AN INTELLIGENT GENERATIVE AI FRAMEWORK FOR AUTOMATED END-TO-END SOFTWARE TEST GENERATION, MULTI-LAYER VALIDATION, AND HALLUCINATION MITIGATION

G RAMESH[1], B YASHASWINI[2], N JASHWANTH[3]
ASSISTANT PROFESSOR [1,2&3]
DEPARTMENT OF ECE, SRI CHAITANYA INSTITUTE OF TECHNOLOGY AND RESEARCH, PONNEKAL, KHAMMAM, TELANGANA-507170

ABSTRACT

The growing complexity and release frequency of software applications increase the effort required to design, maintain, and execute end-to-end (E2E) tests. Generative artificial intelligence (GenAI) and large language models (LLMs) can assist with test authoring, but generated outputs may contain unsupported actions, invalid assertions, inconsistent test data, or code that cannot be executed. This paper presents an implementation-oriented framework, referred to as Rapid TestGen AI, for organizing the workflow from input preparation and feature extraction to AI-assisted test generation, validation, hallucination analysis, execution reporting, and comparative assessment. The supplied implementation evidence consists of screenshots of a web-based application showing project overview and workflow pages, administrative and user dashboards, dataset management and preprocessing, feature extraction and training, generated test-case views, multi-layer validation, hallucination detection and mitigation, execution-system results, and comparative-analysis pages. These screens establish the presence of corresponding user-interface modules; they do not, by themselves, establish the correctness of the underlying algorithms or the validity of displayed performance values. Accordingly, the paper formalizes a reproducible evaluation protocol and defines metrics for syntactic validity, requirement traceability, assertion support, execution success, functional coverage, hallucination incidence, and human review effort. No new experimental scores are claimed because the source package does not include raw datasets, model configuration logs, test execution traces, ground-truth labels, or independently verifiable result files. The contribution is therefore an implementation-grounded system description and an evaluation design that can be completed with auditable experiments. The framework is intended to support human-supervised test engineering rather than replace expert review.

INDEX TERMS: Generative artificial intelligence, large language models, automated software testing, end-to-end testing, test-case generation, hallucination detection, software quality assurance, test validation.

This is an open access article under the creative commons license https://creativecommons.org/licenses/by-nc-nd/4.0/
CC BY-NC-ND 4.0

I. INTRODUCTION

Software testing is a central activity in the software development life cycle (SDLC). End-to-end testing is particularly important because it checks whether application components cooperate to deliver user-visible business functionality. As applications evolve, test engineers must continually update test scenarios, selectors, input data, assertions, and execution environments. Manual test authoring can become a bottleneck when many workflows and application versions must be covered.

LLMs have made it possible to generate test descriptions and automation code from natural-language requirements, source fragments, interface descriptions, or structured configuration. However, fluent output is not necessarily correct output. A model may invent a control that is absent from the application, assume an endpoint or data field that is not specified, create an assertion unsupported by the requirement, or produce syntactically plausible code that fails at runtime. Such errors are especially problematic when generated tests are accepted without systematic validation.

The uploaded reference study by Barathidass and Namburu [1] investigates rapid E2E test generation through abstract-class-based DOM referencing and CRUD-specific single-shot prompting. It reports an

enterprise evaluation and discusses selector mismatch and missing assertions as prominent failure categories. The present work is distinct in scope: it documents the supplied Rapid TestGen AI implementation, whose visible interface organizes dataset management, feature extraction, generated test cases, multi-layer validation, hallucination analysis, execution results, and comparative analysis. The paper does not claim to reproduce the reference study's experiments or its reported metrics.

A. Motivation

A practical test-generation platform needs more than an LLM call. It requires traceable inputs, a structured representation of expected behavior, a validation path, controlled execution, and a record of why a generated test is accepted, rejected, or revised. The implementation screenshots indicate an application workflow intended to bring these activities together. This motivates a system-level description and an evaluation protocol that makes the quality of generated tests measurable.

B. Research Questions

1. RQ1: How can an integrated application workflow organize input preparation, feature extraction, AI-assisted test generation, validation, and execution reporting?
2. RQ2: Which validation checks are needed to identify unsupported or unreliable content in generated test cases?
3. RQ3: How should the framework be evaluated so that generation quality, execution behavior, coverage, and hallucination mitigation are reported separately?

C. Contributions

- An implementation-grounded functional description of the Rapid TestGen AI workflow, based on the supplied interface evidence.
- A layered validation model that separates structural, semantic, requirement-traceability, and execution checks.
- Formal definitions of evaluation measures and a reproducible experiment protocol.
- A transparent results template that prevents dashboard displays or reference-paper results from being misrepresented as new experimental findings.

