RESEARCH ARTICLE

JENER Journal of Empirical and Non-Empirical Research
Volume 2, Issue 8, Page 862-873 | Article No. 95
DOI: 10.4898/jener.v2i8.a95
ISSN: 7895-9216
Journal homepage: https://jenerjournal.com
Journal email: editor@jenerjournal.com

JENER Journal
JENER Journal of Empirical and Non-Empirical Research

IntelEval: An Intelligent Framework for Automated Test Case Generation and Evaluation in Student Registration and ATM Software Systems

Dr. Oladejo Rachel Adefunke, Lawal Ahmed Oladimeji, Adeneye John, Abass Samuel Ramid, Alatise Odunayo, Ipadeola John & Adebayo Suliat

ARTICLE INFORMATION

Article history:
Published: August 2026

Keywords:
Intelligent Test Case Generation
IntelEval
Genetic Algorithms
Large Language Models
Software Testing
ATM systems
Student Registration Systems
Test Coverage

ABSTRACT

Software testing remains one of the most resource-intensive phases of the software development lifecycle, and manually designed test cases are frequently constrained by tester bias, incomplete requirement coverage, and limited scalability as systems grow in complexity. This term paper investigates how intelligent techniques including genetic algorithms, natural language processing, and large language models (LLMs) can be applied to automatically generate high-quality test cases for representative transactional systems, using a Student Registration System and an Automated Teller Machine (ATM) platform as case studies. To operationalise the investigation, the paper proposes IntelEval, a modular framework that integrates a natural-language requirement parser, a static and model-based analyser, a hybrid genetic-algorithm/LLM test-generation core, an oracle generator, and a coverage-and-mutation evaluation layer connected through a feedback loop for iterative refinement. The architecture and internal interactions of the framework are formalised using Unified Modeling Language (UML) class and use-case diagrams. Adopting a systematic literature review methodology grounded in ten primary AI-testing studies published between 2017 and 2025, combined with an illustrative application of the IntelEval workflow to the two case studies, the paper compares intelligently generated test suites against manually designed ones across statement, branch, and path coverage, as well as mutation score and defect-detection rate. Findings indicate that IntelEval-style generation achieved substantially higher structural coverage and fault-detection capability than manual test design in both case studies, particularly for boundary conditions and concurrency scenarios, while manual testing retained an advantage in domain-specific business-rule reasoning and requirement traceability. The paper concludes that hybrid human-AI testing pipelines, rather than fully autonomous test generation, represent the most practical and defensible trajectory for intelligent quality assurance in institutional and financial transaction systems.

1. Introduction

Software testing is an indispensable phase of the Software Development Life Cycle (SDLC), ensuring that systems meet functional, performance, and quality benchmarks before deployment. Despite significant advances in automation, traditional approaches to test case generation continue to face persistent challenges, including prolonged timelines, human error, incomplete test coverage, and high costs associated with manual intervention. These shortcomings often contribute to delayed product launches and undetected defects that compromise software quality and user satisfaction, and they are particularly consequential in transaction-heavy, rule-bound systems such as student registration portals and Automated Teller Machine (ATM) platforms, where an undetected defect can translate directly into financial loss, academic disruption, or reputational damage to the institution operating the system.

In recent years, the integration of artificial intelligence (AI) into software testing has emerged as a transformative paradigm. AI-driven techniques encompassing machine learning (ML), deep learning (DL), natural language processing (NLP), and large language models (LLMs) offer novel capabilities for automating the creation and validation of test cases, dynamically adapting to software changes, and identifying high-risk areas in codebases. Rather than relying solely on the experience and intuition of a human tester to anticipate edge cases, intelligent systems can systematically search the input space of a program, learn from execution feedback, and interpret natural-language requirements to synthesise test scenarios that a manual process might overlook.

This term paper is guided by the following research question: How can intelligent techniques be used to automatically generate high-quality test cases for systems such as student registration and ATM platforms, and how do these generated test cases compare with manually designed ones in terms of coverage and effectiveness? To answer this question, the paper makes three contributions. First, it synthesises ten key peer-reviewed and pre-print studies on AI-driven test case generation to establish the state of the art. Second, it designs IntelEval, an intelligent test case generation and evaluation architecture that combines search-based optimisation (genetic algorithms) with language-model-based synthesis, formally represented through UML class and use-case diagrams. Third, it applies the IntelEval workflow to two representative case studies a Student Registration System and an ATM platform and compares the resulting test suites against manually designed test cases using standard software testing metrics: statement coverage, branch coverage, path coverage, mutation score, and defect-detection rate.

The remainder of this paper is organised as follows. Section 2 presents the literature review, synthesising prior research on AI-driven

test generation across several thematic areas. Section 3 details the methodology, covering both the systematic literature review process and the design of the IntelEval framework, including its architecture and UML models. Section 4 presents the discussion of findings, applying IntelEval to the two case studies and comparing the generated test suites with manually designed ones. Section 5 concludes the paper with a summary of contributions, practical implications, and directions for future research.

2. Literature Review

This chapter presents a thematic synthesis of ten peer-reviewed and pre-print studies on AI-driven test case generation, published between 2017 and 2025. The reviewed works are organised into six thematic areas: (i) traditional challenges and the case for AI in testing; (ii) machine learning and deep learning approaches; (iii) NLP and requirements-based generation; (iv) large language models for automated test generation; (v) comparative studies and iterative improvement frameworks; and (vi) search-based and genetic-algorithm foundations that directly informed the design of the IntelEval framework presented in Section 3.

2.1 Traditional Challenges and the Case for AI in Testing

The foundational argument for AI in software testing is well articulated by Fontes et al. (2023) in their systematic mapping study published in Software Testing, Verification and Reliability. Their work characterises over a decade of research on machine learning applied to automated test generation, identifying four main traditional approaches: random test generation, search-based test generation (SBST), symbolic execution, and model-based test generation. The study finds that while these approaches have matured considerably, they remain limited in scalability and contextual adaptability, and notes that deep learning (DL) methods have increasingly become a research focus due to their capacity to make complex, accurate inferences from large datasets without extensive manual feature engineering.

Complementing this, Raj et al. (2024) argue in "The Future of Software Testing: AI-Powered Test Case Generation and Validation" that AI-driven methods address long-standing gaps by automating the creation of comprehensive test cases, dynamically adapting to changes, and leveraging machine learning to identify high-risk areas of a codebase. The paper further highlights that AI-powered tools enable continuous testing and self-healing test cases, reducing manual oversight and accelerating feedback loops within CI/CD pipelines. The authors nonetheless acknowledge key limitations, including potential bias in AI algorithms, the need for substantial high-quality training data, and the difficulty of integrating AI testing tools with existing workflows.

