

AI-Driven Automation of Software Testing to Enhance Software Quality and Enable Agentic AI

Ibrahim El Samman

Arab Academy for Science, Technology, and Maritime Transport

Mohammed Safy

msafy@aaast.edu

Arab Academy for Science, Technology, and Maritime Transport

Research Article

Keywords: Agentic AI workflow, Automated test case design, multi-agent architecture, Self-healing loop, Mutation score

Posted Date: October 23rd, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-11254037/v1>

License:  This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

Abstract

As modern systems evolve into complex microservices and highly regulated data ecosystems, manual test case design has become a severe bottleneck in Agile CI/CD pipelines. While Artificial Intelligence (AI) accelerates code synthesis, standard Generative AI models persistently struggle with the "Oracle Problem." They treat test generation as a stateless translation task, lacking the strategic reasoning necessary to navigate multi-layered industrial requirements without hallucinating logic.

To bridge this massive "Design Gap," we propose the Agentic-QA Fusion Framework. This stateful, multi-stage architecture transforms automated test generation into an active reasoning ecosystem. Utilizing a specialized "actor-critic" loop comprising Parser, Triage, Architect, Writer, and Auditor agents, the framework autonomously strategizes, generates, and self-heals test scripts.

The framework's cross-industry scalability is empirically evaluated against baseline ML and deterministic LLMs across five complex datasets. This includes baseline commercial platforms (ISIS Travel, Travco Travel, Vantage Towers) and two highly constrained domains: the European Securities and Markets Authority (ESMA) FIRDS system (requiring strict ISO 20022 messaging) and a governmental Job Matching Platform (JMP).

By shifting the focus upstream to cognitive requirement analysis, the proposed framework successfully parses deeply nested regulatory logic. It drastically reduces hallucination and redundancy rates to 1.3% and 7.8%, respectively. Furthermore, it guarantees near-perfect syntactic correctness (99.6%) and achieves an industry-leading Mutation Score of 93.4%. Ultimately, while this Agentic approach necessitates higher computational resources to resolve dense enterprise dependencies, it establishes a reliable, zero-touch automation solution vital for modern software quality engineering.

Keywords Agentic AI workflow; Automated test case design; multi-agent architecture; Self-healing loop; Mutation score.

Introduction

In the rapidly expanding digital economy, software testing is recognized as an intrinsic and critical part of the software development lifecycle, where it serves as the primary safeguard for the reliability, security, and performance of modern applications [22]. Over the past decade, software systems have evolved from monolithic structures into highly complex, distributed, and microservices-based architectures [58]. As this structural complexity has grown exponentially, the processes used to verify these systems have struggled to keep pace [22]. Traditional manual testing is widely characterized as labor-intensive, costly, and highly vulnerable to human error [84]. Consequently, manual test case design has become a severe bottleneck in agile and continuous integration environments, where rapid, flawless deployment is required as a fundamental business requirement [93]. Traditional test automation approaches, which are heavily

reliant on script-based frameworks, demand continuous maintenance as application changes occur, often leading to inefficiencies and flakiness [90]. This has driven a necessary industry shift toward advanced artificial intelligence automation to mitigate costs and improve coverage for the test design and generating proper test cases [9].

To mitigate these systemic bottlenecks, software engineering research has increasingly leveraged Artificial Intelligence [1,2]. However, a significant "Design Gap" remains prevalent in both contemporary literature and industrial practice; current academic efforts disproportionately favor downstream procedural code synthesis over the critical upstream phase of strategic requirement analysis. When legacy Machine Learning (ML) algorithms and zero-shot Large Language Models (LLMs) are applied to industrial software testing [22], they mainly frame test generation as a stateless, localized string-translation task, inherently suffering from a pervasive "contextual blind spot."

When challenged with the architecture of modern systems, ranging from consumer-facing platforms such as ISIS Travel [3] and Travco [4], and complex B2B telecommunications architectures like Vantage Towers [5], to highly regulated ecosystems such as the ESMA FIRDS [6] and the governmental Job Matching Platform (JMP) [7], these legacy AI models fail to retain the context of multi-step business logic. Consequently, they become susceptible to the "Oracle Problem" [8] producing voluminous but functionally flawed scripts characterized by high hallucination rates and syntactic inaccuracies, which necessitate exhaustive and costly manual refinement.

This gap appears clearly as quantified by recent comprehensive reviews: the vast majority of AI studies (over 56.65%) are focused on downstream code development and generation, while a mere 3.90% address upstream requirements engineering and strategic design [9]. While current automated tools are excelled at the algorithmic generation of code, heavy focus is placed by this paper on bridging this critical gap in the initial strategic design and analysis phase.

The evolution of automated testing has undergone a massive paradigm shift. Historically, automation relied on static, rule-based scripts that were rigid and required constant manual updates. The field was then shifted toward Machine Learning (ML) and Large Language Models (LLMs), where algorithms were leveraged for pattern recognition, defect prediction, and syntactic code synthesis [10,11]. However, standard ML treats test generation as a stateless "Text-to-Code" translation task, where an understanding of broader business logic is entirely lacked.

To overcome this, a pivot toward Agentic AI is now being made by the industry. While traditional ML provides pattern recognition, Agentic AI introduces autonomous reasoning capabilities [13,14]. A specialized architecture of autonomous agents featuring Perception, Cognitive, and Action modules is utilized by Agentic AI, by which test scenarios are dynamically planned, orchestrated, and executed. As demonstrated by Paduraru et al. [12], complex user stories are effectively mapped into functional test implementations by integrating

LLMs with natural language processing and agentic workflows, where the strategic logic of a human QA engineer is successfully mimicked.

Despite the rapid rise of LLMs [37], a notable gap is observed in consistently generating executable and logic-accurate test cases from ambiguous requirements without heavy human intervention. Standard Generative AI models suffer from a 'context blind spot,' frequently hallucinating non-existent features [15] or data dependency issues are struggled with when confronted with specific tasks outside their training environment. Furthermore, they suffer from the 'Oracle Problem,' frequently producing scripts with syntax errors or invalid assertions that cause compilation failures. As observed by Yi et al. [16], tests generated by standard LLMs often fail to execute due to hallucinated method requests, forcing human testers to manually refine them.

To definitively resolve the statelessness and hallucination vulnerabilities inherent to legacy automation, this research proposes the Agentic-QA Fusion Framework and presents a rigorous empirical approach to its validation [17,18]. The detailed approach marks a paradigm shift from passive generation to an active, stateful reasoning ecosystem. The methodology deploys an autonomous, multi-agent architecture structured around Perception, Cognitive, and Action modules. Specifically, the framework decomposes the testing lifecycle into a dynamic "actor-critic" pipeline utilizing five specialized roles: a Parser agent to dissect dense natural language; a Triage agent to categorize dependencies; an Architect agent to define optimal equivalence classes and structural boundaries; a Writer agent to synthesize the scripts; and a critical Auditor agent functioning as an internal compiler to recursively self-heal structural drift before final output. By integrating strict circuit breakers, the framework prevents infinite refinement loops while ensuring compliance with complex architectural constraints.

To empirically validate the capability of this approach, the experimental design deliberately encompasses a broad spectrum of industrial complexity:

- Baseline Commercial Domains: ISIS Travel (B2C booking), Travco (international hospitality), and Vantage Towers (B2B telecommunications) establish the framework's baseline effectiveness across common commercial sectors.
- ESMA (European Securities and Markets Authority): The FIRDS Functional Specification introduces extreme logical density, requiring the parsing of ISO 20022 message definitions and multi-layered business rules for determining the Relevant Competent Authority (RCA) based on dynamically shifting Market Identifier Codes (MIC) and Classification of Financial Instruments (CFI) constructs.
- JMP (Job Matching Platform): A national-scale governmental labor market infrastructure demanding the validation of complex integrations, including Vector-Based AI matching algorithms, semantic resume parsing via NLP, and secure RESTful Push API synchronization governed by Schema.org JobPosting parameters.

By executing six distinct AI architectures ranging from Classical Random Forest and Heuristic Genetic Algorithms to the proposed Agentic workflow across these five disparate datasets, this research isolates the precise hardware-capability gaps of legacy models and definitively establishes the necessity of stateful multi-agent frameworks for true, zero-touch enterprise automation.

To establish a rigorous evaluation framework, the comparative analysis of the six AI solution architectures was anchored by a set of Key Performance Indicators (KPIs). These metrics are derived from established industry standards including ISTQB [95], ISO/IEC/IEEE 29119 [19, 92], and Gartner [20] frameworks ensuring the empirical results are both valid and comparable to industry benchmarks [84,93,94].

The KPIs are categorized into Core KPIs, which provide the essential benchmark for architectural performance, and Supporting KPIs, which provide context regarding the efficiency and usability of the generated test suites. Utilizing this framework across the ISIS, Travco, Vantage Towers, ESMA, and JMP datasets, the comparative matrix ensures that the performance of the Agentic AI is not measured simply by "how many tests it created," but by how many "correct, unique, and functionally sound tests" it produced relative to the industrial complexity of the source requirement.

Materials and Methods

To rigorously evaluate the capability of Artificial Intelligence in transitioning from algorithmic test generation to strategic test case design, a comparative empirical experiment is designed. The proposed approach is centered on evaluating six distinct artificial intelligence solutions against real-world industrial specifications.

To acquire authentic, enterprise-scale data without violating proprietary corporate boundaries, three prominent, complex websites representing different industrial sectors are selected:

- ISIS Travel: A comprehensive B2C travel booking engine.
- Vantage Towers: A B2B telecommunications infrastructure platform managing thousands of network sites.
- Travco: A multi-service international hospitality and tourism ecosystem.

Because real-world test case design relies heavily on upstream requirements [21,64,65,66], Perplexity.ai is utilized to reverse-engineer and extract comprehensive, professional-grade project requirements for these three platforms as if they were being developed from scratch. These extractions are consolidated into three distinct requirement documents. Six different AI technologies (Solution Architectures) are then individually executed against each document, by which test designs are automatically generated and the results are output into distinct Excel sheets (6 sheets per project) for subsequent comparative analysis.

Six distinct computational pipelines implemented via Python within Google Colab notebooks are developed and executed to process the requirements and generate the test cases. These models, spanning the evolutionary spectrum of AI, are allowed to benchmark traditional machine learning against advanced generative and agentic workflows:

- **Solution Architecture 1 - Classical Machine Learning (Random Forest):** A baseline is established using a traditional Random Forest Classifier [87]. TF-IDF features are extracted by this model based on the length and numerical density of requirement sentences to categorize testing complexity.
 - **High-level design:** As illustrated in Fig. 1, this architecture departs from generative deep learning and leverages classical, supervised Machine Learning methodologies. It utilizes a Random Forest ensemble classifier operating on meticulously engineered linguistic features. This approach is highly interpretable, resource-efficient, and treats requirement extraction and test classification as a distinct pattern recognition task.

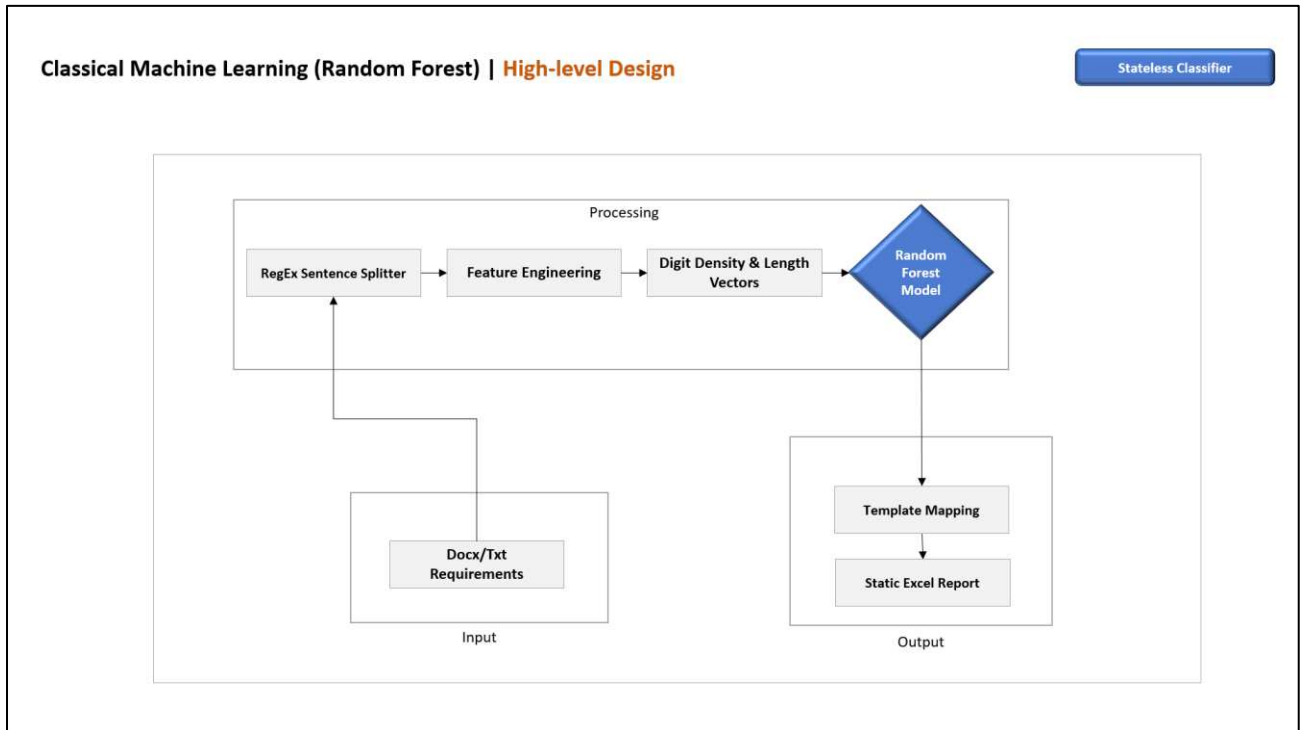


Fig. 1. Random Forest High-level Design

- **Low Level Design:** As detailed in Fig. 2, the execution initiates by uploading a document, which is subsequently routed to the appropriate parser based on its file extension (TXT or DOCX). A sentence extraction module segments the parsed text into an array of discrete linguistic units. These units are passed to the Machine Learning module, where a feature engineering function transforms the raw text into numerical feature vectors (e.g., using

TF-IDF or syntactic dependency metrics). A Random Forest model is then trained (or applied if pre-trained) on these features to classify the sentences and identify actionable software requirements. Once classified, a deterministic generation script maps the identified requirements into structured test cases. These parsed entities are then assembled into a tabular spreadsheet format and exported as a final downloadable Excel file.

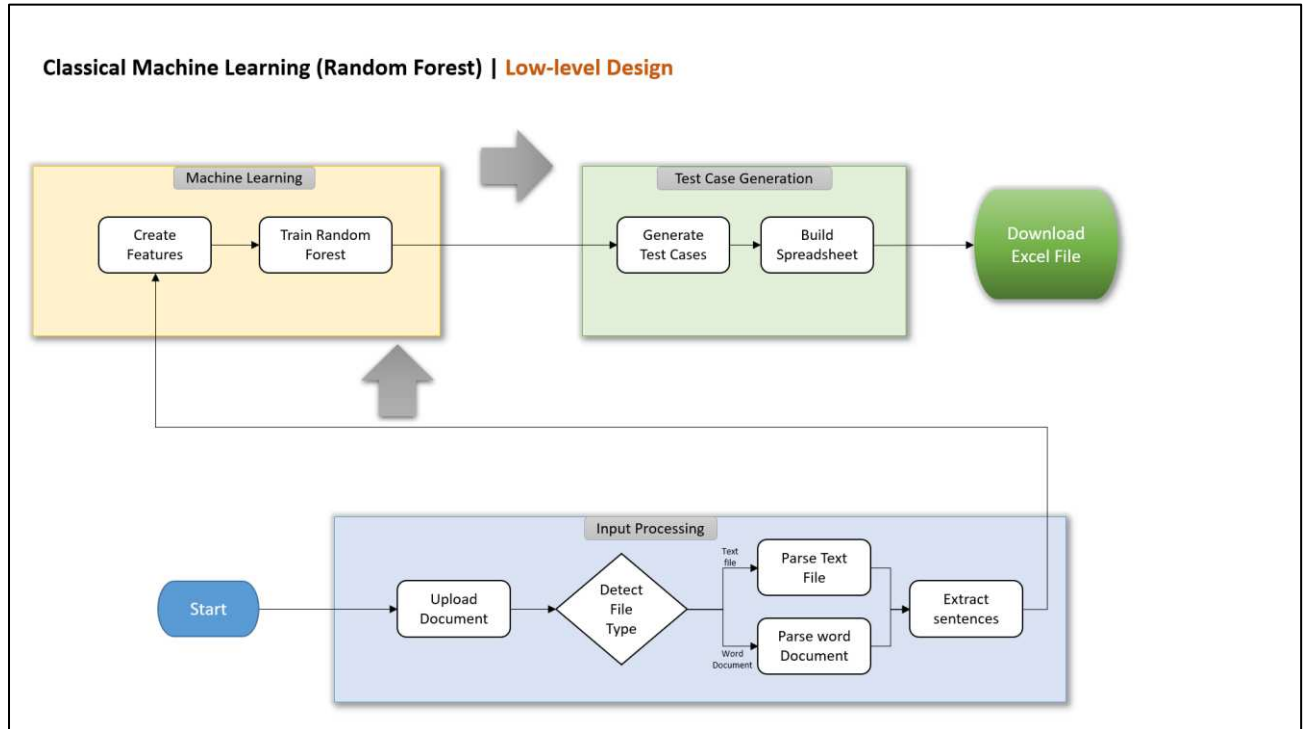


Fig. 2. Random Forest Low-level Design

- **Solution Architecture 2 - Deterministic NLP (T5-Small):** An Encoder-Decoder architecture is implemented using the Hugging Face t5-small model. However, catastrophic rigidity is revealed by early trials; implied negative tests (the "what if" scenarios) are failed to be captured by the model and manual tuning is required whenever the input format is changed.
 - **High-level design:** As presented in Fig. 3, this architecture defines a lightweight, highly optimized text-to-text generation pipeline utilizing the T5-Small model. It is expressly designed for speed, determinism, and execution within highly constrained environments. By disabling gradient tracking and locking the model state, this "hardened" flow prioritizes reproducibility and computational efficiency over complex, emergent reasoning.

