Agentic Vision-Language AI System for Autonomous Desktop Application Testing

Adithya T S, Dr Muruganantham B

Department of Data Science, SRM Institute of Science and Technology, Chennai, India

tsadithya24@gmail.com, muruganb@srmist.edu.in

ABSTRACT

This work presents TestAny, a desktop User Interface (UI) testing application that reduces the effort involved in manual testing and test creation. The software combines multiple technologies, including computer vision, optical character recognition (OCR), and retrieval-augmented generation (RAG), to generate test cases from the test application's official user manuals. The application to be tested is embedded in our system, and its interface is captured through screenshots. From the captured screenshots, the system attempts to find the clickable elements and read any visible labels on the screen. The extracted information is then matched with the relevant portions of the provided user manual, which helps guide the generation of tests to be executed. This combined use of visual and textual references will help us produce more meaningful test cases than relying on a single source of information. Another advantage is that the workflow reduces the amount of manual scripting required and keeps the generated tests traceable to the user manual content. Since this method avoids strict UI selectors, it is also less sensitive to minor UI layout changes. In initial experiments with the sample test application, the UI elements were detected with reasonable accuracy, and OCR correctly identified the labels in most cases. The generated test plans also closely followed the intent of the documented procedures. This paper outlines the system design, main implementation, and overall integration flow, and lays the groundwork for a more detailed evaluation in future work.

***Keywords*:** UI testing, computer vision, Optical Character Recognition (OCR), Retrieval Augmented Generation (RAG), test case generation, and software quality.

1. Introduction

Most current testing applications have a graphical user interface (GUI) to improve user interaction with the application. As a result, UI correctness directly impacts usability, safety, and overall product reliability. The issues will become more critical in enterprise-level software and in camera or imaging applications if the product is released without proper testing, as UI actions may directly control the hardware. UI testing is still one of the more human interactions needed as part of software quality assurance. Automated UI testing during the older days usually depended on test scripts that were manually written for actions that needed element locators, such as XPath, automation IDs, or the screen or UI element coordinates to be fixed. Changing the UI will make the test scripts fail or prevent them from executing the tests properly. This method doesn't work even with a small layout adjustment, localisation updates, or minor visual changes, and it makes the test fail. The traditional methods only work if the UI remains stable and unchanged. Whenever there is a small change in layout, an update to localisation, or any slight modifications to visuals, this method fails the test due to those UI changes. Traditional testing methods yield reliable results only when the user interface (UI) remains constant. Research studies have demonstrated that conventional rule-based automation frameworks do not perform well with rapidly changing software systems or dynamic interfaces [2]. As a result, Testing teams often spend significant time maintaining test scripts rather than improving test coverage. Most recent software products include detailed user manuals or technical documentation for using the software. These documents describe the expected behaviour, explain the different controls for functions, and outline common workflows. In current UI automation testing practices, this information is used only to a limited extent. As a result, automated tests may fail to test some important documented procedures or scenarios, or

Proceedings of ICICCS-2026 (Poster Papers), DMPedia Lecture Notes in Computer Science & Engineering.
ISBN: 978-81-993813-1-5

slowly diverge from the software's intended use cases. To address this gap, we have found a solution: TestAny software, a desktop UI testing tool that combines visual UI analysis, Optical character recognition-based text extraction, and testing using the provided user manual and RAG models. We have implemented a tool that runs the target application to be tested, captures the screen of the application, and identifies the region that appears to have interactive UI elements. It also reads the UI elements' labels and finds the relevant content from the provided user manual for forming the test cases to be tested. Our goal is to completely reduce the manual scripting effort while keeping the automated tests aligned with the official documentation. A new and more advanced technique for analysing the UI of an application has been proposed based solely on the screenshot data, and neither humans nor pre-defined selectors will be used as determining factors for identifying interactive elements. An OCR method has also been incorporated to identify all text associated with each interactive element, and a RAG model is included to assist in retrieving only those documents that will be used in the test creation process. Finally, a new desktop application workflow has been created to enable seamless integration of application embedding, UI analysis, and automated test case generation in a single environment.

The rest of the paper is organised as follows: Section 2 presents related work, and Section 3 presents the proposed methodology. Section 4 discusses the results and discussion, and finally, Section 5 concludes the paper.