II. RELATED WORK

Template-based and model-based testing provide structured mechanisms for deriving tests from specifications, models, and predefined test patterns. Stocks and Carrington [2] describe test templates as a specification-oriented framework. Model-based testing subsequently developed a broad set of techniques for deriving test paths from behavioral models; Schieferdecker [3] provides an overview of this paradigm. These approaches support systematic coverage but depend on the quality and maintenance of the underlying model or template.

Research on automated test generation has also considered web-application models and domain-specific representations. Torsel [4] describes generation from a domain-specific model, while Shrestha and Agha [5] examine model-based test-case generation for web applications. These approaches demonstrate the value of representing application behavior explicitly before deriving test cases.

More recent work explores natural-language processing and generative AI for software testing. Boukhlif et al. [6] review NLP-based software testing. Helmy et al. [7] discuss AI-driven testing and autonomous systems. Alian et al. [8] present feature-driven end-to-end test generation. Karmarkar et al. [9] examine confidentiality considerations in LLM-driven test-data generation. Nettur et al. [10] introduce Cypress Copilot as an AI assistant for web-application testing. These works provide context for the use of language models and automation assistants in testing workflows.

The closest reference is Barathidass and Namburu [1]. Their method combines abstract-class-based DOM references with CRUD-specific prompt templates and a single-shot generation strategy. The article describes file crawling and reading, prompt compilation, LLM generation, test-file writing, abstract classes, and standalone service validation. It also identifies limitations involving highly volatile interfaces, initial abstraction-library investment, and configuration-driven UI assumptions. Its

evaluation uses test accuracy, time efficiency, and test coverage, and discusses selector mismatches and missing assertions.

A. Positioning of the Present Implementation

The supplied Rapid Test Gen AI show a broader application-level workflow with distinct pages for dataset management and preprocessing, feature extraction and training, generated test cases, multi-layer test validation, hallucination detection and mitigation, execution-system results, and comparative analysis. The screenshots support the existence of these interface areas, but they do not expose enough source code or execution logs to verify the internal implementation of each analytical function. Therefore, the present paper positions the system as an integrated test-generation and assessment platform and treats the detailed validation logic as a proposed formalization where implementation internals are not available.

B. Research Gap

The practical gap addressed here is the need to connect generated test artifacts with explicit validation evidence and execution outcomes in one workflow. A generated test should be traceable to an input requirement, contain only supported actions and assertions, pass structural checks, and be evaluated through controlled execution. A dashboard that displays multiple quality indicators is useful for monitoring, but a scientifically defensible evaluation additionally requires definitions, raw observations, ground truth, and repeatable procedures.

III. PROBLEM FORMULATION AND SYSTEM OBJECTIVES

A. Problem Statement

Given a set of software requirements R, application artifacts A, and an LLM-based generation service G, the objective is to produce a set of candidate test cases T that are relevant to R, structurally valid, executable in the target environment, and supported by observable application behavior. The challenge is that G may produce unsupported or inconsistent content. The framework must therefore assess each candidate before it is treated as an accepted test.

Let R = {r1, r2, …, rm} denote normalized requirements and T = {t1, t2, …, tn} denote generated tests. Each test ti contains, at minimum, a test objective, preconditions, actions, input data, expected outcomes, and assertions. The system should preserve a trace relation between each test and one or more requirements. A test is not considered correct merely because it is syntactically valid or because the LLM assigns it a high confidence score.

B. Objectives

- Provide a unified workflow for importing and organizing testing inputs.
- Prepare and extract relevant features from the available input data.
- Generate candidate test cases using a configurable generative-AI component.
- Validate the generated artifacts at multiple levels before execution.
- Identify unsupported actions, selectors, data fields, assumptions, and assertions.
- Execute eligible tests in a controlled environment and record outcomes.
- Present generation, validation, execution, and comparative information in a dashboard.

C. Scope and Assumptions

The evidence package contains screenshots rather than a source-code repository or a reproducible test dataset. Consequently, this paper does not assert a particular programming language, database, model provider, model version, prompt strategy, browser automation library, API framework, or deployment topology unless that detail is explicitly visible in the supplied evidence. The system is described functionally. The proposed evaluation protocol specifies what should be recorded when the implementation is experimentally assessed.

D. Terminology

Term	Operational meaning in this paper
Candidate test	A generated test artifact not yet accepted as valid.
Supported assertion	An assertion whose expected value or behavior is traceable to a requirement or authoritative application specification.
Hallucinated element	A generated action, selector, endpoint, field, state, or expected result unsupported by the supplied evidence.
Validation failure	A check that identifies a structural, semantic, traceability, or execution defect.
Accepted test	A candidate that passes the defined validation policy or is explicitly approved after documented human review.

IV. PROPOSED FRAMEWORK OVERVIEW

The Rapid TestGen AI implementation is presented as a staged workflow. The screenshots show an introductory project overview and workflow pages, followed by application dashboards and functional pages. At the system level, the workflow can be represented as: input acquisition → dataset management and preprocessing → feature extraction → generative test creation → multi-layer validation → hallucination analysis and mitigation → test execution → result analysis and comparison.