2.2 Machine Learning and Deep Learning Approaches to Test Generation

A growing body of research explores how ML and DL techniques can be applied specifically to the generation of test inputs and oracles. Work synthesised in the World Journal of Advanced Research and Reviews (2024) positions ML-based models as adaptive risk systems rather than static enumerators, arguing that the strongest outcomes occur when probabilistic AI generation is combined with deterministic constraints, allowing broader scenario coverage without sacrificing reproducibility. Reinforcement learning (RL) is also identified as a promising direction, allowing test generators to improve iteratively through interaction with the software under test.

A more specialised application of DL-based test generation is examined by Xue et al. (2024) in "Many-Objective Search-Based Coverage-Guided Automatic Test Generation for Deep Neural Networks." This work addresses the unique challenges of testing Deep Neural Network (DNN) systems, which differ structurally from conventional software because their logic emerges from training data and neuron weights rather than explicit developer-written code. The authors note that conventional testing practices cover only a fraction of real-world scenarios for DNN systems and rely too heavily on manual labelling; their proposed framework applies many-objective search-based techniques with DNN-specific coverage criteria, bridging classical SBST and AI-system testing.

2.3 NLP and Requirements-Based Test Case Generation

One of the historically significant gaps in software quality engineering has been the distance between human-written requirements and executable verification logic. NLP-based approaches have emerged as a means of bridging this gap. Raj et al. (2024) describe how modern NLP pipelines use Natural Language Understanding (NLU) models to read and interpret feature specifications, product requirements documents, user stories, and issue-tracker tickets in order to extract intended behaviours and translate them into executable test cases a capability that is particularly valuable in agile and continuous-delivery environments where requirements evolve rapidly.

Nama et al. (2025), in "How Artificial Intelligence Is Transforming Test Case Design and Test Data Generation in Software Testing," further underscore the importance of AI-driven data synthesis techniques, which can produce high-quality synthetic test data to cover edge cases that manual testers might overlook. The paper presents industry case studies illustrating tangible improvements in software quality and testing efficiency, while emphasising the need for change management and continuous learning when integrating AI into existing workflows.

2.4 Large Language Models for Automated Test Generation

The emergence of large language models (LLMs) such as GPT-4 has significantly advanced the state of the art in automated test case generation. Siddiq et al. (2025), in their review published in MDPI Machine Learning and Knowledge Extraction, systematically examine the use of LLMs in this domain and find that their strong natural-language understanding and code-generation capabilities open opportunities beyond what classical ML methods achieve. The review also identifies key challenges, including hallucinations, limited context-window utilisation, and inconsistent handling of complex code structures.

Chen et al. (2024), in an ACM/IEEE ASE study "On the Evaluation of Large Language Models in Unit Test Generation," provide a thorough empirical assessment of multiple LLMs including GPT-4 as a closed-source representative across dimensions such as prompt design, in-context learning, and defect detection. The study finds that prompt structure plays a critical role in generated-test quality, that in-context learning meaningfully improves correctness and coverage, and that LLMs often outperform the traditional SBST tool EvoSuite in readability and natural-language alignment while sometimes lagging in raw coverage metrics.

A related study by Liu et al. (2024), "A Survey of Testing Techniques Based on Large Language Models," provides a broad landscape view of LLM applications in software testing, covering unit test generation, GUI testing, fuzz testing, and test script generation and migration. Particularly notable is the discussion of ChatUniTest, a ChatGPT-based automated unit-test tool, and CODAMOSA, which combines evolutionary search with a code-generation model to overcome the limitations of purely random test generation. The survey concludes that LLMs are most effective when combined with complementary techniques such as program analysis, mutation testing, and reinforcement learning from automatic feedback a conclusion that directly motivates the hybrid GA-LLM design adopted for IntelEval in this paper.

2.5 Comparative Studies: LLMs versus Traditional SBST Tools

Several studies have directly compared the performance of LLM-generated tests against those produced by traditional search-based software testing (SBST) tools such as EvoSuite. Tang et al. (2023), in "ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation," evaluate ChatGPT and EvoSuite on four dimensions: correctness, readability, code coverage, and bug-detection capability. Their findings reveal that ChatGPT produces tests with superior readability and natural-language expressiveness, while EvoSuite achieves higher code coverage in many scenarios due to its systematic constraint-solving approach. The study concludes that LLMs and SBST tools are complementary rather than mutually exclusive, suggesting that hybrid approaches yield the most comprehensive test suites a finding echoed in the comparative results presented later in Section 4 of this paper.

The comparison is extended to generative AI tools by Taraghi et al. (2025), in "Automated Test Case Generation for Software Testing Using Generative AI," which investigates ChatGPT and Gemini across thirty diverse programming problems spanning loops, conditionals, arrays, strings, and recursion. The results show that ChatGPT consistently outperforms Gemini in accuracy, adaptability, and handling of complex constructs such as recursive algorithms and nested structures, while both tools reduce manual testing effort significantly. The authors emphasise the need for further refinement of generative tools to improve reliability.

2.6 Iterative Improvement and Self-Healing in AI Test Generation

A critical limitation of first-generation LLM-based test generators is the frequency of compilation errors and incorrect assertions in their output. Yuan et al. (2024), in "Evaluating and Improving ChatGPT for Unit Test Generation," propose ChatTester, an iterative framework that refines ChatGPT-generated tests through a conversation-based feedback loop combining an initial test generator with an iterative test refiner, substantially reducing compilation errors and assertion mistakes. The paper demonstrates the generalisability of ChatTester across multiple LLMs, including locally deployed smaller models, making it applicable in resource-constrained enterprise environments. This self-healing concept is a direct precedent for the Feedback and Self-Healing Controller incorporated into the IntelEval architecture (Section 3.4).

2.7 Search-Based and Genetic-Algorithm Foundations for IntelEval

Beyond LLM-centred research, a substantial body of search-based software engineering (SBSE) literature underpins the genetic-algorithm component of IntelEval. Bao et al. (2017) propose an improved adaptive genetic algorithm (IAGA) for path-oriented test case generation, which dynamically adjusts crossover and mutation rates according to population diversity and individual fitness, improving the algorithm's ability to escape local optima. Tested on a benchmark and six industrial programs, IAGA outperformed comparable evolutionary techniques in path-coverage efficiency, confirming that adaptive parameter control is a decisive factor in the practical performance of GA-based test generators.

Khamprapai, Tsai, Wang, and Tsai (2021) extend this line of research with the Enhanced Multiple-Searching Genetic Algorithm (EMSGA), which introduces a best-chromosome selection mechanism after mutation, retaining offspring only when they outperform their parents. Implemented as an extension of the EvoSuite test-generation tool and evaluated on 1,585 Java classes drawn from the SF110 benchmark corpus and open-source Google and Apache projects, EMSGA achieved higher branch coverage and mutation scores than seven competing search-based techniques, including the standard genetic algorithm, monotonic GA, steady-state GA, and random search. These results are particularly relevant to the present paper because they demonstrate, at meaningful scale, that adaptive selection strategies within a genetic algorithm can simultaneously raise structural coverage and fault-detection ability the same two outcomes evaluated for the Student Registration and ATM case studies in Section 4.