2. Literature Review

The evolution of UI test automation has taken us from simple record-and-playback scripts to the development of a wide range of sophisticated frameworks for scalable/maintainable test suites. Traditional automation techniques primarily rely on application-specific identifiers (automation IDs, control names, or accessibility trees) for determining which UI element to interact with. These techniques tend to work very well when the UI is stable and the UI elements have not been modified. However, when the interface has frequent layout or theme changes, these types of techniques can become very brittle and not function as expected. Research has also shown that the continued dependence upon conventional rule-based automation frameworks is increasingly resulting in an inability for these frameworks to keep pace with rapidly changing technologies and the dynamic nature of modern software systems [1], [2]. Therefore, there has been a significant amount of research focused on developing intelligent/adaptive testing approaches. Model-based testing (MBT) has also been considered as one possible approach to generating UI test paths. MBT has the potential to improve test coverage of the UI. However, the overhead associated with generating and maintaining accurate models continues to be a challenge and may require significant manual effort. As reported in recent surveys, many traditional automation techniques do not scale and/or provide sufficient flexibility in rapidly changing software environments [2].

In recent years, many researchers have moved toward using computer vision techniques for UI comprehension to surpass the limitations of metadata-based automation. Vision-based methods treat a given UI as an image and employ object detection networks to extract UI components from their images directly (i.e., screenshots). These vision-based methods allow for the automation of tasks even if internal properties of the UI are not available. Nevertheless, pure vision methods still suffer from challenges that stem from a lack of understanding of semantics; they can localise UI components accurately. Still, they often cannot determine the function of a given component because they lack contextual knowledge about the UI. Finally, the landscape of LLM-assisted software testing also notes that GUI comprehension is particularly challenging; existing approaches lack sufficient semantic awareness [3].

Using Optical Character Recognition (OCR) with UI automation enables the extraction of text from button labels, menus, and runtime status messages, for example. Connecting extracted

text to detected UI components enables automation to associate UI components with meaningful actions better. AI-driven testing frameworks are integrating visual and text understanding solutions to enhance their capabilities and reduce reliance on weak selectors [1]. However, OCR accuracy decreases in challenging scenarios such as low contrast, varied fonts, and localisation.

Today's intelligent test systems rely heavily on Natural Language Processing (NLP). The development of Transformer architectures (Bert [7]) enabled the creation of large-scale language-based applications for numerous software engineering tasks, as well as the understanding and generation of language. BERT [6] demonstrates how deep bidirectional pre-training of the model architecture produces highly robust and generalizable contextual representations that can easily transfer across downstream tasks. Many prior sequence-to-sequence learning works laid the groundwork for using neural networks to map structured input to structured output. This has served as the basis for all automated test generation pipelines to date. The most recent addition to this space is the use of Retrieval-Augmented Generation (RAG) [5], a new, effective approach that utilises both parametric language models and external knowledge retrieval to improve the factual accuracy of generated content. This mapping technique will prove invaluable in documentation-focused testing.

Recent advances in the capabilities of Large Language Models (LLMs) indicate that they can significantly improve software testing. An extensive survey shows that LLMs are widely used for unit test generation, bug finding, and bug correction, but they still face challenges in reliability and coverage [3]. Various empirical studies conducted to evaluate LLMs' ability to generate tests reveal that LLMs perform less well for complex reasoning tasks. Multi-agent approaches have been put forth to overcome this limitation. Agent Coder provides an example of a collaborative framework in which 3 agents (Programmer, Test Designer, and Test Executor) work together to iteratively refine the generated code through feedback loops [9]. The framework shows that having separate agents for generating tests both improves the objectivity of the generated tests and increases overall test coverage.

Recent research highlights desktop GUI environments. An example of research is WorldGUI, which is investigated in detail as a large-scale benchmark with commonly used desktop applications and then assesses the robustness of agents with different initial interface states [10]. The results of this research show that when workflows are initiated from a state other than the default state, current GUI agent performance deteriorates substantially, confirming that existing approaches to GUI automation are fragile. Further research on WorldGUI-Agent has also produced a framework that includes multi-stage critique processes to increase the reliability of dynamic GUI environments. These critique processes include post-planning critiques, pre-action validations, and post-action evaluations [10]. Overall, these findings demonstrate the need for more adaptive, context-aware desktop automation systems.

