Invisible Intelligence: Embedding AI/ML for Smarter, Faster Release Testing

Vidhya Ranganathan
vidhya.ranganathan@okta.com

Abstract

This paper explores a practical approach to embedding powerful AI/ML capabilities directly into existing release management workflows. We demonstrate how AI/ML can function as seamless, invisible extensions of your current development and testing infrastructure, unlocking significant efficiency gains without costly overhauls. This paper explores a practical approach to embedding powerful AI/ML capabilities directly into existing release management workflows. We demonstrate how AI/ML can function as seamless, invisible extensions of your current development and testing infrastructure, unlocking significant efficiency gains without costly overhauls. This framework focuses on three critical areas: intelligent test redundancy elimination, automated test strategy generation, and predictive deployment risk assessment. Our case study details specific techniques, including NLP and clustering algorithms, that dramatically reduced validation effort, improved efficiency, and helped maintain rigorous quality standards within the context of existing test codebases.

Biography

Vidhya Ranganathan, with 15 years of experience in the quality domain, began her career as a manual tester and developed a passion for automation and DevOps. Specializing in designing frameworks, curating processes, and forming dedicated quality teams, she has significantly contributed to several startups, helping them bootstrap and scale their quality teams. Currently a Quality Manager at Okta, Vidhya is known for her strategic approach and hands-on expertise in driving excellence in quality assurance.

1. Introduction

In today's fast-paced software development landscape, maintaining high-quality standards while accelerating release velocity presents a significant challenge. Traditional release testing workflows often struggle with manual effort, fragmented data, and a lack of formal planning, leading to costly regressions and delayed deployments. This paper posits that AI and Machine Learning (ML) can be "invisibly" embedded into existing release management workflows, functioning as intelligent extensions rather than disruptive overhauls. By leveraging rich operational data—such as build logs, test results, and release metadata—existing AI/ML tools can be leveraged to enhance predictability, optimize resource utilization, and automate key decisions. This paper presents a pragmatic, experience-based methodology for incrementally integrating AI/ML, focusing on three critical areas where it can deliver immediate benefits: predicting deployment risks, intelligently eliminating redundant tests, and automating test strategy generation via AI agents.

2. Background and Related Work

The integration of Artificial Intelligence is fundamentally reshaping software testing, transitioning it from a reactive, labor-intensive process to a proactive, intelligent, and highly efficient discipline, driven by the demands of modern Agile and CI/CD methodologies. Large Language Models (LLMs) are proving to be a versatile enabler, profoundly influencing areas such as automatic test planning, test automation code creation, test data generation, and solving the critical "Test Oracle Problem" by enabling AI to reliably determine test correctness. This shift is further supported by the rise of codeless automation, self-healing tests, QAOps, and hyperautomation, all aimed at democratizing testing, reducing maintenance burdens, and enhancing overall productivity. AI's capabilities in defect prediction and vulnerability detection are also enabling a strategic move towards preventing issues proactively, optimizing test suites, and prioritizing efforts based on data-driven insights. This trajectory indicates a future where AI is an integral partner in the entire software development ecosystem, driving continuous quality improvement and accelerating innovation.

3. Methodology: Incremental AI/ML Embedding

The methodology for embedding AI/ML into release testing is built on a pragmatic, data-driven, and iterative foundation. The core idea is to tap into the rich operational data already being generated by development and testing activities. This data serves as a valuable resource for training models that can provide genuinely actionable insights. The focus was on two key dimensions, each designed to address a specific pain point experienced in the release process:

