Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

AutoTestAI: Generating Software Test Cases from Screen Recordings using Optical Character Recognition and Vision-Language Models

Swara P. Raut, Raksha R. Kedia, Prathamesh R. Yende, Shripad G. Shiraskar and Dr. G. J. Sawale
Dept. of Computer Science & Engg., Prof. Ram Meghe Institute of Technology & Research, Badnera, Amravati, India

***Abstract*—Browser automation tools like Selenium and Playwright are reliable for running tests. However, tests still need to be written manually and linked to Document Object Model (DOM) locators, which can break if the page layout changes. We introduce AutoTestAI, a system that takes a screen recording of someone performing a workflow and generates test cases that can be turned into Playwright scripts. The system does not need access to the application's source code or API. Users upload screen recordings through a React dashboard to a FastAPI backend. OpenCV samples the video, Tesseract extracts on-screen text, and keyframes are chosen when the text changes. Each keyframe is described by LLaVA 7B, running locally via Ollama, using the recognised text in the prompt. Deterministic rules then create draft test cases for the tester to review, and a template-based generator produces the Python Playwright scripts. In a pilot, only 3 of 20 extracted frames were sent to the vision model, and 3 test cases were generated and stored. Across 5 runs on a laptop with an Intel Core i5-1335U processor and no GPU, the pipeline averaged 31.12 s per run, with 89% of the time spent in the vision model. A generated login script passed all 3 runs, with an average test time of 9.86 s. Text recognition was the weakest part, with a character error rate of 32.04% and a word error rate of 49.65% on 5 hand-transcribed frames. Measuring action-recognition accuracy and test-case correctness will require an annotated dataset, which is planned for future work.**

***Index Terms*—optical character recognition, software testing, test automation, test case generation, vision-language models**

I. INTRODUCTION

Many software teams now ship multiple releases per week, and each release may involve changes to the screens that a regression suite has to test. Selenium and Playwright are good at the execution side of things. The authoring is the problem. Each test starts life as a script that an engineer has to write by hand, pointing at page elements via identifiers, CSS selectors or XPath expressions. If a class is renamed or a button is moved, those locators are no longer valid and the test fails even though the feature is still working. Hammoudi et al. looked at 1,065 breakages across 453 versions of web applications and found that over 73% of the record/replay failures were due to locators [1]. Leotta et al. found that both DOM and visual locators require a lot of effort to maintain as the application evolves [2].

Large language models (LLMs) are now being used for a wide variety of testing tasks [3], but most test generators read source code [4]. That makes them unsuitable for black-box targets such as legacy or third-party systems, where the tester may have neither the code nor an API. The person who best understands a system is the one who has actually used it, and that person is best placed to write effective tests [23]. Testers often record their screens for documentation or bug reports. If the screens and steps can be recovered from these recordings, writing a test becomes a process of inference instead of manual work [24]. Mature OCR engines [5] and vision-language models such as LLaVA [6] make this possible on a regular laptop [25].

The main contributions are:

- a pipeline from screen recording to test case that requires no source code, no DOM access and no API, built from OCR-based keyframe selection, word-level OCR and a local vision-language model;
- a keyframe criterion based on changes in the visible text, a prompt that keeps the vision model grounded in the text it can actually see, and rule-based inference that sends every uncertain step to a human for review;
- a working platform consisting of a React dashboard, a FastAPI backend, PostgreSQL storage and template-based generation of Python Playwright scripts;
- measurements of keyframe reduction, OCR error, stage-wise latency and script execution.

The rest of the paper is organised as follows: Section II reviews related work, Section III describes the method, Section IV covers the implementation, Section V presents results, and Section VI concludes.

