RAG to the rescue: Reimagining Enterprise Unit Test Management with AI

Gaurav Rathor | Ajay Bhosle | Nikhil Yogesh Joshi

g.rathor2210@gmail.com ajaybhosle.sre.data1@gmail.com
nikhilyogesh.joshi@fiserv.com

Abstract

Unit testing in enterprise applications remains a persistent challenge—complexity, time investment, and framework diversity hinder both novice and experienced developers. Legacy monolithic architectures amplify these issues, as declining code coverage drives production defects, doubles bug-fix times, delays releases, and inflates costs. In industries where software failures can threaten public safety or regulatory standing, the stakes are even higher. With U.S. software defects estimated to cost $2.41 trillion [1], automation is no longer optional.

We introduce an AI-driven unit test generation framework that fuses Retrieval-Augmented Generation (RAG) with Model Context Protocol (MCP). Large Language Models (LLMs) have recently been applied to various aspects of software development, including their suggested use for automated generation of unit tests, but need additional training or few-shot learning on examples of existing tests [2]. LLM agents employ RAG to retrieve functional specs, historical test cases, and domain docs, ensuring generated tests validate business intent, rather than code structure alone. MCP then converts this enriched context into maintainable, adaptive test suites that integrate seamlessly with ITIL/ITSM change control processes.

Robust oversight is built into three dimensions: Business Logic Assurance (ensuring test coverage aligns with core functional requirements), Performance & Reliability Assurance (improving runtime efficiency and minimizing flaky behavior), and Model & Data Stewardship (maintaining model accuracy, stability, and trustworthiness). The framework is industry-agnostic yet particularly impactful for mission-critical, compliance-heavy sectors such as fintech and medical systems, meeting global regulatory obligations including SOX, PCI DSS, and SOC, across U.S., EU, and worldwide contexts.

Biography

Gaurav Rathor is a Performance Architect with 17 years of experience optimizing enterprise applications, microservices, and infrastructure performance across diverse technology landscapes. He leads performance benchmarking and optimization initiatives at Omnissa Inc., partnering with product and engineering leadership to embed performance-first practices throughout the development lifecycle.

Ajay Bhosle is a Technical Account Manager at Accenture, based in Houston, Texas, with over 20 years of IT experience across technology, consulting, management & operations supporting innovative cloud & AI-based product development & operations for health care, Oil & Gas, BFSI clients, with strong roots in software architecture, data & AI engineering.

Nikhil Yogesh Joshi, Director of Software Engineering based in Cumming, Georgia, brings in over 18 years of experience leading high-performing teams and delivering complex projects. His career spans multiple industries, demonstrating his expertise in automation, cloud migration, and strategic leadership.

1 Introduction

1.1 Background

Unit testing has long been regarded as a foundational practice in software engineering, ensuring that individual code components function as intended before integration into larger systems. In the financial services domain, where accuracy, reliability, and compliance are paramount, traditional unit testing frameworks have played a critical role in mitigating operational and regulatory risks. These frameworks help verify transaction processing, data integrity, and compliance with industry mandates, forming an essential safeguard against both financial losses and reputational damage. However, as financial applications scale in complexity and regulatory expectations expand, traditional unit testing frameworks are increasingly strained to deliver the efficiency and assurance required in modern enterprise environments. Manually writing these tests is time-consuming and tedious, which significantly escalates the cost of software development [3].

1.2 Challenges

Despite their importance, traditional unit testing methods face persistent challenges. Historically, the prevalence of monolithic systems and diverse testing frameworks has caused coverage gaps, redundant effort, and brittle test designs. Financial and compliance impacts further magnify these weaknesses—defects escaping into production can trigger financial penalties, regulatory scrutiny, severe reputational consequences and actual financial losses to end users. The skills gap in emerging AI-driven quality practices compounds the issue, as organizations often lack expertise to modernize testing pipelines while maintaining regulatory confidence. Together, these challenges create rising costs, longer MTTR, and an unsustainable testing burden that undermines both operational efficiency and risk management. When application complexity increases and modernization is attempted, there is an automatic drop in code coverage and makes it challenging to sustain the same.