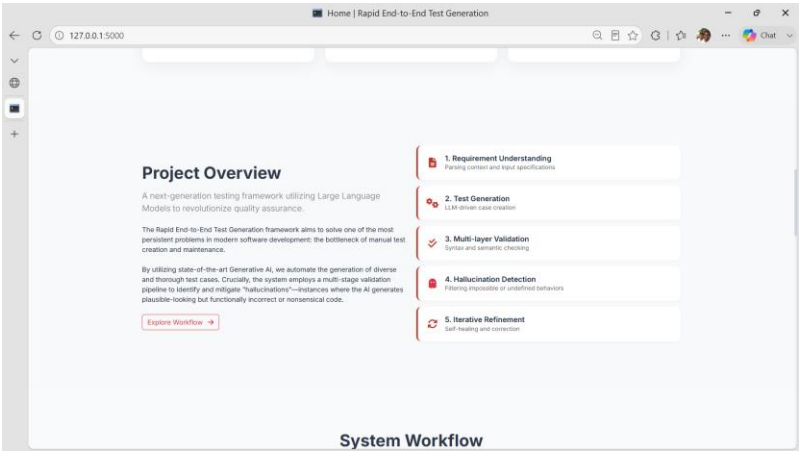


Fig. 1. Project overview screen supplied with the implementation. The image is used as evidence of the product overview interface, not as proof of measured performance.

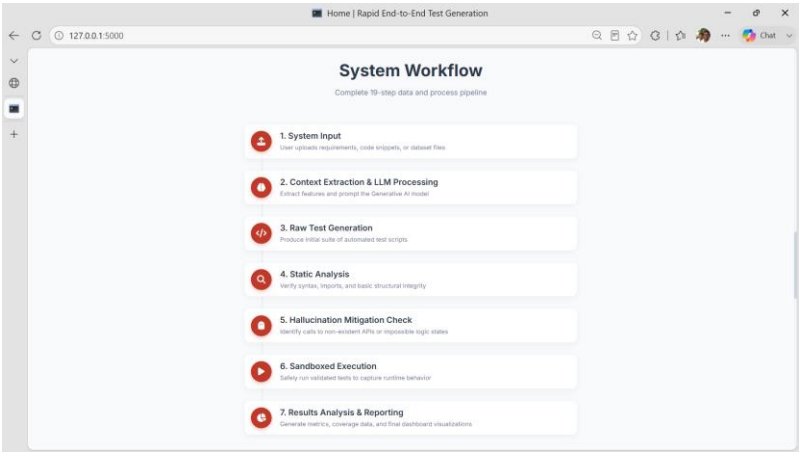


Fig. 2. Workflow screen supplied with the implementation. The visible sequence is interpreted as a user-facing workflow; internal execution details require source-code verification.

A. Functional Components

Component	Role in the proposed workflow	Evidence status
Dataset management	Import, list, and organize input records or test data.	Visible interface page.
Preprocessing	Prepare input records for downstream analysis.	Visible interface page.
Feature extraction and training	Present extracted features and a training-oriented workflow.	Visible interface page; algorithm details unspecified.
Test-case generation	Display generated test artifacts and their attributes.	Visible generated-test page.
Multi-layer validation	Present validation and verification indicators.	Visible interface page; exact checks require implementation verification.
Hallucination detection and mitigation	Present detected issues and mitigation analysis.	Visible interface page; ground truth and classifier details unspecified.
Execution results	Present execution-system results and status information.	Visible interface page; raw logs unavailable.
Comparative analysis	Compare existing and proposed system views.	Visible interface page; comparison methodology must be documented.

B. Data Flow

Each stage should produce a versioned artifact that can be traced to the preceding stage. For example, a generated test should retain the identifiers of the requirement, input record, generation request, model configuration, prompt-template version, and validation run. Execution should additionally record the test-run identifier, environment, browser or runner version, timestamps, logs, and final status. This lineage is essential for debugging and for reproducing the reported metrics.

V. DATA PREPARATION AND FEATURE REPRESENTATION

The implementation screenshots include dataset-management and preprocessing pages, together with a feature-extraction and training page. These screens indicate that input preparation is an explicit part of the platform. The source material does not include the underlying dataset schema, preprocessing code, feature definitions, or training configuration. The following specification therefore defines a reproducible data contract for the framework; it should be aligned with the actual implementation before publication.

A. Input Record Model

A normalized input record should include a unique record identifier, requirement text or structured specification, source artifact reference, application or module identifier, operation type when applicable, preconditions, expected outcomes, and provenance metadata. If source code, UI configuration, API descriptions, or DOM snapshots are used, each artifact should be stored with a version or cryptographic hash so that generation can be tied to the exact input.