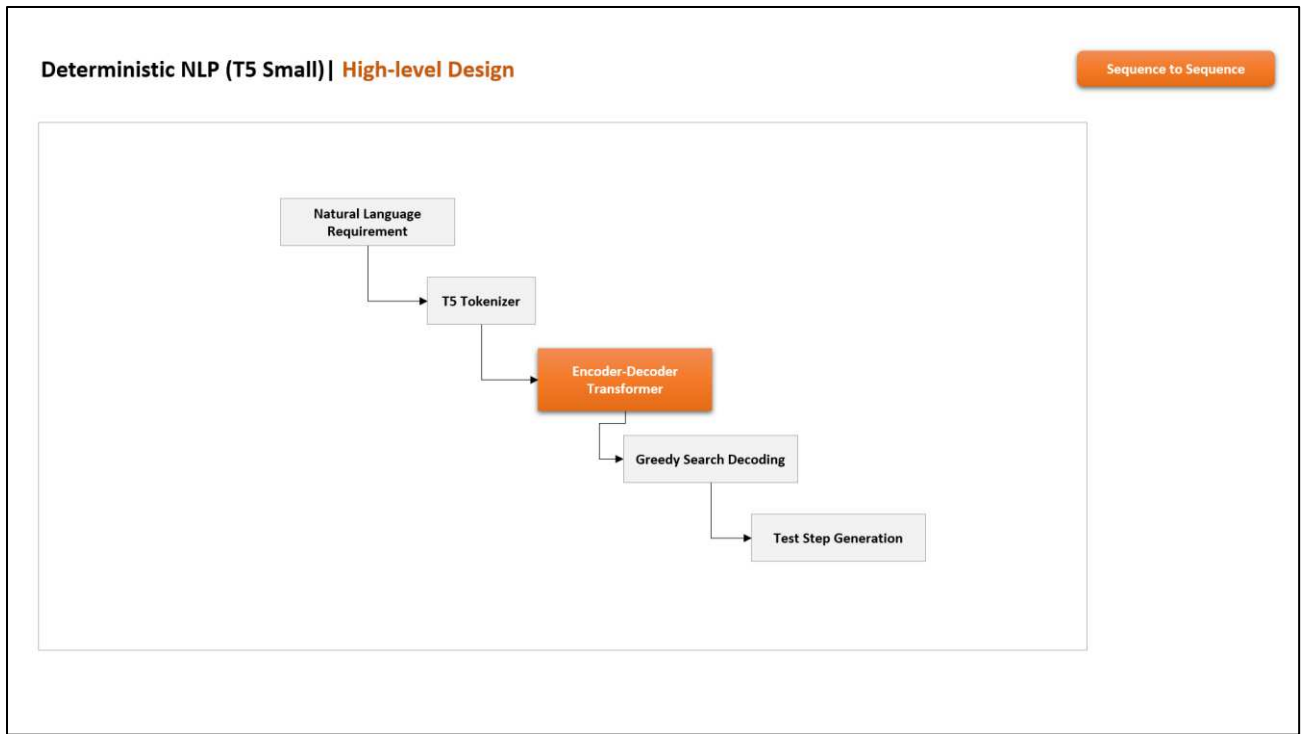


Fig. 3. Deterministic NLP (T5-Small) High-level Design

- **Low Level Design:** As depicted in the flowchart of Fig. 4, the system boots by establishing the environment and executing a strict device selection protocol (routing to CUDA if available, or falling back to CPU). The document handler accepts and validates the uploaded file's MIME type, employing specific parsing logic for TXT versus DOCX formats; unsupported types trigger immediate execution termination. The tokenizer and T5-Small model are loaded, migrated to the selected hardware device, and explicitly set to *evaluation mode* (disabling stochastic elements like dropout layers). Extracted text is tokenized and chunked. The inference engine then executes the generative pass exclusively within a *no_grad()* context, ensuring deterministic output and minimizing memory overhead. Generated tensor outputs are decoded into human-readable strings, structured into corresponding test case rows, and appended to an Excel output file for download.

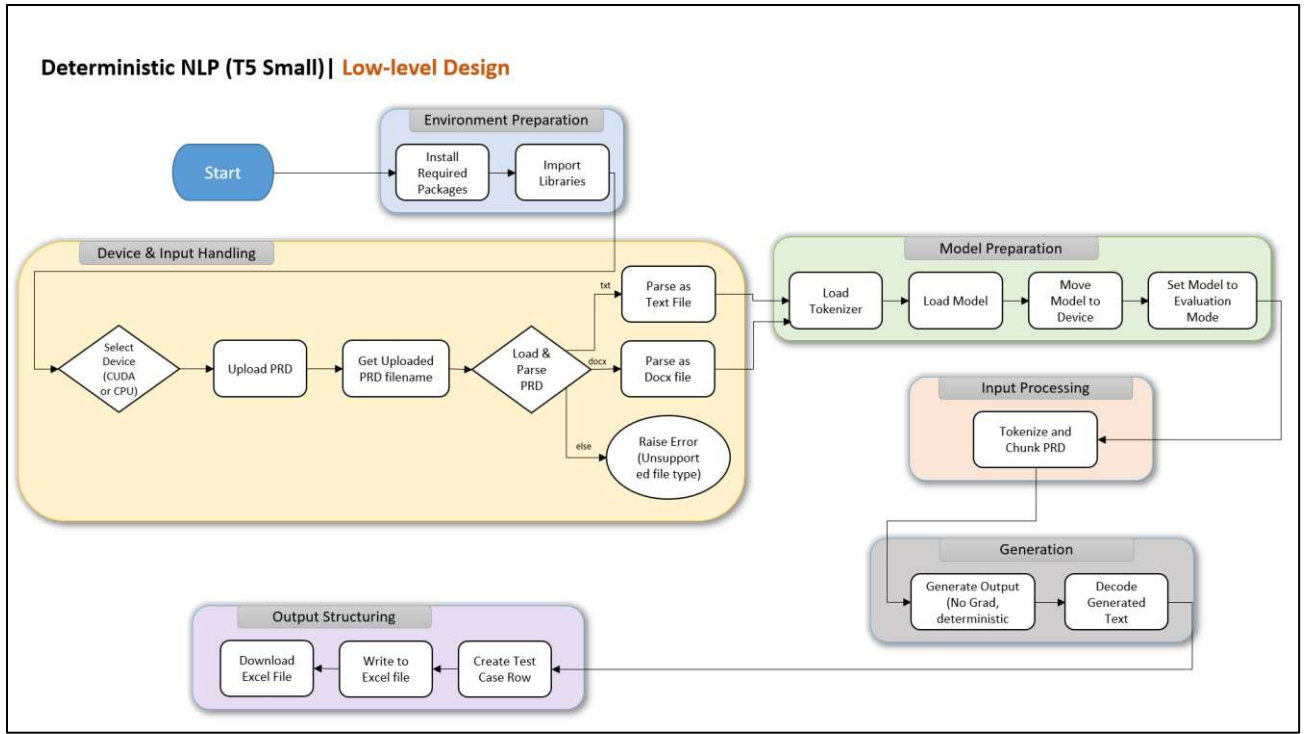


Fig. 4. Deterministic NLP (T5-Small) Low-level Design

- **Solution Architecture 3 - Generative AI with RAG (Retrieval-Augmented Generation):** Requirements are embedded into a FAISS vector index using Sentence Transformer to anchor generation in the actual documentation. A FLAN-T5 model is then queried to synthesize tests based on retrieved context. During implementation, "chunking errors" are encountered and engineered around, where requirements are cut mid-sentence by standard splitting strategies, leading to technically valid but contextually hallucinated test cases.
 - **High-level design:** As shown in Fig. 5, the Retrieval-Augmented Generation (RAG) architecture is engineered to enhance the factual accuracy and traceability of generated test cases. By decoupling the knowledge base from the language model's parametric memory, the system dynamically retrieves semantically relevant document context via a vector database, grounding the LLM's generative output in specific, verifiable requirements.

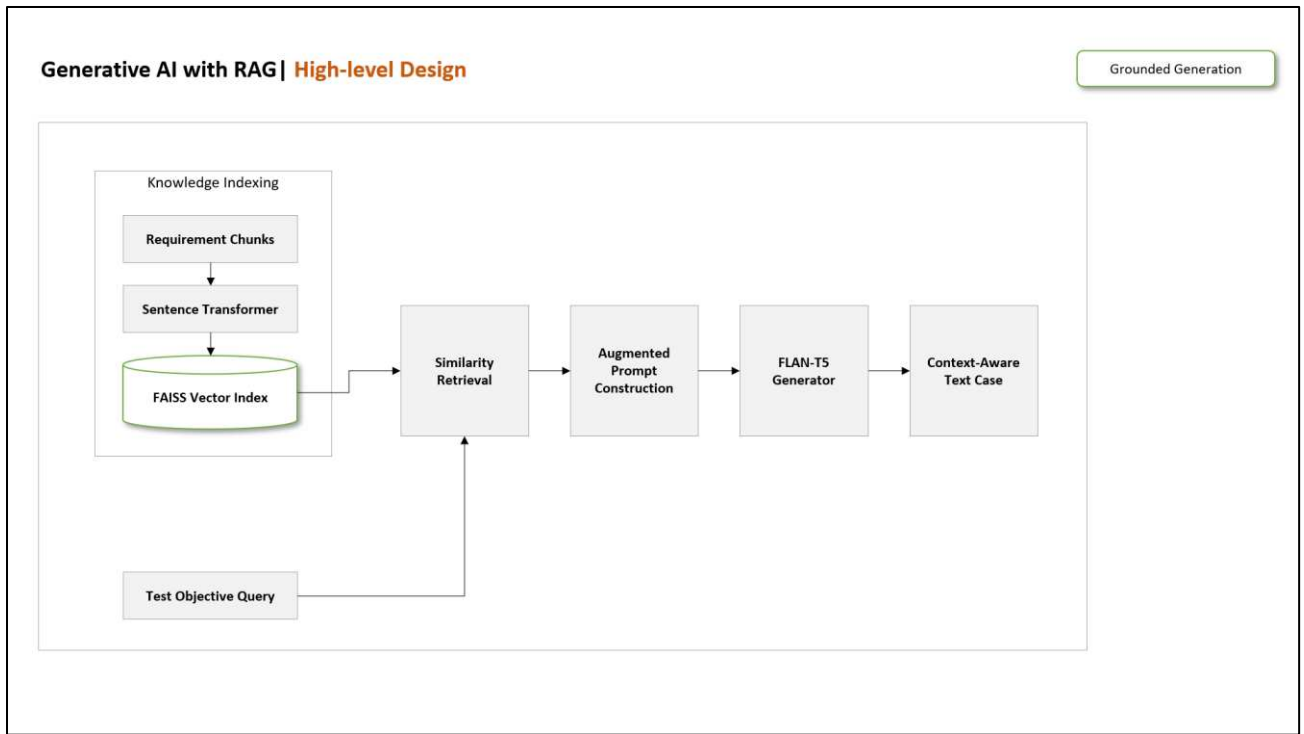


Fig. 5. Generative AI with RAG (Retrieval-Augmented Generation) High-level Design

- **Low Level Design:** Following the data path outlined in Fig. 6, post-initialization, the document input module ingests TXT or DOCX files and applies a chunking algorithm to divide the corpus into manageable, normalized text segments. The "Vector Store" module passes these chunks through an embedding model to generate dense, multi-dimensional vector representations. These vectors are subsequently indexed using FAISS (Facebook AI Similarity Search) to enable rapid nearest-neighbor retrieval. During the generation phase, the pipeline processes requirements iteratively. For each input query, the system polls the FAISS index to retrieve the most contextually similar text chunks. A composite prompt is built, synthesizing the generation instruction with the retrieved context. The language model processes this prompt to yield a highly specific test case. This loop continues until all chunks/queries are exhausted, culminating in the collection and export of the structured results.

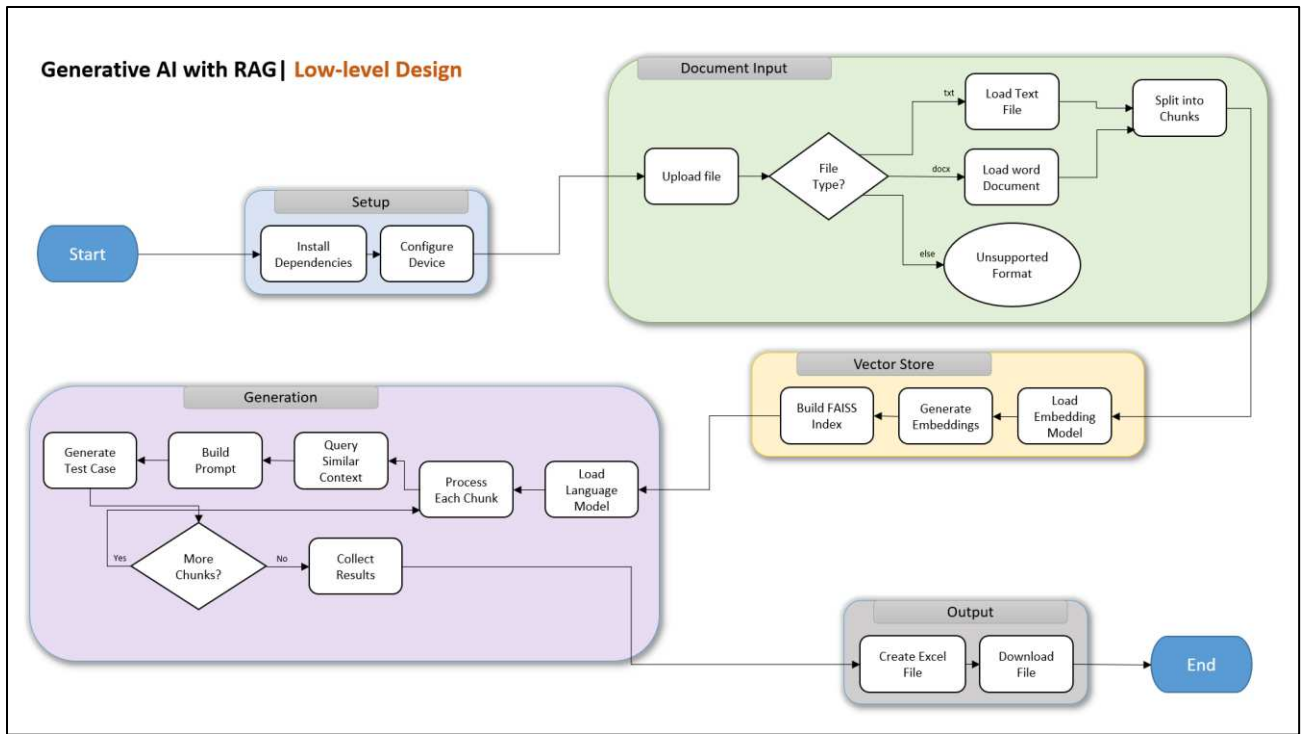


Fig. 6. Generative AI with RAG (Retrieval-Augmented Generation) Low-level Design

- Solution Architecture 4 - Heuristic Optimization (Genetic Algorithm + AI Fitness):** A novel hybrid approach is designed where structural test skeletons (genes) are generated by a Genetic Algorithm, and an LLM (FLAN-T5-base) is acted as an ISTQB [95] expert evaluator. The fitness of the test case against the requirement is scored by the LLM on a 0-10 scale across iterative generations.
 - High-level design:** Fig. 7 illustrates a hybrid evolutionary computing framework that merges traditional Genetic Algorithms (GA) with Large Language Models (LLMs). The system treats test case generation as an optimization problem. Instead of relying on static heuristics, an AI model serves as a dynamic fitness function, evaluating the semantic quality of candidate test cases and guiding the evolutionary process toward optimal test coverage.