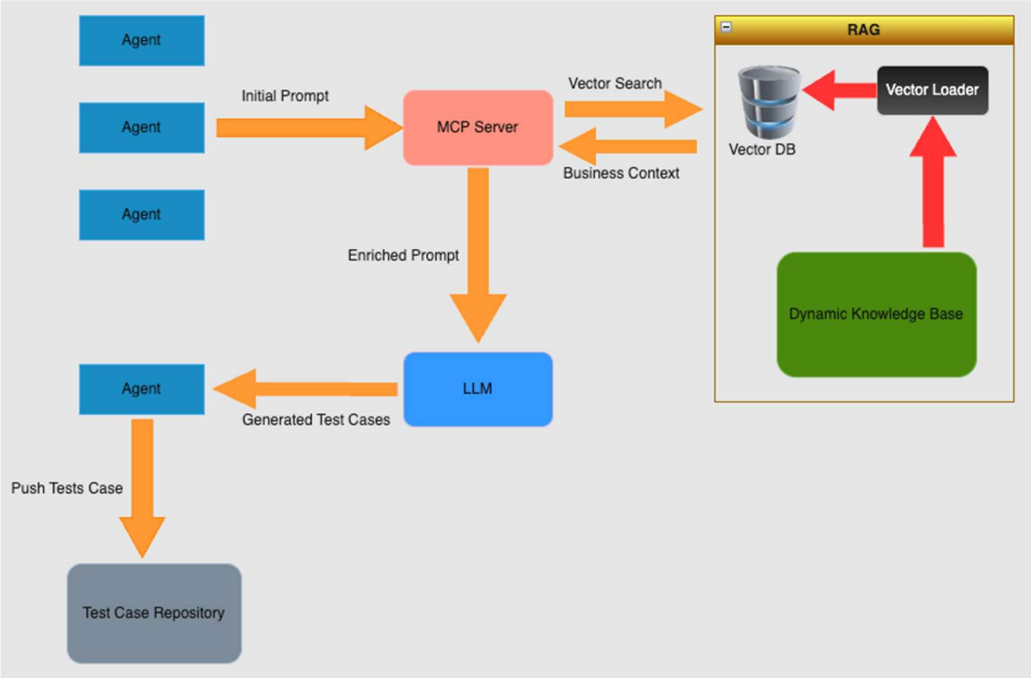
1.3 Objectives

To address these shortcomings, this paper introduces an AI-driven testing framework that combines Retrieval-Augmented Generation (RAG) with Model Context Protocol (MCP). Retrieval Augmented Generation (RAG) has emerged as a solution, enhancing LLMs by integrating real time data retrieval to provide contextually relevant and up-to-date responses [4]. By leveraging enterprise documentation, compliance rules, and historical test cases, the framework generates unit tests that validate business logic as well as technical correctness. The objective is clear: to improve operational efficiency, reduce the cost per bug by catching defects earlier, and shorten the MTTR by aligning tests with regulatory and functional requirements at the outset. This positions AI-enabled testing not as a replacement for traditional frameworks, but as a management-aligned capability that augments enterprise resilience and accelerates delivery.

The target audience for this paper is enterprise management—CIOs, CTOs, QA leaders, compliance officers, and program managers—who are responsible for balancing operational resilience with regulatory compliance and cost efficiency. Instead of viewing unit testing as a purely technical activity, this framework positions it as a management discipline that can be measured, governed, and optimized. Executives can track KPIs such as defect leakage rates, MTTR, audit readiness scores, compliance alignment, and cost per defect fixed, all of which directly impact business outcomes and customer trust. By embedding AI-enabled frameworks into established management processes such as ITIL/ITSM change control, risk management, and continuous improvement cycles, leaders can transform testing from a cost center into a governance-driven capability that strengthens compliance posture, accelerates delivery, and improves return on technology investment.

2 Proposed Framework

2.1 Understanding the architecture



The framework consists of the following building blocks:

- AI Agents - (Orchestrator)
- Model Context Protocol (MCP) Integration Layer
- Retrieval-Augmented Generation (RAG) - Knowledge Layer
- LLM Model - Generator

Agents act as the orchestrators in an AI-driven system. They interpret user intent, decide on actions, and coordinate task execution across various components. Beyond generating responses, agents leverage reasoning, memory, and external tools to solve complex problems. Key components include a planner (breaking down goals into actionable steps), a memory module (short-term conversation state and long-term knowledge persistence), and a tool interface (allowing it to call APIs, databases, or retrieval systems). The agent itself does not directly access external knowledge stores; instead, it relies on MCP to manage these interactions. Each Agent has their own MCP client to communicate with the MCP server. When a request is made, for example, to generate or validate code, the agent sends it to the MCP server, which manages the flow.

The Model Context Protocol (MCP) provides a standardized interface for the agent to interact with external systems. MCP provides a client-server interface for secure tool invocation and typed data exchange [5]. It defines schemas for requests and responses, connectors/adapters for integrating with databases, APIs, or test frameworks, and execution managers to coordinate with tools. By following MCP standards, the agent can access tools and knowledge sources without needing to understand their internal implementation, making system integration modular, robust, and maintainable. Basically, MCP acts like a gatekeeper/ translator/auditor. It ensures that only the right information, in the right format, securely, is passed from the RAG knowledge layer into the LLM so that the generated test cases are accurate, compliant, and audit-ready.

