ARTeMIS: Agent-based Rewriting and Test Case Management with Intelligent Supervision

Gabriel Carvalho
Federal University of Amazonas
Manaus, Brazil
gabriel.pacheco@icomp.ufam.edu.br

Andrés D. Peralta
Federal University of Amazonas
Manaus, Brazil
andres@icomp.ufam.edu.br

André Carvalho
Federal University of Amazonas
Manaus, Brazil
andre@ufam.edu.br

Yan Soares
Federal University of Amazonas
Manaus, Brazil
yan.soares@icomp.ufam.edu.br

Igor Lima
Federal University of Amazonas
Manaus, Brazil
igor.lima@icomp.ufam.edu.br

Nikson Ferreira
Nokia Institute of Technology
Brasilia, Brazil
nikson.fernandes@indt.org.br

Hallyson Melo
Nokia Institute of Technology
Recife, -, Brazil
hallyson.melo@indt.org.br

ABSTRACT

The quality of Test Case (TC) scripts is essential for the execution and automation of test scenarios on mobile devices. It is common to find TC components that are out of date, poorly written or inconsistent with documentation standards. In this work, we propose ARTeMIS, a modular Multi-Agent framework designed to automate the rewriting and validation of non-standardized TC components, integrating semantic retrieval, supervised classification, structured prompting and iterative rule-based validation using Large Language Models (LLMs). Our framework is capable of standardizing these three TC components: Summary, Initial Setup and Test Steps. ARTeMIS assigns each component to specialized agents for classification, rewriting and validation, enabling syntactic consistency and semantic accuracy with minimal human intervention. We performed experiments using, in addition to ARTeMIS, three other LLM-based techniques well established in the literature: Zero-Shot, Few-Shot and Retrieval-Augmented Generation (RAG). Our experiments demonstrated the feasibility and extensibility of our approach, which achieved a higher accuracy compared to the other techniques, highlighting the potential of agent-based architectures to standardize and automate TCs in continuous industrial testing environments.

KEYWORDS

Automated Validation, LLMs, Multi-Agent Systems, Prompt Engineering, RAG and Test Automation

1 Introduction

The quality and effectiveness of Test Cases play a central role in the software development lifecycle, especially in agile and continuous delivery environments. However, the rewriting of existing Test Cases (TCs) remains labor intensive, requiring domain expertise and substantial human effort. Inconsistencies and ambiguities make them difficult to reuse, understand, and maintain in the long term. Therefore, automating this activity represents a promising step forward to ensure consistency, increase coverage, and reduce the costs associated with software testing.

Large Language Models have recently transformed text generation tasks for software engineering. These models have proven capable of generating code, understanding requirements, and even proposing TCs from natural language descriptions [2, 15]. However, their direct application to sensitive tasks such as test rewriting and standardization continues to pose challenges such as logical inconsistencies, syntactic errors, risk of hallucinations and limited contextual grounding of the system under test [13, 17].

Building on these findings, recent studies have explored the use of LLMs in TC generation and reformulation tasks, demonstrating significant progress. Techniques such as Zero-Shot, Few-Shot, and Prompt Chaining have guided models toward creating test artifacts aligned with industry practices [5, 13]. Moreover, combining these techniques with contextual information retrieval, such as Retrieval-Augmented Generation mechanisms, has been key to enhancing the relevance and accuracy of the TCs generated by LLMs [8, 21]. More recently, a new line of research has emerged around Multi-Agent systems driven by transformer-based linguistic models, where different agents perform specific functions [7, 18]. Studies like TestChain and ChatUniTest have shown that this division of responsibilities, combined with automated validation mechanisms, can significantly improve the accuracy, coverage, and standardization of the generated TCs [5, 9].

Recent research has underscored the effectiveness of integrating rule-based mechanisms into LLM-driven text generation and validation workflows. These approaches improve syntactic consistency, accuracy, and interpretability in technical tasks. For example, Elhabbash et al. [3] proposed a system that converts requirement specifications into executable Test Steps using grammatical rules, while Li et al. [10] introduced a rewrite method that enhances query optimization without altering semantics. Additionally, the systematic review by Jayshree and Saravanan [6] reaffirms the importance of explicit rule-based techniques for generating test artifacts with

structural control and traceability. Collectively, these findings support the design of hybrid systems that combine the generalization power of LLMs with the precision of rule-based validation.

In this work, we propose ARTeMIS, a Multi-Agent framework powered by LLMs to automate the rewriting and standardization of TCs, structured through collaboration between specialized agents for classification, rewriting, grammatical and rule-based validation, and task coordination. This approach aims to overcome the limitations of standalone generative language models, by promoting iterative interactions between specialized agents playing different roles. Moreover, it investigates the impact of techniques such as Zero-Shot prompting, Few-Shot prompting and RAG on the individual performance of each agent. The results obtained demonstrate the potential of this approach to automate test validation and maintenance workflows, offering a scalable and adaptable solution that aims to reduce manual workload and ensure quality in continuous industrial testing environments.

The growing complexity and variability of industrial validation scenarios, as well as the high frequency of mobile software updates, demand strategies that go beyond manual or static rule-based approaches. Leveraging LLMs within a Multi-Agent architecture enables more robust handling of TC rewriting and standardization, ensuring syntactic consistency, adaptability, and reduction of human effort. In addition, the modular design of ARTeMIS facilitates the integration of specialized validations and its future extension to new components or domains, reinforcing its applicability in dynamic software engineering environments.

To conduct a rigorous and reliable evaluation of our solution, we performed experiments with both a dataset created and validated by Quality Assurance experts from a company directly linked to the software industry, which we cannot make available and reveal details for confidentiality reasons, and with a synthetic dataset, created through prompt engineering techniques and using the Gemini model, a LLM developed by Google. This work addresses two central questions. First, we investigate how an LLM-based agent framework can be designed and implemented to automatically enforce standardization on black-box test cases. Second, we evaluate how effective a validation-correction cycle is within this framework for iteratively improving the standard adherence of the rewritten test cases.

2 Background

Tasks related to creating, rewriting, and validating Test Case components has become increasingly relevant in software engineering, especially with the advancement of Natural Language Processing (NLP) area. Traditional approaches relied on manual effort, often resulting in time-consuming and error-prone processes. In recent years, several studies applying NLP and Machine Learning techniques have been carried out, aiming to improve the efficiency and reliability of TC generation and maintenance processes. [12, 17].

Data augmentation strategies, such as paraphrasing and sampling, have been applied to enhance training datasets, increasing the robustness of algorithms against linguistic variability. These techniques increase the coverage and reliability of the developed solutions, thus contributing to the production of broad and diverse test scenarios [14].