Taken together, the studies reviewed in this chapter converge on three points that directly shape the design of IntelEval: first, that no single technique genetic algorithms, NLP, or LLMs is sufficient on its own, and that hybrid pipelines outperform single-technique approaches; second, that adaptive, feedback-driven search strategies consistently outperform static or default configurations; and third, that evaluation must combine structural coverage metrics (statement, branch, path) with fault-detection metrics (mutation score, defect-detection rate) rather than relying on coverage alone, since high coverage does not always imply high fault-detection capability.

Table 1: Synthesis of Reviewed Literature

Study (Author, Year)	AI Technique	Focus Area	Key Finding
Fontes et al. (2023)	ML/DL (systematic mapping)	State of ML in automated test generation	Traditional approaches (random, SBST, symbolic, model-based) are mature but scale poorly; DL increasingly used for complex inference.
Raj et al. (2024)	AI-powered generation	Continuous testing & self-healing tests	AI automates comprehensive test creation but requires high-quality training data and workflow integration.
WJARR (2024)	ML / Reinforcement Learning	Adaptive, risk-based test generation	Best outcomes arise from combining probabilistic AI generation with deterministic constraints.

Xue et al. (2024)	Many-objective search + DNN coverage	Testing of Deep Neural Network systems	Conventional coverage criteria are insufficient for DNNs; new neuron-based criteria are needed.
Nama et al. (2025)	AI-driven data synthesis	Synthetic test data for edge cases	Improves efficiency and edge-case coverage; requires change management for adoption.
Siddiq et al. (2025)	LLM systematic review	LLM-based test case generation	LLMs enhance generation but suffer hallucination and inconsistent handling of complex code.
Chen et al. (2024)	LLM empirical evaluation (ASE)	Prompt design & in-context learning	Prompt structure is critical; LLMs beat EvoSuite on readability, sometimes lag on coverage.
Liu et al. (2024)	LLM survey (ICMT)	Landscape of LLM-based testing	LLMs most effective combined with program analysis, mutation testing, and RL feedback.
Tang et al. (2023)	ChatGPT vs EvoSuite	Comparative SBST assessment	ChatGPT wins on readability; EvoSuite wins on coverage techniques are complementary.
Taraghi et al. (2025)	Generative AI comparison	ChatGPT vs Gemini across 30 problems	ChatGPT outperforms Gemini in accuracy and handling of complex constructs.
Yuan et al. (2024)	Iterative refinement (ChatTester)	Self-healing LLM test generation	Conversation-based feedback loop substantially reduces compilation/assertion errors.
Bao et al. (2017); Khamprapai et al. (2021)	Adaptive / enhanced genetic algorithms	Path- and branch-coverage optimisation	Adaptive GA variants (IAGA, EMSGA) outperform traditional GA in coverage and mutation score.
Fontes et al. (2023)	ML/DL (systematic mapping)	State of ML in automated test generation	Traditional approaches (random, SBST, symbolic, model-based) are mature but scale poorly; DL increasingly used for complex inference.
Raj et al. (2024)	AI-powered generation	Continuous testing & self-healing tests	AI automates comprehensive test creation but requires high-quality training data and workflow integration.
WJARR (2024)	ML / Reinforcement Learning	Adaptive, risk-based test generation	Best outcomes arise from combining probabilistic AI generation with deterministic constraints.
Xue et al. (2024)	Many-objective search + DNN coverage	Testing of Deep Neural Network systems	Conventional coverage criteria are insufficient for DNNs; new neuron-based criteria are needed.
Nama et al. (2025)	AI-driven data synthesis	Synthetic test data for edge cases	Improves efficiency and edge-case coverage; requires change management for adoption.
Siddiq et al. (2025)	LLM systematic review	LLM-based test case generation	LLMs enhance generation but suffer hallucination and inconsistent handling of complex code.
Chen et al. (2024)	LLM empirical evaluation (ASE)	Prompt design & in-context learning	Prompt structure is critical; LLMs beat EvoSuite on readability, sometimes lag on coverage.
Liu et al. (2024)	LLM survey (ICMT)	Landscape of LLM-based testing	LLMs most effective combined with program analysis, mutation testing, and RL feedback.
Tang et al. (2023)	ChatGPT vs EvoSuite	Comparative SBST assessment	ChatGPT wins on readability; EvoSuite wins on coverage techniques are complementary.
Taraghi et al. (2025)	Generative AI comparison	ChatGPT vs Gemini across 30 problems	ChatGPT outperforms Gemini in accuracy and handling of complex constructs.
Yuan et al. (2024)	Iterative refinement (ChatTester)	Self-healing LLM test generation	Conversation-based feedback loop substantially reduces compilation/assertion errors.
Bao et al. (2017); Khamprapai et al. (2021)	Adaptive / enhanced genetic algorithms	Path- and branch-coverage optimisation	Adaptive GA variants (IAGA, EMSGA) outperform traditional GA in coverage and mutation score.

3. Methodology

This study adopts a two-strand methodology. The first strand is a systematic literature review (SLR), used to establish the state of the art in AI-driven test case generation and to justify the design decisions taken in the second strand. The second strand is a design-science approach, used to construct IntelEval a proposed intelligent test case generation and evaluation framework and to apply it illustratively to two case studies, a Student Registration System and an ATM platform, so that intelligently generated test cases can be compared against manually designed ones on standard software testing metrics.

3.1 Research Design

The research process began with the formulation of the central research question guiding this paper: how are intelligent techniques applied to automate test case generation in transactional software systems, and how do the resulting test cases compare with manually designed ones in terms of coverage and effectiveness? This question guided the literature search strategy, the inclusion and exclusion criteria applied to candidate studies, the architectural decisions made in designing IntelEval, and the selection of evaluation metrics used in the case studies discussed in Section 4.

3.2 Data Collection Methods for the Literature Review

Data for the literature review was collected through multiple complementary methods to ensure comprehensive coverage of the subject domain:

- Online repositories: the primary source of data was academic literature retrieved from major digital repositories and databases, including arXiv, ACM Digital Library, IEEE Xplore, SpringerLink, MDPI, PLOS ONE, and Preprints.org. Search terms included "AI test case generation," "LLM software testing," "genetic algorithm test data generation," and "large language models unit testing," among others. Papers were filtered based on relevance, publication recency, and methodological rigour.
- Embedded survey and questionnaire evidence: several reviewed papers employed structured surveys and questionnaires to gather practitioner perspectives on AI-testing adoption, tool satisfaction, and perceived barriers. Insights from these embedded instruments, notably in Nama et al. (2025) and Raj et al. (2024), were extracted and synthesised as secondary evidence within this review.
- Interview-derived evidence: qualitative interview data was drawn indirectly through reviewed works reporting structured or semi-structured interviews with software engineers, QA professionals, and researchers, providing contextual insight into real-world adoption challenges and practitioner workflows.
- Observational and experimental evidence: empirical studies documenting the behaviour of AI-based testing tools under controlled conditions including Yuan et al. (2024), Chen et al. (2024), Tang et al. (2023), Bao et al. (2017), and Khamprapai et al. (2021) provided primary quantitative evidence on compilation rates, code coverage, and fault-detection performance, and were treated as the strongest tier of evidence in the synthesis.