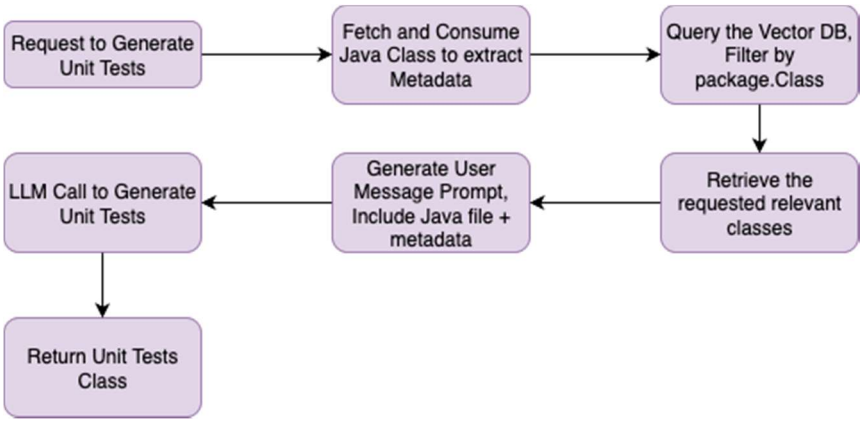
Retrieval-Augmented Generation (RAG) serves as the agent's dynamic knowledge engine, and the vector database is its core component for fast, semantic retrieval. When the agent needs context-specific or up-to-date knowledge, RAG converts the query into embeddings and searches the vector DB for semantically similar documents. The vector DB stores embeddings, supports similarity searches, and may include metadata filtering for relevance, as described in this paper [6]. Retrieved content is then structured by RAG's context builder and injected into the agent's prompt via MCP. This combination ensures the agent, reasons over accurate, current, and domain-specific information, overcoming the limitations of static training data and enhancing real-time decision-making.

All these components are packaged into a microservice.

2.2 Data Flow Diagram

This diagram shows the practical flow of how Agent, MCP, and RAG with a vector database work together to generate unit tests. When a client requests test generation, the agent initiates the process by fetching the relevant code file and extracting metadata. Through the MCP layer, it issues a query to the vector database—filtering by package and class name—to retrieve related classes and supporting context. So, what exactly are these supporting contexts? It can be related classes, interfaces, utility classes, dependencies in the form of mocks and libraries. It can be example test cases, previous implementations, or reusable patterns stored as embeddings that are semantically similar to the target class. It can also be documentation, comments or annotations relevant to the package or module.

This retrieved knowledge, combined with the original file and metadata, is then packaged into a structured prompt for the LLM. The LLM generates the unit test class, which is returned to the client. The enhancements to the framework, such as class-level vector storage for better retrieval, using direct SQL queries instead of additional frameworks, reducing API overhead, and skipping unnecessary system message fetches. This workflow illustrates how the components collaborate to provide accurate, efficient, and context-rich unit test generation.



2.3 Model Selection

To ensure the RAG framework could generate high-quality, domain-specific unit tests for fintech applications, the team began by evaluating several LLMs for suitability. The evaluation criteria focused on financial domain knowledge, code understanding, data privacy, and licensing compliance. Management opted for models with enterprise-friendly licenses, ensuring legal clarity for production use. Given the sensitivity of financial data, the LLM was deployed in an air-gapped environment, completely isolated from the internet. This approach guaranteed that proprietary code and sensitive test data remained secure throughout model training and usage. The training process involved curating internal datasets including historical defects, code snippets, functional specifications, and unit test cases. Subject matter experts collaborated with the technical teams to annotate data, define

prompts, and guide the fine-tuning process. Iterative training cycles were performed to improve accuracy, relevance, and the model's ability to handle complex fintech-specific scenarios.

3 Data Governance & Security first approach

Enterprise adoption of AI-driven testing requires more than technical capability—it demands seamless integration with existing governance frameworks while maintaining the highest security standards. Our RAG-enhanced unit test generation platform embeds security-by-design principles throughout the entire lifecycle, from code analysis to test deployment, ensuring compliance with enterprise change management processes and regulatory requirements.

