Reducing Manual Testing Overhead through AI-assisted Report Automation

Tunahan Catak, Enkhzol Dovdon*, Luiz Schiller, Gabriel Shamon, Frank Van Der Kruijssen*
ICT Group, Center of Excellence, Eindhoven, The Netherlands
{tunahan.catak, enkhzol.dovdon, luiz.schiller, gabriel.shamon, frank.van.der.kruijssen}@ict.nl

***Abstract*—In software projects, executing test specifications is a key element in keeping the quality of the software project high. Manual execution of test specifications also often requires creating a test report, which is a time-consuming task, while it also requires the tester needs to context switch from test execution to report creation. In this paper, we propose the TestPal application, which streamlines manual testing workflows by leveraging large language models (LLMs) combined with retrieval-augmented generation (RAG) and image processing techniques for automated test report generation. Our solution processes screen recordings captured during manual test execution, extracting frames containing meaningful visual events, identifying user actions through AI-powered analysis, storing these actions in a queryable knowledge base, and validating detected behaviors against test case specifications. The system creates a searchable repository of detected actions with visual evidence, enabling automated test validation and evidence-backed report generation. We evaluated the system using a synthetic desktop application that combined command-line and graphical interfaces to simulate real-world industrial scenarios. The results indicate that AI can be used in graphical user interface (GUI) testing tasks to reduce the time consumption of the test report creation process. The overall accuracy rate of correctly labeled test case outcomes with understandable, logical reasoning is 79% in our experiments, while significantly reducing manual effort in test report creation.**

***Index Terms*—GUI Testing, Large Language Models, Retrieval-Augmented Generation (RAG), Computer Vision, Automated Test Reporting, Manual Testing Workflow Automation.**

I. Introduction

In industrial multi-system environments that integrate both stand-alone and embedded applications, which consist of tightly coupled systems with interconnected subsystems [1], [2], introducing a new patch and ensuring complete and reliable synchronization is an inherently complex task. Manual graphical user interface (GUI) testing and documentation with supporting evidence remain significant productivity challenges in such settings. AI-powered software testing [3] and automated reporting [4] present a promising solution because they reduce repetitive and time-consuming activities.

Although existing AI-based test automation solutions, including the studies discussed in Section V, lack the capability to address our scenario, we require a tool that can intelligently analyze screen recordings captured during test execution, leverage contextual understanding within a specific domain to interpret test cases, and generate comprehensive reports with supporting evidence.

Our approach emphasizes non-intrusive testing methods to avoid the "probe effect" [5], wherein test automation tools that directly interact with system interfaces can alter the behavior of the system under test (SUT). In multi-system environments, instrumentation overhead from testing tools can perturb timing and disrupt synchronization in concurrent systems [6]. To mitigate both the probe effect and the oracle problem [7], we adopt screen recording as our primary input modality. This observational approach captures actual user-visible behavior without introducing instrumentation overhead, thereby providing a reliable oracle based on visual evidence of system behavior.

We introduce TestPal, an AI-assisted application designed to automate test report generation by integrating large language models (LLMs) with retrieval-augmented generation (RAG) and image processing techniques. Using screen recordings and test case specifications as inputs, TestPal extracts relevant insights, determines pass or fail outcomes for each test case by incorporating domain-specific context through RAG, and analyzes the results to produce comprehensive reports that include evidence and detailed explanations. Recent research demonstrates that LLMs can effectively support software testing tasks such as interpreting outcomes and generating artifacts when combined with domain knowledge via retrieval, which suggests feasibility for automated analysis and reporting [8].

To validate our methodology, we conducted systematic experiments using a synthetic desktop application that combines command-line and GUI to represent real-world industrial scenarios. Our experimental results demonstrate that TestPal achieves an overall accuracy rate of 79% in correctly determining test outcomes with logical reasoning, while significantly reducing both time consumption and manual effort compared to our previous manual documentation approaches.

This paper is organized as follows. Section II establishes the testing context and identifies opportunities for AI assistance. Section III provides a detailed description of the pipeline architecture. Section IV presents the experimental evaluation. Section V reviews related work, whereas Sections VI and VII offer the conclusions together with directions for future enhancement.

II. Background

A. Software Testing Context

In software projects, testing is an essential part of the software development pipeline. Without it, there is no proof

that it satisfies the expectations of clients and stakeholders. It also verifies whether the functionality of the software aligns with the requirements during the development cycle. Traditionally, it is a manual process, requiring human involvement. Organizations spend a lot of resources to ensure that the quality of their software systems is acceptable [9]. Nowadays, automated testing is increasing [10]. However, manual testing still has its unique place in software testing. Manual testing is a better tool than automated testing for certain concepts such as exploring the software tool, logical replacement, negation insertion, and timer replacement. [11] [12].

A survey shows that between 30% and 50% of the budget of software projects is spent on testing [13]. From a financial perspective, reducing the cost of testing while increasing the quality of the generated report is beneficial for software projects.