The emergence of instruction and prompt-based learning paradigms has further transformed the NLP applications. As shown by Zhang et al. [19], Few-Shot learning, a technique in which Large Language Models receive minimal but structured examples through prompts, has proven to be effective in tasks where labeled data are limited. This technique allows the LLM to generalize better and perform domain-specific tasks with less fine-tuning.

In the context of software testing, the integration of Retrieval-Augmented Generation techniques allows LLMs to generate Test Steps consistent with the context in which they are inserted. Wang et al. [17] demonstrates that retrieving semantically similar examples from existing documentation or code bases improves both the accuracy and domain alignment of the generated content.

Furthermore, the evaluation of LLMs has become a critical area of focus. Recent studies [12, 19] emphasize the importance of rigorous evaluation metrics to assess the performance, reliability, and limitations of LLMs in TC generation and validation tasks. These insights help guide the development of more dependable and explainable systems.

Based on the studies cited, it is clear that the insights generated by them point to a paradigm shift in Software Testing, demonstrating a transition from static, manual processes to dynamic, intelligent, and automated systems. Modern NLP techniques are redefining how TCs are created, rewritten, and validated, enabling greater consistency, maintainability, reliability, and scalability in these processes.

3 Related work

The application of Large Language Models to automate test generation, rewriting, and validation has gained increasing attention in recent years due to their potential to streamline complex processes. Several studies support this trend. Baqar and Khanda [1] explore how Artificial Intelligence and Machine Learning techniques can transform the generating and validating TCs processes, while Mathur et al. [11] and Frister and Hoffmann [4] demonstrate that fine-tuned LLMs enable more efficient test creation, with reduced human error. Collectively, these works highlight the ability of LLMs to deliver scalable, accurate, and automated solutions, positioning them as a disruptive tool in modern software engineering practices.

Zhang et al. [20] introduced GrammarTransformer, a tool designed to automate the transformation of grammars generated in Xtext through configurable rules, ensuring synchronization between the grammar and the metamodel. This approach was evaluated on seven domain-specific languages, such as ATL and Standard ML, and demonstrated a significant reduction in manual changes needed during language evolution. In parallel, Shu et al. [15] presented RewriteLM, a LLM specifically trained for controlled text rewriting using natural language rules and instructions. Based on data from Wikipedia edits and public corpora, their methodology combines supervised fine-tuning and reinforcement learning. Results on the OPENREWRITEEVAL benchmark reveal that RewriteLM outperforms models like Alpaca and Flan-PaLM in complex rewriting and conciseness tasks.

Additionally, Vaswani et al. [16] focused on improving the clarity and consistency of manual TC descriptions through the use of

Natural Language Processing techniques. The research addresses the problem of ambiguous and inconsistent descriptions that can generate errors in testing processes. By applying NLP models, they can refine these descriptions, ensuring that TCs are easier to understand and follow, which improves test execution and the reliability of their results. These approaches are complementary, both aiming to improve the efficiency of testing processes, whether through automation with LLMs or by improving readability and accuracy manually.

Li et al. [10] present a novel approach to query rewriting by leveraging LLMs to enhance the efficiency of SQL queries without altering their results. Traditional methods rely heavily on predefined rewrite rules and DBMS cost estimators, which can be resource-intensive and prone to inaccuracies. LLM-R2 addresses these limitations by training a contrastive learning model that improves the LLM ability to suggest optimal rewrite rules, leading to better query performance. Experimental results demonstrated significant improvements in execution efficiency and robustness compared to baseline methods across various datasets.

Another challenge is utilizing LLMs for multi-style text rewriting, particularly in avoiding issues such as semantic alteration and task confusion. These problems often arise from the model's difficulty in following instructions and its lack of robustness. To tackle these issues, Li and Yuan [9] proposed a methodology that improves, along with justifications, the discriminative capabilities of the model. Experimental results showed that such techniques significantly improve instruction adherence and output robustness in multi-style rewriting contexts. While our literature review considered multi-agent frameworks like *TestChain* and *ChatUniTest*, their primary focus is on test case generation from scratch. In contrast, our work addresses the distinct task of test case standardization and rewriting. Adapting a generation framework to this task would require significant modifications, essentially creating a new method rather than providing a fair baseline comparison. Due to this fundamental difference in tasks, we concluded a direct comparison was not feasible.

4 Proposed method

4.1 Test Cases

This study uses real data from industrial Quality Assurance processes. In our work context, **Test Cases** are composed of three key components: *Summary*, *Initial Setup*, and *Test Steps*. Figure 1 illustrates the hierarchical organization of these components and their relationships. Adherence to documentation standards for each component is essential for developing automatable and sustainable TCs.

However, it is common to encounter TCs that exhibit inconsistencies, ambiguous formulations, or do not follow previously defined documentation. These issues compromise their reusability and long-term maintenance, complicating their automation. To mitigate these issues, the quality assurance experts produced internal documentation to define writing standards for each component, including detailed guidelines, explanations, captions, and correction examples.

The Summary provides an assertive and informative overview of the TCs objectives, offering a concise description to guide its execution and evaluation. The Initial Setup specifies the preconditions and configuration parameters necessary prior to the execution of the Test Steps, which are responsible for describing the sequence of actions performed during the test, encompassing both software-level interactions and hardware-dependent procedures. The adoption of this categorization is based on both formal standards and internal definitions. The macro-classification into Summary, Initial Setup, and Test Step is common in the industry, following standards for Black Box testing (such as IEEE 829, albeit with different names). To allow for a finer-grained evaluation of rewriting strategies, the Test Steps component is further subdivided into four categories: *Conditional*, *Listing*, *Parameterized*, and *Passive*, according to syntactic and functional patterns. This sub-classification resulted from an internal effort by QA experts to establish specific rules to aid in the correction process. This complete structure reflects the standardized practices adopted in Android-based testing workflows and provides a natural segmentation for training, classification, and evaluation tasks.

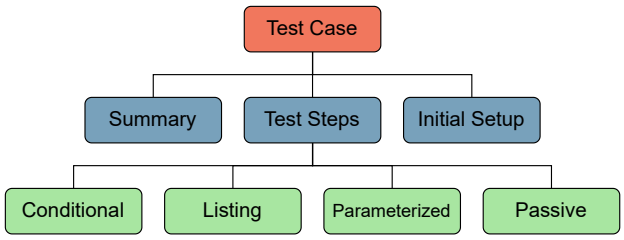

Figure 1: Hierarchical structure of a Test Case.