II. RELATED WORK

A. Visual Perception of User Interfaces

This subsection reviews methods for the visual analysis of user interfaces, covering object detection, text recognition and multimodal understanding. YOLO performs object detection in a single pass in real time [7], though its training set is drawn from natural scenes and contains no interface widgets. In a study of over 50,000 GUI images, Chen et al. compared conventional image processing with deep detectors and found the best results on non-text elements by combining the two approaches [8]. White et al. used image-based widget detection to guide random GUI testing, eliminating the need for source code access, though random testing cannot be relied upon to cover all important business flows [9]. Tesseract uses adaptive thresholding and a two-stage classifier to recognise printed text [5] and has no concept of which widget a piece of text is associated with. Multimodal LLMs come much closer to actually understanding a screen. MP-GUI fuses textual, graphical and spatial information from a screenshot [10], BLIVA is designed for images that are very dense with text [11], and the ScreenSpot benchmark released with SeeClick, covering over 600 screenshots, shows that general-purpose vision-language models struggle to find GUI elements unless they have been pre-trained

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

on GUI data [12]. All of them are single-screenshot models; none of them works on a sequence of screenshots from a video [26].

B. Test Breakage and Repair

Hammoudi et al. examined why record/replay tests break [1], and Leotta et al. examined the practical differences between visual and DOM locators [2]. VISTA fixes broken Selenium tests by visually matching elements between releases; across 2,672 tests from 86 releases it fixed 81% of the breakages [13]. COLOR uses attributes, text, images and position to propose a new locator, and in 77–93% of cases it ranks the correct one first [14]. Both tools start from a test script that used to pass, so neither of them can create a test from scratch.

C. Learning-Based Test Generation

TestPilot takes source code as input to an LLM and achieves a median branch coverage of 52.8% on JavaScript packages [4]. In their survey of 102 studies on LLMs in testing, Wang et al. point out that the vast majority of the work assumes access to code [3]. Other learning-based work includes the use of generative adversarial networks to generate test inputs [15], the identification and repair of defects from commit history [16] and the selection and prioritisation of regression tests [17]. Research into user behaviour modelling learns from logged sequences of interactions [18] rather than from what is on screen.

D. Research Gap

Although detection, OCR and repair techniques have each been extensively explored, deriving test cases directly from a screen recording remains largely open. Avgust generates usage-based tests from videos of mobile app executions [27], and Le et al. generate end-to-end web tests with LLMs and screen transition graphs [24]. AutoTestAI addresses this gap by generating test cases for web applications solely from user-provided screen recordings, without requiring access to source code, the DOM or APIs, which suits black-box scenarios where such access is unavailable. Future development will extend the platform to in-platform test execution and visual locator repair.

III. PROPOSED METHODOLOGY

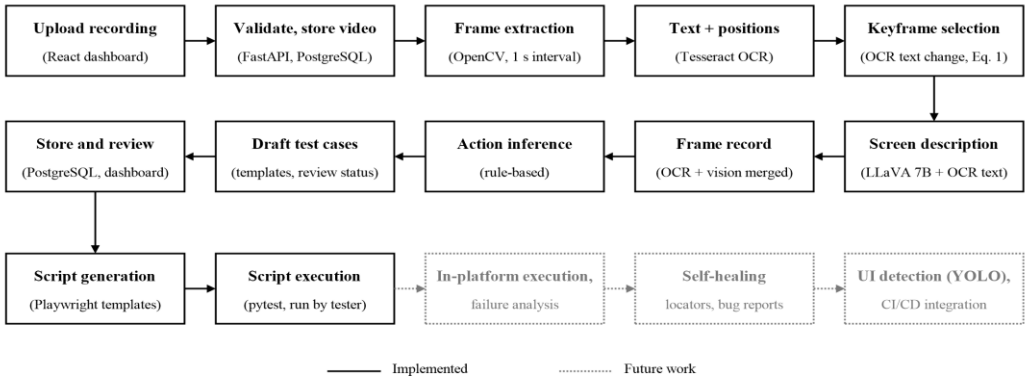


Fig. 1. Workflow of AutoTestAI, from screen recording to reviewed test cases and draft scripts. Solid boxes are implemented; dotted boxes are planned.

