A CONCEPTUAL FRAMEWORK FOR QUALITY CONTROL AND AUTOMATED TESTING OF LARGE LANGUAGE MODELS

Stelian Yolov, Vladimir Valkanov

Abstract. *As large-language models (LLMs) are increasingly made use of across critical applications, efficient quality control and scalable and adaptive testing become essential to ensure reliability, fairness, safety and performance.*

Traditional manual testing is unable to catch up with the constantly evolving nature of LLMs. This article proposes a conceptual automation testing framework designed specifically for LLM quality assurance.

We shape the architecture that integrates (i) specification of quality measurements (e.g. factual accuracy, coherence, bias mitigation, latency, safety), (ii) using pre-defined patterns, test examples, and templates, creating a broad set of test prompts by automatically filing in templates and including hard-to-handle examples, (iii) evaluation of the LLMs response in controlled environment (pipelines), (iv) automated quality assessments enhanced by meta-analysis with CI/CD pipelines.

By connecting these layers into an integrated conceptual architecture, the framework offers a base for scalable, transparent, and reproducible quality control of LLMs. This conceptualization aims to link the theoretical AI evaluation and practical automation strategies and open the way towards standardized methodologies for model verification, clarity, and reliability.

Key words: LLM, Automation Framework, Quality Control, AI, Testing.

Introduction

Transformer-based LLMs now match or exceed human performance in tasks such as summarization, reasoning, and code generation. As these systems are deployed in safety-critical environments, their failure modes require careful scrutiny. Foundational analyses highlight the risks, limitations, and emergent behaviors of large-scale models [1, 4].

Traditional software testing approaches struggle due to:

- **Frequent updates:** Model fine-tuning and architectural changes require continuous regression testing;
- **Non-deterministic outputs:** LLMs generate probabilistic responses based on sampling strategies;
- **Emergent behaviors:** Unpredictable biases or unsafe outputs may arise [3];
- **Code-generation risk:** LLM-generated programs may compile successfully yet contain semantic errors or vulnerabilities [9].

To address these challenges, this article proposes a conceptual testing architecture that unifies specification, automated prompt generation, controlled evaluation, and regression tracking supporting both natural language and code generation tasks.

Background and Related Work

1. LLM Evaluation Challenges

Benchmarks such as MMLU [2], and multidimensional frameworks like HELM [3] provide broad task coverage, but do not fully address durability, continuous integration needs, or code-generation workflows. Additionally, safety, factuality, and bias dimensions also require specialized datasets and red-teaming. Existing automated evaluation tools often lack the option of integration into continuous development pipelines.

2. Code Generation by LLMs

LLMs are now widely used to generate programming code, unit tests, and even bug-fixing patches. Factual studies revealed that while LLMs can generate syntactically correct tests and code, they often struggle with semantic correctness and coverage:

- Schäfer et al. evaluates the automated unit test generation using LLMs, achieving good code coverage for JavaScript functions (∼70% statement coverage) [5];
- Introducing MuTAP, combining LLM test generation with mutation testing to improve the bug detection capability of generated tests [6];
- Using LLMs for generating the inputs for fuzzing various programming languages, showing that LLM-based fuzzers can find more bugs than traditional, language-specific fuzzers [7];
- Meta's TestGen-LLM works at an industrial scale to improve existing human-written test suites by generating additional test cases, increas-

ing the code coverage, and passing reliability filters [8].

Although effective for specific tasks, these approaches do not offer a unified testing framework that reaches both language and programming capabilities.

Conceptual Framework

The proposed architecture consists of three main components:

1. Quality Specification Module
2. Automated Test-Input Generator
3. Controlled Evaluation and Feedback Pipeline

Together, these form a closed-loop workflow for comprehensive LLM testing.

Stage I: Quality Specification Module

Quality attributes must be defined explicitly for each application. Relevant dimensions include factual accuracy, coherence, safety, and bias mitigation [4], as well as code correctness, security, performance, and robustness.

Formal Specification

Using a specification language or schema (e.g., JSON or a domain-specific language), one can express.

- Acceptable error rates (e.g., maximum factual error rate).
- Rule-based safety constraints (e.g., no insecure APIs, no hard-coded credentials).
- Performance thresholds (e.g., runtime, memory).
- Structural requirements (e.g., cyclomatic complexity, modular style).
- Regression budgets (e.g., acceptable drop in test coverage or security metrics across the model versions).