B. Manual GUI Testing Workflow and Report Creation

In our current workflow, a tester gets test case specifications and prepares the SUT environment. Then, the tester executes the manual tests following the steps from the test case specification document. For each step, the tester collects evidence and takes notes for the test report. In this process, the tester needs to continuously switch contexts between execution and data collection. This can reduce the efficiency (focus and speed) of a tester and makes the process more error prone. Once the test execution phase is done, the results have to be documented in the test report to show pass and failure of each test step with their evidences. This tight coupling of execution and report creation tasks creates labor-intensive activities while making the quality of the reports highly dependent on the individual testers' discipline and available time.

C. Bottlenecks in Software Testing

Report creation is time-consuming, especially for large test suites. Capturing screenshots manually and annotating them adds more time-consumption to this task. In addition to these, human error plays a critical role in software testing such as mislabeled screenshots or missing evidence, or weaker explanation for failure scenarios. Also not every tester's knowledge is in the same level in our domain and it creates different depth of knowledge in test reports.

Lack of automated support to extract structured actions and evidence from text execution process may causes the problems mentioned above. Therefore, need of tools can align what happened during text execution with the test case specification documents is obvious.

D. Opportunities for AI Assistance in GUI Testing

The latest developments in artificial intelligence offer promising solutions for manual GUI testing and automated test report generation. In our study, four key technologies have been used for automating test report generation:

LLMs have enabled contextual understanding in natural language processing, and their generation capabilities make them a good candidate for our study. In the software testing context, LLMs can understand natural-language test case specifications, generate explanations of test outcomes, and achieve strong semantic alignment between expected and observed behaviors. Their application to automated testing and test report creation workflows has shown promising results in reducing manual effort [3], [4].

By processing visual information alongside textual prompts, Vision-Language Models (VLMs) enable understanding of GUI elements and user actions across frames. In recent work by Yu et al., the study shows the capability of VLMs to understand GUI widgets and user actions [14].

In a long test suite, retrieving the relevant context is important for enabling LLM to conclude a pass or fail result. Recent applications of RAG in testing contexts have demonstrated improved workflow efficiency [15]. Another benefit of using RAG is improving consistency in evidence retrieval rather than model hallucinations.

Our main hypothesis is that integrating LLMs, VLMs, RAGs, and computer vision techniques can reduce the manual effort required to create test reports without compromising report quality. By automating the evidence collection and test report creation steps of the manual testing pipeline, the system can enhance consistency and reduce effort.

E. Position of the Proposed System in This Context

The TestPal uses video recording of a test execution and test case specification documents as inputs to generate a structured test execution report with evidence for each test step. This process is illustrated in Fig. 1. The proposed model uses three main concepts to generate the test execution report; VLM, LLM and RAG.

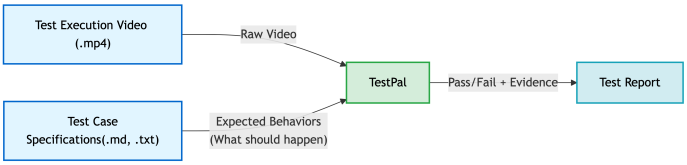

Fig. 1. TestPal Input - Output

We aimed to create an AI-powered application to automatically create test report without requiring testers to switch context between test execution and report creation and focus into their test execution part to improve the quality of the software project. The following chapter mentions details of the proposed approach.

III. Methodology

The proposed approach implements a pipeline architecture to automate test report generation for the manual testing process using VLMs, LLMs, and RAG. TestPal combines computer vision techniques for automated frame selection, a GPT-4 Vision model for understanding user actions, and the creation of textual representations of user actions, vector-based storing for retrieval, and an LLM for analysis of test step validation. This approach addresses the challenges of test report generation.

A. System Architecture

The architecture of the TestPal system, illustrated in Figure 2, comprises five interconnected modules that process screen recordings to generate automated test reports. Similar multi-stage architectures have proven to be effective in automated GUI testing contexts [16].

The pipeline has five main modules: the *Video Processing Module* filters raw screen recordings to retain only frames with significant semantic visual changes, the *VLM Analysis Module* analyzes frame pairs to user actions between each sequential pair, the *RAG-Based Storage & Retrieval Module* maintains a searchable vector space of detected user actions, the *Query & Analysis Module* validates detected user actions against test case specifications, and the *Report Generation Module* synthesizes findings into structured documentation. As an implementation detail, a gRPC-based server architecture is used for modular integration and enables distributed processing of computationally costly operations.

The following subsections elaborate on the technical implementation and theoretical foundations of each pipeline component.

B. Video Processing Module

The video processing module is implemented for reducing thousands of recorded frames to a set that captures meaningful events while minimizing computational cost. SSIM-based filtering is used to identify significant visual differences. The technique has proven effective for GUI testing where structural changes indicate meaningful user actions [17].