3.1. Intelligent Test Redundancy Elimination

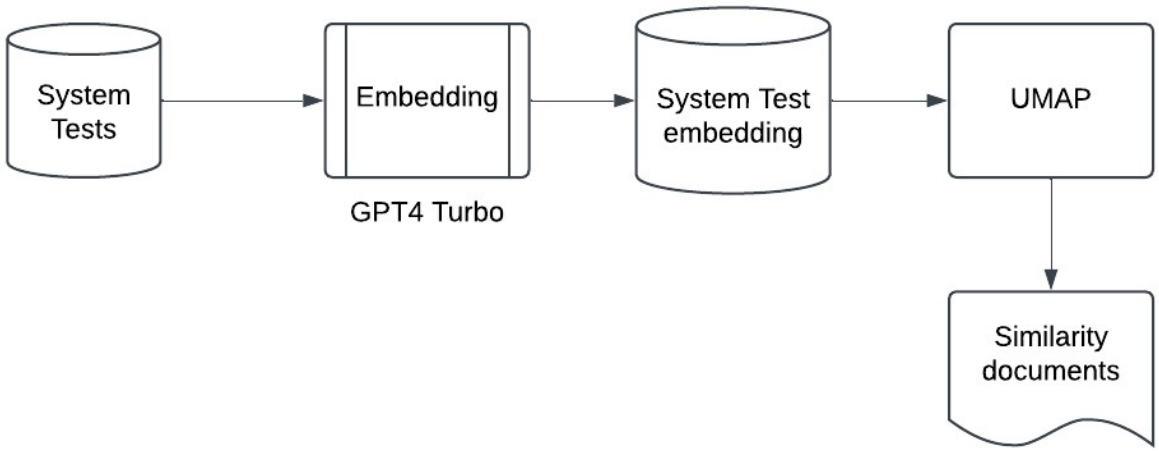
Problem: As test suites grow organically, they often accumulate redundant or overlapping test cases. With over 1200+ system tests, this becomes incredibly resource-intensive to run and

maintain, leading to increased test execution times, higher maintenance costs, and inefficient use of testing resources without necessarily improving quality.

Approach: ML semantic reasoning with UMAP-based clustering and classification algorithms was applied to identify and eliminate redundant test coverage, significantly reducing validation effort while maintaining rigorous quality standards within the existing test codebase. The objective was to ensure that every executed test provided unique and valuable coverage.

Data Sources:

- **Test Case Metadata:** This included test case descriptions, associated requirements, test case type (unit, integration, E2E), and historical modification logs.
- **Test Execution Logs:** Pass/fail status, execution time, and any associated error messages were analyzed.
- **Issue Tracking System:** Test cases were linked to reported bugs or features to understand their historical impact.



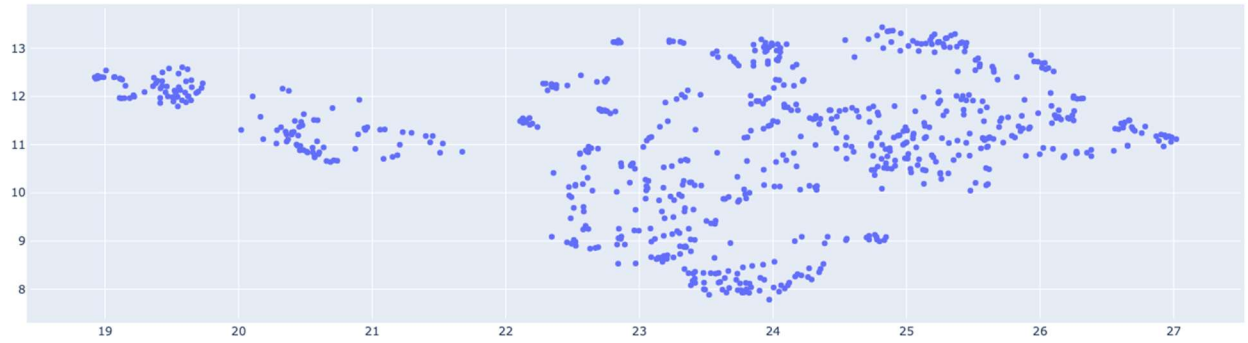
Techniques Employed:

- **Code Embedding for Test Code Analysis:** Test case code was processed by a code embedding model (specifically `GPT-4 Turbo`) to convert it into a numerical vector representation. This process was critical for capturing the semantic meaning of the code, allowing the system to understand what the code "does" beyond a simple keyword match. Once numerical representations were obtained, cosine similarity was used to measure the similarity between test cases.
- **Dimensionality Reduction and Clustering Algorithms (UMAP-based Clustering):** UMAP (Uniform Manifold Approximation and Projection) was chosen over alternatives like t-SNE or PCA because of its ability to more effectively preserve both local and global data structures in a lower-dimensional space, which is critical for correctly identifying subtle semantic relationships between test cases. K-Means clustering was applied to the

feature vectors (including code embeddings and coverage metrics) to group similar tests. Hierarchical clustering was then used to visualize the relationships between test cases and identify natural groupings, providing a more detailed view of the cluster structure. Tests within the same cluster became candidates for redundancy.