Conditional steps express actions that depend on a particular condition and, in some cases, must be split into independent Test Steps according to the conditional flow. *Listing* steps group multiple actions or elements in sequence, typically as inline lists or bullet points, which requires identifying the number of actions and explicitly dividing them into a structured list. In *Parameterized* steps, the actions include parameters, inputs, or commands that must be specified and then executed. Finally, *Passive* steps are written in the passive voice, making it difficult to identify the agent performing the action; therefore, the verb must be transformed into the imperative form. These structural variations affect both the classification accuracy and the overall difficulty of the rewriting process.

4.2 ARTeMIS

In this work, to address the challenge of maintaining standardized Test Cases on Android devices, we propose a modular Multi-Agent framework called **ARTeMIS**, which is an acronym for *Agent-based Rewriting and Test Case Management with Intelligent Supervision*. Our system integrates semantic retrieval, supervised classification, automatic rewriting through structured instructions and reflective learning, and rule-based grammatical validation. The goal is to optimize the standardization of obsolete or incorrect TCs, which is a consequence of factors such as frequent software updates, lack of standardization, or human error, with minimal human intervention, while ensuring agility, syntactic consistency, and semantic fidelity.

ARTeMIS is designed to automate the standardization of TC components in industrial quality control environments. The workflow is organized into three main stages: *Hybrid Classification* (only for Test Steps), *Component Specific Rewriting* (with reflective learning), and *Iterative Validation* (with a defined limit). These stages are illustrated in Figure 2. This architecture enables ARTeMIS to operate iteratively and autonomously, ensuring semantic fidelity and syntactic correctness with minimal human supervision.

The decision to adopt a Multi-Agent architecture in ARTeMIS is grounded in several technical and practical advantages that distinguish it from simpler monolithic pipelines, such as systems based solely on chained prompts and a single LLM.

First, modularity allows each agent to focus on a specific task, such as classification, rewriting, or validation, ensuring that grammatical rules, semantic coherence, and structural corrections are handled with dedicated logic and specialized control. This separation of responsibilities not only facilitates independent debugging and targeted improvements but also increases the system's overall interpretability and traceability.

Second, the iterative control enabled by the *Validator Agent* ensures that rewriting errors are not propagated without supervision. By offering explicit feedback at each cycle, the system guarantees gradual refinement of outputs, allowing adaptive corrections even in complex or ambiguous cases. This process also mitigates common LLM limitations such as hallucinations or semantic drifts, as every generated output must be explicitly validated before acceptance.

4.2.1 Hybrid classification of Test Steps. The operational process begins with loading a Test Case. Each TC component consists of a type (indicating the component) and a sentence (the textual part of the component) pair. Table 1 shows examples of all components, including the four categories into which the **Test Step** component can be classified.

The hybrid classification is a role made by the **Test Step Classification Agent**, leveraging the strengths of both supervised learning and semantic retrieval, ensuring robustness across varied linguistic patterns, by combining a *supervised Random Forest classifier* with a *semantic retrieval mechanism*.

In the proposed architecture, the Random Forest model is implemented as a supervised multiclass classifier responsible for inferring the grammatical category of each test case. For training, the annotated original test cases are converted into feature vectors using the SentenceTransformer model. These representations are then associated with the categorical labels parameterized, conditional, listing, and passive previously defined by QA experts and encoded as numerical values. The Random Forest classifier is trained on this dataset using 100 decision trees and random splits to mitigate overfitting and improve generalization.

The semantic retrieval mechanism classifies an input through stages. First, it generates the input vector embedding using dense embeddings generated by the *all-MiniLM-L6-v2* sentence-transformer model and cosine similarity. Then, it uses cosine similarity to retrieve the five most similar examples from a vector database. Each one of the retrieved examples receives a weighted vote, proportional to its similarity score. The category with the highest total votes is selected as the final prediction, which is then compared with the output of a traditional Random Forest model for evaluation. The final category is determined by consensus between both methods or, in case of conflict, by prioritizing the option with the highest classification probability.

4.2.2 Component specific rewriting. In the second stage, after the Classification process (if the component being treated is a Test Step), the Test Case component is sequentially routed to two specialized agents, for rewriting and validation.

The **Rewriter Agent** follows a structured, component specific approach that incorporates detailed grammatical rules, correction rules, transformation logic, contextual substitutions, and similar examples recovered semantically. These examples are recovered using dense embeddings generated by the *all-MiniLM-L6-v2* sentence-transformer model and cosine similarity. These rules and examples serve as contextual demonstrations that illustrate the necessary transformations to standardize the component. Table 2 shows an example of one rule per component/category received by this agent.

Based on these guidelines, the Rewriter Agent generates a new version of the original component. The generated output is then evaluated by the **Validator Agent**, which applies a set of grammatical and structural rules specific to each component. If the rewritten component does not meet the required criteria, the Validator returns a textual feedback that will be used by the Rewriter Agent in a new iteration, in order to refine the process of rewriting. Analyzing the Table 3, which depicts the System Prompts for both agents, it is possible to visualize the rewriting and validation process.

4.2.3 Iterative validation. *ARTeMIS* incorporates an iterative reflective validation strategy designed to ensure that the rewritten component comply with its defined grammatical, structural, and

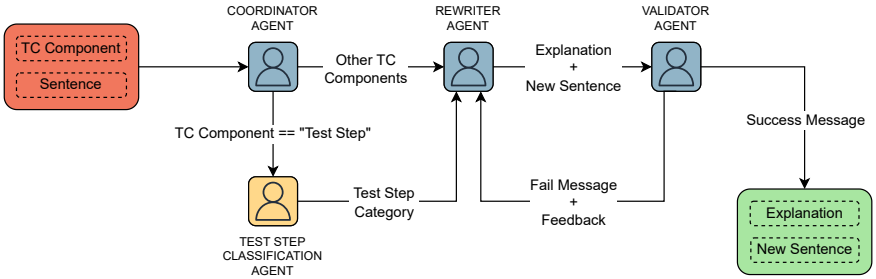

Figure 2: Architectural overview of ARTeMIS.

Component	Category	Original Sentence	Rewritten Sentence
Summary	Not applicable	Using only voice commands, start navigation to a specific place from the map screen.	Voice Input - Map Screen - Start navigation to a specific place using voice commands
Initial Setup	Not applicable	1. Get the NordVPN application installed on the device. 2. The user qa.user@test.com should already be logged in. 3. A connection to server located in Canada is activate.	1. Install the "NordVPN" app. 2. Log in with the account "qa.vpn.user@test.com". 3. Connect to a server in "Canada".
Test Steps	parameterized	Type the password P@ssw0rd123!.	Enter password "P@ssw0rd123!" into the "Password" field.
Test Steps	conditional	If the user is not logged in, verify the "Login" button is displayed on the home screen.	Ensure the user is logged out of the application. Navigate the user is logged out of the application. Verify to the application's home screen.
Test Steps	passive	The "Submit" button is clicked.	Tap the "Submit" button.
Test Steps	listing	Test the data export feature: 1. Go to the settings and find the data export feature. 2. Try to export the data as CSV file to local storage. 3. Then, try to export it as a JSON file to a cloud storage service. 4. After each export, open the file and verify that the data integrity is maintained and the format is correct.	Export user data: - As a CSV file - As a JSON file. - As a local storage. - And verify the integrity of the exported data.