Mathematical Formulation: Let $V = \{F_1, F_2, \ldots, F_n\}$ represent the complete video sequence containing n frames. The frame retention process is defined as follows:

$$F_i \in V_{\text{retained}} \iff \text{SSIM}(F_i, F_{\text{prev}}) < \tau \tag{1}$$

where V_{retained} is the set of retained frames, F_{prev} is the most recently retained frame, and τ is the similarity threshold. The SSIM score is computed as:

$$\text{SSIM}(F_i, F_j) = \frac{(2\mu_i\mu_j + c_1)(2\sigma_{ij} + c_2)}{(\mu_i^2 + \mu_j^2 + c_1)(\sigma_i^2 + \sigma_j^2 + c_2)} \tag{2}$$

where μ_i and μ_j are the mean luminance values, σ_i^2 and σ_j^2 are the variances, σ_{ij} is the covariance, and c_1, c_2 are the stabilization constants.

SSIM-Based Frame Selection: Frame filtering operates through structural similarity analysis. For each frame in the video sequence:

1) Frame conversion from color to grayscale reduces computational complexity while preserving structural information essential for similarity assessment.
2) SSIM score computation between the current frame and the previously retained frame quantifies visual similarity based on luminance, contrast, and structural patterns according to Equation 2.
3) Threshold comparison: frames with SSIM scores exceeding the configurable threshold $\tau = 0.9$ are classified as near-duplicates and discarded according to Equation 1.
4) Retained frames are saved with sequential naming preserving temporal order necessary for action inference.

Wang et al. demonstrated that the SSIM metric is robust to minor pixel-level variations while remaining sensitive to structural changes in the UI [17]. This characteristic of the SSIM metric makes it ideal for frame reduction for GUI testing scenarios.

The approach successfully filters out non-informative variations while preserving frames that show meaningful events, such as window transitions, screen navigation, content updates, error messages, and UI element state changes.

In our study, the threshold value of $\tau = 0.9$ balances frame retention for accurate action detection against computational efficiency and is determined empirically. This filtering typically achieves compression ratios between 10:1 and 50:1, depending on the complexity of UI, reducing datasets from thousands of frames to dozens or hundreds of key frames.

C. VLM Analysis Module

The consecutive frame pairs are analyzed to identify and describe user actions that occurring between frames. Multimodal LLMs have demonstrated exceptional capabilities in understanding GUI semantics and understanding user actions from visual information [16]. Integration with Azure OpenAI's GPT-4 Vision API is achieved through a structured workflow.

Mathematical Formulation: Given the retained frame sequence $V_{\text{retained}} = \{F_1, F_2, \ldots, F_m\}$, consecutive frame pairs are processed to detect actions:

$$A_i = \text{VLM}(F_i, F_{i+1}), \quad i = 1, 2, \ldots, m-1 \tag{3}$$

where A_i represents the detected action between frames F_i and F_{i+1}, and VLM denotes the vision-language model inference function. Each action is represented as a tuple:

$$A_i = (\text{type}_i, \text{target}_i, \text{reasoning}_i, t_i) \tag{4}$$

where type_i is the action classification, target_i identifies the UI element, reasoning_i provides visual evidence explanation, and t_i is the temporal position.

Image Preparation: Frame pairs are encoded in base64 format, creating data URIs suitable for API transmission while maintaining visual fidelity necessary for accurate action detection.

Prompt Engineering: A structured prompt instructs GPT-4 Vision to function as a visual reasoning assistant, comparing two sequential screenshots to determine user actions. The prompt engineering approach draws on best practices from GUI testing applications where clear instruction specification improves model accuracy [16]. The prompt specifies the required output components:

- Action type classification (clicked, navigated, entered text, selected, scrolled)