Several surveys conducted this past year focus on the impact of AI (AI4SE) on the Software Development Life Cycle (SDLC). They show that test automation using AI technologies (ML, DL, RL, LLMs) is advancing to encompass all phases of the SDLC (test planning, test generation, test execution, and test maintenance) [2]. Using AI for testing is also leaning more toward frameworks that incorporate self-healing or autonomous testing through continuous learning from execution history. In addition, analyses show a shift from static, rule-based, scripted test cases to self-healing, or autonomous, testing frameworks that learn from past execution history [1]. However, there are still many unresolved challenges associated with this new technology, such as explainability, dependence on the amount of training data used, and smooth integration with CI/CD [1].

While previous research has produced significant advances in areas such as UI detection, OCR extraction, LLM-based testing, and agentic automation, fully integrated end-to-end desktop testing systems remain limited in number. Historically, there has been more emphasis on

developing isolated components of a testing system instead of developing unified pipelines. Benchmark studies have also shown that even the best-researched GUI agents fall short of providing robust performance in real desktop environments and very often struggle to handle a variety of dynamically changing workflow states [10]. There is also evidence of the need for structured feedback loops in multi-agent code-generation systems for GUI-driven testing workflows. Still, these systems have not been directly designed for these types of testing workflow [9].

TestAny seeks to fill these gaps through four integrated components:

1. Vision-based UI understanding
2. Semantic extraction using OCR
3. Reasoning through RAG-based documentation
4. Multi-agent orchestration of testing

The main objective of this integrated architecture is to provide a more robust and scalable desktop UI testing solution than what is currently available using fragmented approaches.

3. Proposed Methodology

3.1 System Overview

TestAny is an application that mainly consists of 5 stages. Figure 1 illustrates the complete architectural pipeline of the automated UI testing system from application initialisation and screen capture to intelligent UI analysis, test planning, execution, and report generation.

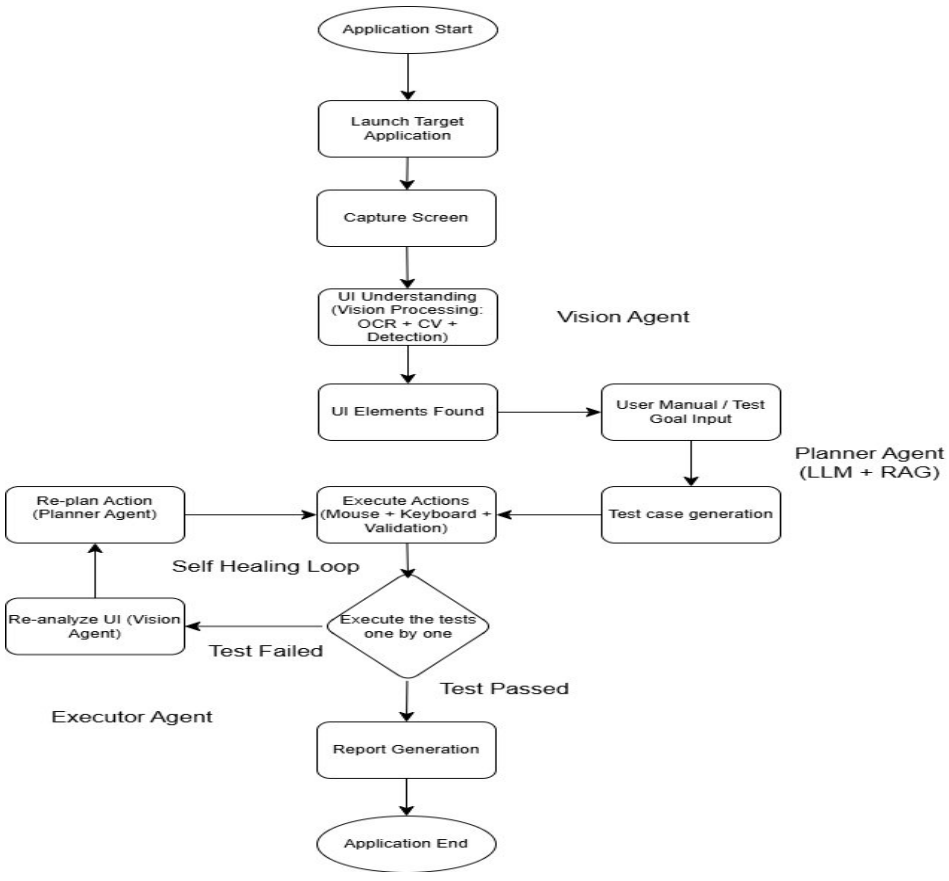

Fig. 1. System Architecture diagram.

3.1.1 Application Embedding

The application to be tested can be embedded in our application's main window interface. This enables the application to automatically interact with the target application and provide our test results.