Table 1: Examples of original and rewritten Test Cases by component and category.

Component	Category	Rule
Summary	Not applicable	- Always use a hyphen surrounded by spaces (-) to separate the main parts (Feature, Location, and the Action phrase).
Initial Setup	Not applicable	- Every sentence must be a direct command that starts with an action verb followed by a specific object.
Test Steps	Parameterized	- Explicitly identify UI elements by name. Always refer to a UI element by its exact visible text label or a clearly defined name (like an accessibility ID).
Test Steps	Conditional	- The Rule of Atomicity: One action per step.
Test Steps	Passive	- Start with an action verb in the imperative mood.
Test Steps	Listing	- Isolate a single action per test case. Each test case should focus on one specific action or function.

Table 2: Rule examples applied to each test case component and category.

Prompt (Rewriter Agent)

You are an Android (mobile device) specialist. Your task is analyze whether an Original *component_name* follows the Grammatical Structure.

If the Original *component_name* does not comply with the Grammatical Structure, justify why and then rewrite it correctly.

[Grammatical Structure]
component_structure

[Components Description]
structure_components_description

[Rules]
- *correction_rules*

[Examples]
Original Sentence: *original_sentence*
Explanation: *correction_explanation*
Rewritten Sentence: *rewritten_sentence*

[Output Instructions]
- Output exactly two lines:
Explanation: <why the *component_name* is incorrect>
Corrected *component_name*: <rewritten *component_name*>
Provide concise and unique explanations only. Avoid repetitive justifications.

Prompt (Validator Agent)

You are an automatic validator. Your task is to assess whether a rewritten sentence complies with the following Grammatical Structure and Rules.

[Grammatical Structure]
component_structure

[Components Description]
structure_components_description

[Rules]
- *validation_rules*

[Output Instructions]
- Output exactly two lines:
Justification: <why the rewritten sentence passes or fails the validation>
Result: <Pass or Fail>
Provide concise and unique explanations only. Avoid repetitive justifications.

Table 3: Examples of the structured prompts used in the Rewriter and Validator agents.

semantic criteria. For each one of them, the **Coordinator Agent** initiates a controlled generation and evaluation cycle, limited to a configurable maximum of K iterations. In each iteration, the original component is processed by the*Rewriter Agent*, which generates a new version based on its particularities.

The rewritten version of the component is immediately evaluated by the *Validator Agent*, which applies a set of formal rules to verify whether the generated content meets the established syntactic and semantic criteria. If the new version of the component is

not accepted, the Validator Agent returns a detailed textual feedback. This feedback is incorporated by the Rewriter Agent in the next cycle, allowing the output to be progressively refined until a valid version is reached, or the maximum number of attempts is exhausted.

The number of iterations in the rewriting-validation loop was empirically defined as $k = 30$, chosen after conducting a series of exploratory tests starting from $k = 3$ and gradually increasing it. We observed that performance improved consistently up to $k = 30$, beyond which the gains plateaued. This setting ensures a balance between convergence of high-quality rewrites and computational cost, and was adopted as the default throughout all experiments.

Additionally, an intelligent monitoring mechanism is implemented to detect consecutive empty, incoherent, or redundant responses. This pattern is interpreted as a loss of conversational context. In such cases, the internal history of the Rewriter Agent is automatically reset, forcing a new generation based solely on the base prompt and the original component. This procedure significantly enhances the system's ability to recover from complex inputs and prevents model stagnation during the rewriting process.

Each iteration logs the individual execution time, validation status, and generated content. All results are stored with complete traceability, enabling auditability of the evolution of each rewrite case. This also ensures transparency in each rewriting cycle, supporting both reproducibility and industrial applicability.

5 Experimental setup

5.1 Datasets

In our experiments, we use two datasets: a **private dataset**, with real outdated Test Cases samples, along with documentation, rewriting rules, component name, original sentences, correction notes and corrected sentences from real Quality Assurance suites; and a **public synthetic dataset**, created for this work in order to ensure reproducibility and comparison with past and future works.

The private dataset is composed of 540 samples, of which 100 belong to the *Summary* component, 100 to the *Initial Setup* component, and 340 to the *Test Steps* component. The dataset was split into 45% for training and 55% for testing. In the training phase, corresponding to the *Test Steps* component, 140 examples were used, specifically selected to train the hybrid classifier and build the semantic retrieval base. These examples were evenly distributed among the four categories defined in the proposed taxonomy (*Conditional*, *Listing*, *Parameterized*, and *Passive*), with 35 samples per category. For each category, 20 examples were used in the supervised training of the `Random Forest` classifier, while the remaining 15 composed the vector base employed by the Retrieval-Augmented Generation mechanism, during the semantic retrieval stage.

Moreover, 50 samples from the Summary component and 50 samples from the Initial Setup component were used to construct the respective vector bases utilized in the RAG mechanism. This configuration enabled the system to retrieve previously validated examples during the rewriting process. The test set consisted of 300 manually annotated examples: 50 corresponding to the Summary component, 50 to the Initial Setup component, and 200 evenly distributed among the four categories of the Test Step component. This division ensured a balanced, stratified, and reproducible evaluation of the ARTeMIS framework's performance on previously unseen inputs.

As an additional step to validate the system's generalization capabilities, as well as reproducibility, we designed and developed a synthetic dataset. For this purpose, a structured guide was designed based on the grammatical rules and linguistic patterns previously defined by us for each of the three components: Summary, Initial Setup, and Test Steps. Using this guide, synthetic component samples, equally divided between syntactically valid and invalid examples, were generated: 540 samples, of which 100 belong to the Summary, 100 to the Initial Setup component, and 340 to the Test Steps component, equally divided between the 4 categories. Both the guide and the samples were generated with the **Gemini 2.5 Pro** language model.

To ensure the quality and relevance of the generated synthetic dataset, the guide and all the samples were manually validated by Quality Assurance experts. This process confirmed that each example either complied with the defined syntactic and semantic standards or intentionally violated them to simulate outdated or incorrect TC components.

Once validated, the four experiments described in Section 6 were also performed on the synthetic dataset, allowing a comparative analysis of system performance on real versus synthetic data under identical experimental conditions. The prompts used for data generation, as well as the complete synthetic dataset, are publicly available in the project repository, aiming to promote transparency and reproducibility.