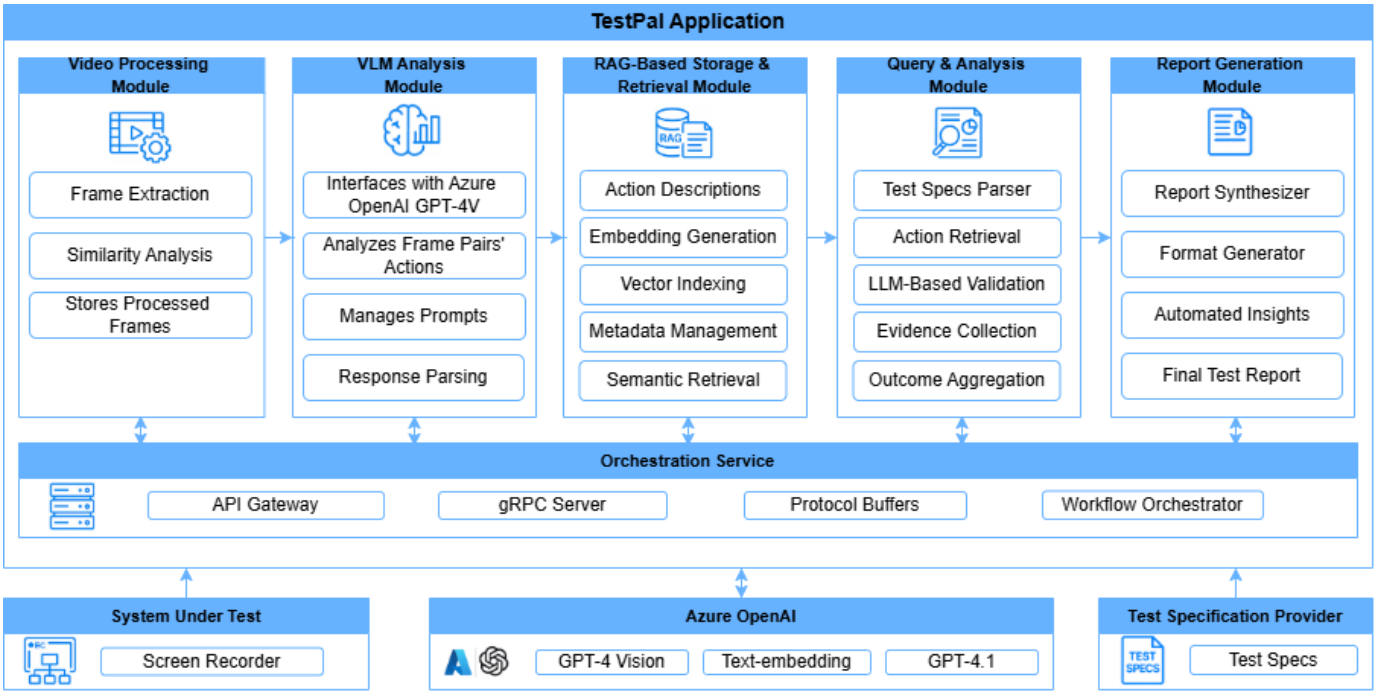

Fig. 2. TestPal System Architecture showing the complete pipeline from screen recording input through feature extraction, VLM processing, context memory (RAG-based storage), query analysis, and report generation

- Target identification (button labels, menu items, input field names, UI element descriptions)
- Reasoning explaining visual evidence supporting the conclusion
- Explicit null response when no meaningful action is detected between frames

API Configuration: The Azure OpenAI API parameters are configured to optimize consistency and accuracy for action detection:

- Model: GPT-4 Vision variants (gpt-4-vision-preview)
- Temperature: 0.2 to ensure near-deterministic, consistent outputs across multiple invocations while preserving natural language fluency
- Max tokens: 800, providing sufficient capacity for detailed action descriptions and reasoning
- Top-p: 1.0 without nucleus sampling restriction
- Frequency penalty: 0, Presence penalty: 0 to avoid biasing against repetitive but accurate descriptions

Response Processing: The API response contains structured action information that is parsed and stored through the RAG module. Each response contains the detected user action along with metadata, such as the frame number.

A comprehensive record of identified user actions for the test execution is produced once all consecutive frame pairs have been processed. This record serves as the foundation for test case validation and automatic report production.

D. RAG-Based Storage and Retrieval Module

Vector-based indexing is implemented to enable semantic search and retrieval of detected actions. The architecture separates concerns between low-level vector operations and high-level storage management. This approach is inspired by context memory patterns used in multi-agent testing systems, which maintain hierarchical information structures to support reasoning and decision-making [16].

Mathematical Formulation: Let $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$ represent the set of all detected actions. The embedding function is defined as:

$$\mathbf{v}_i = E(A_i) \in \mathbb{R}^d \tag{5}$$

where E is the Azure OpenAI text embedding model and d is the embedding dimension. A vector index $\mathcal{V} = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}$ is maintained to support semantic retrieval. For a query q, similarity is computed as:

$$\text{sim}(q, A_i) = \frac{\mathbf{v}_q \cdot \mathbf{v}_i}{\|\mathbf{v}_q\|\|\mathbf{v}_i\|} \tag{6}$$

where $\mathbf{v}_q = E(q)$. The most relevant top-N actions are retrieved as:

$$\mathcal{R}(q, N) = \{A_i \mid \text{sim}(q, A_i) > \theta, \text{ ranked by sim}\} \tag{7}$$

where θ is the minimum similarity threshold and $|\mathcal{R}(q, N)| \leq N$.

Embedding Generation: Action descriptions, which are created by the VLM analysis module, are converted to vector representations using Azure OpenAI's text embedding models (text-embedding-ada-002). This transformation enables semantic similarity comparisons, which is better than just word-based comparisons. So, the retrieval of conceptually similar

actions occurs even when expressed with different terminology.