3.1.2 UI Analysis

Once the application is embedded in our TestAny application, the Launch application button will be enabled. Clicking on the button will launch the application in our embedded window allocated for the target application. Upon successful application launch, the Analyse UI button will be enabled, allowing you to capture a screen from the embedded window. A vision pipeline will detect the UI elements that can be used for our test case generation. The graphical user interface of the TestAny automated UI testing tool is shown in Figure 2

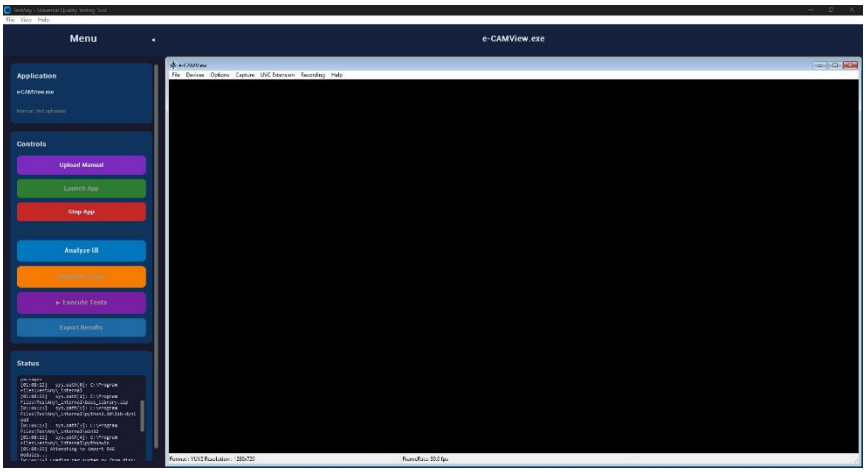

Fig. 2. Desktop application embedding.

3.1.3 OCR Text Extraction

The captured screenshot will be passed to optical character recognition models and algorithms for extracting words/texts from the screenshot. This will produce the word found within a rectangle border highlighted. The system maps the text to UI elements based on spatial proximity. The pre-processed screenshot, which removes the application UI and shows only the embedded application, is shown in Fig 3

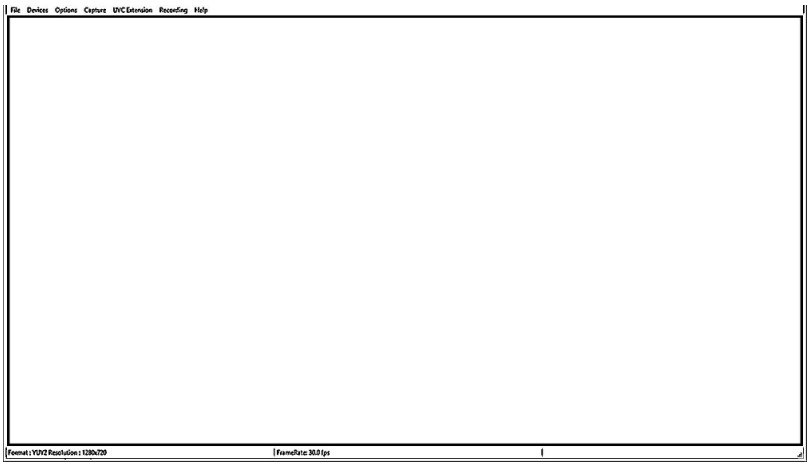

Fig. 3. Pre-processed image before analysis.

3.1.4 RAG Test Generation

The user manual for the application to be tested can be uploaded to our application. Our RAG model will store the text along with vector embeddings in the Chroma vector database and retrieve the relevant section from the user manual.

3.1.5 Execution of Generated Tests

The generated test cases will be sent to the model, which will use the Windows-provided APIs to control mouse clicks and keyboard input events. The mouse click event will be automatically triggered, and the next action will continue until the test results are generated and passed.

3.2 UI Element Detection Pipeline

The UI analysis process will begin with the captured screenshot of the window to be tested. Then the screenshot will be sent to a set of preprocessing steps before test case generation. Preprocessing steps include text resizing, contrast adjustment, normalisation, and noise reduction. The detection model will identify the UI elements and components, and assign the components to the

1. Box covering the text with the coordinates of the UI element area
2. UI element type (button, text box, dropdown box, etc.)
3. Confidence scores

These metrics will be stored as structured metadata and used as a verification tool.