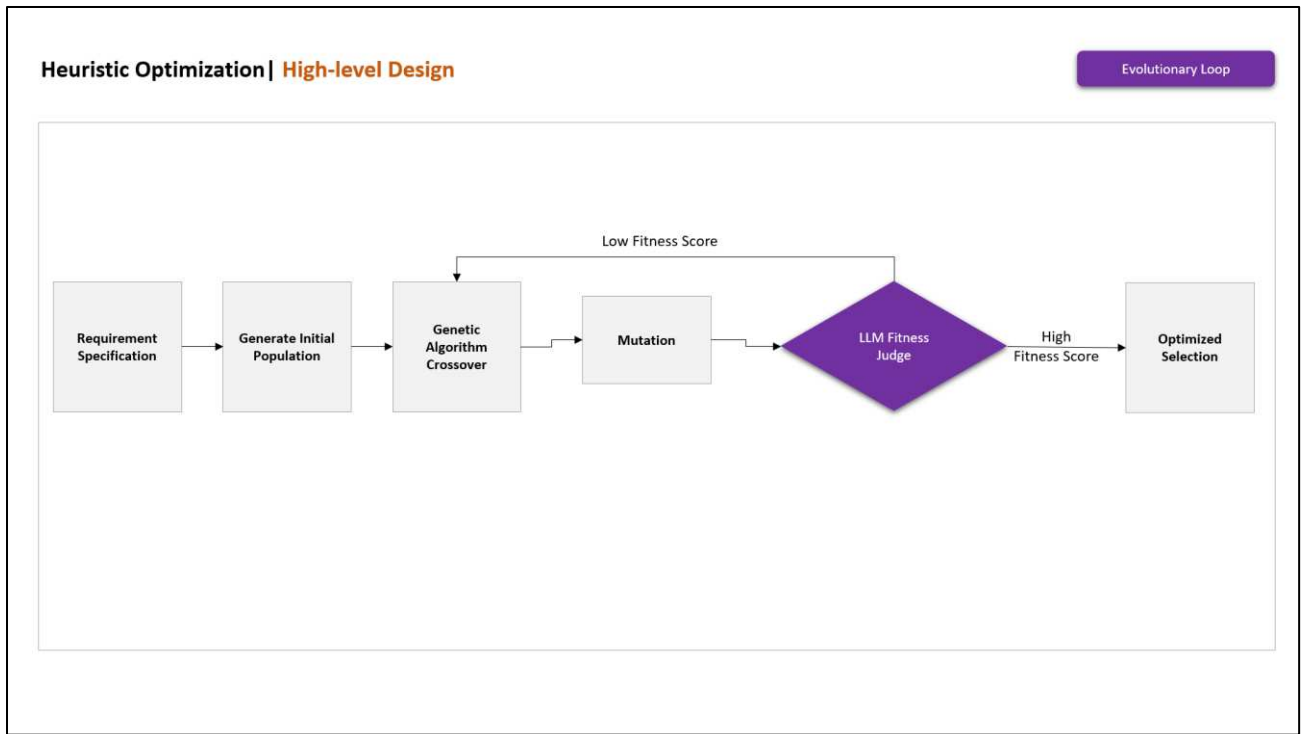


Fig. 7. Heuristic Optimization (Genetic Algorithm + AI Fitness) High-level Design

- **Low Level Design:** As mapped out in Fig. 8, following device configuration and AI model initialization, the system parses incoming requirements from TXT or DOCX files. The GA phase begins by initializing a randomized population of potential test cases. During the "Generation Loop," the system delegates evaluation to an external AI module. Within this AI Evaluation sub-process, an evaluation prompt is constructed and transmitted to the LLM (acting as the TS Model). The natural language response is parsed to extract a quantitative fitness score. Based on these scores, the GA selects parent solutions, applies crossover to generate new cases, and introduces stochastic mutation to maintain population diversity. This loop iterates until a convergence criterion is met, at which point the optimal test solutions are extracted, serialized into test case records, and formatted into an Excel spreadsheet for deployment.

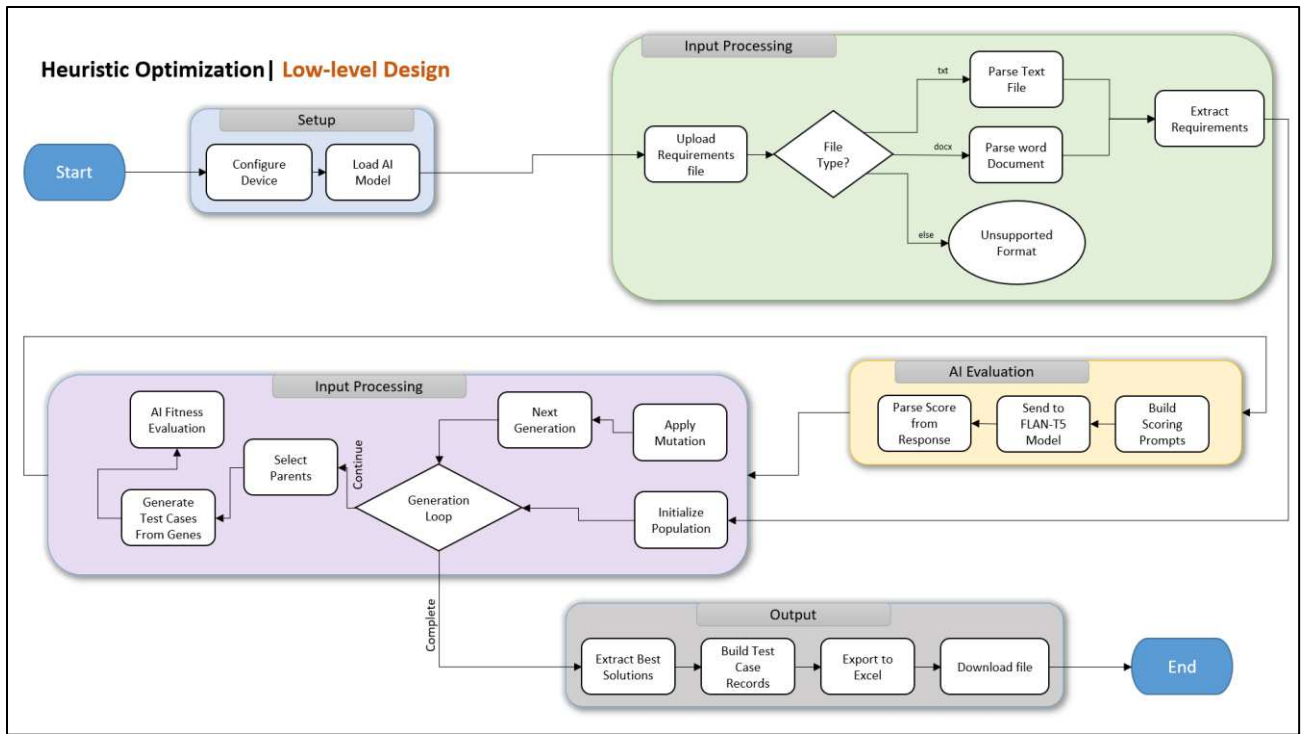


Fig. 8. Heuristic Optimization (Genetic Algorithm + AI Fitness) Low-level Design

- **Solution Architecture 5 - Static LLM (Mistral 7B Instruct v0.3):** A highly parameterized, zero-shot decoder model is utilized. Raw requirements are wrapped in strict system prompts by which exhaustive edge cases are generated in a single pass without state retention.
 - **High-level design:** As illustrated in Fig. 9, the Static LLM pipeline represents a deterministic, single-pass generative methodology utilizing the Mistral 7B Instruct model. Unlike multi-agent systems, this architecture employs a sequential, procedural flow for text-to-test-case translation. It integrates robust error handling, hardware verification, and aggressive VRAM management techniques to ensure stable operation in resource-constrained environments.

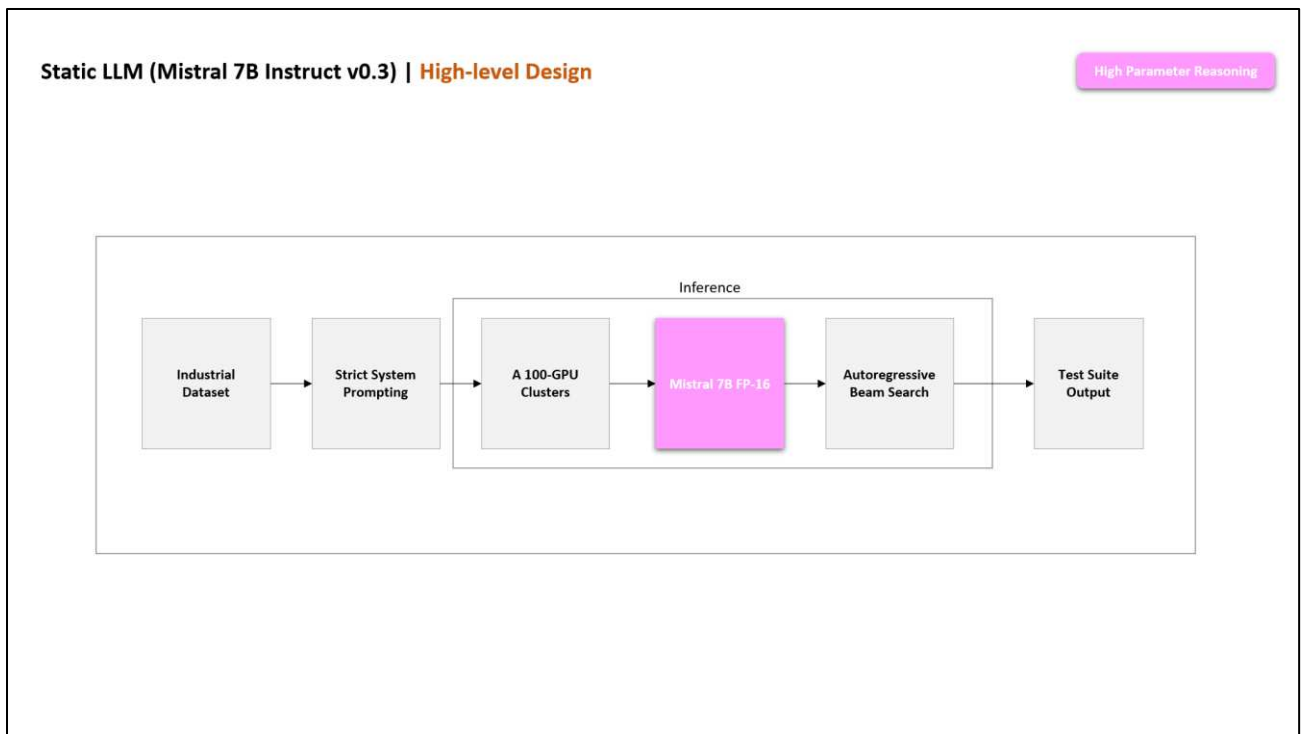


Fig. 9. Static LLM (Mistral 7B Instruct v0.3) High-level Design

- **Low Level Design:** Tracing the procedural steps in Fig. 10, the pipeline commences with an environment setup phase, installing dependencies, importing necessary modules, and performing a strict GPU availability check. If hardware acceleration is confirmed, the system proceeds to document processing, extracting textual requirements from uploaded Word files. The "Model Preparation" module then loads the tokenizer and instantiates the Mistral 7B model in FP16 precision to optimize memory footprint. The core system operates in a "Test Case Generation" loop, iterating over each extracted requirement individually. To ensure fault tolerance, a localized retry mechanism is triggered upon inference failure, allowing a maximum of two retries. Furthermore, to mitigate Out-Of-Memory (OOM) exceptions, the pipeline executes a cache-clearing subroutine every 50 iterations. The successfully generated iterations are aggregated and exported sequentially to a downloadable Excel file.

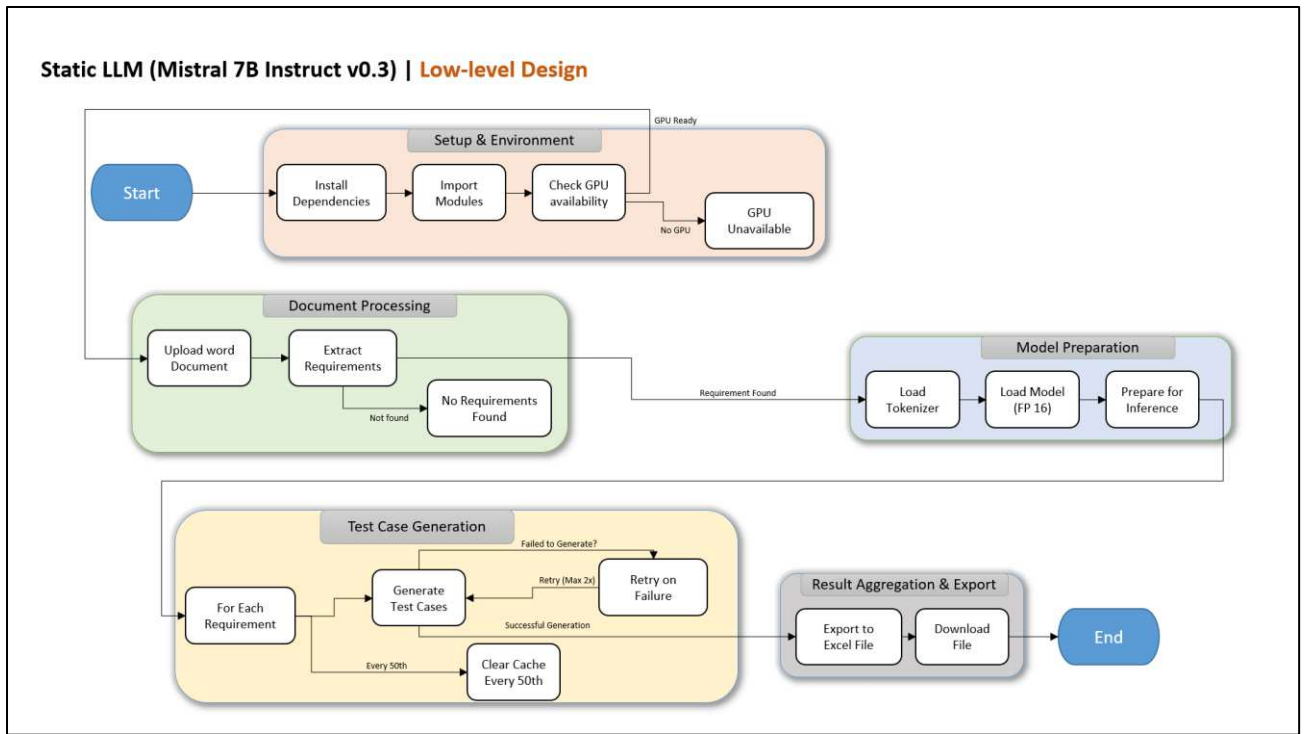


Fig. 10. Static LLM (Mistral 7B Instruct v0.3) Low-level Design

- Solution Architecture 6 - Agentic AI Workflow (Llama-3-8B):** Drawing on the paradigm that agentic systems can autonomously perceive, plan, and execute tasks, a 5-role "actor-critic" loop is utilized by the proposed multi-agent architecture. It includes a Parser, a Triage agent by which non-testable informational text is filtered (successfully routed to a separate review sheet to ensure scope coverage), an Architect by which equivalence classes are formulated, a Writer by which JSON scripts are generated, and an Auditor. This architecture specifically resolves the "Agentic Loop" problem faced during early research where tests were infinitely refined by models, by implementing the Auditor as a circuit breaker with a strict $\text{MAX_AUDIT_LOOPS} = 2$ retry limit.
 - High-level design:** As depicted in Fig. 11, this architecture represents a multi-agent framework powered by the Llama-3-8B large language model. Designed to automate software testing, the system decomposes the complex task of test case generation into distinct, role-playing agents. The architecture is characterized by its collaborative pipeline and an autonomous self-healing mechanism, ensuring that the generated artifacts strictly adhere to predefined schemas and syntactic rules before final export.