Indexing vectors: To facilitate effective nearest-neighbour search across user action embeddings, an FAISS (Facebook AI Similarity Search) index is kept up to date [18]. Even with repositories containing thousands of detected actions, FAISS's optimised algorithms for high-dimensional vector operations enable sublinear search complexity.

Metadata Management: Each stored action maintains associated metadata enabling retrieval and analysis:

- Frame pair identifiers linking to visual evidence
- Temporal information (frame sequence numbers)
- Visual reasoning and confidence indicators

Semantic Retrieval: Query interfaces are exposed that accept test case requirements. To ensure consistency with queries and the user actions in the vector space, the same embeddings were used to convert the test case specifications to word embeddings. Cosine similarity comparisons are performed against stored action embeddings using Equation 6, returning the top 5 ranked results with their relevance scores. Configurable similarity thresholds filter low-confidence matches, ensuring that retrieved actions align semantically with the query.

Test validation workflows are made possible by this RAG architecture, which uses semantic similarity instead of keyword matching or regular expressions to match detected actions with expected behaviours from test case specifications. The method addresses conceptual equivalency between test case languages, terminology variations, phrasing differences, and abstractions in test case specifications. Test case requirements are accepted by query interfaces that are made available. The test case specifications were converted to word embeddings using the same embeddings to guaranty consistency with queries and user actions in the vector space.

E. Query & Analysis Module

Test case validation is implemented by matching detected actions against test case specifications and determining pass/fail outcomes through LLM-based reasoning. This multi-stage verification approach ensures comprehensive validation similar to the mechanisms used in automated testing frameworks [16].

Test Case Processing: To extract specific test steps along with their expected behaviours, validation standards, and acceptance thresholds, the test specifications are parsed. Every test step is transformed into a semantic query that works with the RAG retrieval system.

Action Retrieval: For each test step, the vector space is queried using the expected behavior description and test case specification. The semantic search returns ranked lists of detected actions with similarity scores, ordered by relevance to the test requirement, as in Equation 7. Configurable thresholds determine the minimum similarity required for a detected action to be considered a potential match.

LLM-Based Validation: Retrieved actions are analyzed using LLM reasoning to determine whether they satisfy test requirements. The validation process employs GPT-4 (gpt-4-turbo) configured for analytical tasks:

1) The test step requirement and retrieved action description are provided to the LLM
2) The model evaluates semantic alignment, considering action type, target elements, and context
3) The LLM determines pass/fail status with confidence scoring
4) Reasoning is generated explaining the determination, highlighting matches or identifying deviations

This LLM-based approach handles variations in terminology and accounts for conceptually equivalent actions expressed differently in test specifications versus detected behaviors. For example, a test requirement to "click the Submit button" would match a detected action "user clicked the Send button" if the LLM determines contextual equivalence.

Evidence Collection: For each test step determination, supporting evidence is collected, including:

- Frame pair screenshots showing the visual state before and after the action
- Action description and reasoning from the VLM analysis
- Explanations for failing steps

Outcome Aggregation: Individual test step results are aggregated to produce overall test case outcomes. Pass/fail statistics are computed, and explanations are generated for each test case.

F. Report Generation Module

The analytical results are synthesized into structured test report documentation suitable for review and compliance purposes.

Report Synthesizer: Generated reports contain the following:

- Overall pass/fail label
- Test case results for each test step
- Detailed explanations of seen behavior and comparison with the expected behavior
- Visual evidence galleries with related frames

Format Support: Markdown output format is currently supported in this study, because of its flexibility in integration with other systems.

G. Implementation Technologies

The system is implemented in Python 3.9+ with the following key technologies:

- **Computer Vision:** OpenCV 4.8+ for video processing and frame extraction; scikit-image 0.21+ for SSIM computation
- **AI Services:** Azure OpenAI SDK (openai 1.3+) for GPT-4 Vision (action detection), text embeddings (text-embedding-ada-002 for vector generation), and GPT-4 (validation reasoning)
- **Vector Operations:** FAISS 1.7+ for vector indexing and similarity search [18]

- **Communication:** gRPC 1.59+ for client-server protocol; Protocol Buffers 4.24+ for structured data exchange
- **Numerical Processing:** NumPy 1.24+ for array operations and mathematical computations
- **Report Generation:** Markdown libraries for document formatting; Pillow 10.0+ for image processing

H. Orchestration Service

The development of independent components, testing, and deployment are made possible by the modular architecture. Distributed processing is enabled by the server-based architecture, which allows clients to submit recordings via standardized interfaces while computationally intensive tasks (such as vector indexing and VLM inference) are carried out on specialised hardware.