These formalized specifications enable automated execution by the evaluation system.

Stage II: Automated Test-Input Generation

1. Natural-Language Prompt Templates

Prompts are generated using structured templates, and adversarial examples informed by large-scale evaluation work [3].

- **Predefined prompt formats.** These include the common structures such as Q&A tasks, multi-turn dialogues, and chain-of-thoughts requests. Each format uses a specific capability, while the chain-of-thoughts prompts assess the model's ability to produce coherent reasoning steps.

- **Slot-filling variations.** Templates contain placeholders for entities, contexts, or styles that can be automatically replaced with different values. This creates many controlled variations from single patterns and helps measure whether model performance remains stable across domains and wording changes.

2. Code-Generation Prompt Templates

Code-related prompts may ask the model to assess code-generation capabilities, the framework employs a set of structured prompts targeting various programming tasks. These includes requests to implement functions based on a given signature or docstring, to generate unit tests for existing code, or repair faulty code snippet by fixing errors [5]. Additional templates address optimization – by prompting the model to improve runtime or memory usage, and security by requiring enhancements against issues such as SQL injection or cross-site scripting (XSS).

Together, these templates provide broad coverage of functionality, correctness, efficiency, and security dimensions in generated code.

3. Synthetic and Knowledge-Driven Prompts

Beyond predefined templates, the system generates additional prompts using automated techniques thereby widening test coverage. Knowledge-based generation draws on domain-specific resources, such as APIs or libraries, to create realistic, context-aware tasks. Program-mutation methods introduce controlled bugs or variations in existing code allowing the model's ability to detect and correct errors to be tested [6]. Fuzzing-inspired approaches generate small, often randomized input variations to probe robustness [7]. Adversarial generation creates prompts intentionally crafted to trigger model weaknesses, such as logical errors, unsafe outputs, or insecure code patterns.

Stage III: Controlled Evaluation and Feedback Pipeline

1. Controlled Execution Environment

For natural-language tasks, decoding parameters remain fixed. For code-generation tasks, outputs run in sandboxed environments with controlled resources to ensure safety.

2. Automated Scoring Mechanisms

- **Semantic similarity metrics**. Compare the model's output to reference answers to assess how closely its meaning aligns with expected responses.

- **Instruction-following checks**. Used to verify whether the model accurately follows tasks, requirements, constraints, and formatting rules

- **Safety and toxicity indicators**. Detect harmful, biased, or inappropriate content to ensure outputs meet safety standards [4].

- **LLM-as-judge scoring**. Separate model scores responses for correctness, clarity, and completeness, providing a more nuanced assessment than rule-based metrics.

Code evaluation includes:

- **Compilation and syntax analysis**. Ensuring that the generated code is well-formed and executes without errors.

- **Unit-test execution and coverage measurement**. The output is tested against reference or automatically generated test suites to assess correctness and completeness [5].

- **Mutation-based robustness evaluation**. Using techniques such as MuTAP to expose hidden bugs, weak test coverage or brittle logic [6].

- **Static and dynamic vulnerability scanning**. Applying static or dynamic scanners to detect vulnerabilities such as injection risks, unsafe API usage, or insecure coding patterns.

- **Performance profiling**. Measuring runtime behavior, including execution time, memory consumption, and algorithmic efficiency.

- **Regression comparison across model versions**. Evaluating differences across model versions to identify quality improvements or newly introduced failures.

3. Feedback Loop

The system supports iterative correction by re-prompting based on error traces, mutation signals [6], and fuzzing outcomes [7].