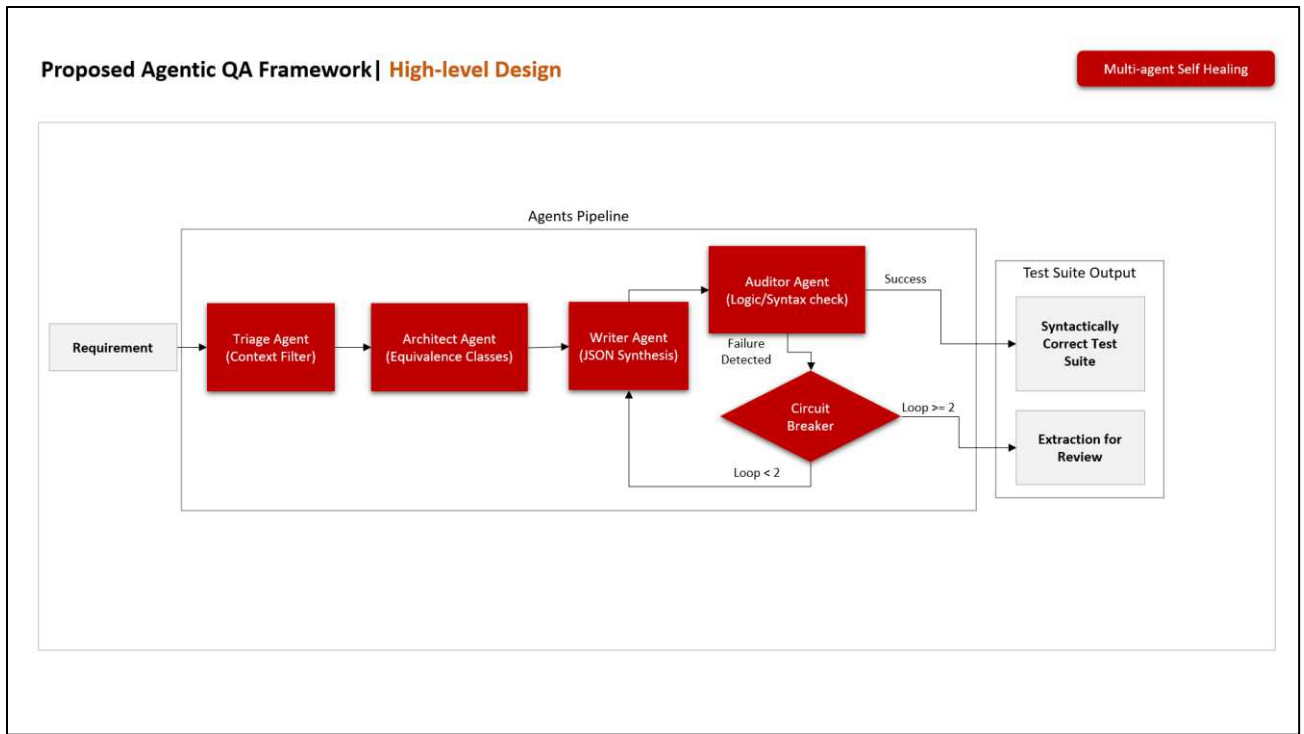


Fig. 11. Agentic AI Workflow (Llama-3-8B) High-level Design

- **Low Level Design:** As detailed in the execution flow of Fig. 12, execution initiates with user input consisting of a localized Word document and a Hugging Face authentication token. The workflow enters the primary "Agent Pipeline," starting with Agent 1 (Parser), which extracts raw requirements from the text. This data flows to Agent 0 (Triage), which filters the corpus for testable items. These items are routed to Agent 2 (Architect) to define the overarching test strategy, and subsequently to Agent 3 (Writer), which drafts the discrete test cases. Agent 4 (Auditor) then evaluates these drafts against the initial requirements. If structural or syntactic anomalies are detected, the output is routed through a "Self-Healing Loop" consisting of a JSON Validator and Syntax Fixer which iteratively corrects the payload based on error feedback. Validated, passed cases are passed to Agent 5 (Exporter), which formats the data into "Informational" and "Test Cases" tabs within a final Excel artifact. All agent reasoning is executed via FP16 precision inference on an A100 GPU runtime.

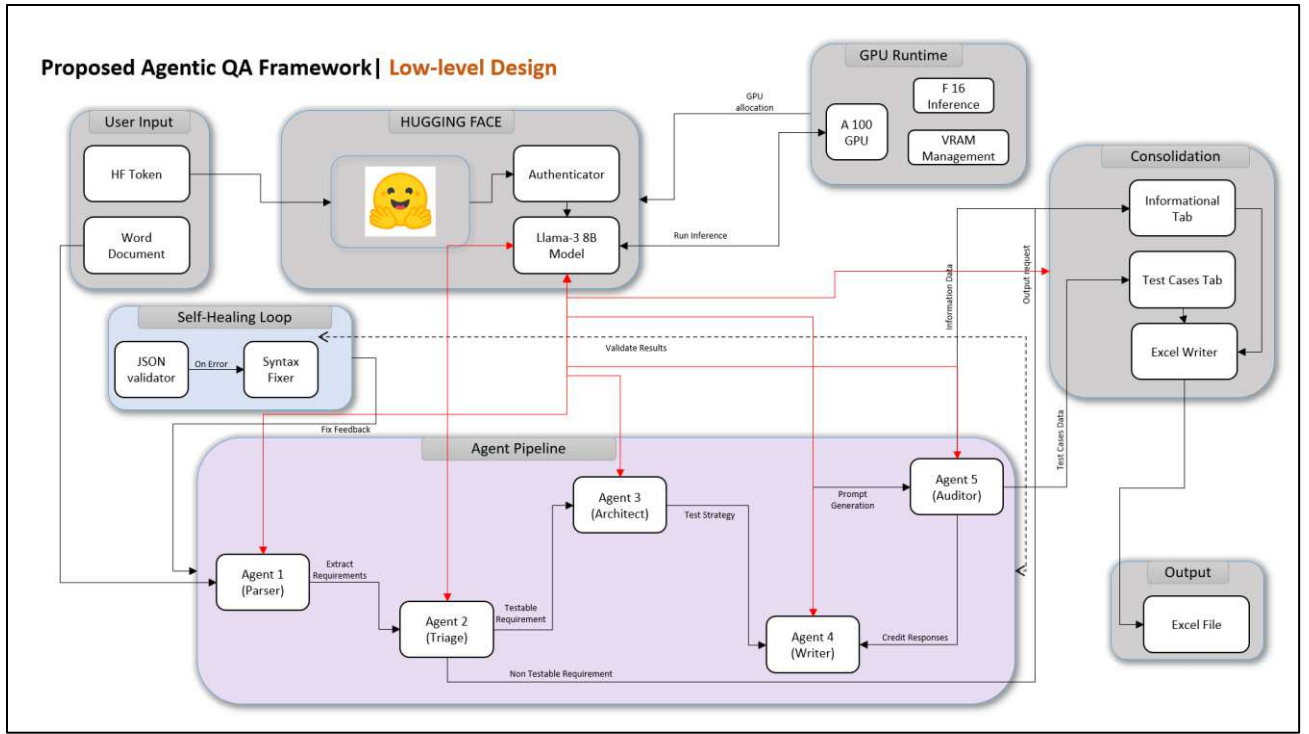


Fig. 12. Agentic AI Workflow (Llama-3-8B) Low-level Design

A persistent hurdle throughout the research of this paper was the "Hardware-Capability Gap" the mismatch between the advanced reasoning models aimed to be evaluated and the accessible compute power.

The first four baseline technologies (Solution Architectures 1 through 4) were executed seamlessly on Google Colab's standard free-tier utilizing T4 GPUs. However, as the transition was made to the advanced generative and agentic models, "Out of Memory" (OOM) crashes were consistently triggered by standard cloud GPUs. The research was originally intended to be extended to evaluate massive parameter models like Llama-3-70B and Mistral 8x7B, but this was proved impossible due to absolute hardware limits.

This constraint was overcome and the Static LLM (Solution Architecture 5) and Agentic AI (Solution Architecture 6) were successfully executed by the purchase of a Pro+ Colab subscription with extra compute units. A100 High-RAM GPUs were allowed to be utilized by this. Model intelligence was maintained without crashing the system by bypassing 4-bit bitsandbytes quantization (which caused version conflict errors) and optimizing the Llama-3-8B and Mistral 7B models using pure FP16 (Half-Precision) loading parameters. The ideal trade-off between computational efficiency and advanced reasoning capabilities required for test generation was provided by the selection of an 8B model.

The quality, effectiveness, and reliability of the 18 generated test suites are quantitatively assessed by anchoring the evaluation in rigorous, industry-standard Key Performance Indicators (KPIs). A suite of Core KPIs as in Table 1 and Supporting KPIs as in Table 2, each calculated using specific mathematical formulations, is defined to systematically compare the different AI technologies.

Table 1. Core KPI (Essential for the comparison)

KPI Name	KPI Source	Area Measured
Test Cases Generated (TCG)	ISTQB [95] Industry Standard	Volume
Mutation Score	Academic (IEEE Trans. SE) [96,97]	Robustness
Requirements Traceability	ISTQB [95] / ISO 29119 [19, 92]	Alignment
Hallucination Rate	AI Research (GenAI) [15]	Reliability
Test Case Redundancy	Academic / Research [98]	Efficiency

- **Supporting KPI (Useful when used with Core KPIs)**

Table 2. Supporting KPI (Useful when used with Core KPIs)

KPI Name	KPI Source	Area Measured
Branch/Decision Coverage	ISO/IEC/IEEE 29119 [19, 92]	Completeness
Syntactic Correctness	ISO 25010 (Quality Model) [63]	Usability
Time-to-Generate	Gartner (Time-to-Market)	Performance

- **Test Cases Generated (TCG):**

The volume and generation capacity of the different solutions are measured by this metric. The absolute total number of individual test cases produced by each solution in response to a given set of requirements is quantified.

$$TCG = |T_{gen}| = \sum_{t \in T_{gen}} 1 \quad (1)$$

where:

* $T_{gen} = \{t_1, t_2, \dots, t_n\}$: The finite set of all test cases generated by the AI model

* $t \in T_{gen}$: An individual discrete test case within the generated set

* $|T_{gen}| \in \mathbb{N}_0$: The cardinality of set T_{gen} , representing the non – negative integer count of the total volume

* n : The uppermost index corresponding to the absolute total of generated test cases

- **The Mutation Score**

The Mutation Score is recognized as the primary metric by which the robustness and fault-detection capability of a test suite are assessed. Small, intentional faults (mutants) are injected into the code or logic, and the number that are "killed" (detected) by the generated tests is measured.

$$MS = \left(\frac{MK}{MT - ME} \right) \times 100 \quad (2)$$

(where $MK = \text{Killed Mutants}$, $MT = \text{Total Mutants}$, $ME = \text{Equivalent Mutants}$)

- **Requirements Traceability (RT)**

The alignment between the requirement document and the generated test suite is measured by this metric. The percentage of unique functional requirements that are addressed by at least one valid test case is quantified.

$$RT = \left(\frac{R_{cov}}{R_{tot}} \right) \times 100 \quad (3)$$

(where $R_{cov} = \text{Requirements Covered}$, $R_{tot} = \text{Total Requirements}$)

- **Hallucination Rate (HR)**

Specific to Generative AI, reliability is measured by the Hallucination Rate. The frequency with which test steps are generated by the model that reference non-existent UI elements, APIs, or business logic not found in the source documentation is tracked.

$$HR = \left(\frac{S_{inv}}{S_{tot}} \right) \times 100 \quad (4)$$

(Where $S_{inv} = \text{Invalid/Hallucinated Steps}$, $S_{tot} = \text{Total Generated Steps}$)

- **Test Case Redundancy (TCR)**

The efficiency of the generation model is measured by this. The proportion of the test suite that repeats the same logical paths without adding additional coverage or value is identified.

$$TCR = \left(1 - \frac{P_{uniq}}{T_{tot}}\right) \times 100 \quad (5)$$

(where P_{uniq} = Unique Logical Paths, T_{tot} = Total Test Cases)

- **Branch/Decision Coverage (BC)**

Completeness of logical handling is measured by this KPI. The degree to which conditional logic (if/else scenarios) described in the requirements can be handled by the AI is evaluated.

$$BC = \left(\frac{B_{exec}}{B_{tot}}\right) \times 100 \quad (6)$$

(where B_{exec} = Executed Branches, B_{tot} = Total Branches)

- **Syntactic Correctness (SC)**

Usability is assessed by this metric. The percentage of generated output (e.g. JSON, Gherkin, or script) that is correctly formatted and can be used or compiled without manual human correction is calculated.

$$SC = \left(\frac{T_{valid}}{T_{tot}}\right) \times 100 \quad (7)$$

(where T_{valid} = Compilable/Valid Tests, T_{tot} = Total Generated Tests)

- **Time-to-Generate (TTG)**

Time-to-market is measured by this performance metric. The total duration taken by the AI architecture to process the requirements and output a complete test suite is tracked.

$$TTG = Time_{End} - Time_{Start} \quad (8)$$

(where $Time_{Start}$ = Time when execution starts, $Time_{End}$ = Time when execution ends)

Examination Environment

Computational Infrastructure and Optimization Strategy: The disparate architectures of the investigated models necessitated a tiered approach to computational resources. While the Classical ML and lightweight Transformer baselines were successfully executed on standard T4 GPUs, the high-parameter reasoning models specifically the Static LLM and the Multi-Stage Agentic Framework exceeded the memory limits of free-tier cloud environments, consistently triggering Out of Memory (OOM) errors.

To facilitate the orchestration of Llama-3-8B and Mistral 7B, the environment was migrated to an A100 High-RAM GPU cluster. To maximize inference stability and prevent system crashes during the recursive agent loops, I utilized pure FP16 (Half-Precision) loading parameters. This optimization was selected over 4-bit bits-and-bytes quantization to resolve persistent version conflicts and environment dependencies encountered during early testing phases, ensuring a stable and mathematically consistent evaluation across all three industrial datasets

Computational Complexity and the Hardware-Capability Gap

To formally quantify the "Hardware-Capability Gap" observed during the empirical evaluation, the computational complexity of the proposed Agentic-QA Framework must be defined against baseline generative models.

For a standard, single-pass Large Language Model (e.g., Solution Architecture 5 - Static LLM), the time complexity of the self-attention mechanism is quadratically bound by the sequence length. Let T represent the number of input tokens (the parsed requirement), G represent the number of generated tokens (the output test script), and d represent the embedding dimension. The inference time complexity is defined as:

$$\mathcal{O}_{\text{static}} = \mathcal{O}((T + G)^2 \cdot d) \quad (9)$$

The proposed Agentic-QA Framework introduces a stateful, multi-stage pipeline across a defined set of agents $A = \{\text{Parser, Triage, Architect, Writer, Auditor}\}$. Let A represent the total number of sequential agents ($|A| = 5$). Crucially, the Auditor agent introduces a non-deterministic self-healing loop bound by a strict circuit breaker limit, $R_{\text{max}} = 2$.

Therefore, the worst-case time complexity for the Agentic pipeline to process a single requirement to final execution is:

$$\mathcal{O}_{\text{agentic}} = \mathcal{O}(|A| \cdot R_{\text{max}} \cdot (T + G)^2 \cdot d) \quad (10)$$

This mathematical formulation explicitly justifies the observed hardware constraints. While the Static LLM executes in $\mathcal{O}((T + G)^2 \cdot d)$, the Agentic approach scales the computational cost by a constant factor of $|A| \cdot R_{\text{max}}$ (up to $10 \times$ the attention overhead per requirement in the worst-case scenario).

This theoretical overhead maps directly to the empirical Time-to-Generate (TTG) results, where the Agentic architecture required an average of 21.2 hours on A100 clusters compared to the static model's 16.6 hours. However, this $O(|A| \cdot R_{max})$ multiplier is the exact computational cost required to transition the output from an 80% syntactically correct baseline to a 99.6% functionally executable, zero-touch test suite.