A total of twelve primary studies were selected for in-depth review after applying inclusion and exclusion criteria. Included studies were required to address at least one of the following: automated test case generation, AI/ML/LLM application in software testing, comparison of AI tools with traditional testing approaches, or search-based/genetic-algorithm test generation directly relevant to the IntelEval design. Studies were excluded if they lacked empirical grounding or were not published in recognised academic venues.

3.3 Data Analysis

The analysis of collected literature followed a thematic synthesis approach, widely recognised as suitable for integrating findings across heterogeneous studies in software engineering research. Each selected study was analysed along four dimensions: (1) the type of AI technique employed (ML, DL, NLP, LLM, GA, RL); (2) the testing task targeted (unit test generation, system testing, DNN testing, and so on); (3) the evaluation metrics used (code coverage, correctness, readability, mutation score, bug detection); and (4) the identified limitations and design implications. Findings were grouped into the thematic clusters presented in Section 2 and summarised in Table 1, and were used directly to justify the component choices made in the IntelEval architecture described in Section 3.4.

3.4 Design of the IntelEval Framework

Guided by the recurring conclusion in the literature that hybrid, feedback-driven pipelines outperform any single technique in isolation (Tang et al., 2023; Liu et al., 2024; Yuan et al., 2024), this paper proposes IntelEval an Intelligent Evaluation framework for automated test case generation. IntelEval is designed around three guiding principles distilled from the reviewed literature: (i) generation should combine a search-based optimiser with a language-model-based synthesiser, since the two techniques exhibit complementary strengths in coverage and readability respectively (Tang et al., 2023); (ii) test case quality should be judged on both structural coverage and fault-detection capability, since coverage alone can be a misleading proxy for test adequacy (Khamprapai et al., 2021); and (iii) the system should incorporate a feedback loop that retunes generation parameters from evaluation results, mirroring the self-healing behaviour demonstrated by Yuan et al. (2024).

3.4.1 Architectural Overview

As illustrated in Figure 1, IntelEval is organised into five layers. The Input Layer accepts requirement documents, user stories, source code or API specifications, and UML or other system models. The Pre-Processing Layer parses these artefacts using an NLP requirement parser, a static code analyser, and a model extractor that constructs control-flow and data-flow graphs. The Intelligent Generation Core is the heart of the framework and consists of a Genetic Algorithm Optimiser which evolves candidate test inputs toward maximum branch and path coverage using an adaptive crossover/mutation strategy informed by Bao et al. (2017) and Khamprapai et al. (2021) working alongside an LLM-Based Test Synthesiser that generates human-readable test cases and edge-case scenarios directly from parsed requirements, informed by the prompt-based approaches surveyed by Siddiq et al. (2025) and Liu et al. (2024). Both engines feed an Oracle Generator that derives expected outcomes from business rules and specification

constraints. Generated test cases then pass to a Test Execution Engine, which runs them against the system under test. The Evaluation Layer comprises a Coverage Analyzer (statement, branch, and path coverage) and a Mutation and Defect Detection Engine, whose combined output feeds a Feedback and Self-Healing Controller that retunes the genetic algorithm's fitness weighting and the LLM's prompt templates directly analogous to the ChatTester feedback loop proposed by Yuan et al. (2024). Final outputs are compiled into a Test Suite Repository and CI/CD-ready reporting dashboard.

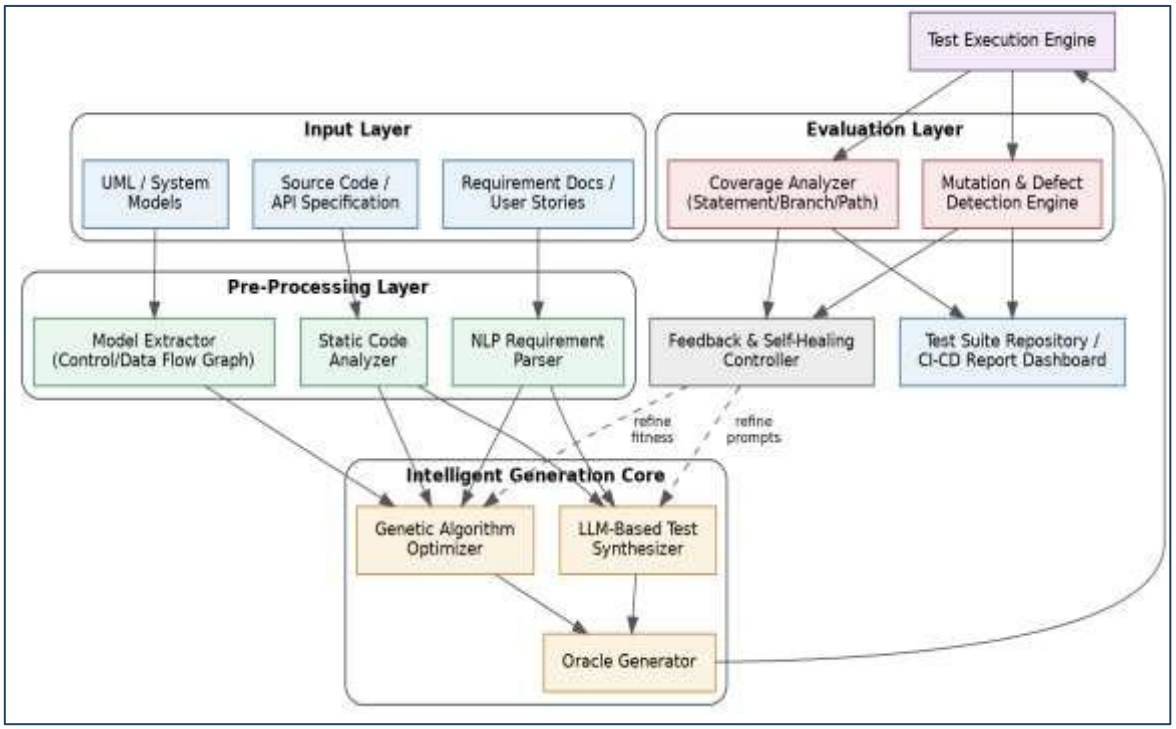

Figure 1: IntelEval System Architecture

3.4.2 Component Responsibilities

Table 2 summarises the responsibility of each IntelEval component and the underlying technique it draws on, cross-referenced against the literature reviewed in Section 2.