A. System Overview

Fig. 1 shows the workflow. After logging in to the React dashboard, the tester picks a project and uploads a recording of a single workflow. The FastAPI backend [19] then validates the upload, stores the video on the server and logs it in PostgreSQL. OpenCV extracts the frames, Tesseract reads the text from each frame, and keyframes are selected at any point where the text changes. LLaVA 7B then describes each keyframe with the OCR text as context, rules infer the actions, and the resulting draft test cases are saved. In the dashboard the tester can view each case with its status, confidence and source frame, add hand-written cases to a recording, and generate a Python Playwright script for any stored case. In-platform execution, failure analysis and self-healing locators are the next steps.

B. Frame Extraction and Text Recognition

Frames are extracted from the video at one per second using OpenCV [20], maintaining their original order. Each frame is then processed by Tesseract, which returns the words along with their bounding boxes and confidence scores, and any word with a confidence score below 40 is discarded. The remaining words are lower-cased, symbols are stripped, and single characters as well as the usual OCR noise are removed [28], resulting in a token set A_t for frame t.

C. Keyframe Selection

Rather than comparing pixels, AutoTestAI compares the text visible to the tester. In a form-based web application, a meaningful step will almost invariably alter some label, value or message on screen. The difference between frame t and the frame prior to it is measured as

$$D_t = \begin{cases} \dfrac{|A_t \triangle A_{t-1}|}{|A_t \cup A_{t-1}|}, & A_t \cup A_{t-1} \neq \emptyset \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

where $\triangle$ is the symmetric difference, i.e. the tokens that appeared or disappeared. The first frame is always taken as the initial state. Subsequent frames with $D_t \geq \tau$ are candidates, and the K candidates with the highest D_t are passed on to the vision model in chronological order; this caps the cost of the slowest stage. As D_t tracks the recognised text rather than the visual change, it can be fooled by OCR errors or by text that keeps changing, such as a clock.

D. OCR-Grounded Vision-Language Description

Keyframes are sent to LLaVA 7B [6], which Ollama [21] runs locally, so no screen content ever leaves the machine. Standard vision-language models are poor at identifying GUI

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

elements and at times describe things that are not actually present [12]. To deal with this, we include the OCR text of the frame in the prompt and instruct the model to use it for any visible text, not to make up applications, buttons or fields, and to return "unknown" if it is not sure. The model responds in JSON with the application name, screen type, visible controls, input fields, UI state and most likely user action. If a call times out or returns malformed output, the analysis proceeds with an "unknown" record and a warning. For each keyframe there is one record containing the OCR words (with positions and confidences) and the model's JSON.

E. Action Inference and Test Case Generation

Actions are inferred via a set of fixed rules, with OCR text as the primary evidence. The first keyframe is labelled *initial screen*. Every subsequent keyframe is compared with the previous one using (1): a change of 0.35 or more is a *UI change* with medium confidence, a smaller but non-zero change in visible text is a *minor text change* with low confidence, and no change at all is *no significant text change* [27]. If LLaVA describes a different screen, that is recorded as supporting evidence but is never taken as proof of navigation. For each action type the generator fills in a template to produce a test case with a title, preconditions, steps listing the text that appeared or disappeared, an expected result and the source frame. Confidence is carried over from inference, and every case other than a high-confidence initial screen is set to *needs review*, so the system never treats a generated case as verified. Both stages are deterministic and very fast. Algorithm 1 gives the full procedure.

Algorithm 1 Recording-to-Test-Case Generation

Require: recording V; threshold τ; frame limit K
Ensure: draft test-case set T

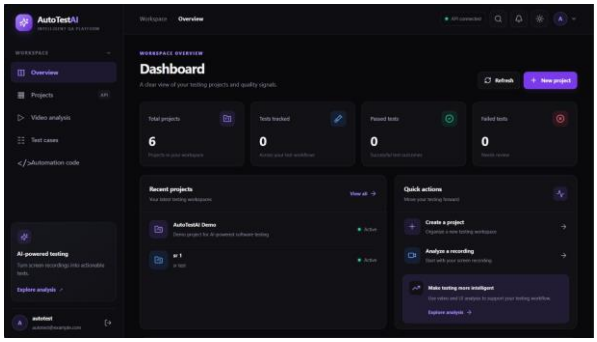
```
extract frames F_1,…,F_n from V at a fixed interval
for i = 1 to n do
    A_i ← normalise(Tesseract(F_i)), drop low-confidence tokens
end for
S ← {F_1} ∪ top-K of {F_i : D_i ≥ τ} by Eq. (1)
for each F_j ∈ S in temporal order do
    v_j ← LLaVA(F_j, prompt with OCR text of F_j)
    c_j ← merge(OCR words of F_j, v_j)
end for
A ← infer actions from ordered c_j (rules)
T ← apply case templates to A
set status from confidence; save T in PostgreSQL
return T
```