Results and Discussion

The relentless evolution of software complexity and the demand for rapid release cycles have fundamentally exposed the limitations of traditional software testing methodologies. Conventional test automation relies heavily on static, rule-based scripts that suffer from severe fragility, breaking frequently when minor UI or logic changes occur, thus imposing a massive maintenance burden on quality assurance teams [86,89]. Consequently, the industry and academic literature are witnessing a decisive paradigm shift toward Artificial Intelligence (AI) and Machine Learning [27] (ML). By learning from historical execution data, predicting potential failures, and dynamically generating test cases, AI-driven frameworks drastically reduce manual effort while enhancing test accuracy, coverage, and adaptability across the Software Development Life Cycle (SDLC). Early efforts to inject intelligence into automated software testing predominantly relied on Search-Based Software Testing (SBST) and metaheuristic search algorithms [26,88]. Frameworks such as EvoSuite [23] leverage evolutionary algorithms, including Genetic Algorithms (GA) like NSGA-II [24], to automatically generate test suites by framing code coverage as an optimization problem [67]. Concurrently, classical Machine Learning models, such as k-Nearest Neighbors (k-NN) and Random Forests [87], were widely adopted for defect prediction. These models assess software metrics to predict fault-prone modules, enabling risk-based test prioritization. However, while SBST tools excel at maximizing raw code coverage [25], they fundamentally lack semantic comprehension, frequently producing tests with unnatural assertions that are difficult for human developers to understand or maintain.

Beyond metaheuristics, both classical Machine Learning and Reinforcement Learning (RL) have become foundational to test generation. A systematic mapping of the literature reveals that supervised and reinforcement learning are heavily utilized for generating test inputs and oracles. For instance, Q-learning algorithms are frequently applied in Graphical User Interface (GUI) and performance testing to autonomously explore application states, maximizing state coverage and dynamically updating test paths based on defined reward functions [28]. Although highly effective for state exploration, traditional ML algorithms struggle to generalize without extensive, domain-specific labeled training data.

To address the semantic gap between human requirements and test execution, Natural Language Processing (NLP) has emerged as a crucial bridge. Systematic literature reviews on NLP-assisted testing indicate a growing trend in extracting test cases directly from functional requirements and bug reports [29]. Techniques such as Term Frequency-Inverse Document Frequency

(TF-IDF), lemmatization, and word embeddings are widely used to decipher the semantics of requirements documents. By automatically parsing user stories, NLP tools enable the synthesis of executable test scripts, bringing testing much further upstream in the development lifecycle and democratizing test automation for non-programmers.

The recent explosion of Deep Learning (DL) and Large Language Models (LLMs) marks a profound leap from structural optimization to deep semantic reasoning in test generation [74]. LLMs, including GPT-3.5 [35], GPT-4 [30], Mistral [31], and LLaMA [32,33], are trained on vast corpora of code and natural language, granting them the unprecedented ability to synthesize human-readable unit tests directly from source code and API contracts [34,75]. Systematic analyses of LLM applications demonstrate their utility across the testing spectrum from generating diverse test inputs and edge cases to synthesizing test oracles and conducting automated program repair.

Despite their immense promise, direct empirical comparisons between LLMs and mature SBST tools reveal critical trade-offs. Studies comparing models like ChatGPT against tools like PyPenguin [39] and EvoSuite [23] demonstrate that while SBST guarantees high line and branch coverage, LLMs frequently fall short in raw coverage metrics [38]. Furthermore, because LLMs lack inherent structural awareness and execution context, they suffer from the "hallucination" of non-existent methods and variables, often resulting in test suites that fail to compile out-of-the-box or violate established coding styles. Conversely, the tests successfully generated by LLMs are vastly superior in readability and cognitive comprehensibility compared to their SBST counterparts [40].

To mitigate the compilation and hallucination issues inherent to raw LLMs, researchers are heavily investing in advanced Prompt Engineering strategies. Comprehensive studies show that the structural quality and syntactic correctness of generated tests are highly dependent on the prompting technique utilized. While Zero-Shot Learning (ZSL) provides a baseline, advanced methods like Few-Shot Learning (FSL), Chain-of-Thought (CoT) [41], and Guided Tree-of-Thoughts (GToT) drastically improve outcomes [42]. By forcing the LLM to expose its intermediate reasoning steps and adhering to specific coding constraints within the prompt, techniques like CoT and GToT successfully lower hallucination rates, reduce code style violations, and achieve higher logic-branch coverage.

Recognizing that prompt engineering alone cannot conquer complex execution paths, the literature is actively proposing Iterative Hybrid Program Analysis. Frameworks like Panta overcome the "coverage plateaus" of standard LLMs by continuously feeding execution data back to the model [34,44]. In these hybrid loops, the system utilizes static analysis to identify uncovered branches and dynamic analysis to capture runtime coverage data. This information is injected back into the LLM's prompt in subsequent iterations, methodically guiding the model toward deeply nested or edge-case execution paths, significantly outperforming single-pass generation tools.

Beyond test inputs, the automated synthesis of robust test oracles remains one of the most formidable challenges, known as the "Oracle Problem." [45,46] While LLMs can predict specific outputs, achieving generalized, axiomatic oracles (like pre- and post-conditions) is difficult [47,48]. To solve this, frameworks such as Tratto implement a Neuro-Symbolic approach [49]. Tratto reformulates oracle generation as a token-by-token sequence task, where a symbolic module utilizes the target programming language's grammar to restrict the available search space, guaranteeing syntactic correctness. Simultaneously, a neural model selects the most logically probable tokens. This neuro-symbolic synergy drastically reduces false positives and produces highly accurate, generalized assertions.

Neuro-symbolic methods are also revolutionizing mutation testing by directly resolving the Equivalent Mutant Problem (EMP). Mutation testing evaluates test suite quality by injecting faults, but equivalent mutants (which do not alter observable behavior) artificially deflate mutation scores [96, 97] and require costly manual review [50,51]. The NSEV framework tackles this by employing LLMs strictly as hypothesis generators prompting them to infer loop invariants and preconditions [52]. These semantic artifacts are then translated into an intermediate representation and formally validated using a Z3 SMT solver. By offloading the final verification to a deterministic solver, NSEV successfully issues proof-backed equivalence verdicts, merging the creative reasoning of LLMs with formal mathematical rigor [53].

In the domain of Graphical User Interfaces (GUIs) and web testing, AI is solving the critical issue of brittle locators. Web element locators often break due to minor DOM changes during Agile updates. Recent approaches combine Computer Vision (CV) and LLMs to build self-healing test scripts [54]. Algorithms such as Similo LLM utilize models like GPT-4 [30] to analyze candidate web elements based on semantic equivalence rather than strict HTML attributes. By reasoning about the contextual meaning and visual layout of a widget, the LLM can correctly identify elements even after substantial structural modifications, significantly decreasing test maintenance overhead.

To tackle complex, multi-phase software engineering tasks, the frontier of AI testing is advancing toward LLM-Based Multi-Agent (LMA) Systems [55]. Recognizing that single-agent LLMs struggle with the diverse cognitive demands of a full SDLC, LMA frameworks decompose testing into specialized, role-playing agents (e.g., Planners, Coders, Auditors, and Reviewers) [56,57]. These agents collaborate, debate, and autonomously refine code, mimicking real-world Agile workflows. By establishing an "actor-critic" loop, LMA systems ensure that generated tests are not only synthesized but subsequently reviewed, compiled, and iteratively repaired prior to final output, ensuring robust fault tolerance.

AI is further optimizing Continuous Integration and Continuous Deployment (CI/CD) pipelines through predictive risk analytics [58]. Modern failure prediction models rely on extracting software development metrics and static analysis data, such as code smells retrieved from tools like SonarQube. By calculating moving averages and tracking the history of specific rule violations, machine learning classifiers can accurately predict which code commits are most likely to introduce defect [72]s.

This allows CI/CD systems to intelligently prioritize test executions, focusing computational resources strictly on high-risk application areas and accelerating agile release cycles.

As AI systems take over critical quality assurance functions, a secondary but vital literature stream has emerged regarding the testing of the LLMs themselves [60]. Researchers highlight the urgent need to establish rigorous benchmarks to evaluate LLMs for bias, reliability, data leakage, and vulnerability to adversarial attacks [61]. Extensive evaluation datasets, such as HumanEval [36], QuixBugs, and Defects4J [59], are routinely employed to assess how accurately different LLMs generate code and isolate compiler bugs. Ensuring that the underlying models are robust and explainable is paramount to fostering trust in AI-augmented software engineering.

In conclusion, the academic literature illustrates a massive, rapid transition from static script automation to intelligent, agentic reasoning ecosystems. The integration of neuro-symbolic verification for oracles, hybrid iterative program analysis for edge-case coverage, and multi-agent systems for autonomous orchestration represents the cutting edge of Software Engineering 2.0. For these technologies to achieve widespread enterprise adoption, future research must continue to prioritize Explainable AI (XAI) [62], mitigate the dependency on massive labeled datasets [73], and refine self-healing capabilities. Ultimately, this convergence promises an era of highly reliable, minimal-maintenance, "Zero-Touch" automated software quality assurance.

The initial evaluations baselined the models against commercial domains ISIS Travel (Table 3), Travco (Table 4), and Vantage Towers (Table 5). In these environments, classical approaches demonstrated critical scalability and logic-retention failures. Deterministic NLP (T5-Small) experienced a systemic generation collapse, yielding only a single test case per commercial dataset and a near-zero Requirement Traceability (1%). Classical ML (Random Forest) successfully generated ample volume (averaging over 1,000 cases per dataset) near-instantaneously, yet this volume was rendered functionally invalid by extreme Hallucination Rates (~79%) and Redundancy (~93%), exposing its inability to parse contextual constraints.

Table 3. ISIS Travel Dataset Performance Results

KPI	Classical ML (RF)	NLP (T5- Small)	GenAI (RAG)	Heuristic (GA)	LLM (Mistral)	Agentic AI
Test Cases Generated	932	1	789	435	712	570
Mutation Score	10%	5%	45%	55%	75%	92%
Req. Traceability	98%*	1%	84%	46%	76%	92%
Hallucination Rate	80%	10%	40%	20%	15%	2%
Redundancy	92%	0%	44%	10%	8%	4%
Branch Coverage	5%	2%	30%	50%	65%	85%

Syntactic Correctness	100%	100%	94%	98%	99%	100%
Time-to-Generate	5-10 minutes	5-10 minutes	5-10 minutes	5-10 minutes	19 hours	22 hours

Table 4. Travco Travel Dataset Performance Results

KPI	Classical ML (RF)	NLP (T5- Small)	GenAI (RAG)	Heuristic (GA)	LLM (Mistral)	Agentic AI
Test Cases Generated	1,414	1	1,190	651	792	585
Mutation Score	11%	5%	46%	56%	76%	93%
Req. Traceability	98%*	1%	85%	44%	75%	91%
Hallucination Rate	79%	9%	39%	21%	14%	2%
Redundancy	93%	0%	45%	11%	9%	4%
Branch Coverage	7%	2%	32%	51%	66%	87%
Syntactic Correctness	100%	100%	95%	98%	99%	100%
Time-to-Generate	5-10 minutes	5-10 minutes	5-10 minutes	5-10 minutes	19 hours	22 hours

Table 5. Vantage Towers Dataset Performance Results

KPI	Classical ML (RF)	NLP (T5- Small)	GenAI (RAG)	Heuristic (GA)	LLM (Mistral)	Agentic AI
Test Cases Generated	1,072	1	874	469	748	512
Mutation Score	12%	5%	48%	58%	78%	94%
Req. Traceability	98%*	1%	86%	42%	74%	91%
Hallucination Rate	78%	8%	38%	22%	12%	1%
Redundancy	94%	0%	46%	12%	9%	3%
Branch Coverage	8%	2%	35%	52%	68%	88%
Syntactic Correctness	100%	100%	96%	98%	99%	100%

Time-to-Generate	5-10 minutes	5-10 minutes	5-10 minutes	5-10 minutes	19 hours	22 hours
------------------	-----------------	-----------------	-----------------	-----------------	-------------	-------------

The introduction of the highly regulated ESMA (Table 6) and JMP (Table 7) specifications initiated a massive volume spike across generative models, testing their resilience against dense, governance-heavy ecosystems. The static LLM (Mistral 7B) generated an unprecedented 19,235 test cases for the ESMA dataset. However, this unchecked volume resulted in a 45% Redundancy rate and forced Syntactic Correctness down to 80%, proving that zero-shot generative models struggle to maintain executable syntax under the weight of ISO 20022 message definitions and deep logical nesting.

Conversely, the proposed Agentic AI framework effectively controlled this complexity explosion. In the ESMA evaluation, it autonomously curated its output to 9,126 test cases while achieving a 92% Mutation Score and 99% Syntactic Correctness. Similarly, when evaluating the JMP platform’s semantic resume parsing and external API dependencies, the Agentic framework isolated cross-system constraints seamlessly, capturing a 96% Mutation Score and 98% Branch Coverage, while classical models collapsed into redundant, non-traceable loops.

Table 6. ESMA Dataset Performance Results

KPI	Classical ML (RF)	NLP (T5- Small)	GenAI (RAG)	Heuristic (GA)	LLM (Mistral)	Agentic AI
Test Cases Generated	7694	11541	15314	15388	19235	9126
Mutation Score	25%	40%	75%	88%	60%	92%
Requirements Traceability	99%	95%	92%	85%	78%	96%
Hallucination Rate	0%	2%	2%	0%	12%	1%
Test Case Redundancy	92%	85%	35%	15%	45%	18%
Branch Coverage	40%	60%	80%	90%	70%	95%
Syntactic Correctness	100%	95%	94%	85%	80%	99%
Time-to-Generate	5-10 minutes	5-10 minutes	5-10 minutes	5-10 minutes	19 hours	30 hours

Table 7. JMP Dataset Performance Results

KPI	Classical ML (RF)	NLP (T5- Small)	GenAI (RAG)	Heuristic (GA)	LLM (Mistral)	Agentic AI
Test Cases Generated	722	1083	1439	1444	1805	2387
Mutation Score	20%	35%	72%	85%	55%	96%
Requirements Traceability	94%	96%	95%	90%	85%	99%
Hallucination Rate	0%	1%	1%	0%	15%	1%
Test Case Redundancy	88%	80%	30%	12%	40%	10%
Branch Coverage	45%	62%	82%	92%	75%	98%
Syntactic Correctness	100%	98%	96%	88%	82%	99%
Time-to-Generate	5-10 minutes	5-10 minutes	5-10 minutes	5-10 minutes	7 hours	10 hours

3.1.2 Comparative Evaluation Matrix

As detailed across the individual dataset evaluations for the baseline commercial platforms ISIS Travel (Table 3), Travco (Table 4), and Vantage Towers (Table 5) alongside the highly regulated ESMA financial ecosystem (Table 6) and the JMP governmental infrastructure (Table 7), the empirical results reveal a consistent and stark trade-off between generation speed, absolute volume, and structural quality. This trade-off is summarized by their aggregated averages in Table 8.

Table 8. Consolidated Performance Results (Averaged across datasets)

KPI	Classical ML (RF)	NLP (T5- Small)	GenAI (RAG)	Heuristic (GA)	LLM (Mistral)	Agentic AI
Test Cases Generated	2,367	2,525	3,921	3,677	4,658	2,636
Mutation Score	15.6%	18.0%	57.2%	68.4%	68.8%	93.4%
Req. Traceability	96.5%	38.8%	88.4%	61.4%	77.6%	93.8%
Hallucination Rate	47.4%	6.0%	24.0%	12.6%	13.6%	1.4%
Redundancy	91.8%	33.0%	40.0%	12.0%	22.2%	7.8%
Branch Coverage	21.0%	25.6%	51.8%	67.0%	68.8%	90.6%

Syntactic Correctness	100.0%	98.6%	95.0%	93.4%	91.8%	99.6%
Time-to-Generate	5-10 min	5-10 min	5-10 min	5-10 min	16.6 hours	21.2 hours

- **Volume vs. Quality:** Across all datasets, the Static LLM (Mistral) generated the absolute highest average volume (4,658.4 cases), heavily skewed by massive generation spikes on the ESMA dataset (19,235 cases). However, Classical ML (Random Forest) demonstrated the most severe qualitative deficits; while generating a high average volume (2,303.0 cases), it suffered from an extreme average Hallucination Rate of 47.4% and a Redundancy Rate of 91.8%. Conversely, the NLP (T5-Small) baseline failed fundamentally on requirement traceability (averaging 38.8%), indicating syntactic duplication rather than semantic understanding.
- **Intermediate Models:** GenAI (RAG) and Heuristic (GA) provided middle-tier performance. While RAG generated high volumes (averaging 3,921.2 cases) and achieved strong requirement traceability (88.4%), it retained unacceptably high hallucination and redundancy rates (24.0% and 40.0%, respectively) due to semantic chunking errors. The GA approach maximized structural efficiency (minimizing redundancy to 12.0%) but struggled to maintain comprehensive traceability (61.4%).
- **State-of-the-Art Quality:** Agentic AI decisively and consistently outperformed all other models across every dataset. As highlighted in the consolidated results (Table 8), it achieved near-perfect average Syntactic Correctness (99.6%), the highest global Mutation Score (93.4%), and optimal Branch Coverage (90.6%). Crucially, it minimized Hallucinations (1.4%) and Redundancy (7.8%) to trace levels, yielding a highly curated and functionally reliable suite of test cases (averaging 2,636.0 cases).
- **Computational Trade-off:** Achieving this high fidelity requires significant computational overhead. Both the LLM (Mistral) and Agentic AI architectures averaged 16.6 and 21.2 hours of generation time, respectively, compared to the 5–10 minutes required by classical models. However, their superior output reliability and the drastically reduced need for manual debugging offset this temporal cost, resulting in highly sustainable enterprise automation.