3.1 Change Control (ITIL/ITSM)

- Automated Change Requests: Unit test updates map to ITIL workflows—e.g., standard change for login validation updates, normal change for new KYC workflow tests (CAB approval), and emergency change for fraud-detection defect fixes.
- ITSM Integration: Connectors (ServiceNow, Jira) auto-create change records—e.g., Jira ticket logs AI model version and unit test metadata for a new payment authorization module.
- Pipeline Governance:
  - Approval Gates: Unit test generation flows through staged checks: RAG context retrieval → AI test generation → static/dynamic security validation → change approval → deployment.
  - Traceability: Every step records lineage - e.g., AML document retrieved as test context, model prompts used, and reviewer approvals with timestamps.
  - Rollback & Recovery: Failed regression or coverage gaps trigger auto-rollback to last stable test suite. Example: if new unit tests for transaction limits degrade fraud-detection coverage, rollback is initiated.
  - Emergency Disable: AI-generated tests can be temporarily disabled while keeping manual test baselines active—critical for production banking environments where availability is non-negotiable.

3.2 Model Integrity & Security

- Model Provenance: All AI models must be cryptographically signed and versioned. Example: fraud-detection unit tests can only be generated with an approved LLM version, preventing use of tampered models. [7]
- Training Data Lineage: Documentation of data sources (e.g., regulatory guidelines, financial specs) with bias indicators—ensuring AML/KYC tests aren't skewed by incomplete rule sets. [8]
- Secure Context Management: RAG vector databases storing proprietary fintech rules (e.g., card transaction thresholds) are encrypted at rest and in transit.
- Third-Party Integration Security:
  - API Security: OAuth 2.0/OIDC enforced when pulling compliance rules (e.g., Basel III liquidity requirements) from external services. [9]
  - Network Segmentation: Unit test generation runs in isolated environments to prevent cross-contamination with production financial systems.
  - Data Minimization: Least-privilege enforced—AI retrieves only the subset of financial rules needed for the specific unit test (e.g., interest calculation, not full loan book).

Recent studies confirm that AI-generated code carries higher defect and vulnerability rates than human-written code (Cotroneo, Improta, and Liguori 2025), making rigorous validation and provenance controls essential in fintech environments. [10]

3.3 Security Reporting & Guidance

AI-generated unit tests provide regulator-ready visibility into security posture:

- Executive Dashboards: Real-time risk scores across payment flows (e.g., PCI DSS cardholder data tests, SOX reporting checks), with trend lines showing faster remediation of flaws (e.g., crypto misuse reduced from 14 to 3 days).
- Compliance Reporting: Auto-generated evidence packages map unit tests to controls—e.g., PCI DSS 3.4 tied to AES-256 encryption checks. Full audit trails capture model prompts, approvals, and exception handling for traceability.
- Continuous Improvement: CVE feeds and fintech-specific threat models (fraud transfer race conditions, double-spend checks) continuously refine test templates, while false positives feed back to improve generation accuracy.

3.4 Implementation Considerations

Successful rollout depends on organizational readiness and measurable KPIs:

- Readiness: Teams trained to handle AI-test risks (e.g., no private key exposure), with clear escalation playbooks for flagged unit tests and alignment to existing CAB/risk boards.
- Technology Fit: Integration with fintech-standard tools (SonarQube, ServiceNow), hybrid deployment for sensitive AML/KYC data, and adapters for legacy mainframe-based transaction systems.
- KPIs: MTTSR cut from 5→2 days for fraud defects; coverage of payment edge cases raised from 70%→95%; audit prep time reduced by 40%; zero-trust validation shown via anonymized test data access.

This security-first approach ensures that AI-driven test generation becomes a force multiplier for organizational security capabilities rather than introducing new attack vectors or compliance gaps.

4 Management Case Study

4.1 The Problem

The leadership team identified a series of systemic challenges that were increasingly impacting delivery timelines, software quality, and cost efficiency. Several technical gaps had emerged over time, particularly as systems evolved into monolithic structures. Legacy codebases were difficult to maintain, and even small refactoring efforts frequently caused code breakages. The absence of automated unit testing and performance checks at the method level left these applications fragile, and the lack of standardized frameworks meant that the technical debt continued to grow unchecked. Manual test case creation was slow and inconsistent, resulting in incomplete test coverage and heightened risk of defects escaping into production.

In parallel, functional gaps were also apparent. Teams were under pressure to meet aggressive deadlines, often leading to the bypassing of critical gating processes. Without strong controls, the discipline of writing meaningful unit test cases deteriorated. Code coverage fell drastically, and management's ability to ensure consistent quality across releases diminished. The pressure to deliver quickly created a trade-off where long-term quality was sacrificed for short-term gains, compounding

the technical debt problem and eroding customer trust. Furthermore, aged applications lacked proper documentation, and functional insights were often trapped in siloed teams, making knowledge transfer and test creation even more difficult.