3.3 OCR Extraction and Label Mapping

Optical character recognition (OCR) extracts text and other strings from captured screenshots. The model then associates the recognised texts with the UI element found in the nearest coordinate with the bounding box. This further produces the semantic labels for UI elements and enables test steps, for example:

1. Click the “Start stream” button.
2. Enter a value in the “Resolution” field.
3. Select “Device Settings” from menu options.

The OCR layer uses PaddleOCR models for their excellent UI detection performance and support for other languages. The bounding box is drawn for the detected UI elements using OCR, as shown in Figure 4

Fig. 4. UI Detected from the screenshot captured.

3.4 Manual Processing and Retrieval

The User manual (PDF) is processed by:

1. Text extraction and cleaning
2. Chunking them into sections
3. Embedding using sentence-transformers
4. Storing embeddings in a vector database (Chroma DB)

When generating tests, the application builds a query based on the UI element context and finds the top-k related document chunks. This context is passed to the AI Generation models to ensure that the test scripts reflect what is specified in the user manual.

3.5 Test Case Generation

The test generation module produces a structured test plan with the following details:

1. Test case ID
2. Test Title
3. Preconditions
4. Steps to perform the test
5. Expected results

The output is saved in JSON format for easier integration, allowing the executor agent to run the tests one by one with the test case ID automatically. The use of user manual contexts will enable the model for traceability and reduce hallucinated requirements.

3.6 Test executioner Agent

The executor agent will parse the generated JSON file, count the test cases, and, based on the test case ID, automatically execute mouse clicks and keyboard inputs in ascending order of the test case ID.

1. Find the coordinates of the UI element.
2. Moves the mouse automatically to that position and triggers a click event.
3. The screenshot will be captured again after the mouse clicks to capture the post-click event.
4. If the post, click event screenshot content matches the expected results from the generated test cases, then the test is allocated a PASS.

Once all the tests are complete, it will generate a report with the number of passed and failed tests. We can view all the reports and the intermediate output screenshots in the output folder.

3.7 Test completion and Report generation

The report shows the target application's test results, with screenshots of the before and after states after mouse clicks and keyboard inputs. Some of the contents included in the report are:

1. Execution ID
2. Timestamp
3. Application name
4. Total number of tests
5. Count of Passed tests
6. Count of Failed tests
7. Tests resulted in warnings
8. Total time taken for each test
9. Screenshots of before and after the event triggered

4. Results and Discussion

The proposed AI–driven autonomous testing framework was tested with a desktop application using the automatically generated test cases from the UI Analysis, along with the user manual guidance. The vision agent successfully extracted UI elements from the captured screenshots using OCR and Computer Vision (CV) models. It successfully identified the textual menu options/components available in the application menu bar, such as File, Devices, Options, UVC Extension, Recording, and Help. A total of 7 test cases were generated by the Planner Agent and executed one by one by the Executor Agent. Figure 5 shows the execution results, indicating that 4 test cases passed successfully. In comparison, 3 test cases resulted in warnings due to a lack of visible UI changes, with no complete failures observed.

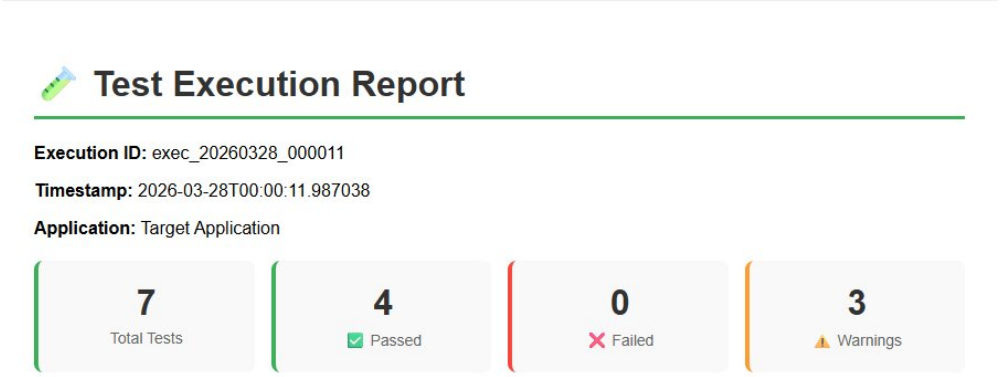

Fig. 5. Test Execution Report.

The tool enables automatic interaction with UI elements by calculating clickable coordinates and performing actions, such as menu selection. Figure 6 shows each test execution, with before-and-after screenshots, to validate UI state transitions and ensure the traceability and reproducibility of results.