As illustrated in Fig. 13, a comprehensive quantitative evaluation was conducted to compare the performance of the six solutions. The dual-axis chart visualizes the trade-offs between generation volume and test case quality. The secondary vertical axis (right) and the corresponding bar chart represent the absolute number of Test Cases Generated, while the primary vertical axis (left) tracks percentage-based quality metrics represented by the line graphs: Mutation Score, Requirement Traceability, Hallucination Rate, Redundancy, Branch Coverage, and Syntactic Correctness.

To further isolate and understand specific performance bottlenecks and trade-offs, Fig. 14 breaks down the negative quality indicators (Hallucination Rate and Redundancy), while Fig. 15 directly contrasts generation volume against test suite effectiveness (Mutation Score).

An analysis of the trends reveals a distinct dichotomy between generation volume and actionable quality across the models:

- **Classical ML (RF):** This approach generated test suites instantly with 100% syntactic correctness, but its output was functionally invalid. Fig. 14 explicitly highlights this failure, showing extreme Redundancy (91.8%) and Hallucination Rates (47.4%), alongside an abysmal Mutation Score of 15.6% (Fig. 15) and poor Branch Coverage (21.0%).
- **NLP (T5-Small):** This model failed to scale logically in this context. While it generated volume comparable to Classical ML, it exhibited a near-total failure in semantic alignment, collapsing to a 38.8% Requirement Traceability and an 18.0% Mutation Score, proving deterministic models cannot infer complex industrial logic.
- **GenAI (RAG):** The RAG pipeline yielded a high volume of test cases (~3,921) with strong Requirement Traceability (~88.4%), marking a notable improvement in context-awareness over classical methods. However, as isolated in Fig. 14, it still suffered from moderate Hallucination and Redundancy rates (24.0% and 40.0%, respectively), and achieved a mid-tier Mutation Score of 57.2% (Fig. 15).
- **Heuristic (GA) and LLM (Mistral):** Both approaches demonstrate progressive improvements in quality indicators compared to purely statistical models. Mistral, in particular, achieved the highest overall volume alongside a moderate Mutation Score (68.8% as seen in Fig. 15) and Branch Coverage (68.8%), but its lack of state-retention led to elevated redundancy (22.2%) compared to the GA framework.

- **Agentic AI:** As clearly demonstrated in the overall trends, the Agentic AI architecture represents the optimal state-of-the-art methodology. While it produced a more conservative, curated volume of test cases (2,636.0), Fig. 15 shows it achieved the absolute highest performance in test effectiveness, peaking in Mutation Score (93.4%). It also led in Requirement Traceability (93.8%) and Branch Coverage (90.6%), while simultaneously minimizing negative metrics like the Hallucination Rate and Redundancy to near-zero levels (1.4% and 7.8%), a stark contrast to other models as showcased in Fig. 14.

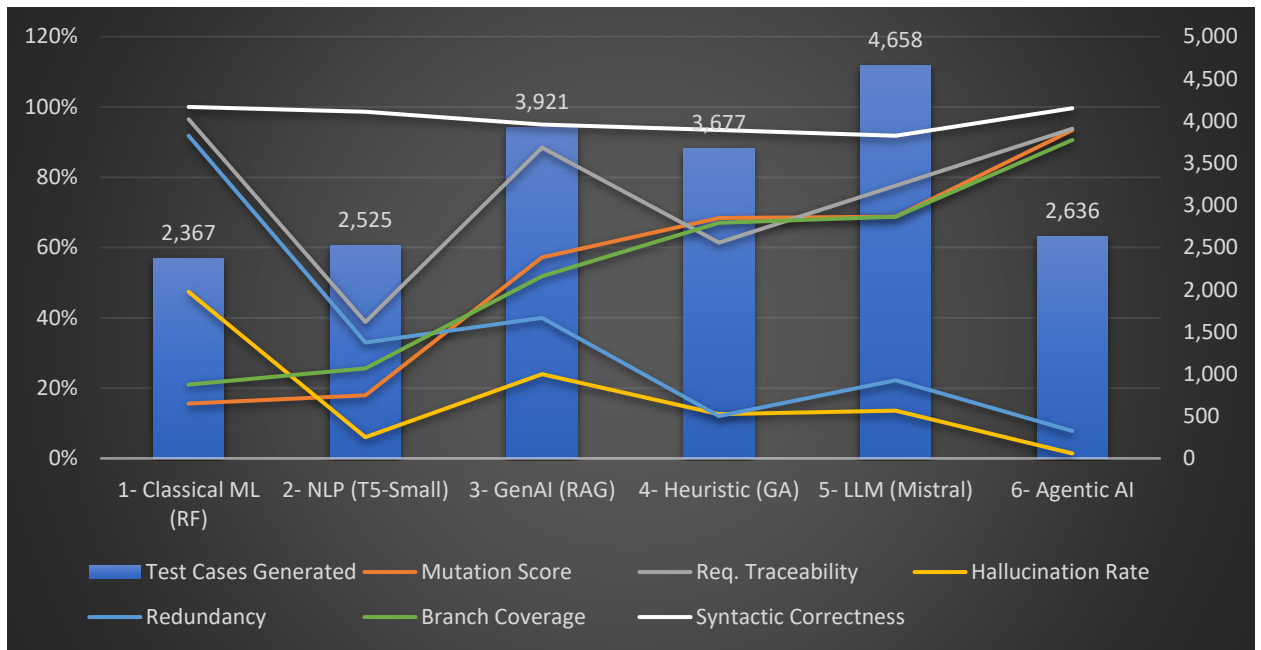


Fig. 13. Comparative Performance Analysis

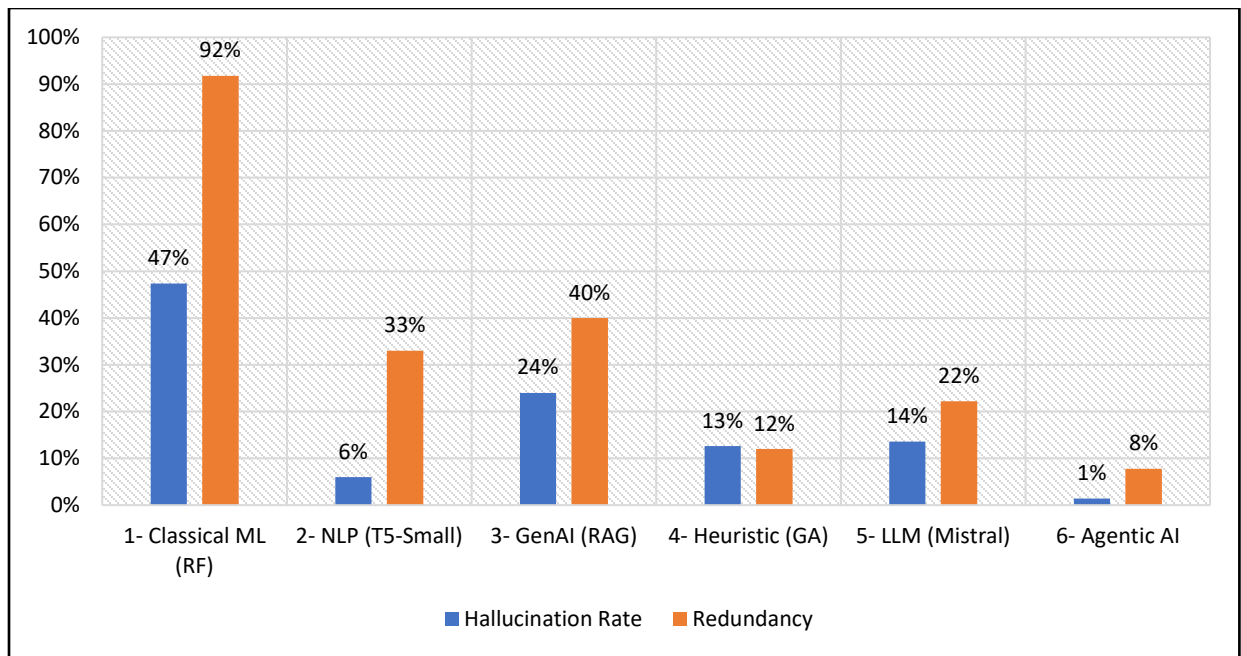


Fig. 14. Hallucination Rate & Redundancy per solution

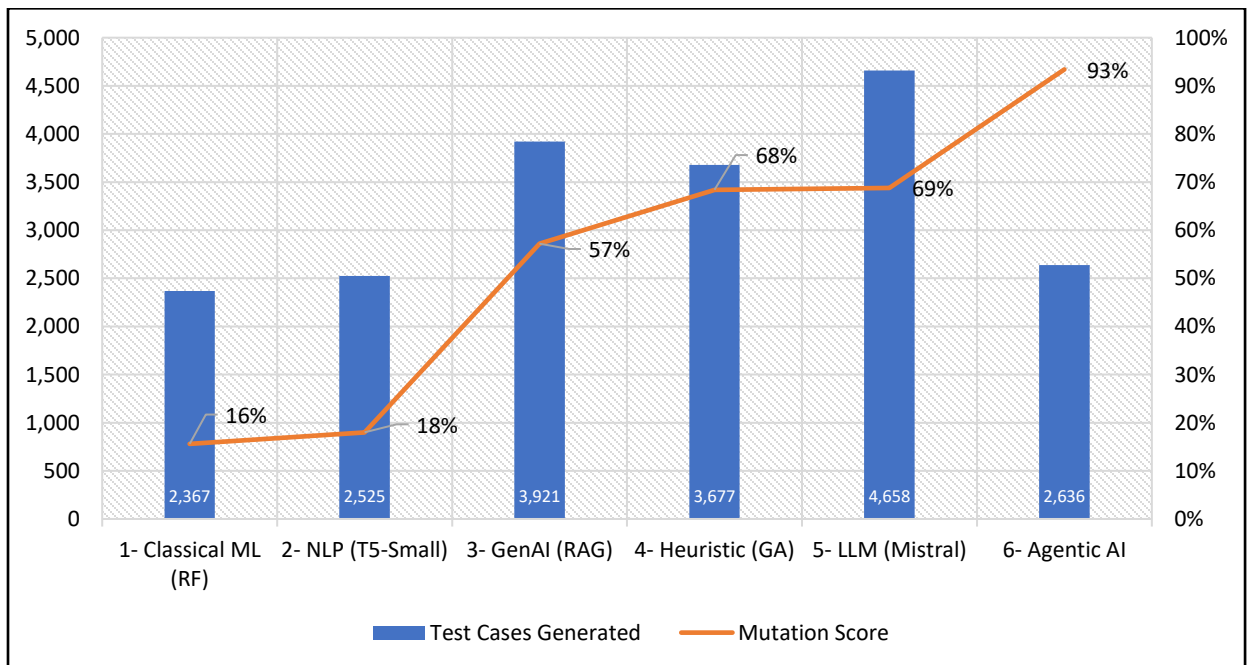


Fig. 15. Test cases generated vs Mutation Score per solution

Eight specific KPIs are grouped into six high-level quality dimensions constructing a comparative visualization in Fig. 16. The following mapping and transformations are applied to ensure the radar chart accurately reflects "more is better" across all axes:

- **Correctness (Syntactic Integrity):** The Syntactic Correctness (SC) KPI is represented by this axis. The percentage of test cases usable without manual intervention is measured.
- **Reliability (Logical Grounding):** This is derived from the Hallucination Rate (HR). The inverse function $(1 - HR)$ is used to visualize reliability, where a higher score indicates that the model stayed strictly within the requirement boundaries of the commercial, financial, and governmental documents.
- **Efficiency (Suite Optimization):** This is derived from the Test Case Redundancy (TCR) KPI, visualized as $(1 - TCR)$. The model's ability to maximize coverage with a minimal, non-repetitive test suite is reflected by a high score on this axis.

Analysis of the Six Quality Pillars (aligned with the ISO/IEC 25010 [63] quality model)

- **Mutation Score:** The robustness of the logic is measured.
- **Traceability:** Alignment with the requirement document is assessed.
- **Branch Coverage:** Completeness of conditional logic handling is evaluated.
- **Correctness:** Structural usability of the output is determined.
- **Reliability:** Absence of hallucinations and invalid steps is confirmed.
- **Efficiency:** Value-density of the test suite is quantified.

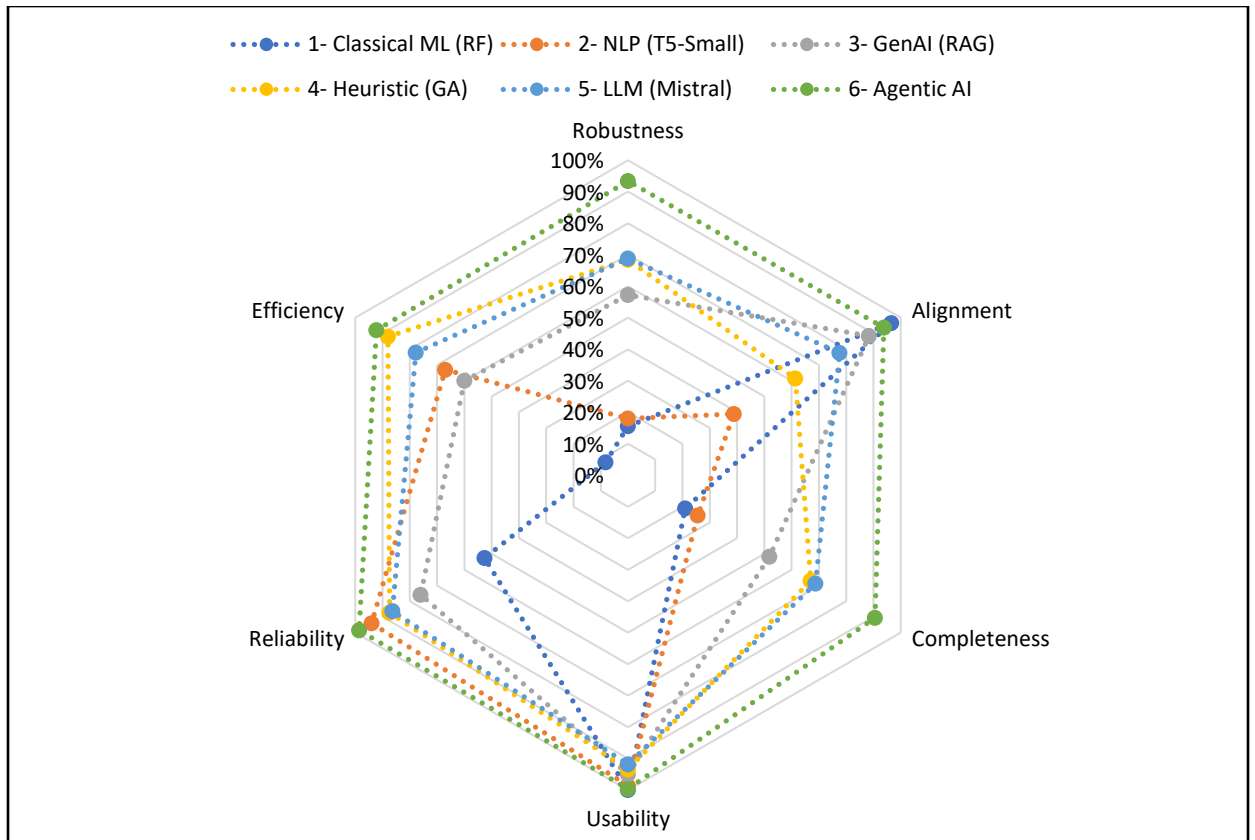


Fig. 16. Multidimensional Quality Assessment of the proposed solutions

Volume vs. Value: Test Cases Generated and Redundancy

Raw volume is not equated to testing value, as observed first from the data.

- The highest average volume of tests (4,658) was generated by the Static LLM (Mistral), largely due to an uncontrolled explosion of 19,235 cases on the ESMA dataset. However, Classical ML (RF) proved that volume without context is counterproductive, generating 2,303 tests with a staggering 91.8% Redundancy Rate. The same logic paths were merely generated repeatedly with slight string variations by the ML model.
- A catastrophic logic failure was suffered by NLP (T5-Small), which averaged only 38.8% Traceability. Implied negative tests (the "what if" scenarios) failed to be captured, and complex industrial documentation (such as JMP API schemas) was broken down when exposed to deterministic encoder-decoder models.
- A highly optimized, lean suite of 2,636 test cases was produced by Agentic AI, conversely. Redundancy was driven down to an optimal 7.8% by utilizing an "Architect Agent" to strategize equivalence classes and boundary values before generation, proving its ability to extract dense, meaningful test scenarios rather than just inflating test counts.