The cost impact was significant. Bugs that escaped into production were expensive to fix, increasing the overall cost per defect. Go-To-Market timelines stretched, release cycles slowed, and support costs grew as rework became a norm. Customers were becoming increasingly dissatisfied due to unpredictable delivery schedules and inconsistent release quality. In some cases, management even faced the challenge of aligning internal teams and external stakeholders because expectations were not properly managed. The pressure to deliver faster while reducing cost created an unsustainable cycle of compromise.

Recognizing these risks, management realized that the problem was not only technical but also cultural and procedural. The organization lacked discipline and governance around quality assurance. Teams often generated technical debt to meet short-term delivery goals, and hard gates were either softened or removed altogether to accelerate releases. This led to a decline in confidence—both internally among leadership and externally with customers. These challenges created the urgency to develop a robust, tool-driven framework that could reintroduce process discipline, reduce technical and functional gaps, and deliver measurable value to the business.

4.2 Strategy Using the 4 P's: Planning, People, Process, and Performance

Planning: Leadership worked closely with technical leads to analyze the current state. They documented technical and functional gaps, quantified the cost of rework, and mapped areas where efficiency gains were possible. The roadmap included phased implementation, starting with pilot projects to validate the framework's value. The plan outlined how integrating tools like agents, MCP, RAG, and vector DB would create a cohesive ecosystem for test generation, functional quality validation, static analysis, and performance metrics. An important consideration was the knowledge base (KB) layer — validated and curated by business teams, as described in paper [11]. These KB articles fed into a vector database, ensuring that the integration layer could query accurate, business-aligned knowledge during test generation [12]. This eliminated functional blind spots and allowed the framework to integrate context-aware insights into every test scenario. A data governance framework was designed to ensure that only sanitized, business-approved knowledge base (KB) articles were ingested into the vector database.

- Role-based access control (RBAC) and encryption policies were defined to protect sensitive data during storage and transmission.
- Clear audit and retention policies were established to meet regulatory requirements (GDPR, SOC 2). This ensured that the integration layer could query accurate, audit-compliant, and business-aligned knowledge during test generation while maintaining the confidentiality and integrity of the generated test cases.

People: Management identified that success depended on upskilling and creating a culture of discipline. Developers were capable but needed enhanced technical skills to use advanced tooling. Training programs and internal champions were introduced to accelerate adoption. Leaders also set clear expectations for adhering to testing practices and gating processes, recognizing that discipline had to be enforced at both technical and management levels. This shift required managers to protect quality timelines and resist shortcuts, ensuring that customer expectations were managed realistically.

Process: The framework enforced best practices to eliminate ad-hoc behaviors. The framework promoted measurable and enforceable steps—defining thresholds for coverage, embedding quality checks early, and reducing friction by automating retrieval and analysis using MCP, RAG, and vector DB. This created transparency and predictability, even for complex refactoring tasks in legacy applications. Standardized workflows were enhanced with governance checkpoints:

- All data entering the framework underwent classification and sanitization.
- Integration points (Agents, MCP servers, vector DB) were secured with encryption at rest and in transit.
- Test data followed least-privilege access principles, and secrets were kept out of source code.

Management approved investments to automate static code analysis, unit test generation, performance testing, and gating—while ensuring that every step was auditable. The focus was not only on quality but also on traceability and security, giving leadership confidence in compliance and risk management.

Performance: The business case demonstrated clear metrics to track value. Testing frequency increased, developer efficiency improved, and critical modules could be tested with confidence. The framework captured unit-level performance metrics, giving visibility into potential issues before release. Leadership tied performance to business outcomes: reduced defect leakage, shorter release cycles, lower cost per bug, and more predictable delivery timelines. These indicators became central to evaluating ROI and ensuring continuous support for the initiative.

4.3 Implementation and Adoption Journey

Pilot Phase: Testing the Framework on a Microservice

To ensure the proposed framework was robust and practical, the development team initiated a pilot by selecting a smaller but critical microservice. This provided a controlled environment to validate the framework's components without disrupting larger monolith systems. Development teams worked on automating unit test generation for critical modules, demonstrating quick wins like improved coverage and reduced manual effort. Static code analysis was used to highlight technical debt and prioritize refactoring, which resonated with management when presented as quantified risk reduction. The MCP server integrated various tools, ensuring seamless orchestration and retrieval of supporting context through the vector DB. Feedback loops were established to refine the framework with every iteration.