- **Classification for Redundancy:** Following clustering, a pre-trained classification service was used to pinpoint the truly redundant tests within those clusters. This service considered several factors:
    - Overlap in test coverage (e.g., which specific parts of the system were exercised by multiple tests).
    - Semantic similarity of test code—this was particularly effective at revealing cases where **nearly identical steps with minor variations** occurred across different test procedures.
    - Historical defect detection rate (tests that rarely found unique defects were strong candidates for redundancy).
    - Maintenance history (tests frequently modified together within the test management system often indicated a relationship, or potential redundancy).
- **Actionable Insights:** The output from this process provided clear recommendations for test cases that could be de-prioritized, merged, or retired. Crucially, these recommendations included justifications based on coverage and semantic overlap, facilitating review and action by test leads.

Redundancy Map for tests:



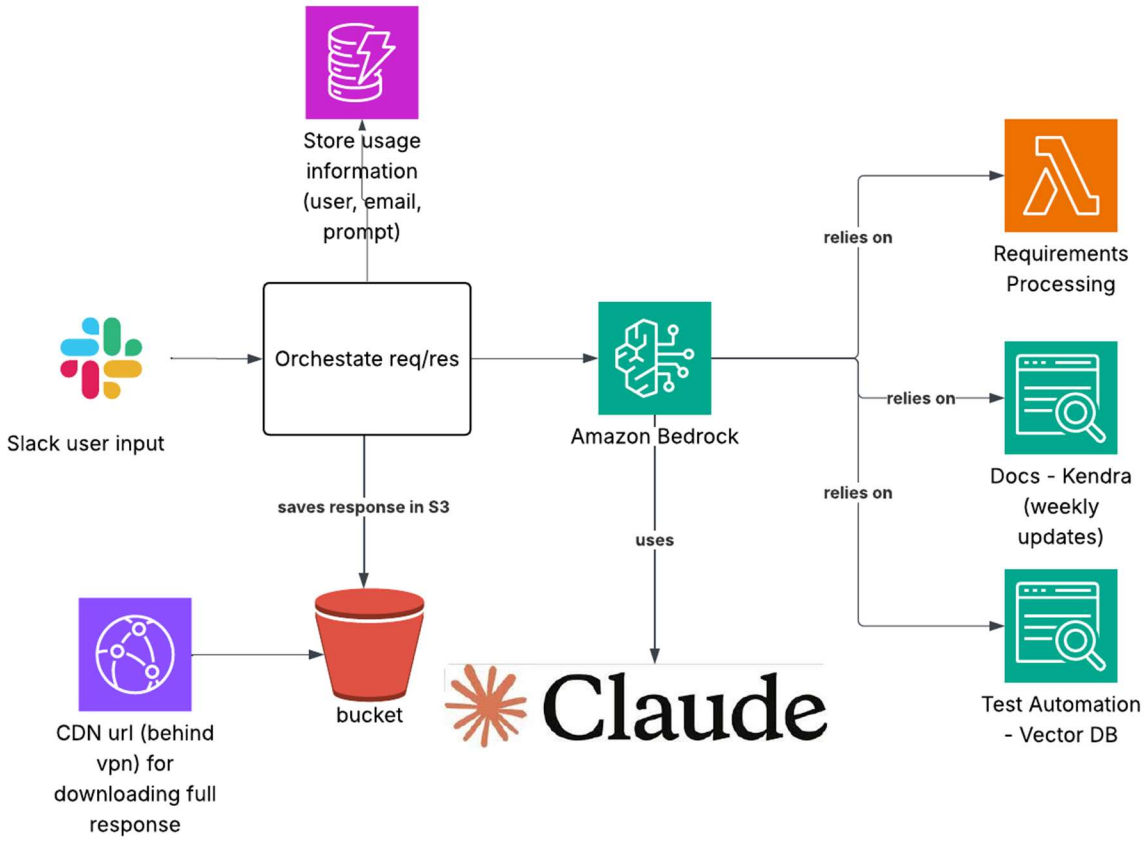
3.2. Automated Test Strategy Generation

Problem: Manually crafting targeted test plans for every new release or feature is a time-consuming task, prone to human bias or oversight. Testers often struggle to identify the most impactful tests to run given the specific changes in a release, leading to a noticeable lack of formal test planning during release cycles.

Approach: An AI agent system was developed to automatically generate targeted test plans and system test code that could adapt to the evolving scope of releases and new features.