5.2 Evaluation metric

To assess the performance of the rewriting and classification strategies, we used **accuracy** as the primary evaluation metric. Accuracy measures the proportion of correctly predicted or rewritten sentences over the total number of evaluated sentences, and is defined as:

$$\textbf{Accuracy} = \frac{\text{Number of correct outputs}}{\text{Total number of outputs}} \times 100\% \tag{1}$$

This metric was uniformly applied across all experiments, for both automatic and manual validation. Its use enables a consistent and comparable assessment of each approach's effectiveness in generating standardized and semantically coherent results.

5.3 Baselines

To evaluate the effectiveness of the automatic rewriting and validation strategy, we compared our proposed method, *ARTeMIS*, with three baselines: **Zero-Shot**, **Few-Shot**, and **RAG**. All experiments were performed on the same datasets, under controlled conditions to ensure fairness.

The following key aspects were evaluated:

- **Manual Rewriting Accuracy:** the rate of TC components correctly rewritten. For this manual evaluation, QA experts were instructed to check not only for compliance with the grammatical structure and specific transformation rules, but also to ensure that no essential information or meaning was lost during the rewriting process.

- **Automatic Validation Accuracy**: the rate of TC components correctly rewritten according the automated evaluation performed by the *Validator Agent*. It is presented as ACC_a in the experiments.
- **Execution Time**: the average time required to standardize each TC component.

In addition to the three proposed baselines, we also present ARTeMIS results before and during the reflective rewriting process (i.e. accuracy before the first and for all subsequent iterations).

The methodological details of each method are described below:

5.3.1 Zero-Shot. In this baseline we aim to evaluate the capability of the Large Language Model to rewrite a component using only explicit grammatical and structural rules, without relying on contextual examples. To achieve this, we designed a single, rule-intensive *prompt*, tailored to each component (*Summary*, *Initial Setup*, and *Test Step*) or Test Step category (*parameterized*, *conditional*, *passive*, and *listing*).

The **prompt** contained detailed grammatical guidelines, formal structural instructions, and lexical substitution rules, ensuring that the model followed a strictly controlled rewriting standard. However, no sample examples were included, and no correction demonstrations were provided. This configuration allowed us to isolate the model's inherent ability to generalize based purely on syntactic and structural definitions.

The rewriting process in this approach was performed in a single attempt, without iterative refinement cycles or intermediate feedback. This setup mirrors a straightforward one-pass rewriting scenario and does not involve dynamic correction loops. All other experimental conditions, including dataset composition and evaluation metrics, were kept identical to the subsequent approaches to ensure comparability across techniques.

5.3.2 Few-Shot. We also evaluated the *few-shot* approach to assess whether providing static examples could guide the model toward producing more accurate and stylistically consistent rewrites compared to the *Zero-Shot* approach.

For this purpose, each **prompt** was carefully constructed to include explicit grammatical rules, structural instructions, and a curated set of correct and incorrect examples. These examples were manually selected and tailored tailored to each component (*Summary*, *Initial Setup*, and *Test Step*) or Test Step category (*parameterized*, *conditional*, *passive*, and *listing*). The examples served as static demonstrations to illustrate both correct and incorrect rewriting patterns.

The rewriting process in this approach was performed in a single attempt, without iterative refinement cycles or intermediate feedback. This setup mirrors a straightforward one-pass rewriting scenario and does not involve dynamic correction loops. The inclusion of examples was designed to help the model better capture syntactic patterns and domain-specific nuances, with the aim of improving structural adherence and stylistic alignment over purely rule-based prompts. All other experimental conditions and evaluation criteria were kept consistent with the other experiments to ensure direct comparability.

5.3.3 RAG. Using Retrieval-Augmented Generation, our objective was to evaluate whether dynamically retrieved context could enhance rewriting accuracy and semantic adequacy compared to purely static examples. To achieve this, we employed a RAG-based approach, which involved retrieving the top five most semantically similar examples for each component from a pre-built train dataset. The retrieval was performed using dense vector embeddings and cosine similarity, allowing the system to select examples that were contextually close to the input.

The final **prompt** included grammatical rules, structural rewriting instructions, and the dynamically retrieved examples. Unlike the *Few-Shot* approach, no static examples were predefined; instead, the examples varied depending on the specific semantics of each input.

This configuration allowed us to assess the potential of semantic context retrieval to improve syntactic adherence and domain relevance, while still operating without iterative correction mechanisms. All other experimental conditions were kept identical to ensure fair comparability across techniques.

5.4 Runtime environment

The experiments were conducted using the **Google Colab** environment, equipped with an Intel Xeon CPU operating at 2.30 GHz, 13 GB of RAM, and a virtual storage space of 235 GB. An NVIDIA Tesla T4 GPU with CUDA 12.4 support was utilized to accelerate the processing of LLMs and intensive embedding computations.

This cloud-based setup provided the computational power needed to efficiently perform model inferences, data encoding, and rewriting tasks, ensuring reproducibility and high-quality experimental results.

6 Results

In this section, we present the results of ARTeMIS and the baselines in the real and synthetic datasets.

Table 4 presents both the manual and automatic validation accuracies for ARTeMIS and all baseline methods, evaluated on the real and synthetic datasets. Overall, the final iteration of ARTeMIS (ART_f) achieved the best results across all components, with a particularly large improvement in the manual evaluation. This indicates that, although the automatic validation results are relatively close among the approaches, the manually validated results clearly show that ARTeMIS produces substantially higher-quality rewrites. This difference is especially evident in the *Test Steps* component, where the manual accuracy increased from 41% in the initial iteration (ART_i) to 80% in the final one, a gain of 39 percentage points, while the automatic accuracy rose by 37%.

While these results attest to the quality of the proposed approach, comparing the manual (ACC_m) and automatic (ACC_a) validation accuracies provides further evidence of the validator's reliability. Table 5 presents the Root Mean Square Error (RMSE) and its Standard Deviation (SDE) between the manual and automatic validation scores. As shown, there was a strong agreement between both validations for the *Test Steps* component in both datasets, as indicated by the lowest RMSE and SDE values. In contrast, the *Summary* component exhibited the largest discrepancies, particularly in the real dataset, where the RMSE reached 0.40. For the *Initial Setup* component, the automatic validator tended to slightly overestimate performance relative to the manual evaluation, suggesting that