Adoption was accelerated by transparent communication: dashboards showed time saved, defect reduction, and faster turnaround times. Champions were identified in each team to drive adoption, and leadership tracked KPIs like adoption rate, code coverage improvement, and defect leakage reduction to validate impact.

4.4 ROI Analysis

In a real-world enterprise setting, below ROI analysis assumes a mid-sized development team managing a critical microservice that undergoes frequent bug-fix releases. The baseline for comparison is a manual unit testing process, where each test case requires approximately four hours to design, implement, and review, factoring in team handoffs and code reviews. The automated framework leverages a Retrieval-Augmented Generation (RAG) model integrated with orchestration and knowledge base layers, reducing this effort to roughly one hour per case, including creation and peer validation.

An initial one-time investment covers infrastructure setup, model license procurement, prep and training, knowledge base curation, staff upskilling, and an estimated 10% attrition cost. Cost per development hour is assumed to be $100/hr. Initially, our unit test case for the SUT microservice was 9,000 test cases, requiring significant design, implementation, and review effort. In subsequent cycles, only smaller increments of test cases—reflecting new fixes or updates—need to be created or reworked, reducing effort while maintaining coverage.

Iteration (Bug-Fix Cycle)	Approx # Test Cases (new/updates)	Manual Efforts (hrs)	Manual Cost	RAG-Driven Efforts (hrs)	RAG-Driven Cost	Cumulative Savings
1	1000	4000	$400,000	1000	$100,000 + $150,000 (setup) = $250,000	$150,000 (saved, but offset by setup)
2	800	3200	$320,000	800	$80,000	$240,000
3	700	2800	$280,000	700	$70,000	$350,000
4	600	2400	$240,000	600	$60,000	$430,000
5	600	2400	$240,000	600	$60,000	$510,000

Investment: One-time setup: $150,000

Effort-hour savings:

- Manual cost: $1,480,000 (sum of all manual costs)
- RAG-driven cost: $520,000 (including $150k setup)
- Cumulative savings: $510,000
- ROI % = (Savings – Investment) /Investment × 100 = (510,000−150,000)/150,000 * 100 = 240 %

Observations

- Significant upfront savings: ~75% reduction in effort per cycle after framework adoption, even with initial setup costs.
- Break-even: Achieved after ~5 bug-fix cycles.
- Scalability: Once implemented, the framework can support other microservices or additional bug-fix streams without repeating setup costs.

Qualitative Benefits

- Faster bug resolution reduces MTTR.
- Consistent test coverage minimizes regression risk.
- Knowledge retention improves as the framework documents functional gaps.

4.5 Performance KPIs - Improvements

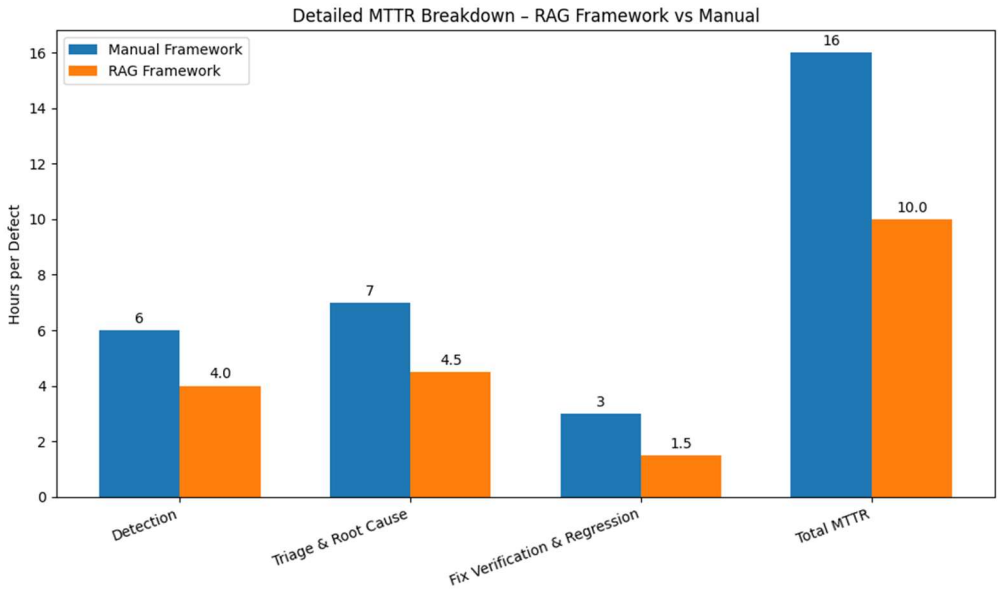
4.5.1 Reduced MTTR (Mean Time to Resolve)

Improvement: ~30–40% reduction in average resolution time.