Field	Description	Validation rule
record_id	Unique identifier for an input item.	Nonempty and unique within a dataset.
requirement_id	Identifier of the originating requirement.	Must resolve to a source requirement.
requirement_text	Normalized behavior description.	Must be nonempty and reviewable.
artifact_ref	Reference to code, UI, API, or configuration input.	Must resolve to a retained artifact.
operation	Operation category, where relevant.	Use a controlled vocabulary or explicitly mark unknown.
expected_behavior	Expected observable result.	Must be grounded in a requirement or specification.
version_metadata	Dataset and artifact version information.	Must be recorded for reproducibility.

B. Preprocessing

A defensible preprocessing stage should detect empty records, malformed encodings, duplicate requirements, missing identifiers, inconsistent operation labels, and contradictory expected outcomes. It should preserve the original input and write transformations to a separate normalized representation. Deduplication must not silently remove distinct requirements that happen to share similar wording. Any excluded record should carry an exclusion reason.

C. Feature Extraction

Feature extraction can convert unstructured or semi-structured input into a representation useful for test generation. Candidate feature groups include entities and fields, user-interface controls, navigation transitions, operation categories, constraints, validation rules, preconditions, expected state changes, and observable outputs. The exact feature extractor used by the implementation is not identified in the available evidence; no particular NLP model, embedding method, or classifier is claimed here.

D. Dataset Governance

The evaluation dataset should be partitioned by application, feature, or requirement group rather than randomly splitting near-duplicate test cases across training and test partitions. Sensitive credentials, personal information, production secrets, and customer data should be removed or masked. Dataset lineage, annotation instructions, label definitions, and reviewer agreement should be retained.

VI. GENERATIVE TEST-CASE CREATION

The test-generation stage transforms normalized requirements and application context into candidate test artifacts. The implementation includes a generated-test-case view. The precise LLM, model parameters, prompting approach, output schema, and retry policy are not verifiable from the screenshots. This section therefore specifies a model-agnostic generation contract rather than claiming a specific model implementation.

A. Structured Test Representation

Each candidate test should be represented in a machine-readable structure containing: (i) test identifier; (ii) requirement identifier; (iii) test objective; (iv) preconditions; (v) ordered actions; (vi) test data; (vii) expected results; (viii) assertions; (ix) dependencies; and (x) provenance metadata. A structured representation is preferable to free-form prose because downstream validation and execution require explicit fields.

Element	Illustrative content (schema only)
Objective	Verify that a permitted user can create a record.
Precondition	User is authenticated and the relevant page is available.
Action sequence	Open form; enter specified fields; submit.
Test data	Values drawn from an approved synthetic test-data set.
Expected result	A record is displayed with the submitted values.
Assertions	Visible confirmation and persisted field values match the requirement.
Traceability	Requirement ID and source artifact version.

B. Generation Constraints

- The generator must not invent selectors, endpoints, fields, permissions, or expected values when they are absent from the source context.
- Every test must reference at least one requirement or be explicitly marked as exploratory.
- Expected results must be derived from authoritative requirements, specifications, or approved oracles.
- The output must conform to a declared schema; malformed output should be rejected or sent for repair.
- The system should record model name, model version, prompt version, decoding parameters, input references, and generation timestamp.

C. Prompt and Context Design

A generation request should provide the model with the target requirement, relevant application context, allowed actions, output schema, and explicit prohibitions against unsupported assumptions. Where examples are used, they should be reviewed and versioned. A model should be instructed to return an uncertainty marker or request missing information rather than fabricate a test precondition or expected result.

D. Human Oversight

Generated tests should remain candidate artifacts until validation is complete. Human reviewers should be able to inspect the source requirement, generated steps, assertions, validation findings, and execution evidence together. Any manual edit should be recorded as a separate revision so that original generation quality can be distinguished from post-edit quality.

VII. MULTI-LAYER VALIDATION AND HALLUCINATION MITIGATION

The implementation contains separate screens for multi-layer test validation and hallucination detection and mitigation. This separation is important because a test can be syntactically correct yet semantically

unsupported, or semantically plausible yet non-executable. The following validation model formalizes the checks that should be reported independently.

A. Layer 1: Structural and Schema Validation

Check that the generated artifact conforms to the declared schema, contains mandatory fields, uses valid data types, and has a well-formed action sequence. A failure at this layer indicates malformed output, not necessarily a hallucination.

B. Layer 2: Requirement Traceability

Map every generated action and assertion to one or more source requirements. A step without a traceable source should be flagged as unsupported or exploratory. Traceability should be reviewed at the level of individual actions and assertions rather than only at the test-case level.

C. Layer 3: Semantic and Context Validation

Check whether referenced entities, controls, routes, API operations, fields, roles, and state transitions are present in the supplied application context. For UI tests, a locator should be resolved against the versioned interface artifact or verified during controlled execution. A direct text similarity score alone is not sufficient evidence that an assertion is correct.