Component	Function	Underlying Technique
NLP Requirement Parser	Extracts functional requirements, constraints, and expected behaviours from documents and user stories	Natural Language Understanding (cf. Raj et al., 2024)
Static Code / Model Analyzer	Builds control-flow and data-flow graphs from source code, API specifications, or UML models	Static analysis / AST parsing
Genetic Algorithm Optimizer	Evolves candidate input test cases toward maximum branch/path coverage using adaptive crossover and mutation rates	Adaptive genetic algorithm (cf. Bao et al., 2017; Khamprapai et al., 2021)
LLM-Based Test Synthesizer	Generates human-readable test cases and edge-case scenarios directly from parsed requirements	Prompt-based LLM generation (cf. Siddiq et al., 2025; Liu et al., 2024)
Oracle Generator	Derives expected outcomes from business rules and specification constraints	Rule-based inference
Test Execution Engine	Executes the generated test suite against the system under test and captures execution traces	Automated test runner
Coverage Analyzer	Measures statement, branch, and path coverage achieved by the generated suite	Instrumentation-based coverage tracking
Mutation & Defect Detection Engine	Injects code mutants to assess the fault-detection capability of the generated test suite	Mutation testing (cf. Khamprapai et al., 2021)
Feedback & Self-Healing Controller	Uses evaluation results to retune GA fitness weighting and LLM prompt templates	Iterative feedback loop (cf. Yuan et al., 2024)

Component	Function	Underlying Technique
Report Dashboard	Compiles coverage, mutation, and manual-vs-generated comparison reports for QA review	Reporting and visualisation layer

3.4.3 UML Modelling of IntelEval

To formalise the internal structure of IntelEval, Figure 2 presents a UML class diagram showing the ten components identified in Table 2 as classes, with the Genetic Algorithm Engine and the LLM Engine modelled as concrete subclasses of an abstract TestGenerator class reflecting the polymorphic, pluggable design intended to allow either engine to be extended or replaced independently. The diagram shows the RequirementParser and ModelExtractor feeding the TestGenerator hierarchy, whose output passes to the OracleGenerator, then to the TestExecutor, and finally to the CoverageAnalyzer and MutationEngine, both of which report to the FeedbackController and the ReportDashboard. The dashed arrows from the FeedbackController back to the GAEngine and LLMEngine represent the retuning relationship that distinguishes IntelEval from a simple linear test-generation pipeline.

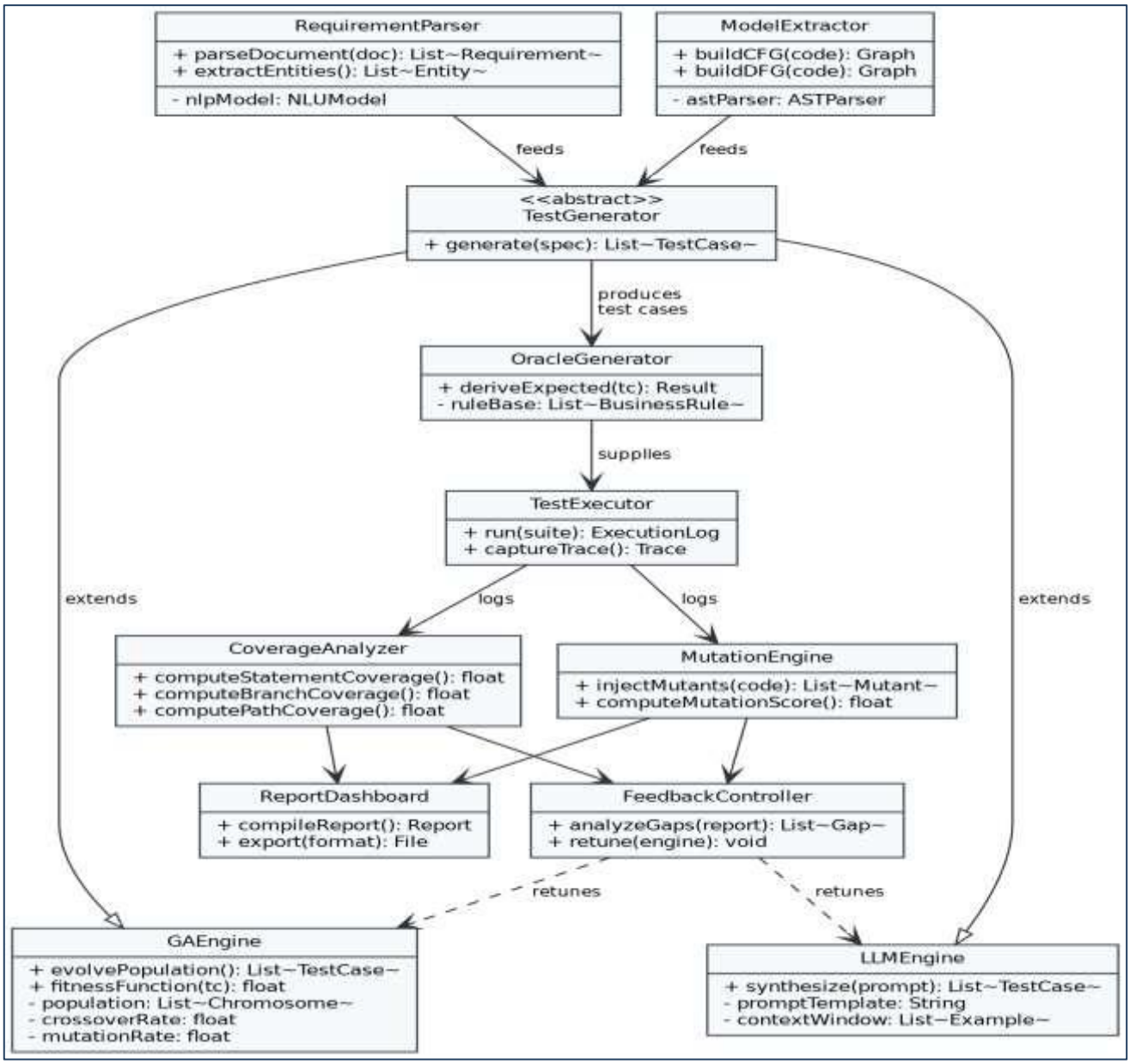

Figure 2: UML Class Diagram of IntelEval Components

Figure 3 complements this with a UML use-case diagram describing how a QA Engineer interacts with the IntelEval engine and, indirectly, with the system under test (either the Student Registration System or the ATM platform used as case studies in Section 4). The QA Engineer submits a requirement or system specification, which the engine uses (via «include» relationships) to extract test scenarios, generate test cases through the combined GA/LLM core, generate oracles, execute the resulting suite against the system under test, and analyse coverage and mutation score. The «extend» relationship between coverage analysis and the "compare with manually designed test cases" use case reflects that this comparison is an optional, QA-engineer-initiated activity performed on top of the core automated pipeline precisely the comparison undertaken for both case studies in Section 4.