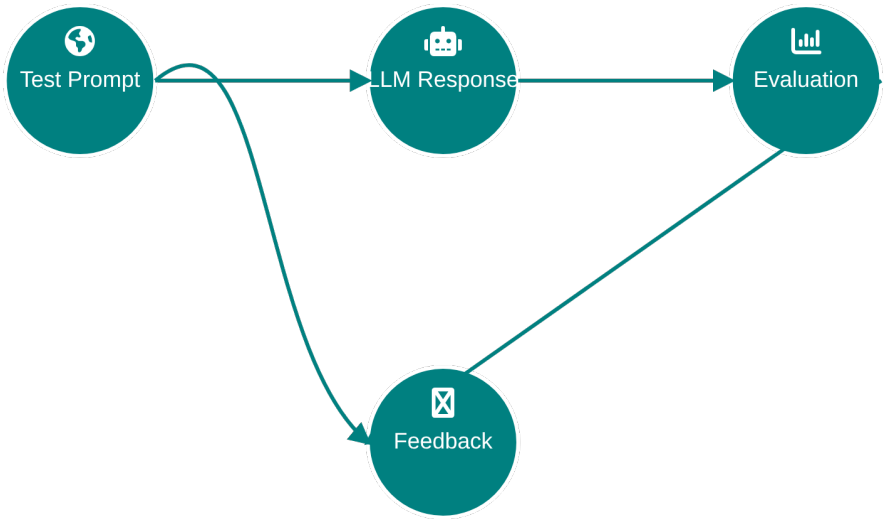

Figure 1. Feedback Loop & Iteration

The flow diagram illustrates this continuous cycle. Each test prompt generates an LLM response, which undergoes evaluation, producing feedback that influences the next generation of test prompts.

System Architecture

The overall architecture incorporates:

1. **Specification Module:** holds formal definitions of quality metrics.
2. **Prompt Generator:** generates natural language prompts and code-related tasks.
3. **Execution Engine:** runs LLM, captures outputs under fixed conditions.
4. **Evaluation Module:** scores outputs using the mechanisms described above.
5. **Feedback Controller:** iteratively refines prompts based on failure or performance.
6. **Storage & Analytics:** logs, prompts, responses, code, metrics, test histories.
7. **Dashboard / CI Integration:** integrates into continuous integration (CI) pipelines for regression tracking, alerts, and reports.

Figure 2 illustrates the implementation of the three stages as a complete system. Architecture employs a modular design, enabling components to be updated or replaced as testing methodologies evolve.

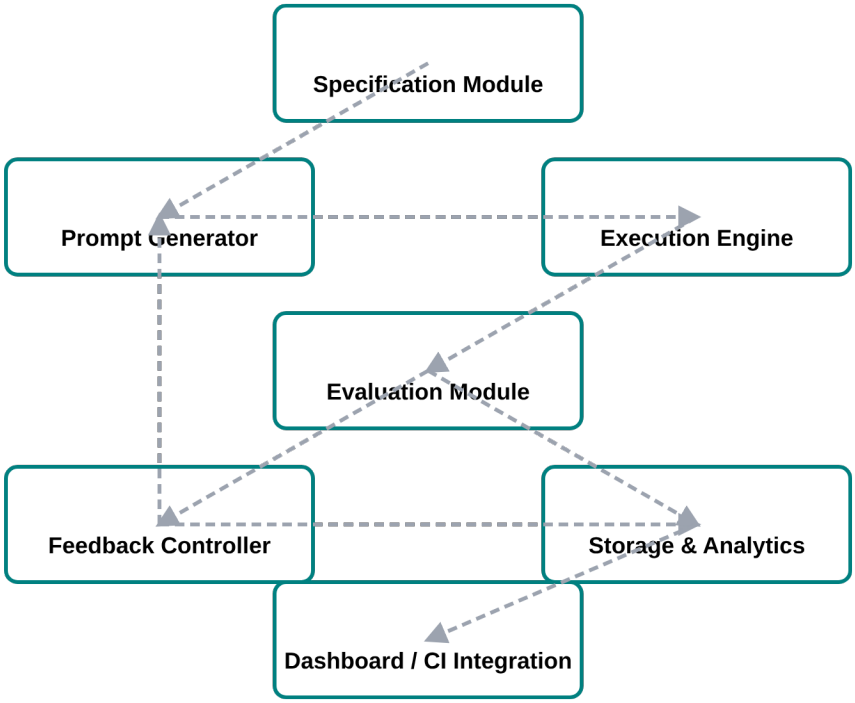

Figure 2. System architecture

Evaluation Metrics

The framework utilizes the following evaluation metrics.

Natural Language

Natural language metrics include semantic similarity scores and count of safety violations, which together indicate the quality, consistency, and risk level of model outputs. Correctness is measured by factual accuracy and adherence to instructions, ensuring responses are both accurate and usable. Robustness is evaluated by assessing model performance under adversarial prompts, which helps identify weaknesses in reasoning or safety behavior.

Code

For code-related tasks, syntactic correctness is assessed through compilation success rate, while functional correctness is measured using unit-test pass rates and coverage across lines and branches. Code robustness is evaluated using mutation scores and fuzzing-based bug detection, which highlight fragile or error-prone logic. Security metrics track the number and severity of detected vulnerabilities, ensuring generated code avoids issues such as injection flaws or unsafe API usage. Performance profiling records runtime and memory usage to evaluate efficiency. Finally, regression metrics compare results across model versions to track improvements or detect new failures introduced over time.