F. Automation Script Generation

The automation studio takes a saved test case, plus an optional target URL, and turns it into a Python script for Playwright [22]. It does this locally from templates, with no call to any external AI service. The script navigates to the target URL, confirms that the page has loaded and takes a screenshot as evidence. If the test case and URL match one of the verified interaction templates, it goes on to perform the workflow and check the outcome. The only verified interaction template at present is a login flow for a public practice website with published demo credentials. For any other target, the script has a clearly marked placeholder where verified actions should go, rather than guessed selectors. Each script is marked as a draft and lists why it needs review (e.g. missing URL, no matching template). The templates locate elements by what the user sees, such as the label of an input or the accessible role and name of a button, rather than by CSS or XPath paths; the latter depend heavily on page structure and are the principal cause of locator breakage [1], [29].

IV. IMPLEMENTATION

The backend is written in Python 3.12 using FastAPI and Uvicorn. Each stage has its own module: video processing, OCR, change detection, the vision service, frame analysis, action inference, test-case generation and script generation. SQLAlchemy maps users, projects, videos and test cases to PostgreSQL, Alembic handles schema migrations, and the server's file system holds the recordings and extracted frames. Authentication uses JSON Web Tokens issued through FastAPI's OAuth2 password flow; passwords are stored only as salted hashes, and each request is checked against the project owner. The dashboard (Fig. 2) is built with React, Vite and Tailwind CSS. It provides a workspace view, project management, video analysis, a list of test cases with their status, confidence and source frame, and the automation studio. Table I shows the configuration used in the experiments.



(a) Workspace overview

(b) Automation studio generating a Playwright script

Fig. 2. The AutoTestAI web dashboard, built with React, Vite and Tailwind CSS.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

TABLE I
IMPLEMENTATION AND EXPERIMENTAL CONFIGURATION

Component	Tool / setting
Frontend	React 18, Vite, Tailwind CSS
API and server	FastAPI, Uvicorn (Python 3.12.2)
Storage	PostgreSQL, SQLAlchemy, Alembic; local file system
Authentication	JWT, OAuth2 password bearer
Frame extraction	OpenCV, 1 frame per second
OCR	Tesseract via pytesseract, word confidence ≥ 40
Keyframe selection	$\tau = 0.35$, $K = 3$
Vision-language model	`llava:7b` via Ollama (llama.cpp), 4.8 GB loaded, context 4096, timeout 10 s connect / 120 s read
Scripts	Playwright 1.64.0, pytest 9.1.1, headless Chromium
Processor	Intel Core i5-1335U (13th Gen), 100% CPU inference, no GPU
Memory, OS	16 GB RAM; Windows 11 Home (build 26200)

V. RESULTS AND DISCUSSION

We report measured results for keyframe reduction, OCR error, latency and script execution. Metrics that require an annotated dataset are listed in Section V-F along with how they will be measured; we make no claims as to their values.

A. Dataset and Metrics

Table II details the data supporting each reported result. The pilot used a single screen recording, processed end to end, to evaluate the full workflow. To measure latency, the analysis pipeline was executed five times on this same recording with $K = 3$. The OCR evaluation used five frames from a recording of the login and registration pages of the Tricentis Demo Web Shop; each frame was cropped to the page area, omitting the browser tabs and address bar, and its visible text was typed by hand. A hand-written login case was used to test script execution. This allowed a targeted assessment of each component, but the data lack diversity, as all recordings come from one application and one workflow scenario. OCR quality is measured via the character and word error rates