Data Sources:

- **Requirements Management System:** Functional and non-functional requirements were pulled directly from the product requirements documents.
- **Test Case Repository:** The existing repository of test cases was used, along with their coverage data and historical execution information.
- **Kendra-based Knowledge Base:** A comprehensive knowledge base containing up-to-date public documentation about the product, serving as a primary context source for the AI agent.



Techniques Employed:

- **AI Agent-Powered Generation (LLM via Claude + AWS Bedrock):** The specialized system functions as an AI agent powered by Large Language Models (LLMs) via Claude 4 Sonnet + AWS Bedrock. This agent operates via **prompt chaining**, taking step-by-step actions and leveraging a Kendra-based knowledge base that incorporates up-to-date public product documentation as context. This intelligent process enables the agent to:
    - Infer mappings between features, tests, and owners by analyzing requirements and product documentation.

        - Generate test plans directly from requirements documents, feature names, or descriptions by understanding the semantic intent of the input.
        - Provide near-production-ready code snippets and guidance for writing high-quality automated system tests, significantly lowering the barrier to entry for test automation, speeding up test development, and promoting consistency in testing practices within the existing methodology.
- **Dynamic Test Suite Selection:** The ultimate output from this system is a prioritized list of recommended test cases or test suites, which can then be automated and triggered via CI/CD pipeline.

4. Case Study and Implementation Details

Implementing this pragmatic framework within an existing enterprise software development environment was a key part of the initiative. The essential aspect of "invisible intelligence" was ensuring these AI/ML components were lightweight services that simply consumed existing data streams and produced outputs that current tools could easily understand and use, thereby avoiding the need for new testing tools from scratch.

Architecture:

- **Data Ingestion Layer:** Python scripts and API integrations were built to regularly pull data from various sources: Jenkins/TeamCity for build logs, Jira/Confluence for requirements, issues, and release notes, and internal test management tools for test results, test case metadata, and historical modification logs.
- **Data Lake/Warehouse (Snowflake):** A centralized, accessible, and scalable source of truth for all feature-test ownership data was established in Snowflake. This serves as the data backbone, storing both raw and processed data, essential for historical analysis. For the storage of embeddings, a dedicated vector database (VectorDB) was utilized, integrated with this data infrastructure.
- **AI/ML Integration Services:** Each AI/ML component (redundancy elimination and test strategy generation) was implemented as its own microservice. These services primarily use Python to call external LLM APIs and other AI services, exposing straightforward REST APIs.
- **Integration Points:**
    - **Slack Bot for Test Management:** A Slack bot was introduced, functioning as a central quality partner for teams. It offers a range of capabilities:
        - Quickly listing tests per feature.
        - Helping identify test owners.
        - Assisting teams in writing structured test plans.
        - Checking for existing test coverage for any given scenario.
        - Proactively notifying service owners and relevant teams whenever a system test changes its lifecycle state. This ensures immediate visibility into test health and allows for swift action.

    - **Release Dashboard/Planning Tools:** Dashboards were developed to track key release metrics. This includes issues detected *before* reaching production and "triage" metrics for any unexpected increases in failures or regressions. These dashboards provide real-time risk feedback during release planning and give insights into overall coverage completeness. A dedicated Feature Coverage Dashboard, initially populated with the system's inferred mappings, offers a clear visual representation of feature test coverage.
    - **"Intelligent Test Redundancy Elimination"** service periodically analyzes the entire test suite, providing recommendations to test leads for manual review and pruning.

A Practical Example: Automated Test Strategy Generation with an AI Agent

A new feature is being built, and the product team has finalized the requirements document. The development team is actively iterating on the use cases. While Test-Driven Development (TDD) is being used for unit and integration tests, we want to leverage our AI system to accelerate the broader release cycles.

- **Step 1: Requirements Ingestion:** The comprehensive requirements document is fed into the AI system.
- **Step 2: Test Plan Generation:** The AI system, utilizing its knowledge of the product, analyzes the requirements document. It generates a comprehensive test plan with detailed test priorities and mappings to specific requirements. This test plan also includes detailed test steps and the necessary test data setup for each test.
- **Step 3: Test Code Generation:** A specific test from the generated plan is selected. The system is then queried to generate the test automation code for that particular test. Given the system's knowledge of existing test cases and the established coding patterns, it generates a near-production-ready system test automation. This significantly accelerates the test development phase and ensures consistency in the test codebase.