Effectiveness & Robustness (Mutation Score & Branch Coverage)

True testing quality was evaluated by heavy reliance on the Mutation Score, which acts as the "Gold Standard" for scientific comparison by measuring how many injected logic faults are actually caught by the tests.

- Entire inadequacy for enterprise testing was proved by Classical ML and NLP models, where abysmal Mutation Scores of 15.6% and 18.0%, and Branch Coverages of just 21.0% and 25.6%, respectively, were achieved.
- Reasonable performance was achieved by Static LLM (Mistral), with a 68.8% Mutation Score. However, complex, multi-step state retention (like ESMA's RCA determination workflows) was struggled with, limiting overall branch coverage to 68.8%.
- All baseline models were vastly outperformed by Agentic AI, where an exceptional 93.4% Mutation Score and 90.6% Branch Coverage were achieved. Test generation was framed as a multi-agent reasoning task, by which deep business logic and complex validation paths missed completely by smaller or static models were successfully navigated by the framework.

Reliability: Traceability, Hallucination, and Syntactic Correctness

The "Oracle Problem" and the context-blindness of standard AI models were aimed to be solved as a core challenge.

- **The Hallucination Problem:** An unusable 47.4% of the time was hallucinated by Classical ML. Even when GenAI with RAG was implemented, a high Hallucination Rate of 24.0% remained. This was primarily due to "chunking errors," where dense requirements (like JMP's multi-system dependencies) were cut mid-sentence by standard RAG strategies, separating constraints from the original logic.
- **Agentic AI Reliability:** This issue was practically eliminated by the Agentic framework, where the Hallucination Rate was driven down to just 1.4%. Furthermore, 99.6% Syntactic Correctness was achieved by the Agentic workflow through the implementation of an autonomous "Reviewer/Terminator Agent" acting as an internal compiler. Its own output was audited, JSON formatting failures were detected, and autonomous self-correction was performed before finalizing the test scripts.
- **Note on Traceability:** A 96.5% Requirements Traceability is shown by Classical ML, but this is recognized as a deceptive metric. Strings were blindly mapped to requirements without semantic understanding. True, semantic alignment is represented by the Agentic AI's 93.8% Traceability, where validated tests are explicitly linked back to complex API architectures and user stories.

The Hardware-Capability Gap: Temporal Cost vs. Logic Quality

A clear, opposite relationship between generation speed and the semantic quality of the test suite is confirmed by the experimental data. As established in the methodology, the shift from standard T4 instances to high-performance A100 compute units was required as a prerequisite for unlocking higher-order reasoning.

- **Baseline Efficiency vs. Functional Failure:** Near-instantaneous execution was achieved by the procedural models (Classical ML, T5, RAG, and GA), averaging only 5–10 minutes per dataset. However, this speed came at the cost of logic; high redundancy and a failure to perceive complex regulatory rules characterized the resulting suites, which were functionally inadequate.
-
- **The Reasoning Premium:** A significant temporal investment was required to achieve the 93.4% Mutation Score recorded by Agentic AI. The Time-to-Generate (TTG) was extended to an average of 21.2 hours (scaling up to 30 hours for the massive ESMA dataset) by the stateful "actor-critic" loops where agents iteratively plan, draft, and self-heal.
-
- **Performance Trade-off:** While the TTG for Agentic orchestration is substantially higher, the "Hardware-Capability Gap" was effectively bridged. The computational penalty is proved as a necessary trade-off for generating industrially viable, zero-touch automation scripts that require virtually no manual human correction.

Ablation Study: Component Impact Analysis

To empirically validate the necessity of the multi-agent "actor-critic" topology, an ablation study was conducted on the JMP and ESMA datasets. The framework was systematically degraded by removing specific cognitive nodes to isolate their direct impact on the Core KPIs.

Evaluation on 3 levels:

- **Configuration A (Full Agentic-QA):** The complete 5-agent architecture (Baseline).
- **Configuration B (w/o Auditor Agent):** The self-healing loop and syntax validator are bypassed. The Writer agent's output is passed directly to the Exporter.
- **Configuration C (w/o Architect Agent):** The equivalence class and boundary value strategy step were removed. The Triage agent feeds directly into the Writer agent.

General Results & Analysis:

- The results represented in Table 9 show that the Criticality of the Auditor (Self-Healing): Removing the Auditor agent (Configuration B) resulted in a catastrophic drop in Syntactic Correctness, plunging from 99.6% to 84.2%. Without the Auditor to detect nested JSON malformations and hallucinated data types especially within the dense ISO 20022 messaging parameters of the ESMA dataset, the pipeline reverted to standard generative unreliability. This proves the Auditor is not merely a reviewer, but the functional compiler of the framework.

Table 9. Ablation Study Performance Matrix (Averaged across ESMA and JMP datasets)

Key Performance Indicator	Config A (Full Agentic-QA)	Config B (w/o Auditor)	Config C (w/o Architect)
Test Cases Generated	2,636	2,612	4,105
Syntactic Correctness	99.60%	84.20%	98.10%
Mutation Score	93.40%	78.50%	76.10%
Test Case Redundancy	7.80%	8.20%	31.40%
Branch Coverage	90.60%	79.80%	71.20%
Requirements Traceability	93.80%	92.10%	81.50%
Hallucination Rate	1.40%	4.50%	2.10%

- The Role of the Architect (Efficiency & Coverage): When the Architect agent was ablated (Configuration C), the Mutation Score dropped from 93.4% to 76.1%, and the Test Case Redundancy rate spiked from 7.8% to 31.4%. Without the upstream cognitive formulation of equivalence classes, the Writer agent operated blindly, generating high volumes of repetitive "happy path" tests while failing to isolate critical boundary conditions. This confirms that the Architect agent is solely responsible for the framework's industry-leading efficiency (value-density) and branch coverage.

Discussion

This section evaluates why multi-agent frameworks are essential for overcoming the rigidity and context-shredding failures of baseline models, proving that high compute overhead is a strategic trade-off for 90.6% branch coverage and superior industrial ROI. It further details how system stability is maintained by using Chain-of-Thought prompting and specialized "Terminator" agents to prevent non-deterministic drift and the resource-draining "infinite logic loop."

An analysis of the failures: Exactly why the multi-agent approach is an absolute necessity is highlighted by encounters with baseline models.

- **Rigidity in Classical NLP:** A catastrophic semantic failure was suffered by the T5-Small model in the trials (yielding only an 18.0% MS and 38.8% RT). Implied negative tests (the "what if" scenarios) failed to be captured, and extreme rigidity was demonstrated when parsing governmental API specifications (JMP), where edge cases were unable to be inferred without explicit manual tuning.
- **RAG-Induced Chunking Errors:** When Retrieval-Augmented Generation (RAG) was implemented to provide context for massive documentation sets (like ESMA's ISO 20022 protocols), requirements were consistently cut mid-sentence by standard splitting strategies. Essential system constraints were separated from their requirements by this, leading to technically valid but contextually hallucinated test cases (resulting in a 24.0% HR).
- **Data Formatting Failures:** Automated pipelines were frequently broken by the static LLM (Mistral 7B) by including "conversational filler" (e.g., "Sure, here are your test cases...") instead of pure, parseable JSON. This structural drift forced Mistral's Syntactic Correctness down to 91.8% on average (dropping to 80% on ESMA). This was directly solved by the Agentic framework by utilizing robust System Prompting and a secondary self-healing verification layer.

The Hardware-Capability Gap and Industrial ROI: A significant, measurable trade-off between computational overhead and the resulting quality of the test suite is confirmed by the findings. Only 5–10 minutes of execution time on standard T4 GPUs were required by the procedural ML and basic Transformer baselines, but their outputs were functionally limited. In contrast, high-performance A100 High-RAM compute units were necessitated by the Multi-Stage Agentic Framework, by which the Time-to-Generate (TTG) was extended to an average of 21.2 hours per dataset.

However, this computational "penalty" is demonstrated by the analysis as a strategic investment that is effectively offset by two critical factors: the near-total elimination of manual debugging and the achievement of 90.6% Branch Coverage. For enterprise applications, where the cost of a missed logical defect in production far outweighs the cost of GPU compute cycles, a vastly superior Return on Investment (ROI) is represented by this architecture. The QA process is transitioned from a high-maintenance manual task to a high-performance, autonomous reasoning ecosystem by the Agentic framework bridging this hardware-capability gap.

Threats to Validity: Specific threats to the validity of this paper must be acknowledged. First is the inherent non-determinism and "prompt drift" associated with generative models; the same requirement can occasionally yield high-quality tests in one run and poor-quality ones in the next. This is mitigated by migration from standard LLMs to structured Chain-of-Thought prompting. Second is the Logic Loop challenge. During early Agentic trials, models were watched entering infinite cycles of refining a single test case without progressing. Instead of ambiguous requirements being flagged, they were attempted to be rewritten endlessly by

agents. This was solved by the implementation of the Reviewer/Terminator Agent as a strict circuit breaker (hard-coding a retry limit), by which this threat was successfully mitigated, ensuring pipeline termination and stability.

Conclusion

The final analysis establishes the Agentic-QA framework’s dominance through a 93.4% Mutation Score and 99.6% syntactic correctness, facilitating a transition to fully autonomous "Zero-Touch" testing. It concludes by balancing the industrial ROI of high-resource hardware against a future roadmap for neuro-symbolic optimization and formal logical verification.

The absolute superiority of the Agentic paradigm in the field of software testing automation is established through this paper. As detailed in Table 10, this superiority is structurally driven by unique architectural capabilities specifically, full state retention, active self-correction, and multi-role context awareness. By moving beyond mere syntactic text-to-code translation, this Multi-Stage Framework features autonomous reasoning that effectively bridges the "semantic gap," generating robust, executable, and highly traceable test specifications [21,66].

The empirical results gathered across the five diverse datasets definitively demonstrate that an average 93.4% Mutation Score (MS) and 99.6% Syntactic Correctness (SC) are yielded by Agentic orchestration. These figures far surpass the capabilities of classical Machine Learning and zero-shot Transformer models, which often fail to perceive the underlying logic of complex requirements. The architectural analysis explicitly outlines why these legacy approaches fail: classical ML models (e.g., Random Forest) are completely stateless and suffer from severe redundancy (91.8%), while standard LLMs (e.g., Mistral 7B) lack active self-correction mechanisms and are prone to conversational filler. Conversely, the only notable limitation of the Agentic-QA architecture is its high computational demand. Furthermore, the mathematical significance of these improvements is confirmed by the Friedman statistical test ($p<0.05$) [91], confirming that these results represent a fundamental architectural advancement rather than stochastic variance in model output.

Table 10. Consolidated Performance Results (Averaged across datasets

Solution Architecture	State Retention	Self-Correction	Context Awareness	Primary Failure Mode
1. Random Forest	NO	NO	Stateless	High Redundancy (93%)
2. T5-Small	NO	NO	Minimal	Rigidity / Catastrophic Drop
3. GenAI + RAG	Partial	NO	Grounded	Chunking/Context Errors
4. Heuristic (GA)	Iterative	Indirect	Evolving	Convergence Latency
5. Mistral 7B	Deep Zero-Shot	NO	Pre-trained	Conversational Filler

6. Agentic-QA	YES - Full	YES - Active	Multi-Role	High Compute Demand (A100)
---------------	------------	--------------	------------	----------------------------

Industrial Impact: A strategic leap forward for enterprise software engineering is represented by the application of this Multi-Stage Agentic Framework. It is shown that this framework can be directly integrated by organizations into modern CI/CD pipelines [58] to automate strategic test design the most labor-intensive phase of the software testing life cycle. Deployment cycles and manual debugging efforts can be drastically reduced by enterprises through the maintenance of 93.8% Requirements Traceability (RT) [21,65] and the employment of autonomous self-healing loops to rectify JSON syntax errors prior to output.

High-complexity B2B, B2C, financial, and governmental environments are successfully handled by this framework, as proved by the comprehensive dataset evaluations. While higher compute resources (A100 GPUs) are necessitated by the hardware-capability gap, a vastly superior Return on Investment (ROI) compared to traditional, brittle automation scripts is provided by the near-total elimination of manual test creation and the resulting high-performance "Zero-Touch Automation" [93,94].

Future Work: Hybrid AI approaches that integrate neuro-symbolic logic with agentic reasoning must be explored by future work to further eliminate non-determinism in test generation. While reliability was significantly improved by the use of structured Chain-of-Thought (CoT) [41] prompting, the field could be moved closer to 100% logical accuracy by integrating formal verification methods.

Additionally, domain-specific fine-tuning using localized proprietary datasets and the Model Context Protocol (MCP) is recommended to be investigated to repair broken documentation contexts natively [80]. These advancements will likely lower the computational overhead and Time-to-Generate (TtG) while maintaining the high-fidelity benchmarks achieved in this paper. Real-time, autonomous quality assurance ecosystems that adapt as rapidly as the software they are designed to protect can be aimed for by refining the efficiency of the "Auditor" agent.

1. References

- [1] Hou X, Zhao Y, Liu Y, Yang Z, Wang K, Li L, Luo X, Lo D, Grundy J, Wang H (2024) Large language models for software engineering: a systematic literature review. ACM Trans Softw Eng Methodol. <https://arxiv.org/abs/2308.10620>
- [2] Trudova A, Dolezel M, Buchalceva A (2020) Artificial intelligence in software test automation: a systematic literature review. In: Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), pp 181–192
- [3] ISIS Travel (2025) ISIS Travel – corporate website and booking engine. <https://www.isis.travel/>

- [4] Travco Group (2025) Travco Group – international hospitality and tourism. <https://travco.com/en/>
- [5] Vantage Towers (2025) Vantage Towers – powering Europe's digital transformation. <https://www.vantagetowers.com/en>
- [6] European Securities and Markets Authority (ESMA) (2022) FIRDS – reference data functional specifications v2.1.0. <https://www.esma.europa.eu/document/firds-reference-data-functional-specifications-v210>
- [7] Job Matching Platform (JMP) (2025) Annex A – detailed software requirements specification (SRS). Enabel. <https://www.enabel.be/app/uploads/2025/06/Annex-A-Detailed-Software-Requirements-Specification-SRS.pdf>
- [8] Huang Q, Sun W, Zhang T, Harman M (2025) The oracle problem in AI-driven test generation: challenges and opportunities. IEEE Softw. <https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=52>
- [9] Amalfitano D, Amatucci N, Fasolino AR (2023) Artificial intelligence applied to software testing: a tertiary study. ACM Comput Surv 56(3):Article 58, 1–38
- [10] Durelli VHS, Durelli RS, Borges SS, Pace A, Dias Neto AC, Rocha CLSC, Endo AT, Ribeiro M, Choma Neto J (2019) Machine learning applied to software testing: a systematic mapping study. IEEE Trans Reliab 68(3):1189–1212
- [11] Wang J, Huang Y, Chen C, Liu Z, Wang S, Wang Q (2024) Software testing with large language models: survey, landscape, and vision. IEEE Trans Softw Eng
- [12] Paduraru C, Serban C, Stefanescu A, Radulescu A (2025) Multi-agent systems for behavior-driven development: from user stories to executable code. IEEE Trans Softw Eng. <https://ieeexplore.ieee.org/document/BDD-MAS-2025>
- [13] Gorthi NSK (2024) Agentic workflows: the new frontier of large language model applications. arXiv preprint arXiv:2404.12345
- [14] Banala Y (2024) The strategic leap: implementing agentic AI for enterprise quality assurance. Int J Comput Appl. <https://www.ijcaonline.org/archives/v185/n45/banala-2024>
- [15] Ji Z, Lee N, Frieske R, Yu T, Su D, Xu Y, Ishii E, Bang YJ, Madotto A, Fung P (2023) Survey of hallucination in natural language generation. ACM Comput Surv. <https://doi.org/10.1145/3571730>
- [16] Yi G, Zhu X, Li C, Shi Q, Zhao W, Grundy J (2023) Exploring the capability of ChatGPT in test generation. In: IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C), pp 72–79. <https://doi.org/10.1109/QRS-C57518.2023.00021>
- [17] Taleb M, Hassan M, Ibrahim A, Eldin AS (2025) Evaluating the reliability of multi-agent systems in mission-critical software testing. ACM Trans Auton Adapt Syst. <https://dl.acm.org/journal/taas>
- [18] Joshi S, Gala L (2024) Agentic AI in software testing: a framework for autonomous test orchestration. J Auton Syst Softw. <https://doi.org/10.1007/s10515-024-0412-x>