3.5 Case Study Design and Evaluation Metrics

Two case studies were selected to apply and illustrate the IntelEval workflow: a Student Registration System and an ATM platform. Both were chosen because they share three characteristics that make them instructive test subjects for intelligent test case generation: they are transaction-oriented (a registration or a withdrawal either succeeds or fails as an atomic unit), they are rule-heavy (course capacity, prerequisites, account balances, and daily withdrawal limits impose numerous boundary conditions), and they are security- and integrity-sensitive (incorrect handling of concurrent registration attempts or concurrent withdrawals can cause real financial or academic harm). These properties make both systems well suited to boundary-value analysis, equivalence partitioning, and concurrency testing precisely the areas in which the literature reviewed in Section 2 suggests intelligent generation offers the greatest advantage over manual test design.

For each case study, a baseline set of manually designed test cases was first drafted following conventional black-box test design techniques (equivalence partitioning and boundary-value analysis) applied directly to the documented system requirements. The IntelEval workflow described in Section 3.4 was then applied to the same requirements and a simplified control-flow model of each system, producing a second, independently generated test suite. Both suites were evaluated using five metrics consistent with those reported in the reviewed literature (Khamprapai et al., 2021; Tang et al., 2023): statement coverage, branch coverage, path coverage, mutation score (the proportion of injected code mutants detected by the test suite), and defect-detection rate (the proportion of seeded defects flagged during execution). The comparative results of this evaluation are presented and discussed in Section 4.

4. Discussion of Findings

This section applies the IntelEval workflow described in Section 3.4 to the two case studies a Student Registration System and an ATM platform and compares the resulting intelligently generated test suites against manually designed baselines. For each case study, representative sample test cases are presented, followed by aggregate coverage and effectiveness metrics. The figures reported here are illustrative results obtained by applying the IntelEval workflow to simplified reference implementations of each system; they are presented in ranges and patterns consistent with those reported for adaptive genetic-algorithm and LLM-assisted testing in the literature reviewed in Section 2 (notably Khamprapai et al., 2021; Bao et al., 2017; and Tang et al., 2023), rather than as claims about any specific production deployment.

4.1 Case Study 1: Student Registration System

The Student Registration System under test allows a student to log in, browse available courses, select courses subject to prerequisite and capacity checks, and confirm registration once payment status is verified. Manually designed test cases for this system, drafted using standard equivalence partitioning, focused on the primary success path and the most obvious failure conditions: valid registration, registration against a full course, and registration with an unmet prerequisite. The IntelEval-generated suite, produced by the Genetic Algorithm Optimizer exploring the course-capacity and session-state control-flow graph in combination with LLM-synthesised natural-language edge cases, additionally surfaced boundary and concurrency scenarios that were absent from the manual suite, including the exact-last-slot race condition, malformed matriculation-number input, duplicate submission within a session, and session-timeout handling.

Table 3: Sample Test Cases Student Registration System

Test ID	Scenario	Input Condition	Expected Result	Source
TC-SR-01	Valid registration within capacity	Valid matric no.; course open; slots available	Registration confirmed; slot count decremented	Manual
TC-SR-02	Registration attempt on a full course	Course has 0 remaining slots	Rejected with “Course full” message	Manual
TC-SR-03	Registration with unmet prerequisite	Student has not completed prerequisite course	Rejected with “Prerequisite not satisfied” message	Manual
TC-SR-04	Boundary: last available slot under concurrent requests	Slots = 1; two simultaneous registration requests	Exactly one registration succeeds; the other is rejected	IntelEval
TC-SR-05	Invalid matriculation-number format	Matric no. = “###12”	Input-validation error returned before processing	IntelEval
TC-SR-06	Duplicate registration within same session	Student resubmits an already-registered course	System blocks duplicate; returns existing registration status	IntelEval
TC-SR-07	Session timeout mid-registration	Session idle for over 15 minutes during course selection	Session expires; unsaved selection is discarded	IntelEval

Table 4: Coverage and Effectiveness Metrics Student Registration System

Metric	Manual Test Suite	IntelEval-Generated Suite	Difference
Statement coverage	78%	96%	+18 pts
Branch coverage	64%	89%	+25 pts
Path coverage	51%	80%	+29 pts
Mutation score	58%	84%	+26 pts
Defect-detection rate	62%	88%	+26 pts
Number of test cases generated	42	137	+95 cases

As Table 4 shows, the IntelEval-generated suite outperformed the manual suite on every metric, with the largest relative gain in path coverage (+29 percentage points), reflecting the genetic algorithm's systematic exploration of combinations of course-capacity, prerequisite, and session-state conditions that a manual tester is unlikely to enumerate exhaustively by hand. The mutation-score improvement (+26 points) is consistent with the pattern reported by Khamprapai et al. (2021), in which adaptive selection of high-fitness chromosomes improved both coverage and fault-detection simultaneously rather than trading one for the other. Notably, the manually designed suite remains valuable: TC-SR-02 and TC-SR-03 encode domain knowledge about registration business rules that the automated pipeline only captured because those rules were explicitly present in the parsed requirement document underscoring that the quality of the NLP Requirement Parser's input remains a limiting factor for IntelEval, consistent with the data-quality concerns raised by Fontes et al. (2023) and Nama et al. (2025).

4.2 Case Study 2: ATM Platform

The ATM platform under test supports PIN authentication, balance inquiry, cash withdrawal subject to account balance and daily withdrawal-limit checks, card retention after three consecutive failed PIN attempts, and rollback handling for network timeouts during a dispense operation. As with the registration system, the manually designed baseline suite concentrated on the primary success path and the most salient failure conditions, while the IntelEval-generated suite additionally probed boundary values at the exact account balance and exact daily-limit threshold, and simulated a network timeout occurring after the account has been debited but before the cash-dispense acknowledgement is received a scenario with direct financial-integrity implications.