D. Layer 4: Assertion and Oracle Validation

Assertions should be checked against expected behavior defined by the requirement or a trusted test oracle. The framework should identify missing assertions, contradictory assertions, assertions with unsupported values, and checks that do not verify the stated objective.

E. Layer 5: Static and Runtime Validation

Static checks should inspect syntax, imports, unresolved variables, unsupported commands, and unsafe operations. Runtime checks should execute approved tests in an isolated environment and record pass, fail, skipped, timeout, and infrastructure-error states separately. A runtime pass is not sufficient proof of semantic correctness; a test may pass while asserting the wrong behavior.

F. Hallucination Taxonomy

Category	Example of unsupported output	Detection evidence
Invented UI element	A button or field absent from the UI specification.	Artifact/DOM resolution.
Invented selector	A locator that does not resolve in the target version.	Selector lookup and runtime trace.
Invented API detail	An endpoint or request field not in the API contract.	API specification comparison.
Unsupported assertion	Expected value not grounded in a requirement.	Assertion-to-requirement trace.
Invalid workflow	Action sequence violates documented state transitions.	Model/state-transition validation.
Fabricated test data	Values violate constraints or are not approved.	Schema and boundary-rule checks.

III. FORMAL MODEL AND ALGORITHM

Let the input context be $X = (R, A, P)$, where R is the set of requirements, A is the set of application artifacts, and P is the set of approved policies and constraints. The generation component G produces a candidate test $t = G(X, \theta)$, where θ represents the recorded model and prompt configuration. The validation function V evaluates the candidate against the input context and returns a vector of checks rather than a single opaque score.

Define $V(t, X) = (s, q, c, a, e)$, where s denotes structural validity, q denotes requirement traceability, c denotes contextual consistency, a denotes assertion support, and e denotes execution status. The first four components are validation outcomes; e is populated only after controlled execution. Each component should retain evidence and a reason code.

A. Acceptance Policy

A test may be accepted automatically only when all mandatory checks pass and the execution policy permits it. A human-approved exception must be recorded with reviewer identity, rationale, and timestamp. The acceptance decision can be expressed as:

$$\mathbf{Accept(t) = S(t) \wedge Q(t) \wedge C(t) \wedge A(t) \wedge E_allowed(t)}$$

Here, E_allowed(t) indicates that execution is either successful or not required under the documented policy. A test that is structurally valid but lacks traceability should not be accepted as a requirement-based regression test.

B. Proposed Algorithm

```
Algorithm 1: Human-supervised test generation and validation
Input: requirement set R, application artifacts A, policy set P
Output: validated test set T_valid and review queue T_review

X ← normalize_and_version(R, A, P)
F ← extract_features(X)
T_candidate ← generate_tests(F, recorded_model_config)
for each test t in T_candidate do
    if not schema_valid(t) then
        record(t, STRUCTURE_FAILURE); add t to T_review; continue
    end if
    trace ← map_actions_and_assertions_to_requirements(t, R)
    context ← verify_entities_locators_and_transitions(t, A)
    oracle ← verify_assertions_against_expected_behavior(t, R, P)
    static ← run_static_checks(t)
    findings ← aggregate(trace, context, oracle, static)
    if mandatory_checks_pass(findings) then
        result ← execute_in_isolated_environment(t)
        record_execution_evidence(t, result)
        if execution_policy_passes(result) then
            add t to T_valid
        else
            add t to T_review
        end if
    else
        add t to T_review
    end if
end for
return T_valid, T_review
```

C. Hallucination Incidence

For an annotated evaluation set, let H be the number of generated tests containing at least one confirmed unsupported element and N be the total number of evaluated generated tests. The test-level hallucination incidence is HI = H/N. A complementary element-level rate can be calculated as the number of confirmed unsupported actions or assertions divided by the total number of reviewed actions or assertions. Both rates require human-reviewed labels or an independently validated oracle.

IX. IMPLEMENTATION DESCRIPTION AND INTERFACE EVIDENCE

The paper avoids naming unverified frameworks or asserting that each displayed indicator is backed by a validated algorithm.

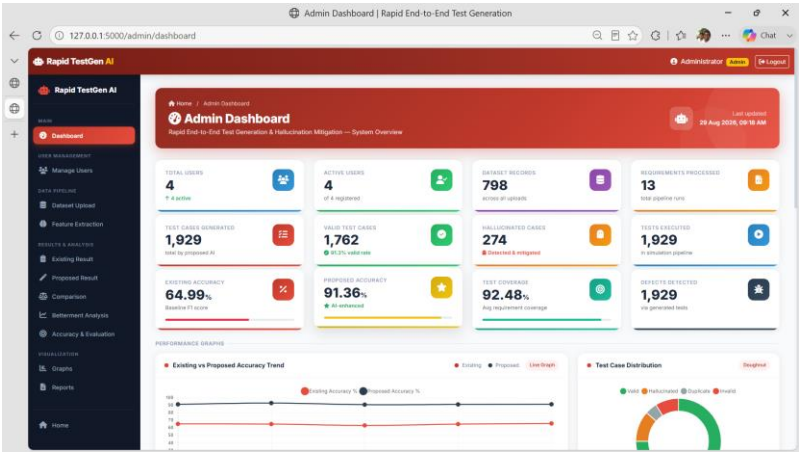


Fig. 3. Administrative dashboard screenshot. It evidences a dashboard view with summary cards and visual analytics; displayed values are not adopted as experimental findings in this manuscript.