Cause Analysis:

- RAG-generated tests are context-rich: they include reproducible inputs, detailed assertions, and environment stubs, reducing the time engineers spend reproducing issues.
- Automated triage and prioritization highlight high-risk or customer-impacting defects earlier, cutting the “diagnosis” phase.
- As fixes are deployed, RAG automatically updates or creates regression guards, eliminating repeat investigation cycles.

Impact: Faster turnarounds mean teams deliver bug fixes in hours or days instead of days or weeks, which directly shortens release delays and improves customer confidence.



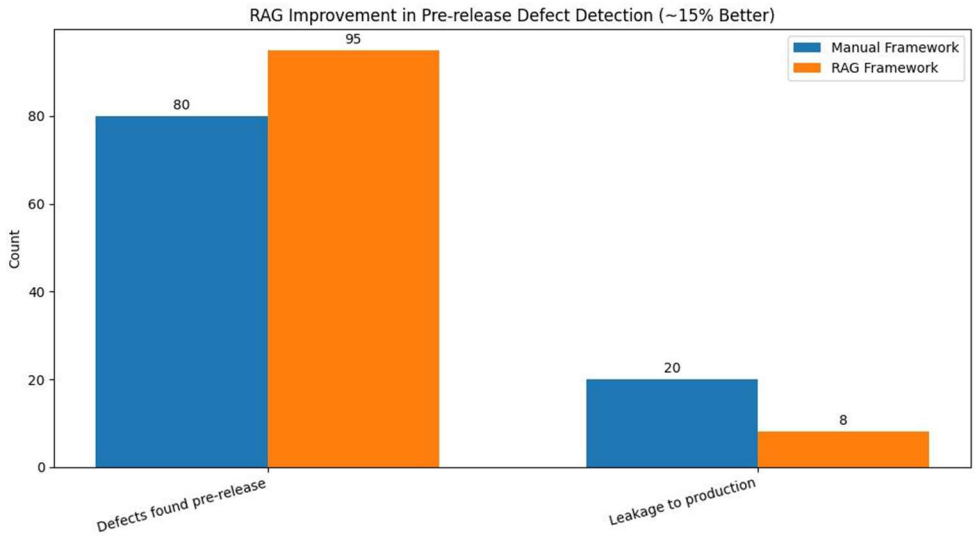
4.5.2 Defect Leakage Reduction

Improvement: ~ 20-30% fewer escaped defects in UAT or production.

Cause Analysis:

- RAG leverages LLM along with a knowledge base to generate broader and deeper test coverage, including boundary, negative, and integration scenarios that are commonly missed in manual cycles.
- Knowledge retention: previously discovered issues are coded as reusable tests across components, preventing re-introduction of the same class of bugs.
- Dynamic updates: as code changes, RAG can re-generate impacted tests, reducing “blind spots” caused by stale or missing tests.

Impact: Fewer critical incidents and hotfixes in production, less firefighting, and lower support costs.



4.5.3 Code Quality Improvements

Improvement: Notable reduction in code smells, security vulnerabilities, and compliance violations flagged by SCA or static analysis tools.

Cause Analysis:

- RAG can embed secure coding and compliance rules into test generation, ensuring risky patterns are detected early.
- Tests encourage design-for-testability: developers refactor code for clearer seams and fewer hidden dependencies, reducing complexity and technical debt.
- Automated guardrails catch misconfigurations or insecure defaults before they ship.

Impact: Cleaner, safer, and more maintainable codebase, reducing long-term remediation effort and audit risk.

The RAG framework doesn't just reduce manual effort—it improves speed, quality, and risk management simultaneously. By catching more defects early, reducing resolution time, ensuring compliance, and lowering cost per defect, it provides measurable business value that compounds over successive release cycles.

4.6 Lesson Learned

Lessons learned included the importance of aligning business and technical teams early, the need for visible management support, and the impact of structured KB layers on functional accuracy. Most importantly, the framework underscored that true ROI lies not just in tools, but in culture and governance supported by disciplined leadership.