IV. Experiments

A. Problem Statement

To evaluate the performance of the TestPal, we specified requirements representing its capacity to recognize user interactions and to detect information in both CLI and GUI contexts, or the lack thereof:

R1 Recognize commands put in the terminal.
R2 Detect when CLI commands are skipped.
R3 Recognize the output of commands in the terminal.
R4 Recognize when the GUI application is opened.
R5 Recognize substantial GUI changes.
R6 Recognize textual information present in the GUI.
R7 Recognize information being put into the GUI.
R8 Detect when buttons are pressed in the GUI.
R9 Detect when a button press is skipped.
R10 Detect information not being put into the GUI.

Test cases are defined as a list of test steps and corresponding expected results. The TestPal produces a report containing a OK/FAIL assessment and the reasoning behind it for each test step. Table I shows an instance of a generated test report. The values therein ("A", "100") are arbitrary. LLMs are known to produce hallucinations, so we inspect the reasoning to ensure it is coherent. Current AI models are nondeterministic even when configured to minimize randomness [19]. For this reason, we also assess the consistency of the outputs through multiple test runs.

B. System Under Test

The SUT is composed of mock CLI scripts and a simple GUI application, implemented using Tkinter. The former represent querying the version of a system, performing an upgrade, and starting the GUI application. These are simple shell scripts that echo a fixed output.

The latter presents multiple widget types to enable assessing the TestPal's capacity to recognize and to interpret the relation between them. The example report in Table I is the result of submitting a form with three text input fields: Batch, Expected, and Actual. Doing so adds an entry to a table showing the inputs and an Accepted/Rejected result. Any errors are logged to a file and may also be searched for in the GUI.

C. Test cases

We specified six test cases for the TestPal, each of them based on one of three possible scenarios. The test cases consist of executing one of these scenarios or a variation thereof where a step is skipped. We repeat each test case three times and compare their results to evaluate consistency.

Scenario 1 consists of filling in the GUI form, clicking the button to submit it and checking that no error was triggered. **Scenario 2** mixes CLI and GUI steps and requires leaving input fields empty, triggering an error when processing the form, which must then be found in the logs. **Scenario 3** repeats CLI and GUI steps with multiple expected outputs and is meant to test whether the length of the test affects the results.

D. Results

Each test step is associated to one or more requirements. We considered a failed test step to have failed all associated requirements regardless of the reasoning. This is because once the result or reasoning is incorrect, even the parts of the reasoning that seem to fit may well be hallucinations.

Table II shows for each requirement: how many test steps are associated with it, how many of them pass or fail, and the accuracy as the portion of steps that passed relative to the total. In this context, passing means that the model produced the expected OK/FAIL result along with a coherent explanation.

The poor performance in identifying the fullfillment of *R4*, the GUI application being opened, owed to ambiguous word choice in specifying this step: "Start the system". This was mistaken for another step that was also present: "Initialize subsystems", referring to initializing subsystems using a button in the GUI. This is also the reason for the failed steps relative to *R1*, as the application was started from the terminal. This result indicates a limitation of the model in interpreting the test specifications depending on the level of abstraction. It also suggests that care must be taken when choosing how to write them until this limitation is overcome.

R6 and *R10* had the least consistent results, as the model alternated between correct and incorrect results for the same test steps. The low accuracy also indicates two limitations in the model: reading static data from the GUI and detecting when text inputs are skipped. In the instances of the test executions where the text input fields were explicitly highlighted and left empty, the model was more likely to detect it correctly. It could also be considered a matter of abstraction, whether it is easier to interpret an explicit instruction not to perform an action or simply having no instruction at all. *R2* and *R9* also refer to the absence of interaction, but they had the least representation in our test cases and their results, though positive, are the least significant.

On the positive side, *R5*, *R7*, and *R8* had more significant representation and accuracy above 90%. These requirements represent the most common operations: to fill and submit GUI forms. When submitting the forms provides instant visual feedback, this is also detected by the model. However, when

TABLE I
GENERATED TEST REPORT INSTANCE

Step	Action	Expected Result	LLM Response
1	Set the input for Batch to “A”	Batch is A	Yes, the input for Batch was set to “A” in the video. According to the provided context, in Frame 0 and Frame 1, the ”Batch” field changes from empty to ”A”, indicating the user typed ”A” into the ”Batch” field. This fully matches the expected result of Batch being ”A”. OK
2	Set the inputs for Expected and Actual to the same value	Expected and Actual have the same value	Yes, the inputs for Expected and Actual were both set to ”100” (Frame 0–Frame 1: Expected=”100”; Frame 1–Frame 2: Actual=”100” entered), and after processing (Frame 2–Frame 3), both values appear as ”100” in the table, matching the expected result. OK
3	Execute the Process	Batch A is accepted	Yes, the process was executed for Batch A with ”Expected” = 100 and ”Actual” = 100. The table shows ”Acc/Rej” as 1/0, meaning Batch A was accepted, which matches the expected result. OK
4	Verify that there is no error in the log	No error found.	There is no evidence in the provided data that ”Verify that there is no error in the log” was checked or that ”No error found” was confirmed as an expected result. No explicit verification or comparison for errors in the log is described in the provided frames. FAIL

TABLE II
RESULTS PER REQUIREMENT

Req	Steps	Pass	Fail	Accuracy
R1	21	15	6	71%
R2	3	3	0	100%
R3	12	12	0	100%
R4	15	0	15	0%
R5	24	23	1	96%
R6	30	17	13	57%
R7	42	42	0	100%
R8	54	49	5	91%
R9	3	3	0	100%
R10	24	16	8	67%
Total	228	180	48	-
Average	23	18	5	79%

the cause and effect are separated in time, it may not associate the events.