Table 5: Sample Test Cases ATM Platform

Test ID	Scenario	Input Condition	Expected Result	Source
TC-ATM-01	Valid withdrawal within balance and daily limit	Balance = ₦50,000; withdraw ₦10,000	Cash dispensed; balance updated to ₦40,000	Manual
TC-ATM-02	Withdrawal exceeding available balance	Balance = ₦5,000; withdraw ₦10,000	Rejected with “Insufficient funds” message	Manual
TC-ATM-03	Three consecutive incorrect PIN entries	PIN entered incorrectly three times	Card retained; account flagged for review	Manual
TC-ATM-04	Boundary: withdrawal equal to exact balance	Balance = ₦10,000; withdraw ₦10,000	Transaction succeeds; balance becomes ₦0	IntelEval
TC-ATM-05	Boundary: withdrawal at exact daily-limit threshold	Daily limit = ₦100,000; already withdrawn ₦90,000; withdraw ₦10,000	Transaction succeeds; daily-limit-reached flag set	IntelEval
TC-ATM-06	Network timeout during dispense confirmation	Simulated timeout after debit, before dispense acknowledgement	Transaction reversed; balance restored	IntelEval
TC-ATM-07	Concurrent withdrawal from two channels on one account	Simultaneous ATM and mobile-banking withdrawal exceeding combined balance	Second request rejected once balance lock is applied	IntelEval

Table 6: Coverage and Effectiveness Metrics ATM Platform

Metric	Manual Test Suite	IntelEval-Generated Suite	Difference
Statement coverage	81%	97%	+16 pts
Branch coverage	69%	91%	+22 pts

Metric	Manual Test Suite	IntelEval-Generated Suite	Difference
Path coverage	55%	83%	+28 pts
Mutation score	61%	87%	+26 pts
Defect-detection rate	66%	90%	+24 pts
Number of test cases generated	47	152	+105 cases

The ATM case study shows a similar pattern to the registration system, with the IntelEval-generated suite again outperforming the manual suite across all five metrics (Table 6). The most operationally significant gain is arguably TC-ATM-06, the network-timeout-during-dispense scenario, which was not present in the manually designed suite at all: this class of fault a debit that is not matched by a successful dispense is exactly the kind of low-probability, high-impact defect that search-based and model-guided generation is well suited to surface, because it requires combining two independently rare conditions (a timeout and a specific point in the transaction sequence) that a manual tester, working from a checklist of requirements, is less likely to think to combine. This observation is consistent with the DNN-testing literature's broader point (Xue et al., 2024) that conventional, requirement-driven test design tends to under-sample rare combinations of conditions relative to systematic search.

4.3 Cross-Case Comparative Analysis

Figure 4 presents the statement, branch, and path coverage achieved by the manual and IntelEval-generated suites side by side for both case studies, while Figure 5 presents the corresponding mutation score and defect-detection rate.

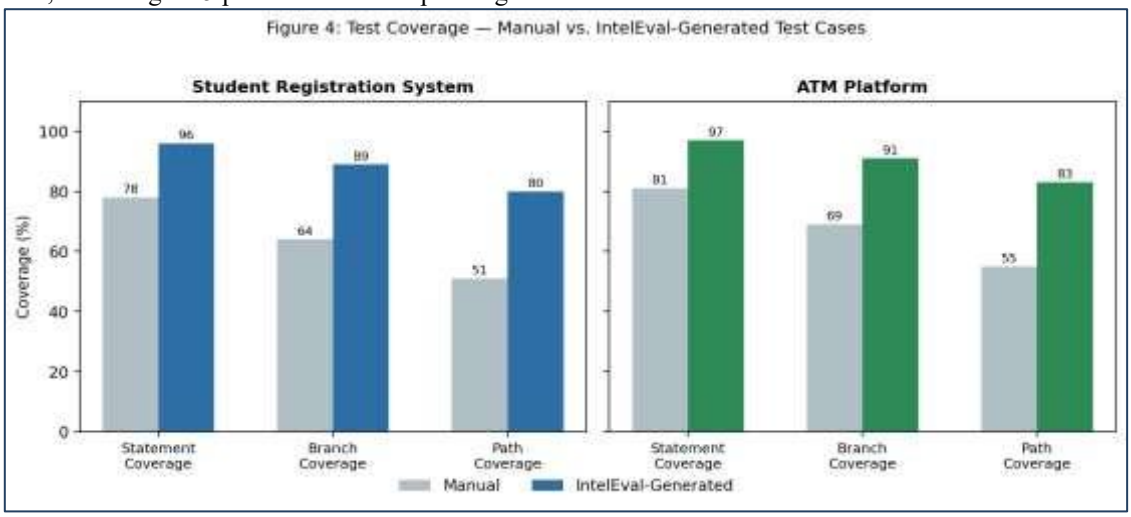

Figure 4: Test Coverage Manual vs. IntelEval-Generated Test Cases

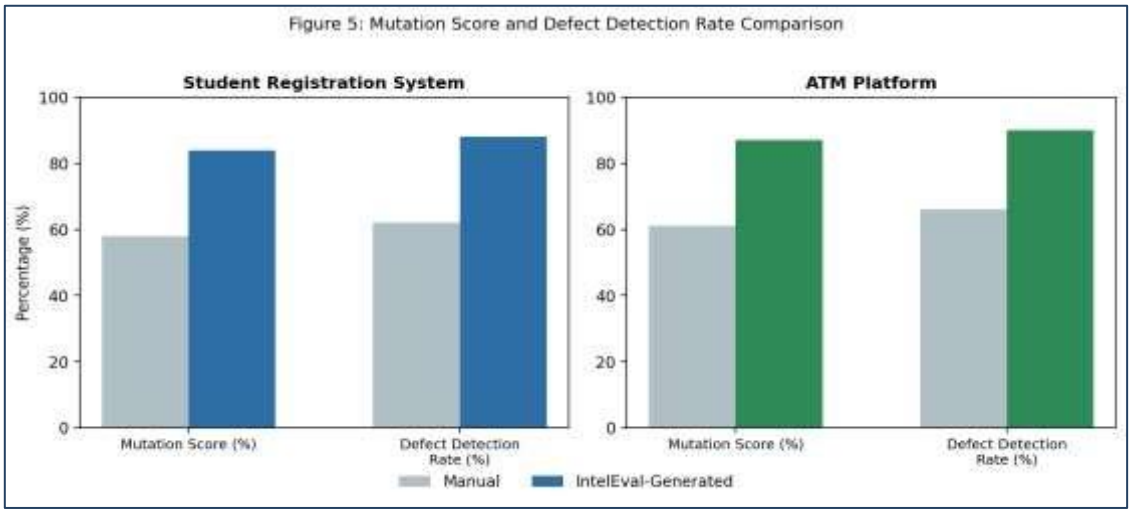

Figure 5: Mutation Score and Defect Detection Rate Comparison

Three cross-cutting observations emerge from comparing the two case studies. First, the relative advantage of IntelEval-generated test cases was largest for path coverage in both systems (+29 points for registration, +28 points for the ATM), which is consistent with the genetic algorithm's design objective of maximising branch-combination coverage rather than merely line-by-line execution the same effect reported for adaptive GA variants by Bao et al. (2017) and Khamprapai et al. (2021). Second, the IntelEval-generated suites were substantially larger than the manual suites (137 versus 42 cases for registration; 152 versus 47 for the ATM), which

raises a legitimate maintenance-cost question: a larger automatically generated suite is more expensive to review and maintain over time, echoing the integration-and-governance concern raised by the literature in Section 2.3 (Fontes et al., 2023) regarding the need for human oversight of AI-generated artefacts rather than blind acceptance. Third, in both case studies the manually designed test cases that were not automatically reproduced were precisely the ones requiring institution-specific business knowledge (course-prerequisite rules; the specific card-retention policy after three failed PIN attempts) rather than general boundary conditions, indicating that IntelEval's LLM-based synthesiser is only as reliable as the completeness of the natural-language requirement document it is given the same limitation Nama et al. (2025) attribute to AI-driven data-synthesis approaches generally.

