AI Solution Architecture

This section details the technical AI architecture for the AI-Driven QA Automation Platform, including LLM-based test generation, RAG for requirements retrieval, self-healing mechanisms, and the continuous learning pipeline.

Architecture Overview

The platform implements an **LLM + RAG + Specialized ML Hybrid Architecture**. A foundation LLM generates test scripts from natural language requirements, grounded by RAG retrieval over requirements documents. Specialized ML models handle self-healing (DOM change adaptation) and flaky test detection, while a continuous learning engine improves coverage over time.

Core Components:

1. **Test Generation Engine:** LLM-powered conversion of requirements to executable test scripts
2. **Requirements RAG Pipeline:** Retrieval over PRDs, user stories, and acceptance criteria for context-aware generation
3. **Self-Healing Module:** ML-based element locator adaptation when UI changes break tests
4. **Flaky Test Detector:** Classification model identifying non-deterministic test failures
5. **Continuous Learning Engine:** Feedback loop improving test quality from execution results

LLM-Based Test Generation

Model Selection

Use Case	Recommended Model	Rationale
Test Script Generation	Claude 3.5 Sonnet or GPT-4 Turbo	Superior code generation; 128K+ context for full requirement sets; strong instruction following
Test Maintenance	GPT-4o-mini or Claude 3 Haiku	Cost-efficient for high-volume locator updates and minor script modifications
On-Premise Option	Code Llama 70B or DeepSeek Coder 33B	No external API calls for sensitive government/enterprise environments

Generation Pipeline

1. **Requirement Parsing:** Extract testable assertions from user stories and acceptance criteria via structured extraction prompt
2. **Context Retrieval:** RAG fetches related requirements, existing test patterns, and application context
3. **Test Generation:** LLM generates test script with assertions, setup/teardown, and parameterization
4. **Validation:** Syntax checking, linting, and dry-run execution in sandbox environment
5. **Human Review:** QA engineer approval before promotion to regression suite

Prompt Template: Test Generation

```
[SYSTEM]
```

```
You are a senior QA automation engineer. Generate complete, executable test
scripts following best practices: explicit waits, descriptive assertions,
proper error handling, and data parameterization. Output in {framework}
format (Selenium/Playwright/Cypress).
[CONTEXT]
Application: {app_name} | Module: {module} | Test Type:
{functional|integration|regression}
Requirements: {retrieved_requirements}
Existing Patterns: {similar_tests_from_rag}
[OUTPUT]
Generate: 1) Test class with setup/teardown, 2) Individual test methods with
Given/When/Then comments, 3) Assertions covering all acceptance criteria, 4)
Data-driven parameters where applicable.
```

Requirements RAG Pipeline

RAG grounds test generation in actual requirements, reducing hallucination and improving requirement traceability.

Component	Specification
Document Corpus	PRDs, user stories (Jira/Azure DevOps export), acceptance criteria, UI specifications, API contracts (OpenAPI), existing test suites
Embedding Model	text-embedding-3-large for requirements text; CodeBERT embeddings for existing test code similarity
Vector Database	Weaviate (hybrid BM25 + vector) or ChromaDB (lightweight self-hosted); metadata filtering by project, sprint, component
Chunking Strategy	User story-level chunks (one story = one chunk); acceptance criteria as sub-chunks with parent linkage; code files chunked by function/class
Retrieval Method	Query decomposition (split requirement into testable components); retrieve top-10, rerank to top-5; include at least one similar existing test for style consistency

Self-Healing Test Maintenance

When UI changes break element locators, the self-healing module automatically proposes fixes without manual intervention.

Mechanism

6. **Failure Detection:** Test execution captures ElementNotFound exceptions with DOM snapshot at failure point
7. **Candidate Generation:** ML model (trained on historical locator→element mappings) proposes alternative selectors based on element attributes, position, and surrounding context
8. **Validation:** Proposed locator tested against current DOM; highest-confidence match selected
9. **Auto-Repair:** If confidence > 85%, locator updated automatically; otherwise flagged for human review
10. **Learning:** Human-approved fixes feed back into training data for model improvement

Model Architecture

- **Base Model:** Siamese network comparing original element features to candidate elements in changed DOM

- **Features:** Element tag, class names, ID, text content, XPath position, CSS selectors, visual position (if available), parent/sibling context
- **Training Data:** Historical element changes from version control; UI change logs; manually mapped element migrations
- **Performance:** Target 80%+ auto-repair success rate for simple locator changes (ID/class renames); 60%+ for structural changes

Flaky Test Detection

A classification model identifies tests with non-deterministic failures, distinguishing true bugs from environmental/timing issues.

- **Model:** Random Forest classifier on test execution metadata
- **Features:** Pass/fail history variance, execution time variance, failure error types, time-of-day patterns, concurrent test interference signals, resource utilization at failure
- **Output:** Flakiness score (0-100); tests >70 flagged for quarantine and root cause analysis
- **Actions:** Auto-quarantine flaky tests from blocking CI/CD; generate suggested fixes (add waits, isolate test data, etc.)

Continuous Learning Pipeline

The system improves over time by learning from test execution outcomes and human feedback.

- **Feedback Signals:** Test pass/fail results, human edits to generated tests, defects caught vs. missed, coverage gaps identified in production incidents
- **Prompt Refinement:** A/B testing of prompt variations; winning prompts promoted based on test quality metrics
- **RAG Corpus Updates:** High-quality generated tests (human-approved, high coverage) added to retrieval corpus as examples
- **Model Retraining:** Self-healing and flaky detection models retrained monthly with new labeled data

Infrastructure & Compliance

Requirement	Specification
Generation Latency	< 30 seconds for single test generation; < 5 minutes for full requirement set (batch)
Self-Healing Latency	< 10 seconds per locator repair attempt
Deployment	Azure Government (FedRAMP) or AWS GovCloud for government clients; private cloud option for on-premise requirements
Traceability	Full audit trail: requirement ID → generated test ID → execution results; immutable logging for compliance
CI/CD Integration	Jenkins, Azure DevOps, GitHub Actions plugins; webhook triggers for test generation on story completion

[End of AI Solution Architecture Section]