Consider the scenario of the feature: *"Implemented new user authentication flow with OAuth2 support"*

- **Data Ingestion:** The requirements document is fed to the AI system.
- **NLP Processing (by the AI Agent):**
    - The AI agent tokenizes the release notes and extracts key phrases: "user authentication," "OAuth2 support", "user profile."
    - Crucially, the AI agent, leveraging its LLM capabilities, infers mappings between these features and the relevant functional areas and existing test cases, drawing from its extensive knowledge base of internal documentation.
- **Output:** The system then generates a prioritized list of test cases.
    - `TC_AUTH_001: Verify successful login with valid credentials.` (High priority, as it is core functionality)

- `TC_AUTH_005: Test OAuth2 token refresh mechanism.` (High priority, directly related to the new feature)
- `TC_SEC_010: Authentication bypass vulnerability test.` (Medium priority, given the security implications of a new auth flow)
- `TC_UI_003: Check the user dashboard display after login.` (Medium priority, potentially impacted by either the authentication or profile changes)
- `TC_PROFILE_007: Verify user profile fields display correctly.` (High priority, directly addressing the UI glitch fix)

5. Results and Discussion

The incremental embedding of AI/ML capabilities, particularly through the specialized AI system, has yielded significant positive outcomes across release testing processes.

5.1. Efficiency Gains

- **Reduced Test Execution Time:** By intelligently eliminating redundant tests (e.g., identifying cases of "same steps with just one line change" across test cases) and generating targeted test strategies, a remarkable **30-40% reduction in overall test execution time** for major releases has been observed. This translates to faster feedback loops and earlier detection of issues.
- **Faster Release Cycles:** The confidence gained from predictive risk assessment and optimized testing has allowed for a **15% reduction in the average time-to-market** for new features. Teams can now release with greater certainty and less manual gatekeeping.
- **Optimized Resource Utilization:** Testing teams can now reallocate resources from simply executing redundant tests to more valuable activities such as exploratory testing, performance testing, or developing entirely new test automation. This has led to a more strategic use of human capital.

Metric	Baseline	Post AI Integration	% Improvement
Test Execution Time	3.12 hrs	1.64 hrs	47.44%
Release Cycle Time-Test planning	2 weeks	1 day	92.86%
Release Cycle Time - Test Automation	3 Weeks	3 days	85.71%

5.2. Discussion and Limitations

While the results are highly promising, it is crucial to acknowledge the limitations and ongoing needs of the system.

Limitations:

- **Dependency on Third-Party Models:** The effectiveness of the system is dependent on the performance and consistency of the external AI/ML services used. If an external model experiences **model drift**, its recommendations may degrade. This necessitates a regular review of the prompts and service configurations to ensure their continued accuracy and effectiveness.
- **Edge Cases:** While the AI-generated test plans are highly effective for standard features and bug fixes, they can occasionally fail to account for highly specific or unexpected **edge cases** that a human tester with deep domain knowledge would recognize. A feedback loop from manual exploratory testing is essential to continuously improve the system's recommendations.

Future Work: Future enhancements to the system include integrating predictive analytics for performance testing, expanding the AI agent's capabilities to automatically prioritize and recommend tests based on code changes from GitHub Copilot's recommendations, and exploring the use of reinforcement learning to further optimize the test suite over time. We also propose creating a **Model Context Protocol (MCP)** to serve as a standardized, plug-and-play tool that allows the system to seamlessly integrate with any developer ecosystem. This protocol will manage the context and data exchange between the AI/ML services and various developer tools, ensuring maximum velocity for new features.

6. References

- McInnes, L., Healy, J., & Melville, J. (2018). UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *arXiv preprint arXiv:1802.03426*. https://arxiv.org/abs/1802.03426
- Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach*. Pearson. https://books.google.com/books/about/Artificial_Intelligence.html?id=koFptAEACAAJ
- Muller, R., & Padberg, J. (2003). Automated Test Case Generation for UML Models. In *Software Engineering and Applications (SEA)*. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=2e23b066ff5b44ae7b7a51c716a0fe7b5d72ae63
- Kohavi, R., & Provost, F. (1998). Glossary of Terms for Knowledge Discovery and Machine Learning. *Machine Learning, 30*(2-3), 271-274. https://www.scirp.org/(S(ny23rubfvg45z345vbrepxrl))/reference/referencespapers?referenceid=2264480
- Brown, T. B., Mann, B., Ryder, N., et al. (2020). Language Models are Few-Shot Learners. *arXiv preprint arXiv:2005.14165*. https://arxiv.org/abs/2005.14165