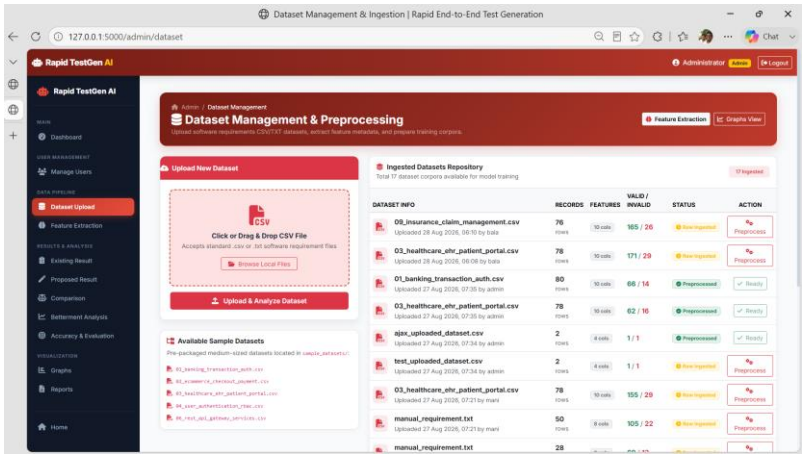


Fig. 4. Dataset management and preprocessing screen. The interface displays dataset-oriented controls and records; underlying preprocessing operations require source-code verification

A. Feature Extraction and Training Interface

A dedicated feature-extraction and training screen is present in the implementation evidence. This supports the existence of a feature-oriented stage in the user workflow. The available screenshots do not establish the feature vector, extraction algorithm, training labels, model architecture, optimization procedure, or train/validation/test split. These details must be added from the implementation source before submission.

B. Generated Test-Case Interface

A generated-test page presents a list/table-oriented view of test artifacts. For a reproducible implementation, each displayed row should be exportable with its test identifier, originating requirement, generated steps, assertions, validation status, and version metadata. If the application currently stores only summary fields, the manuscript should be updated to describe the actual stored schema.

C. Validation, Hallucination, and Execution Interfaces

Separate screens are visible for multi-layer validation, hallucination detection and mitigation, execution-system results, and comparative analysis. These pages provide evidence that the application is designed to expose multiple quality and execution views. The screenshots alone cannot verify the formula behind each indicator, the classifier or rules used for hallucination detection, the source of the ground truth, or whether execution was performed in an isolated environment.

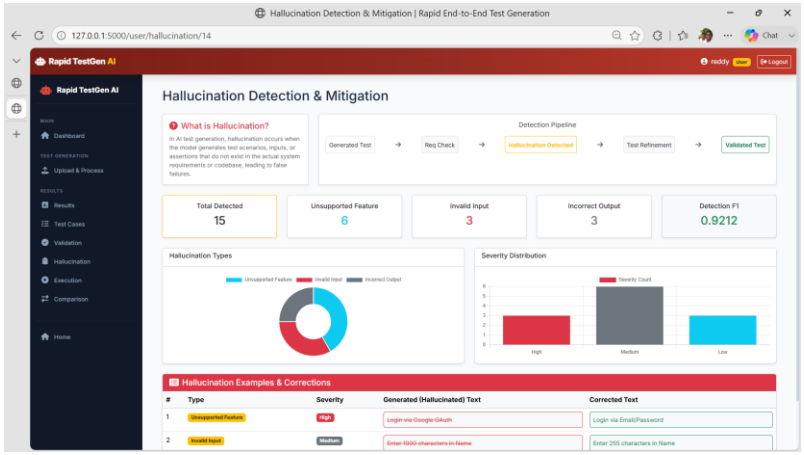


Fig. 5. Hallucination detection and mitigation interface. The screen is included as implementation evidence; its displayed counts and rates are not treated as validated experimental measurements

X. EXPERIMENTAL DESIGN AND EVALUATION PROTOCOL

A rigorous evaluation should compare the proposed workflow with a documented baseline under identical input requirements and execution conditions. The supplied package does not contain the raw experimental dataset, baseline test suite, run logs, annotation labels, model configuration, or a complete result export. Therefore, this section defines the experiment needed to substantiate the framework rather than reporting unsupported results.