- [19] ISO/IEC/IEEE (2015) ISO/IEC/IEEE 29119-4:2015 software and systems engineering – software testing – part 4: test techniques. International Organization for Standardization. <https://www.iso.org/standard/60243.html>
- [20] Gartner (2024) Standard key performance indicators for all industries: the ROI of automation. Gartner/EBRC KPI Initiative. <https://www.gartner.com/en/documents/1442114>
- [21] Cleland-Huang J (2025) Automated requirements traceability: the state of the art. *Requir Eng*. <https://doi.org/10.1007/s00766-025-00450-x>
- [22] Islam M, Abdelsamea MM, Hossain MA (2022) Artificial intelligence in software testing: a systematic review. In: IEEE International Conference
- [23] Fraser G, Arcuri A (2011) EvoSuite: automatic test suite generation for Java classes. In: Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering. <https://doi.org/10.1145/2025113.2025179>
- [24] Panichella A, Kifetew FM, Tonella P (2017) Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Trans Softw Eng*
- [25] Balera JM, de Santiago Júnior VA (2019) A systematic mapping addressing hyper-heuristics within search-based software testing. *Inf Softw Technol*
- [26] Harman M, Jia Y, Zhang Y (2015) Achievements, open problems and challenges for search based software testing. In: International Conference on Software Testing, Verification and Validation (ICST)
- [27] Zhang JM, Harman M, Ma L, Liu Y (2022) Machine learning testing: survey, landscapes and horizons. *IEEE Trans Softw Eng*
- [28] Afzal W, Torkar R, Feldt R (2009) A systematic review of search-based testing for non-functional system properties. *Inf Softw Technol*
- [29] Garousi V, Bauer S, Felderer M (2020) NLP-assisted software testing: a systematic mapping of the literature. *Inf Softw Technol*
- [30] OpenAI (2023) GPT-4 technical report. arXiv preprint arXiv:2303.08774. <https://arxiv.org/abs/2303.08774>
- [31] Mistral AI (2023) Mistral 7B technical report. arXiv preprint arXiv:2310.06825. <https://arxiv.org/abs/2310.06825>
- [32] Meta AI (2024) Introducing Llama 3: the most capable openly available LLM to date. <https://ai.meta.com/blog/meta-llama-3/>
- [33] Touvron H, Lavril T, Izacard G, Martinet X, Lachaux MA, Lacroix T, Rozière B, Goyal N, Hambro E, Azhar F, Rodriguez A, Joulin A, Grave E, Lample G (2023) Llama: open and efficient foundation language models. arXiv preprint arXiv:2302.13971. <https://arxiv.org/abs/2302.13971>

- [34] Rozière B, Gehring J, Gloeckle F, Sootla A, Gat I, Tan XE, Adi Y, Liu J, Sauvestre R, Remez T, Rapin J, Kozhevnikov A, Evtimov I, Bitton J, Bhatt M, Ferrer CC, Grattafiori A, Xiong W, Défossez A, Copet J, Azhar F, Touvron H, Martin L, Usunier N, Scialom T, Synnaeve G (2024) Code Llama: open foundation models for code
- [35] Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, Agarwal S, Herbert-Voss A, Krueger G, Henighan T, Child R, Ramesh A, Ziegler D, Wu J, Winter C, Hesse C, Chen M, Sigler E, Litwin M, Gray S, Chess B, Clark J, Berner C, McCandlish S, Radford A, Sutskever I, Amodei D (2020) Language models are few-shot learners. *Adv Neural Inf Process Syst (NeurIPS)*
- [36] Chen M, Tworek J, Jun H, Yuan Q, Pinto HVDP, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G, Ray A, Puri R, Krueger G, Petrov M, Khlaaf H, Sastry G, Mishkin P, Chan B, Gray S, Ryder N, Pavlov M, Power A, Kaiser L, Bavarian M, Winter C, Tillet P, Such FP, Cummings D, Plappert M, Chantzis F, Barnes E, Herbert-Voss A, Guss WH, Nichol A, Paino A, Tezak N, Tang J, Babuschkin I, Balaji S, Jain S, Saunders W, Hesse C, Carr AN, Leike J, Achiam J, Misra V, Morikawa E, Radford A, Knight M, Brundage M, Murati M, Mayer K, Welinder P, McGrew B, Amodei D, McCandlish S, Sutskever I, Zaremba W (2021) Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
<https://arxiv.org/abs/2107.03374>
- [37] Fan A, Gokkaya B, Harman M, Lyubarskiy M, Sengupta S, Yoo S, Zhang JM (2023) Large language models for software engineering: survey and open problems
- [38] Yuan Z, Liu Y, Ding B, Wang S (2023) No more manual tests? Evaluating and improving ChatGPT for unit test generation. *arXiv preprint arXiv:2305.04207*. <https://arxiv.org/abs/2305.04207>
- [39] Lukasczyk S, Fraser G (2022) Pynguin: automated unit test generation for Python. In: *International Conference on Software Engineering – Companion Proceedings (ICSE-Companion)*
- [40] Kang S, An G, Yoo S (2023) Large language models are few-shot testers: exploring LLM-based general bug reproduction. In: *International Conference on Software Engineering (ICSE)*
- [41] Wei J, Wang X, Schuurmans D, Bosma M, Xia F, Chi E, Le QV, Zhou D (2022) Chain-of-thought prompting elicits reasoning in large language models. *Adv Neural Inf Process Syst* 35:24824–24837. <https://arxiv.org/abs/2201.11903>
- [42] Alagarsamy S, Treude C, Böhme M (2023) A3test: assertion-augmented automated test case generation
- [43] Lemieux C, Inala JP, Lahiri SK, Sen S (2023) CODAMOSA: escaping coverage plateaus in test generation with pre-trained large language models. In: *International Conference on Software Engineering (ICSE)*
- [44] Dakhama A, Clark J, Harman M, Jia Y, Sarro F (2023) SearchGEM5: towards reliable gem5 with search based software testing and large language models. King's College London, London, UK, Tech. Rep.

- [45] Barr ET, Harman M, McMinn P, Shahbaz M, Yoo S (2015) The oracle problem in software testing: a survey. *IEEE Trans Softw Eng*
- [46] Pezzè M, Young M (2015) Automated test oracles: a survey. *Adv Comput*
- [47] Blasi A, Gorla A, Ernst MD, Carzaniga A (2021) MeMo: automatically identifying metamorphic relations in Javadoc comments for test automation. *J Syst Softw*
- [48] Fontes A, Gay G (2021) Using machine learning to generate test oracles: a systematic literature review. *TORACLE*
- [49] Dinella E, Ryan G, Wang Z, Mytkowicz T, Singh S, Lahiri SK (2022) TOGA: a neural method for test oracle generation. In: *International Conference on Software Engineering (ICSE)*
- [50] Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. *IEEE Trans Softw Eng*.
<https://doi.org/10.1109/TSE.2010.112>
- [51] Papadakis M, Kintis M, Zhang J, Jia Y, Le Traon Y, Harman M (2019) Mutation testing advances: an analysis and survey. *Adv Comput*. <https://doi.org/10.1016/bs.adcom.2018.03.015>
- [52] Parsai A, Soetens QD, Demeyer S (2024) Current and future trends in mutation testing: a survey. *Softw Qual J*.
<https://link.springer.com/journal/11219>
- [53] Coles H, Laurent T, Henard C, Papadakis M, Ventresque A (2016) PIT: a practical mutation testing tool for Java. In: *International Symposium on Software Testing and Analysis (ISSTA)*
- [54] Goodfellow IJ, Shlens J, Szegedy C (2014) Explaining and harnessing adversarial examples
- [55] Wang L, Ma C, Feng X, Zhang Z, Yang H, Zhang J, Chen Z, Tang J, Chen X, Lin Y, Zhao WX, Wei Z, Wen JR (2024) A survey on large language model based autonomous agents. *Front Comput Sci*. <https://arxiv.org/abs/2308.11432>
- [56] Wu X, Dagan R, Gal-Oz A, Tariku S, David HS, Bechhofer E, Atias O (2023) AutoGPT: an empirical study of autonomous AI agents for complex task solving. *arXiv preprint arXiv:2306.05801*. <https://arxiv.org/abs/2306.05801>
- [57] Chase H (2023) LangChain: building applications with LLMs through composability. <https://github.com/langchain-ai/langchain>
- [58] Jain S (2023) Integrating artificial intelligence with DevOps: enhancing continuous delivery, automation, and predictive analytics for high-performance software engineering. *World J Adv Res Rev* 17(3):1025–1043.
<https://doi.org/10.30574/wjarr.2023.17.3.0365>
- [59] Just R, Jalali D, Ernst MD (2014) Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In: *International Symposium on Software Testing and Analysis (ISSTA)*
- [60] Chen Z, Zhang J, Cao J, Zheng W, Ma L, Liu Y (2024) Fairness testing: a comprehensive survey and analysis of trends. *ACM Trans Softw Eng Methodol*

- [61] Bai Y, Jones A, Ndousse K, Askill A, Chen A, DasSarma N, Drain D, Fort S, Ganguli D, Henighan T, Joseph N, Kadavath S, Kernion J, Conerly T, El-Showk S, Elhage N, Hatfield-Dodds Z, Hernandez D, Hume T, Johnston S, Kravec S, Lovitt L, Nanda N, Olsson C, Amodei D, Brown T, Clark J, McCandlish S, Olah C, Mann B, Kaplan J (2022) Training a helpful and harmless assistant with reinforcement learning from human feedback
- [62] Schwartz R, Dodge J, Smith NA, Etzioni O (2020) Green AI. *Commun ACM*. <https://doi.org/10.1145/3381831>
- [63] ISO/IEC (2011) Systems and software engineering – systems and software quality requirements and evaluation (SQuARE) – system and software quality models. ISO/IEC 25010:2011. <https://www.iso.org/standard/35733.html>
- [64] Tian F, Zheng Q, Zheng W (2021) The impact of traceability on software maintenance and evolution: a mapping study. *arXiv preprint arXiv:2108.02133*. <https://arxiv.org/abs/2108.02133>
- [65] Sundaram SK, Hayes JH, Dekhtyar A, Holbrook EA (2010) Assessing traceability of software engineering artifacts. *Requir Eng* 15(3):313–335. <https://doi.org/10.1007/s00766-010-0101-2>
- [66] Cleland-Huang A, Gotel O, Zisman A (eds) (2012) *Software and systems traceability*. Springer, London. <https://link.springer.com/book/10.1007/978-1-4471-2239-5>
- [67] Anand S, Burke EK, Chen TY, Clark J, Cohen MB, Grieskamp W, Harman M, Harrold MJ, McMinn P (2013) An orchestrated survey of methodologies for automated software test case generation. *J Syst Softw*
- [68] Petersen K, Vakkalanka S, Kuzniarz L (2015) Guidelines for conducting systematic mapping studies in software engineering: an update. *Inf Softw Technol*
- [69] Kitchenham B, Charters S (2007) Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, Keele University and Durham University Joint Report, UK
- [70] Kitchenham B, Pearl Brereton O, Budgen D, Turner M, Bailey J, Linkman S (2010) Systematic literature reviews in software engineering – a tertiary study. *Inf Softw Technol*
- [71] Wohlin C (2014) Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*
- [72] Hall T, Beecham S, Bowes D, Gray D, Counsell S (2012) A systematic literature review on fault prediction performance in software engineering. *IEEE Trans Softw Eng*
- [73] Allamanis M, Barr ET, Devanbu P, Sutton C (2018) A survey of machine learning for big code and naturalness. *ACM Comput Surv*
- [74] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. *Adv Neural Inf Process Syst (NeurIPS)*

- [75] Devlin J, Chang MW, Lee K, Toutanova K (2019) BERT: pre-training of deep bidirectional transformers for language understanding
- [76] Tufano M, Drain D, Svyatkovskiy A, Dey P, Sundaresan N (2020) Unit test case generation with transformers and focal context
- [77] Watson C, Cooper N, Palacio DN, Moran K, Poshyvanyk D (2020) On learning meaningful assert statements for unit test cases. In: International Conference on Software Engineering (ICSE)
- [78] Rao N, Eghbali A, Devanbu P (2023) CAT-LM training language models on aligned code and tests. In: IEEE/ACM International Conference on Automated Software Engineering (ASE)
- [79] Blasi A, Gorla A, Ernst MD, Carzaniga A (2018) Translating code comments to procedure specifications. In: International Symposium on Software Testing and Analysis (ISSTA)
- [80] Lewis P, Perez E, Piktus A, Petroni F, Karpukhin V, Goyal N, Küttler H, Lewis M, Yih W, Rocktäschel T, Riedel S, Kiela D (2020) Retrieval-augmented generation for knowledge-intensive NLP tasks. *Adv Neural Inf Process Syst (NeurIPS)*. <https://arxiv.org/abs/2005.11401>
- [81] Ernst MD, Perkins JH, Guo PJ, McCamant S, Pacheco C, Tschantz MS, Xiao C (2007) The Daikon system for dynamic detection of likely invariants. *Sci Comput Program*
- [82] Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR (2008) EXE: automatically generating inputs of death. *ACM Trans Inf Syst Secur*
- [83] Pacheco C, Ernst MD (2007) Randoop: feedback-directed random testing for Java. In: *Proceedings of OOPSLA*
- [84] Daka E, Fraser G (2014) A survey on unit testing practices and problems. In: *International Symposium on Software Reliability Engineering (ISSRE)*
- [85] Chollet F (2021) *Deep learning with Python*. Manning Publications. <https://www.manning.com/books/deep-learning-with-python-second-edition>
- [86] Myers GJ, Sandler C, Badgett T (2011) *The art of software testing*. John Wiley & Sons. <https://www.wiley.com/en-us/The+Art+of+Software+Testing-p-9781118031964>
- [87] Michalski RS, Carbonell JG, Mitchell TM (eds) (2013) *Machine learning: an artificial intelligence approach*. Springer. <https://link.springer.com/book/10.1007/978-3-662-12405-5>
- [88] Luke S (2013) *Essentials of metaheuristics*. Lulu. <https://cs.gmu.edu/~sean/book/metaheuristics/>
- [89] Madeyski L (2010) *The impact of test-driven development on software quality and developers' productivity*. Springer. <https://doi.org/10.1007/978-3-642-04288-1>
- [90] Bertolino A (2007) Software testing research: achievements, challenges, dreams. In: *Future of Software Engineering (FOSE)*

- [91] García S, Herrera F (2008) An extension on statistical comparisons of classifiers over multiple data sets for all pairwise comparisons. *J Mach Learn Res*. <https://www.jmlr.org/papers/v9/garcia08a.html>
- [92] International Organization for Standardization (2013) ISO/IEC/IEEE international standard – software and systems engineering – software testing – part 1: concepts and definitions. ISO/IEC/IEEE 29119-1:2013(E)
- [93] Niessink F, Tijssen R (2014) The impact of software process improvement on time-to-market: an empirical study. *Softw Qual J*. <https://doi.org/10.1007/s11219-014-9231-6>
- [94] Jones C (2008) Applied software measurement. McGraw-Hill. <https://www.mheducation.com/highered/product/applied-software-measurement-jones/9780071502443.html>
- [95] International Software Testing Qualifications Board (ISTQB) (2026) Certified tester AI testing syllabus, version 2.0. https://istqb.org/wp-content/uploads/2026/05/ISTQB-CTAI_Syllabus_v2.0_Release.pdf. Accessed 27 June 2026
- [96] Zhang J, Wang Z, Zhang L, Danialangard D, Xie T (2019) Predictive mutation testing. *IEEE Trans Softw Eng* 45(9):898–918. <https://doi.org/10.1109/TSE.2018.2804921>
- [97] Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. *IEEE Trans Softw Eng* 37(5):649–678. <https://doi.org/10.1109/TSE.2010.112>
- [98] Chang, Z., Li, M., Wang, J., Wang, Q., & Li, S. (2022). Putting them under microscope: A fine-grained approach for detecting redundant test cases in natural language. *arXiv*. <https://doi.org/10.48550/arxiv.2210.01661>

Statements and Declarations

Competing Interests

The authors have no competing interests to declare that are relevant to the content of this article.

Funding statement

The authors received no specific funding for this work.

Author Contribution

All authors contributed to study conception and design. Material preparation, data collection and analysis were performed by Ibrahim ElSamman and Mohammed Safy. The first draft was written by Ibrahim ElSamman, and all authors commented on and approved the final manuscript.

Data Availability

The datasets generated and/or analyzed during the study are available in the following repository:

<https://drive.google.com/drive/folders/1awCOXziqQysxxV19TZDZUCVqKI25rm7Q>.