V. RELATED WORK

Research on automating testing and reporting has evolved in multiple domains, including report generation, GUI automation, and LLM-based workflows. Existing approaches primarily focus on reducing manual effort through GenAI, RAG, and multi-agent systems, often targeting mobile or web environments. Although these solutions demonstrate the potential of AI in testing and reporting. We review key contributions in report generation, GUI automation testing, and LLM with RAG-driven testing.

Report Generation. Gupta et al. [4] propose an LLM and RAG-based system that automates data extraction and chart updates in government reports, significantly reducing manual effort and error rates. Similarly, Modak [20] demonstrates how GenAI can streamline the creation of business reports, including financial, market, and customer feedback reports, in various industries. In contrast, our approach focuses on automating the analysis and reporting of GUI software tests using screen recordings.

GUI Automation Testing. To reduce the manual testing overhead in UI automation testing, the solutions proposed by Feng et al. [21] and Sampath et al. [22] combine Generative AI, RAG, and local LLMs to generate and maintain test scripts. Yu, Shengcheng, et al. [14] introduced ScenGen, a multi-agent framework that uses LLMs to automate scenario-based GUI testing for mobile applications. Extracting and understanding the semantics of GUI elements from an application, as well as bridging the gap between GUI testing and business logic, are particularly relevant to our work; however, their focus was on mobile apps rather than desktop or multi-system environments. AUITestAgent by Zhang et al. [23] proposes an automated GUI testing framework for mobile apps that focuses on automatically executing GUI tests based on requirements, whereas our approach focuses on automating report generation after manual execution.

LLM-Based Reporting. Franzosi et al. [24] propose an approach that combines multiple LLM-based agents for tasks such as widget detection, business logic planning, operation execution, and scenario matching, ultimately generating an HTML report with test summaries and action sequences. However, their work is limited to augmented testing using the Scout tool [25] in mobile app environments. In contrast, our approach focuses on automated report generation from test cases and screen recordings captured during test execution.

RAG for Testing. Wang et al. [15] propose an AI-driven testing system that synchronizes bug detection with code updates using LLMs and RAG, achieving a 10.5% higher user acceptance rate. This highlights RAG’s role in improving testing workflows and aligns with our approach to contextual analysis.

These studies show significant progress in test generation, execution, and reporting; we found a lack of research addressing scenarios similar to ours. The reviewed approaches are only partially relevant, as they differ in scope and objectives, which highlights the novelty of our approach.

VI. CONCLUSION

In this paper, we propose TestPal, a combination of VLM, LLM, RAG, and computer vision techniques to address two main limitations of test report creation processes: the effort required to create test reports and context switching during manual test execution. Based on our experimental setup, detecting common actions (button clicks, tab switching, screen changes, filling GUI fields, and using the command-line tool) on the GUI and aligning the observed behaviors with the

expected results in test case specifications is around 90% accurate. It shows that our proposed model is a good candidate for common user actions on GUI tools. Besides that, our results show that TestPal is less effective at capturing static data from the GUI. The accuracy of capturing static data ranges from 57% to 67%. It clearly shows that our proposed feature extraction method is too tight to capture user actions on the GUI. Finally, the worst results are in the test cases which are defined more abstractly than clear expectations that can be seen directly in the GUI. The diversity of our accuracy ratios across these related requirements is too high as well; they range from 0% to 71%. It also indicates this topic can be covered in future work. Since the test case specifications are written in natural language, the model should be able to correctly interpret abstract instructions.

VII. Future Plans

In this paper, we focus on an intermediate step: automatically analyzing executed GUI tests and creating a test report. Our long-term goal is to extend this to recognize ambiguous test cases, execute GUI tests from test specification documents automatically, and create a test report. This would reduce human involvement in manual testing to critical parts, allowing more focus on higher-level tasks, like designing test specifications.

To recognize ambiguous test cases, we plan to integrate the Model Context Protocol (MCP) [26], enabling TestPal to access external domain knowledge sources for enhanced contextual understanding.

To automatically execute test specifications, the TestPal application should have the ability to operate the computer. The most promising method looks like creating a computer user agent with OmniParser V2 [27]. In future works, adding the enhanced extractor model into our pipeline can help to understand static data on GUI widgets and enable operating a computer.