A. Research Dataset

The dataset should contain a representative collection of application requirements and associated source artifacts. It should include normal workflows, boundary conditions, invalid inputs, authorization cases, error handling, and regression scenarios. Each requirement should be independently reviewed and labeled with the expected actions and assertions. Dataset size, application count, source types, inclusion criteria, exclusion criteria, and annotation procedure must be reported.

B. Baselines

- Manual test authoring by qualified test engineers using the same requirements.
- A conventional template- or rule-based generator, if available and configured for the same application.
- An LLM-only generation condition without the proposed validation stages, to isolate the effect of validation.
- The complete Rapid TestGen AI workflow.

Comparisons should use the same requirement set, application version, test environment, and execution policy. If a baseline cannot be implemented, the paper should state that the comparison is descriptive rather than experimental.

C. Evaluation Metrics

Metric	Definition	Required evidence
Schema validity	Valid generated artifacts / all generated artifacts.	Raw outputs and schema validator logs.
Traceability rate	Actions/assertions mapped to source requirements / all reviewed actions/assertions.	Requirement-to-test annotation.
Execution success rate	Tests completing with expected runner status / tests attempted.	Runner logs and failure classification.
Functional coverage	Covered requirement behaviors / total in-scope behaviors.	Coverage matrix and test-to-requirement map.
Hallucination incidence	Tests with at least one confirmed unsupported element / tests reviewed.	Independent human labels or verified oracle.
Human correction effort	Review and repair time per accepted test or requirement.	Time logs and edit history.
Generation latency	Elapsed time from request submission to candidate output.	Timestamped generation records.

D. Experimental Controls

Record model and prompt versions, temperature and other decoding parameters, number of retries, token usage if available, application version, runner and browser versions, hardware or service configuration, and concurrency settings. Repeat stochastic generation runs and report mean, dispersion, and confidence intervals where appropriate. Separate model failures, application defects, test-oracle defects, and infrastructure failures

XI. RESULTS STATUS, DISCUSSION, AND THREATS TO VALIDITY

A. Results Supported by the Supplied Evidence

The supplied screenshots support the following implementation-level observations: (1) the project has an overview and workflow presentation; (2) the application includes administrative and user-facing dashboard views; (3) dataset management and preprocessing are represented as separate interface areas; (4) a feature-extraction and training interface is present; (5) generated test cases are displayed in a dedicated view; and (6) distinct pages are provided for multi-layer validation, hallucination detection and mitigation, execution-system results, and comparative analysis. These are observations about the interface and its stated workflow.

The screenshots also contain numerical cards, charts, percentages, and comparison graphics. However, no underlying CSV/JSON exports, execution logs, ground-truth labels, model configuration, or metric calculation code were included in the evidence package. These displayed values cannot be

independently recomputed. They are therefore not reproduced as experimental results, and no accuracy, coverage, time-saving, hallucination-reduction, or comparative-superiority claim is made in this manuscript.

B. Results Table to Complete After Reproducible Runs

Metric	Baseline	Rapid TestGen AI	Evidence / notes
Schema validity (%)	Not supplied	Not supplied	Populate from validator output.
Traceability (%)	Not supplied	Not supplied	Populate from reviewed mapping.
Execution success (%)	Not supplied	Not supplied	Attach raw runner logs.
Functional coverage (%)	Not supplied	Not supplied	Attach requirement coverage matrix.
Hallucination incidence (%)	Not supplied	Not supplied	Requires independent annotation.
Median generation latency (s)	Not supplied	Not supplied	Use timestamped repeated runs.
Human correction time (min/test)	Not supplied	Not supplied	Use reviewer time logs.

C. Discussion

The principal value of the implementation, as evidenced by its interface, is workflow integration: data preparation, generation, validation, hallucination analysis, execution, and comparison are surfaced in one application. This can improve operational visibility and make the testing lifecycle easier to inspect. The research value will depend on whether the platform preserves traceability between requirements, generated tests, validation findings, and execution records. A dashboard is an observability mechanism; it is not a substitute for an evaluation protocol.

D. Threats to Validity

- Construct validity: dashboard labels may not map directly to standard definitions of accuracy, coverage, or hallucination.
- Internal validity: without raw logs and controlled baselines, causal claims about the validation stages cannot be established.
- External validity: the screenshots do not reveal the diversity of applications, domains, or UI architectures tested.
- Reproducibility: model versions, prompts, dataset versions, and runtime configuration are not available in the supplied evidence.
- Annotation validity: hallucination labels require a clear policy and independent review to avoid subjective classification.

XII. LIMITATIONS AND FUTURE WORK

A. Current Limitations

The current manuscript is limited by the evidence supplied with the project. The implementation document is primarily visual and contains no readable source-code description. The exact technology stack, model identity, model parameters, feature-extraction method, validation rules, hallucination-detection mechanism, execution runner, data schema, and storage architecture remain unspecified. In addition, the available screenshots do not provide raw experimental observations or reproducible metric calculations.