Fig. 6. Test Execution Report showing PASS results with before and after screenshots

The results confirm that integrating vision-based perception with LLM-driven planning enables effective automated testing of desktop applications without relying on source code or predefined scripts. The results obtained in the experiments reveal the pros and cons of the proposed system. The Vision Agent correctly identifies UI elements and extracts text with high confidence, enabling the Planner Agent to generate relevant test cases based on the application's functionality. The proposed system shows strong potential for autonomous testing flows, which can significantly reduce manual work in GUI testing. The approach also

demonstrates strong adaptability to dynamic UI layouts. The use of Retrieval-Augmented Generation (RAG) improves the relevance of test cases by providing more context from the user manuals. The experiments have validated the proposed approach for reducing manual work in GUI testing by dynamically adapting to different UI layouts. Future work can improve the testing approach by enabling multi-step interaction reasoning for complex workflows.

5. Conclusions

This article provides a framework for developing an AI-based autonomous test for desktop applications using vision-based UI understanding. The framework was successfully demonstrated to be capable of detecting a UI element, generating test case scenarios, executing interactions, and generating reports, without using any scripts or source code. The results of the experimentation indicate that the framework has the potential to develop reliable automation of the application with moderate success, but also highlight difficulties in the UI validation process and the complexity of interactions. It appears that a combination of computer vision (CV), Optical Character Recognition (OCR), and RAG may be a viable path toward developing next-generation software testing. While the proposed framework has demonstrated positive results, the current implementation faces several limitations. Currently, the implementation primarily relies on pixel-based verification to confirm UI changes after an interaction. Still, these verification techniques may not always provide an accurate and complete representation of the logical or internal state changes. Also, under certain conditions, such as low contrast, complex fonts, overlapping UI elements, or low-quality screenshots, OCR accuracy may be reduced. In addition, the current implementation primarily supports verification of simple single-step interactions. At the same time, it provides little support for verifying complex multi-step workflows or navigation scenarios that depend on one another. The current implementation is also limited, as it lacks contextual knowledge of longer execution sequences and dynamic user interface behaviour, which will not increase reliability in highly interactive desktop environments. Future work will continue to enhance the ability to validate interactions, apply reinforcement learning to develop adaptive testing strategies, and expand the framework's capabilities to manage more complex multi-step interactions and cross-platform applications.

Funding source

No funding was received for this study.

Conflict of Interest

The authors declare no conflict of interest.

References

[1]. Pandhare, H. V. (2025). Future of software test automation using AI/ML—International Journal of Engineering and Computer Science, 13(05). doi: 10.18535/ijecs/v14i05.5139.

[2]. Faraji, A., & Pombo, N. (2025). AI-Driven Software Test Automation: An AI4SE-Oriented Survey of Techniques, Tools, and Challenges. IEEE Access. doi: 10.1109/ACCESS.2025.3623944.

[3]. Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering, 50(4), 911-936. doi: 10.1109/TSE.2024.3368208.

[4]. Li, K., & Yuan, Y. (2024). Large language models as test case generators: Performance evaluation and enhancement. arXiv preprint arXiv:2404.13340. doi: 10.48550/arXiv.2404.13340.

[5]. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., ... & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in neural information processing systems, 33, 9459-9474. doi: 10.48550/arXiv.2005.11401.

[6]. Shrivastava, G., Peng, S. L., Bansal, H., Sharma, K., & Sharma, M. (Eds.). (2020). New age analytics: Transforming the internet through machine learning, IoT, and trust modeling. CRC Press.

[7]. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., ... & Rush, A. M. (2020, October). Transformers: State-of-the-art natural language processing. In Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations (pp. 38-45). doi: 10.18653/v1/2020.emnlp-demos.6.

[8]. Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence-to-sequence learning with neural networks. Advances in Neural Information Processing Systems, 27. doi: 10.48550/arXiv. 1409.3215.

[9]. Huang, D., Zhang, J. M., Luck, M., Bu, Q., Qing, Y., & Cui, H. (2023). Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. arXiv preprint arXiv:2312.13010. doi: 10.48550/arXiv.2312.13010.

[10]. Zhao, H. H., Yang, K., Yu, W., Gao, D., & Shou, M. Z. (2025). Worldgui: An interactive benchmark for desktop GUI automation from any starting point. arXiv preprint arXiv:2502.08047. doi: 10.48550/arXiv.2502.08047.