	Summary		Initial Setup		Test Steps	
Real Dataset						
	ACC$_a$	**ACC$_m$**	**ACC$_a$**	**ACC$_m$**	**ACC$_a$**	**ACC$_m$**
ART$_i$	0.54	0.44	0.56	0.38	0.45	0.41
ART$_f$	0.96	0.84	0.96	0.74	0.82	0.80
Zero$_i$	0.34	0.20	0.54	0.32	0.43	0.38
Zero$_f$	0.98	0.32	1.00	0.28	0.64	0.61
Few$_i$	0.54	0.28	0.58	0.52	0.43	0.39
Few$_f$	0.96	0.32	0.98	0.64	0.69	0.65
RAG$_i$	0.20	0.64	0.38	0.36	0.48	0.43
RAG$_f$	0.98	0.58	1.00	0.62	0.73	0.77
Synthetic Dataset						
ART$_i$	0.24	0.22	0.76	0.58	0.50	0.46
ART$_f$	0.98	0.84	1.00	0.76	0.91	0.87
Zero$_i$	0.26	0.18	0.72	0.66	0.41	0.36
Zero$_f$	1.00	0.30	1.00	0.70	0.73	0.66
Few$_i$	0.30	0.36	0.74	0.58	0.49	0.44
Few$_f$	0.94	0.46	1.00	0.70	0.75	0.73
RAG$_i$	0.42	0.56	0.50	0.32	0.50	0.47
RAG$_f$	0.98	0.72	1.00	0.34	0.85	0.83

Table 4: Results of all experiments – Zero-Shot, Few-Shot, RAG, and ARTeMIS – evaluated in their initial ($_i$) and final ($_f$) iterations.

more refined validation strategies could further improve consistency for this specific component.

Dataset	Component	RMSE	SDE
Real	Summary	0.405	0.352
	Initial Setup	0.338	0.220
	Test Steps	0.039	0.029
Synthetic	Summary	0.324	0.284
	Initial Setup	0.310	0.180
	Test Steps	0.043	0.017

Table 5: Root Mean Square Error (RMSE) and Standard Deviation of the Error (SDE) between automatic and manual validation, grouped by dataset and component.

A component-by-component analysis of the real dataset shows that the performance of the baselines and ART$_i$ was relatively similar, though still well below that of ART$_f$. This reinforces the crucial role of the reflective iterative validation process.

Although reflective automatic validation leads to substantially better results, the computational cost of the rewriting cycles must be addressed. As shown in Figure 3 and Figure 4, ARTeMIS requires more processing time than the baselines in all scenarios due to its use of multiple agents. However, this increased computational cost is offset by consistent gains in rewriting accuracy.

In the real dataset, the *Summary* and *Initial Setup* components required fewer iterations (approximately five) to surpass the accuracy of the baselines. In contrast, the *Test Steps* component posed a greater challenge, with accuracy improving more gradually over a larger number of iterations. This is consistent with the fact that *Test Steps* are more diverse and are categorized into four distinct categories, which increases the complexity of the rewriting task.

Furthermore, as indicated in Table 5, the automatic validator is more prone to false positives for the *Summary* and *Initial Setup* components. This likely occurs because they are assessed against more general rules, which can lead to greater ambiguity compared to the specific, granular rules for each *Test Step* category. This may cause incorrect sentences to be accepted prematurely, leading to an earlier plateau in accuracy for these components.

6.1 Classification evaluation

To assess the performance of the *Test Step Classification Agent*, we present a confusion matrix in Figure 5, showing that it achieved consistent results across most categories, with minor confusions in syntactically similar cases. In addition to the confusion matrix, the classifier achieved a Precision of 88%, a Recall of 87%, and an F1-score of 87%, providing a more comprehensive assessment of its balanced performance.

The Test Step Classification Agent demonstrated robust performance across most categories. In the Parameterized category, it achieved near-perfect accuracy, correctly identifying 49 out of 50 samples with only one misclassification as Conditional. The Passive category also showed strong results, with 45 correct predictions and minor confusion with Conditional (2 instances) and Listing (3 instances).

More variability was observed in the Conditional category, where 42 out of 50 classifications were correct. A notable six instances were incorrectly labeled as Listing, suggesting an overlap in structural patterns, particularly in steps involving enumerations or conditional phrases. The Listing category presented the highest level of confusion, with 11 instances misclassified as Conditional, likely due to syntactic proximity between list-based actions and conditional expressions. Overall, the *Test Step Classification Agent* exhibited strong generalization capabilities and high accuracy across the four subcategories. The remaining misclassifications underscore the importance of syntactic disambiguation for structurally similar test instructions. These findings validate the proposed taxonomy and highlight the necessity of defining clear linguistic boundaries for effective rewriting prompts and validation rules.

In the Zero-shot, Few-shot, and RAG baselines, an ideal scenario was initially assumed where each step was perfectly pre-labeled with its correct category. To facilitate a more realistic comparison, we manually evaluated their performance by integrating our supervised classifier to predict the category before prompt selection.

As shown in Table 6, incorporating the classifier into the baselines resulted in a slight drop in rewriting accuracy on both real and synthetic datasets, primarily due to misclassifications during the prediction step. This decrease was approximately 10% when using our classifier agent, indicating that misclassifications indeed presented a more challenging scenario for ARTeMIS. Nevertheless, even with this disadvantage, ARTeMIS demonstrated superior results, confirming that its classification quality is sufficiently high to enable very accurate performance.