In addition to our long-term goal, creating test cases to assess the current level of abstract instruction understanding of our model would be a good enabler study to improve the level of abstract instruction understanding.

References

[1] F. Giertzsch, M. Blecken, and R. God, "Towards a model-based systems engineering framework for the design and configuration of communication networks in a data-driven and interconnected aircraft cabin," in *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, 2024, pp. 216–226.

[2] S. Pasupuleti, "Robotic process automation for enhancing workflow automation in multi-system environments," *IJAIDR-Journal of Advances in Developmental Research*, vol. 15, no. 1.

[3] M. Baqar and R. Khanda, "The future of software testing: Ai–powered test case generation and validation," in *Intelligent Computing-Proceedings of the Computing Conference*. Springer, 2025, pp. 276–300.

[4] R. Gupta, G. Pandey, and S. K. Pal, "Automating government report generation: A generative ai approach for efficient data extraction, analysis, and visualization," *Digital Government: Research and Practice*, vol. 6, no. 1, pp. 1–10, 2025.

[5] J. Gait, "A probe effect in concurrent programs," *Software: Practice and Experience*, vol. 16, no. 3, pp. 225–233, 1986.

[6] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs." in *OSDI*, vol. 8, no. 2008, 2008.

[7] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[8] M. L. Siddiq, A. Islam-Gomes, N. Sekerak, and J. Santos, "Large language models for software engineering: A reproducibility crisis," *arXiv preprint arXiv:2512.00651*, 2025.

[9] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge: Cambridge University Press, 2016.

[10] E. Enoiu, D. Sundmark, A. Čaušević, and P. Pettersson, "A comparative study of manual and automated testing for industrial control software," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 412–417.

[11] K. S. Thant and H. H. K. Tin, "The impact of manual and automatic testing on software testing efficiency and effectiveness," *Indian journal of science and research*, vol. 3, no. 3, pp. 88–93, 2023.

[12] E. Enoiu, D. Sundmark, A. Čaušević, and P. Pettersson, "A comparative study of manual and automated testing for industrial control software," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 412–417.

[13] Z. Q. Zhou, A. Sinaga, W. Susilo, L. Zhao, and K.-Y. Cai, "A cost-effective software testing strategy employing online feedback information." *Inf. Sci.*, vol. 422, pp. 318–335, 2018. [Online]. Available: http://dblp.uni-trier.de/db/journals/isci/isci422.html#ZhouSSZC18

[14] S. Yu, Y. Ling, C. Fang, Q. Zhou, C. Chen, S. Zhu, and Z. Chen, "Llm-guided scenario-based gui testing," *arXiv preprint arXiv:2506.05079*, 2025.

[15] Y. Wang, S. Guo, and C. W. Tan, "From code generation to software testing: Ai copilot with context-based rag," *IEEE Software*, 2025.

[16] S. Yu, Y. Ling, C. Fang *et al.*, "Llm-guided scenario-based gui testing," *IEEE Transactions on Software Engineering*, 2025.

[17] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, 2004.

[18] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, 2019.

[19] H. He and T. M. Lab, "Defeating nondeterminism in llm inference," *Thinking Machines Lab: Connectionism*, 2025, https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/.

[20] R. Modak, "Generative ai for automated business report generation and analysis," *World Journal of Advanced Engineering Technology and Sciences*, vol. 15, no. 2, pp. 10–30 574, 2025.

[21] S. Feng, H. Lu, J. Jiang, T. Xiong, L. Huang, Y. Liang, X. Li, Y. Deng, and A. Aleti, "Enabling cost-effective ui automation testing with retrieval-based llms: A case study in wechat," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1973–1978.

[22] V. Sampath, "Transforming ui automation testing with genai, rag, and local llms," https://www.altimetrik.com/white-papers/ui-automation-genai-rag-local-llms, 2025.

[23] Y. Hu, X. Wang, Y. Wang, Y. Zhang, S. Guo, C. Chen, X. Wang, and Y. Zhou, "Auitestagent: Automatic requirements oriented gui function testing," *arXiv preprint arXiv:2407.09018*, 2024.

[24] D. B. Franzosi, E. Alégroth, and M. Isaac, "Llm-based labelling of recorded automated gui-based test cases," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2025, pp. 453–463.

[25] M. Nass, E. Alégroth, and R. Feldt, "On the industrial applicability of augmented testing: An empirical study," in *2020 ieee international conference on software testing, verification and validation workshops (icstw)*. IEEE, 2020, pp. 364–371.

[26] X. Hou, Y. Zhao, S. Wang, and H. Wang, "Model context protocol (mcp): Landscape, security threats, and future research directions," 2025. [Online]. Available: https://arxiv.org/abs/2503.23278

[27] Y. Lu, J. Yang, Y. Shen, and A. Awadallah, "Omniparser for pure vision based gui agent," 2024. [Online]. Available: https://arxiv.org/abs/2408.00203