Applications and Use Cases

This testing framework supports a range of practical applications across both industry and research. In LLM-as-a-programmer scenarios, it can be used to validate the reliability and safety of code assistants such as Copilot or specialized code-focused LLMs, prior to deployment in enterprise environments. The framework also enables LLM-assisted testing workflows, where models generate unit tests, integration tests, or fuzzing inputs to improve the quality of legacy or complex software systems.

In safety-critical domains, the framework helps ensure that automatically generated code does not introduce security vulnerabilities or unstable behavior. Within continuous deployment pipelines, it integrates naturally into MLOps [8] processes, enabling ongoing regression tracking and quality monitoring as models evolve. Finally, in academic research, the framework provides a structured way to benchmark new LLM architectures, prompting methods, or evaluation strategies across both natural-language tasks and program synthesis.

Analysis

Strengths

The framework offers several notable advantages. Scalability is achieved through automated prompt generation and evaluation, which significantly reduces the need for manual oversight. Its comprehensive scope enables simultaneous assessment of natural-language responses and code-generation quality, providing a unified view of model performance. Reproducibility is ensured by conducting evaluations in controlled environments and using formally defined specifications, supporting consistent comparisons across model versions. The inclusion of security testing and adversarial prompt generation enhances overall safety and robustness, enabling proactive detection of high-risk behaviors. The framework aligns with CI/CD workflows, supporting continuous monitoring, alerting, and regression analysis as models evolve.

Limitations

Despite these benefits, several challenges remain. The automated oracle problem complicates the definition of clear ground truth for open-ended language tasks, and reliance on humans or LLM-as-judge systems may introduce bias. Resource costs are significant, as code execution, fuzzing, and mutation testing require substantial compute time. The framework must address prompt design debt, since creating and maintaining meaningful adversarial tests requires ongoing domain expertise. There is also risk of overfitting, where models

perform well within the framework without improvement in real-world behavior. Security blind spots persist, as static or dynamic analysis tools may fail to detect subtle vulnerabilities, and generated tests may overlook rare or sophisticated exploit paths.

Future Work

Potential future expansions include adaptive adversarial generation using reinforcement learning, meta-evaluation of the framework's effectiveness, and the incorporation of a human-in-the-loop component for high-risk assessment. Additional development may involve standardized cross-model test suites, and interpretability tools for understanding model failures.

Conclusion

The article presents a conceptual framework for automated evaluation of large language models that unify natural-language assessment with comprehensive testing of LLM-generated code. By incorporating formal specifications, diverse prompt generation, controlled execution, and multi-layer scoring mechanisms, the framework establishes a replicable foundation for ensuring LLM reliability. As language models become increasingly integrated into software engineering and other critical domains, comprehensive testing architectures of this nature will be essential.

Acknowledgments

This study is supported by the project FP25-FMI-010 "Innovative interdisciplinary research in informatics, mathematics and educational pedagogy" at the Paisii Hilendarski University of Plovdiv.

References

[1] R. Bommasani et al., *On the Opportunities and Risks of Foundation Models*, 2021

[2] D. Hendrycks et al., *Measuring Massive Multitask Language Understanding (MMLU)*, 2021

[3] P. Liang et al., *Holistic Evaluation of Language Models (HELM)*, 2022

[4] E. Bender, T. Gebru, *On the Dangers of Stochastic Parrots*, 2021

[5] M. Schäfer et al., *TestPilot: Automated Unit Test Generation with LLMs*, 2023

[6] A. Dakhel et al., *MuTAP: Mutation Testing-Aided Prompting*, 2023

[7] X. Zheng et al, *Fuzz4All: Universal Fuzzing with LLMs*, 2023

[8] Y. Tian et al., *Test Generation Using LLMs at Meta Scal*e, 2024
[9] M. Chen et al., *Evaluating Large Language Models Trained on Code*, 2021
[10] J. Kaplan et al., *Scaling Laws for Neural Language Models*, 2020

Stelian Yolov[1], Vladimir Valkanov[1]
[1] Paisii Hilendarski University of Plovdiv,
Faculty of Mathematics and Informatics,
236 Bulgaria Blvd., 4027 Plovdiv, Bulgaria
Corresponding author: `syolov@uni-plovdiv.bg`