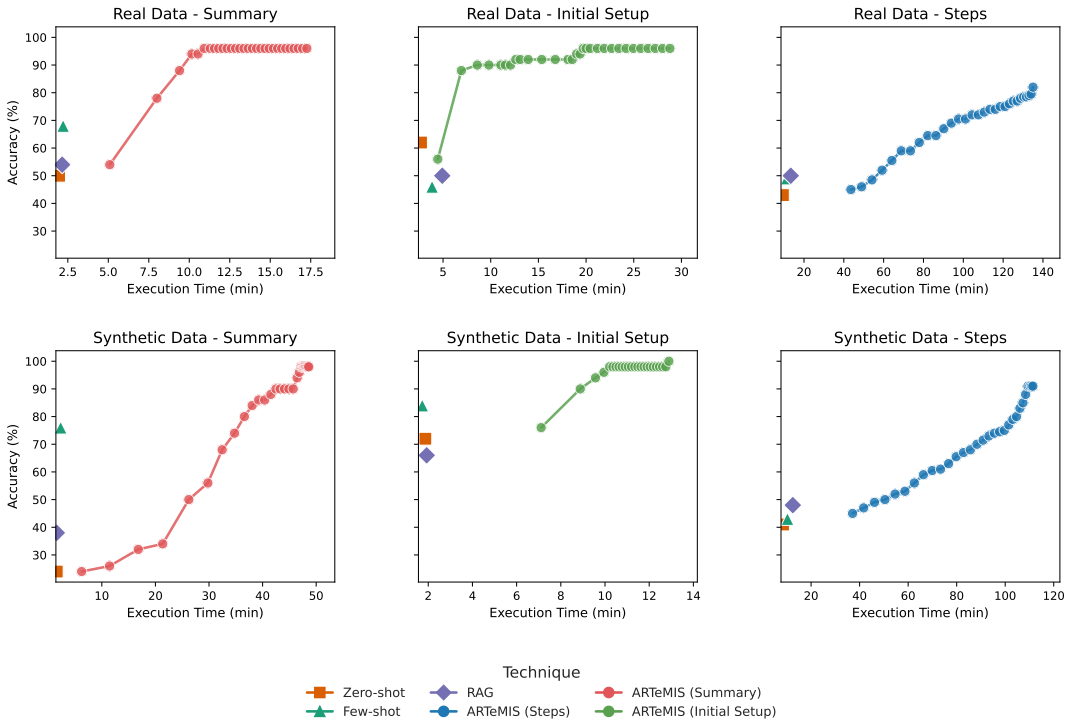

Figure 3: Comparison of rewriting accuracy and execution time across four rewriting strategies: Zero-Shot, Few-Shot, RAG, and ARTeMIS on the real-world dataset, for 50 Summary samples, 50 Initial Setup samples and 200 Test Step samples.

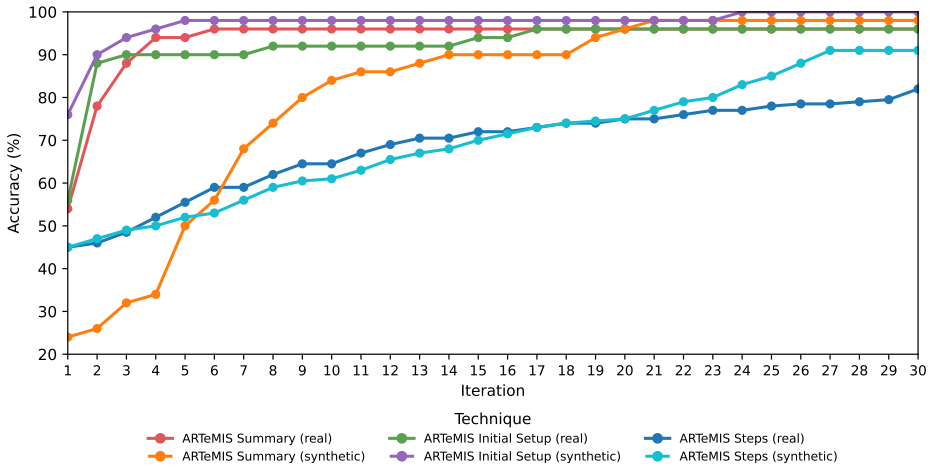

Figure 4: Comparison of rewriting accuracy per iteration across four strategies: Zero-Shot, Few-Shot, RAG, and ARTEMIS.

6.2 Discussion

These results strongly support our initial hypothesis that combining structured prompting, semantic retrieval, and multi-stage iterative validation is substantially more effective than isolated strategies. Despite using a compact model such as the *Meta-Llama-3.1-8B-Instruct*, our framework was able to generate high-quality, standardized outputs thanks to its modular agent-based architecture and carefully tailored prompting strategies for each linguistic pattern. Overall, the demonstrated performance confirms ARTeMIS as a robust and scalable solution for rewriting and standardizing Test Cases. Its design and operational flexibility make it highly applicable in industrial automation scenarios, where maintaining consistent and high-quality documentation is critical for continuous testing and software validation workflows.

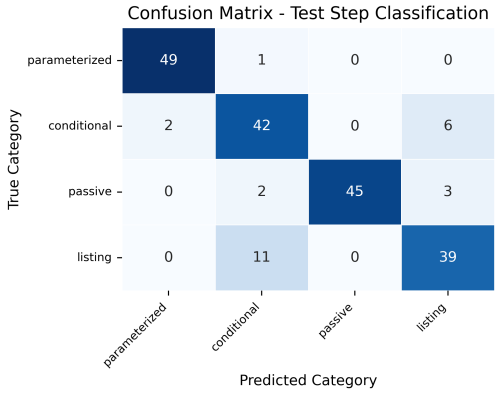

Figure 5: Confusion matrix illustrating the performance of the Random Forest classifier across the four Test Step categories: Parameterized, Conditional, Passive, and Listing.

	No classifier		With classifier	
	Real	Synthetic	Real	Synthetic
Techique	**ACC$_m$**	**ACC$_m$**	**ACC$_m$**	**ACC$_m$**
Zero-Shot	0.32	0.31	0.43	0.41
Few-shot	0.37	0.40	0.49	0.43
RAG	0.42	0.44	0.50	0.48

Table 6: Performance results (ACC) of the rewriting techniques (Zero-Shot, Few-shot, and RAG), with and without the use of the classifier, using real and synthetic data.

One of the main motivations for designing the *Validator Agent* in ARTeMIS was precisely to mitigate the risk of hallucinations and semantic drifts inherent in LLMs. Thanks to iterative validation and the application of strict grammatical rules, the system significantly reduces the likelihood of producing fictitious outputs. However, we acknowledge that when working with LLMs, a residual risk of hallucination remains—especially in highly ambiguous scenarios or with inputs containing substantial errors. This controlled risk reinforces the importance of the feedback cycle and the modular supervision architecture.

To the best of our knowledge, there are no publicly available frameworks that integrate a Multi-Agent pipeline with iterative validation, explicit grammatical rules, and controlled feedback mechanisms, as implemented in ARTeMIS. For this reason, Zero-Shot, Few-Shot, and RAG-based techniques were adopted as representative proxies of baseline strategies found in the state of the art, enabling a fair comparison with approaches that lack iterative validation or modular division of responsibilities. Likewise, as evidenced in the study, ARTeMIS allows agent-by-agent evaluation to verify individual impact, highlighting the importance of maintaining their interconnected roles.

7 Conclusions

In this work, we demonstrated that integrating specialized agents within an iterative validation loop significantly improves the standardization and rewriting of Test Case components. Our proposed framework, ARTeMIS, offers a scalable and robust solution for industrial environments by combining supervised classification, semantic retrieval, and modular agents. While the iterative cycles increase processing time, the notable improvements in quality justify this computational cost, confirming its value for continuous validation workflows and industrial test automation scenarios.

The primary contribution of this research is demonstrating that even compact models like Meta-Llama-3.1-8B-Instruct can achieve high-quality results when paired with iterative strategies and specialized agents. The modular architecture of ARTEMIS is central to this success. It enhances reliability by mitigating hallucinations and offers high adaptability, as the framework can be tailored to new contexts by modifying agent prompts and RAG documentation, avoiding the need for model fine-tuning. This process ensures structural coherence and compliance with predefined standards, enabling seamless integration into real-world CI/CD pipelines

For future work, several avenues warrant exploration. First, applying this framework to other types of technical or requirements documentation could validate its adaptability beyond TCs. Second, investigating alternative, potentially more efficient, Large Language Models could help optimize the trade-off between computational cost and performance. Finally, developing more sophisticated *Validator Agents*, particularly for the *Initial Setup* component where we identified a higher rate of false positives, could further enhance the system's overall accuracy.

