcement learning integrated agentic rag for software test cases authoring. In *Proc. 2026 IEEE 5th International Conference on AI in Cybersecurity (ICAIC)*, pages 1–6, 2026. URL https://ieeexplore.ieee.org/document/11395683.
10. Kiran Raj K M et al. Ai-driven devops for intelligent automation in continuous software delivery pipelines. In *Proc. 2025 IEEE International Conference on Distributed Computing and Electrical Circuits and Electronics (ICDCECE)*, pages 1–6, 2025. URL https://ieeexplore.ieee.org/document/11382867.
11. Dimitrije Stojanović et al. Unit test generation using multi-agent large language models. 2024 32nd Telecommunications Forum (TELFOR), 2024. URL https://ieeexplore.ieee.org/document/10819096.
12. Weiyuan Xu et al. Logsage: An llm-based framework for ci/cd failure detection and remediation with industrial validation. In *Proc. 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1–12, 2025. URL https://ieeexplore.ieee.org/document/11334373.
13. Pedro Luís Fonseca et al. Streamlining acceptance test generation for mobile applications through large language models: An industrial case study. In *Proc. 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1–12, 2025. URL

ISSN 1015-7999 (print) ISSN 2313-7835 (online)

https://ieeexplore.ieee.org/document/11334650.
14. Anshuman Chhabra et al. Agentic ai security: Threats, defenses, evaluation, and open challenges. *IEEE Access*, 14:1–25, 2026. URL https://ieeexplore.ieee.org/document/11447227.
15. G. Martin and J. Olszewska. Automation testing framework for reliable autonomous agentic ai. In *Proc. 2025 IEEE Engineering Reliable Autonomous Systems (ERAS)*, pages 1–6, 2025. URL https://ieeexplore.ieee.org/document/11135911

Puligadda, S. (2026). Eliminating Human Intervention in the Software Development Lifecycle: A Novel Agentic Paradigm for Scalable Infrastructure Maintenance. *South African Computer Journal,* 38(2), 12–21.

ISSN 1015-7999 (print) ISSN 2313-7835 (online)