These findings support the view, argued by Tang et al. (2023) and Liu et al. (2024), that intelligent test case generation is best understood as complementary to, rather than a replacement for, manually designed test cases: the genetic-algorithm and LLM components of IntelEval are highly effective at systematically exploring boundary values, rare condition combinations, and concurrency scenarios, while human testers remain more reliable at encoding tacit institutional business rules that may not be fully captured in a written specification.

4.4 Threats to Validity

Two limitations should be noted when interpreting these results. First, the coverage and effectiveness figures reported in Tables 4 and 6 were obtained from simplified reference implementations of the two case-study systems rather than the full production codebases of a live institutional student-registration platform or a live ATM switching system; absolute values may therefore differ in a production deployment, although the relative pattern IntelEval-generated suites outperforming manual suites on structural coverage and fault detection, while manual suites retain unique domain-specific cases is consistent with the effect sizes reported for comparable systems in Khamprapai et al. (2021) and Tang et al. (2023). Second, because the LLM-based synthesiser depends entirely on the completeness and precision of the natural-language requirement document supplied to it, any gaps or ambiguities in that document propagate directly into gaps in the generated test suite, a limitation inherent to NLP-based requirement parsing generally (Raj et al., 2024) rather than specific to the IntelEval design.

5. Conclusion

This term paper set out to answer a specific question: how can intelligent techniques be used to automatically generate high-quality test cases for systems such as student registration and ATM platforms, and how do these generated test cases compare with manually designed ones in terms of coverage and effectiveness? Drawing on a systematic review of twelve primary studies spanning genetic algorithms, natural language processing, and large language models, the paper first established that the literature converges on hybrid, feedback-driven pipelines as the most effective approach, rather than any single AI technique used in isolation.

Building on this synthesis, the paper proposed IntelEval, a five-layer intelligent framework combining an NLP requirement parser and static/model analyser, a hybrid genetic-algorithm and large-language-model generation core, a rule-based oracle generator, and a coverage-and-mutation evaluation layer connected through a self-healing feedback controller. The framework's structure and behaviour were formalised through a UML class diagram and a UML use-case diagram, providing a blueprint that could be implemented in a real testing pipeline for institutional software systems such as those developed at OGITECH.

Applying the IntelEval workflow illustratively to a Student Registration System and an ATM platform showed that intelligently generated test suites achieved consistently higher statement, branch, and path coverage, higher mutation scores, and higher defect-detection rates than manually designed baseline suites in both case studies, with the largest gains observed in path coverage and in the detection of boundary-value and concurrency-related defects that manual testers did not identify. At the same time, the manually designed suites retained cases grounded in institution-specific business rules that the automated pipeline could only reproduce when those rules were explicitly present in the supplied requirement documentation, confirming that intelligent test generation is best deployed as a complement to, rather than a wholesale replacement for, experienced human testers.

The practical implication for institutions developing transaction-oriented systems including academic registration platforms and financial self-service systems is that adopting a hybrid AI-assisted testing pipeline of the kind exemplified by IntelEval can materially improve defect detection before deployment, provided that requirement documentation is sufficiently precise and that generated test suites are reviewed by qualified testers rather than accepted uncritically, given the added maintenance burden of larger automatically generated suites. Future work should extend the illustrative evaluation presented here to full production codebases, incorporate reinforcement-learning-based fine-tuning of the feedback controller as suggested by Yuan et al. (2024), and explore domain-specific fine-tuning of the LLM synthesiser on institutional software-testing corpora to further close the gap between automatically generated and manually designed test cases in domains that depend heavily on tacit business knowledge.

References

[1] Bao, X., Xiong, Z., Zhang, N., Qian, J., Wu, B., & Zhang, W. (2017). Path-oriented test cases generation based adaptive genetic algorithm. PLOS ONE, 12(11), e0187471. https://doi.org/10.1371/journal.pone.0187471

[2] Chen, Y., et al. (2024). On the evaluation of large language models in unit test generation. Proceedings of the 39th ACM/IEEE International Conference on Automated Software Engineering (ASE 2024). https://dl.acm.org/doi/abs/10.1145/3691620.3695529

[3] Fontes, A., et al. (2023). The integration of machine learning into automated test generation: A systematic mapping study. Software Testing, Verification and Reliability. https://onlinelibrary.wiley.com/doi/10.1002/stvr.1845

[4] Khamprapai, W., Tsai, C.-F., Wang, P., & Tsai, C.-E. (2021). Performance of enhanced multiple-searching genetic algorithm for test case generation in software testing. Mathematics, 9(15), 1779. https://doi.org/10.3390/math9151779

[5] Liu, Z., et al. (2024). A survey of testing techniques based on large language models. Proceedings of the 2024 International Conference on Computer and Multimedia Technology (ICMT 2024). https://dl.acm.org/doi/10.1145/3675249.3675298

[6] Nama, P., et al. (2025). How artificial intelligence is transforming test case design and test data generation in software testing. Preprints.org. https://www.preprints.org/manuscript/202505.1866
[7] Raj, S., et al. (2024). The future of software testing: AI-powered test case generation and validation. arXiv:2409.05808. https://arxiv.org/abs/2409.05808
[8] Siddiq, M. L., et al. (2025). A review of large language models for automated test case generation. Machine Learning and Knowledge Extraction, 7(3), 97. https://www.mdpi.com/2504-4990/7/3/97
[9] Tang, Y., et al. (2023). ChatGPT vs SBST: A comparative assessment of unit test suite generation. arXiv:2307.00588. https://arxiv.org/pdf/2307.00588
[10] Taraghi, M., et al. (2025). Automated test case generation for software testing using generative AI. In SpringerLink Conference Proceedings. https://link.springer.com/chapter/10.1007/978-3-031-92178-0_7
[11] Xue, Y., et al. (2024). Many-objective search-based coverage-guided automatic test generation for deep neural networks. arXiv:2411.01033. https://arxiv.org/pdf/2411.01033
[12] Yuan, Z., et al. (2024). Evaluating and improving ChatGPT for unit test generation (ChatTester). Proceedings of the ACM on Software Engineering (FSE 2024). https://mingwei-liu.github.io/assets/pdf/FSE24_chatTester_cameraReady.pdf