In conclusion, ARTeMIS provides a reliable and extensible framework for standardizing TCs with minimal human intervention, establishing a strong foundation for future research in automated technical document rewriting and validation.

ARTIFACTS AVAILABILITY

To ensure reproducibility and support future research, we publicly share the main artifacts used in our experiments. These include the complete synthetic dataset generated to evaluate the ARTeMiS system. The data were generated using Gemini 2.5 Pro, based on structured prompts tailored to each one of the three components: *Summary*, *Initial Setup*, and *Test Steps*, including all its categories.

In total, 540 samples were produced of which 100 belong to the *Summary* component, 100 to the *Initial Setup* component, and 340 to the *Test Step* component, equally divided between its four categories. This ends up resulting in a diverse and representative evaluation dataset. Additionally, we provide the detailed prompts used during the data generation process. All examples were manually validated by Quality Assurance experts, to ensure correctness and consistency. These artifacts are available in a public anonymous repository for verification and reproducibility: [1].

ACKNOWLEDGMENT

This work was partially supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES-PROEX) - Finance Code 001, and by the Fundação de Amparo à Pesquisa

[1] https://anonymous.4open.science/r/ARTeMIS-344D/

do Estado do Amazonas - FAPEAM - through the POSGRAD 2024 project.

REFERENCES

[1] M. Baqar and R. Khanda. 2024. The Future of Software Testing: AI-Powered Test Case Generation and Validation. 19 pages. https://arxiv.org/abs/2409.05808

[2] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote. 2024. System Test Case Design from Requirements Specifications: Insights and Challenges of Using ChatGPT. https://arxiv.org/abs/2412.03693

[3] Alaa Elhabbash, John McDermid, Annika Menzel, Tom Crick, Robert Alexander, and Neil Walkinshaw. 2024. From Requirements to Test Cases: An NLP-Based Approach for High-Performance ECU Test Case Automation. *arXiv preprint* (2024). https://arxiv.org/abs/2505.00547

[4] D. Frister and J. Hoffmann. 2024. Generating Software Tests for Mobile Applications Using Fine-Tuned Large Language Models. In *2024 IEEE/ACM International Conference on Automation of Software Test (AST)*. 76–77. doi:10.1145/3644032.3644454

[5] Junxiao Han, Chuan Xu, Valerio Terragni, Haoye Zhu, Junjie Wu, and Lingming Zhang. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE)*. ACM, 572–576. doi:10.1145/3663529.3663801

[6] R. Jayshree and V. Saravanan. 2021. A Review on Test Automation for Test Cases Generation using NLP Techniques. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)* 12, 10 (2021), 2687–2693. https://turcomat.org/index.php/turkbilmat/article/view/2687

[7] Cristian Jimenez-Romero, Alper Yegenoglu, and Christian Blum. 2025. Multi-Agent Systems Powered by Large Language Models: Applications in Swarm Intelligence. doi:10.48550/arXiv.2503.03800

[8] Heiko Koziolek, Virendra Ashiwal, Soumyadip Bandyopadhyay, and K. R. Chandrika. 2024. Automated Control Logic Test Case Generation using Large Language Models. In *Proceedings of the 29th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, Padova, Italy, 1–8. doi:10.1109/ETFA61755.2024.10711016

[9] K. Li and Y. Yuan. 2024. Large Language Models as Test Case Generators: Performance Evaluation and Enhancement. https://arxiv.org/abs/2404.13340

[10] X. Li, Y. Wang, Z. Zhang, and L. Chen. 2024. LLM-R2: A Large Language Model Enhanced Rule-based Rewrite System for Boosting Query Efficiency. *Journal of Database Management* 45 (2024), 123–135. https://arxiv.org/abs/2404.12872

[11] A. Mathur, D. Patel, S. Pradhan, R. Regunathan, and P. Soni. 2023. Automated Test Case Generation Using T5 and GPT-3. In *Proceedings of the 2023 9th International Conference on Advanced Computing and Communication Systems (ICACCS)*. 1986–1992. doi:10.1109/ICACCS57279.2023.10112971

[12] Wendkûuni C. Ouédraogo, Kader Kaboré, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. 2024. Large-scale, Independent and Comprehensive Study of the Power of LLMs for Test Case Generation. *arXiv* (2024). https://arxiv.org/abs/2407.00225

[13] C. Paduraru, M. Zavelca, and A. Stefanescu. 2025. Agentic AI for Behavior-Driven Development Testing Using Large Language Models. In *Proceedings of the 17th International Conference on Agents and Artificial Intelligence - Volume 2: ICAART*. INSTICC, SciTePress, 805–815. doi:10.5220/0013374400003890

[14] Ye Shang, Quanjun Zhang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. 2024. A Large-scale Empirical Study on Fine-tuning Large Language Models for Unit Testing. *arXiv* (2024). https://arxiv.org/abs/2412.16620

[15] L. Shu, L. Luo, J. Hoskere, Y. Zhu, Y. Liu, S. Tong, J. Chen, and L. Meng. 2024. RewriteLM: An Instruction-Tuned Large Language Model for Text Rewriting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 18970–18980. https://ojs.aaai.org/index.php/AAAI/article/view/29863

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, Vol. 30. https://arxiv.org/abs/1706.03762

[17] Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025. TESTEVAL: Benchmarking Large Language Models for Test Case Generation. https://arxiv.org/abs/2406.04531

[18] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 76:1–76:24. doi:10.1145/3660783

[19] J. Zhang, Y. Chen, H. Huang, Q. Chen, Y. Wang, J. He, L. Liu, and Y. Ma. 2024. Supporting meta-model-based language evolution and rapid prototyping with automated grammar transformation. *Journal of Systems and Software* 212 (2024), 111846. https://www.sciencedirect.com/science/article/pii/S0164121224001146

[20] Quanjun Zhang, Ye Shang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. 2024. TestBench: Evaluating Class-Level Test Case Generation Capability of Large Language Models. *arXiv* (2024). https://arxiv.org/abs/2409.17561

[21] Yaqin Zhou, Zhe Wang, Ziyuan Li, Lingming Zhang, and Jun Wang. 2023. LLM4Test: A Study and Dataset on Using Large Language Models for Test Case Generation. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1–12. doi:10.1109/ASE56732.2023.00012