The framework description also assumes that input requirements and application artifacts can be made available in a machine-readable or reviewable form. Incomplete requirements, unstable interfaces, inaccessible test environments, and unreliable test oracles may still lead to invalid tests. Human review remains necessary for ambiguous business rules and safety-critical workflows.

B. Future Research Directions

- Implement requirement-to-action and assertion-level traceability with auditable provenance.
- Introduce a benchmark dataset with expert-reviewed test cases and hallucination labels.
- Evaluate the contribution of each validation layer through ablation studies.
- Compare LLM-only generation, validation-assisted generation, and manual authoring under controlled conditions.
- Add robust handling of dynamic interfaces, role-based access, API workflows, and multi-step state transitions.

- Measure cost, latency, token consumption, correction effort, and environmental impact alongside test quality.
- Provide exportable experiment manifests, raw execution logs, metric scripts, and anonymized datasets to support replication

XIII. CONCLUSION

This paper has presented an implementation-grounded description of Rapid TestGen AI, an application intended to support automated end-to-end test generation and quality assessment using generative AI. The supplied screenshots show a connected workflow covering project overview, dataset management and preprocessing, feature extraction and training, generated test cases, multi-layer validation, hallucination detection and mitigation, execution results, and comparative analysis. Based on this evidence, the work contributes a structured system description, a multi-layer validation model, a formal acceptance policy, and a reproducible evaluation protocol. The available evidence is insufficient to establish numerical performance or to verify the internal algorithms behind the displayed indicators. Accordingly, this manuscript intentionally reports no new experimental scores and does not reuse the quantitative findings of the related IEEE Access article as results of the present implementation. The next step toward journal submission is to attach the implementation source details, dataset description, model and prompt configuration, independent labels, and raw execution records, then run the evaluation protocol and populate the results tables. With those additions, the system can be assessed transparently for test quality, execution reliability, functional coverage, hallucination incidence, and human-effort reduction.

REFERENCES

[1] C. Barathidass and K. Namburu, “Rapid end-to-end test generation and hallucination mitigation using generative artificial intelligence,” IEEE Access, vol. 14, pp. 20605–20619, 2026, doi: 10.1109/ACCESS.2026.3657407.

[2] P. A. Stocks and D. A. Carrington, “Test templates: A specification-based testing framework,” in Proc. 15th Int. Conf. Software Engineering, Baltimore, MD, USA, 1993, pp. 405–414, doi: 10.1109/ICSE.1993.346025.

[3] I. Schieferdecker, “Model-based testing,” IEEE Software, vol. 29, no. 1, pp. 14–18, Jan. 2012, doi: 10.1109/MS.2012.13.

[4] A.-M. Torsel, “Automated test case generation for web applications from a domain specific model,” in Proc. IEEE 35th Annual Computer Software and Applications Conference Workshops, Munich, Germany, 2011, pp. 137–142, doi: 10.1109/COMPSACW.2011.32.

[5] N. Shrestha and G. Agha, “Model-based test case generation for web applications using UML models,” in Proc. IEEE 10th Int. Conf. Software Testing, Verification and Validation Workshops, Tokyo, Japan, 2017, pp. 48–55, doi: 10.1109/ICSTW.2017.15.

[6] M. Boukhlif, M. Hanine, N. Kharmoum, A. R. Noriega, D. G. Obeso, and I. Ashraf, “Natural language processing-based software testing: A systematic literature review,” IEEE Access, vol. 12, pp. 79383–79400, 2024, doi: 10.1109/ACCESS.2024.3407753.

[7] M. Helmy, O. Sobhy, and F. ElHusseiny, “AI-driven testing: Unleashing autonomous systems for superior software quality using generative AI,” in Proc. Int. Telecommunication Conference (ITC-Egypt), Cairo, Egypt, 2024, pp. 1–6, doi: 10.1109/ITC-Egypt61547.2024.10620598.

[8] P. Alian, N. Nashid, M. Shahbandeh, T. Shabani, and A. Mesbah, “Feature-driven end-to-end test generation,” arXiv:2408.01894, 2024.

[9] H. Karmarkar, S. Agrawal, A. Chauhan, and P. Shete, “Navigating confidentiality in test automation: A case study in LLM driven test data generation,” in Proc. IEEE Int. Conf. Software Analysis, Evolution and Reengineering (SANER), Rovaniemi, Finland, 2024, pp. 337–348, doi: 10.1109/SANER60148.2024.00041.

[10] S. B. Nettur, S. Karpurapu, U. Nettur, and L. S. Gajja, "Cypress copilot: Development of an AI assistant for boosting productivity and transforming web application testing," IEEE Access, vol. 13, pp. 3215–3229, 2025, doi: 10.1109/ACCESS.2024.3521407.