Lesson Learned	Description	Impact on KPIs

High Initial Investment	Significant upfront cost for infrastructure, LLM fine-tuning, training, and integration.	Requires budget approval, clear ROI justification, and patience before savings are realized (~4–5 cycles).
Training and Onboarding	Teams need guidance on interpreting RAG-generated tests.	Improves adoption rate, increases test effectiveness, enhances coverage improvement.
Early Knowledge Capture	Codifying resolved defects as reusable tests early prevents repeat issues.	Reduces defect leakage, lowers cost per defect, improves pre-release detection, accelerates ROI.
Metrics & KPI Alignment	Continuous tracking of MTTR, defect leakage, coverage, release cadence, and cost per defect.	Provides management with visibility to measure ROI, track improvements, and make data-driven decisions.
Technical Debt Awareness	Design-for-testability drives code refactoring.	Reduces hidden dependencies and complexity, improves code quality, lowers long-term maintenance costs.
Compliance & Audit Readiness	Embedding regulatory and internal policy checks into the process.	Higher audit scores, reduced manual compliance effort, mitigates organizational risk.
Change Management & Organizational Adoption	Framing RAG benefits in business terms improves acceptance.	Teams embrace framework → faster realization of MTTR, leakage reduction, coverage, and cost savings
Oversight & Governance Needs	AI-generated tests may create redundant, overly complex, or irrelevant tests if unchecked.	Management must define review processes, approval gates, and metrics to maintain test efficiency.
Incremental ROI Expectation	Benefits accumulate over multiple release cycles; upfront investment is significant.	Sets realistic expectations for management; helps plan resources, budgets, and timelines; demonstrates compounding business impact.

5 Conclusion

The RAG-driven unit test framework demonstrates a transformative approach to improving software quality, efficiency, and governance. By integrating a domain-tuned, air-gapped LLM with a retrieval-augmented knowledge base, the framework automates unit test creation and enforces guardrails, resulting in reduced manual effort, faster release cycles, lower MTTR, and improved pre-release

defect detection. Pilot implementations validated measurable business benefits while ensuring audit readiness and compliance alignment.

However, the initiative also revealed key challenges. High initial investment, workflow integration complexity, change management hurdles, and the need for ongoing governance require careful management oversight. Teams must continually monitor metrics, refine test generation rules, and maintain internal knowledge quality to maximize ROI.

Future improvements may include enhancing domain adaptation of LLMs through continual fine-tuning with fintech-specific defect corpora to improve accuracy. Cross-team collaboration features such as shared dashboards and approval workflows would reduce adoption friction across DevOps, risk, and compliance teams. Finally, scaling for multi-service orchestration, coordinating tests across distributed microservices and hybrid infrastructures, remains a key aspect to enable enterprise-wide adoption.

References

[1] Consortium for Information & Software Quality (CISQ). 2022. The Cost of Poor Software Quality in the US: A 2022 Report. Retrieved September 2, 2025 (https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/)

[2] Yang, L., Yang, C., Gao, S., Wang, W., Wang, B., Zhu, Q., ... & Chen, J. (2024). An empirical study of unit test generation with large language models. arXiv preprint arXiv:2406.18181.

[3] Kumar, D., & Mishra, K. K. (2016). The impacts of test automation on software's cost, quality and time to market. Procedia Computer Science, 79, pages 8-15.

[4] Singh, A., Ehtesham, A., Kumar, S., & Khoei, T. T. (2025). Agentic retrieval-augmented generation: A survey on agentic rag. arXiv preprint arXiv:2501.09136.

[5] Ehtesham, A., Singh, A., Gupta, G. K., & Kumar, S. (2025). A survey of agent interoperability protocols: Model context protocol (mcp), agent communication protocol (acp), agent-to-agent protocol (a2a), and agent network protocol (anp). arXiv preprint arXiv:2505.02279.

[6] Shin, J., Aleithan, R., Hemmati, H., & Wang, S. (2024). Retrieval-augmented test generation: How far are we?. arXiv preprint arXiv:2409.12682.

[7] Marc Ohm, Arnold Sykosch, and Michael Meier. Towards detection of software supply chain attacks by forensic artifacts. In Proceedings of the 15th international conference on availability, reliability and security, pages 1–6, 2020

[8] Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. Software composition analysis for vulnerability detection: An empirical study on java projects. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 960–972, 2023.

[9] Fett, Daniel, Pedram Hosseyni, and Ralf Küsters. 2019. “An Extensive Formal Security Analysis of the OpenID Financial-Grade API.” arXiv. February 1. Preprint. arXiv:1901.11520

[10] Cotroneo, Domenico, Cristina Improta, Pietro Liguori. 2025. “Human-Written vs. AI-Generated Code: A Large-Scale Study of Defects, Vulnerabilities, and Complexity.” arXiv. August 29. Preprint. arXiv:2508.21634

[11] Johnsson, N. (2024). An in-depth study on the utilization of large language models for test case generation.

[12] OpenAI, 2025. ChatGPT (GPT-5) OpenAI URL: https://www.openai.com/