$$\mathrm{CER} = \frac{S_c + D_c + I_c}{N_c}, \quad \mathrm{WER} = \frac{S_w + D_w + I_w}{N_w} \tag{2}$$

where S, D and I are the substitutions, deletions and insertions required to transform the OCR output of a frame into its transcription of N characters (c) or words (w). Both rates are computed on the full text of each frame using the jiwer library. Latency is the wall-clock time of each stage, from OCR of the extracted frames through action inference, as logged by the backend; frame extraction and saving of the test cases are not included. For action recognition and test-case quality we use

$$\mathrm{Acc} = \frac{N_{\text{correct actions}}}{N_{\text{actions}}}, \quad \mathrm{Corr} = \frac{N_{\text{correct cases}}}{N_{\text{cases reviewed}}} \tag{3}$$

where a case is counted as correct if its steps and expected result match the recorded workflow.

TABLE II
DATA USED IN THE EVALUATION

Evaluation	Data
Pilot (end to end)	1 recording, 20 extracted frames
Latency	5 runs of the pipeline on 1 recording
OCR accuracy	5 frames of a Demo Web Shop recording, transcribed by hand
Script execution	1 hand-written login case (TC-MANUAL-006), 3 runs

B. Pilot Run

The pilot results are in Table III. Of the 20 frames extracted, the change criterion and frame limit sent 3 to LLaVA, giving it 85% fewer frames and reducing the input to the slowest stage. This result is from a single run and does not show whether the omitted frames were important. The pipeline inferred three actions and produced three test cases, which were saved in PostgreSQL and retrieved through the API. In the pilot, two of the three cases were flagged for review; the current generator flags every generated case (Section III). We also noticed that LLaVA described the same frames differently in different runs, which is another reason to keep a human involved.

TABLE III
MEASURED PILOT RESULTS (ONE RECORDING)

Measure	Value
Frames extracted	20
Frames sent to LLaVA	3
Reduction in vision-stage input	85%
Actions inferred	3
Test cases generated, stored and retrieved	3
Test cases flagged for review	2

C. OCR Accuracy

For the five transcribed frames, Tesseract had a CER of 32.04% and a WER of 49.65% (Table IV). These pages used a light theme, so the results are poor for an engine that usually works well with printed documents [5]. Looking at the OCR output, we found three main causes. First, Tesseract reads the multi-column layout in a different order from the transcription, sometimes placing the search box before the site title. Since both metrics compare the full frame text, words read correctly but in a different order still count as errors. Second, words with confidence below 40 are dropped, which counts as deletions. Third, small menu text is often garbled or merged, so “Log in” might become “Log ii”. The first cause inflates the error rates. Still, OCR is the weakest stage, since keyframe selection (Eq. 1), the vision prompt and action inference all rely on OCR tokens. We plan to add upscaling, layout-aware reading order and an order-independent word-recall measure. Five frames are enough to show the scale of the problem, but not to determine an exact rate.

TABLE IV
MEASURED OCR ERROR, LATENCY AND SCRIPT EXECUTION

Measure	Value
OCR CER, Eq. (2), 5 frames	32.04%
OCR WER, Eq. (2), 5 frames	49.65%
Mean pipeline latency, 5 runs	31.12 s
Latency range, 5 runs	15.07–82.44 s
Mean frame-analysis (OCR) time	3.32 s
Mean vision-model (LLaVA) time	27.80 s
Change detection, selection, inference	<1 ms each
Script executions passed	3 of 3 (100%)
Script test time, mean (range)	9.86 s (5.16–13.61 s)

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

D. Latency

We ran the pipeline five times on an Intel Core i5-1335U laptop, with Ollama running the model entirely on the CPU. The average time was 31.12 s, with runs ranging from 15.07 s to 82.44 s (Fig. 3). Of this average, 27.80 s (89%) was spent in the vision model and 3.32 s in OCR of the extracted frames. Change detection, frame selection and rule-based inference each took less than a millisecond. In the slowest run, LLaVA alone took 75.97 s of the 82.44 s. This variation is likely due to the first request after an idle period, which requires loading the 4.8 GB model into memory. Processing time depends on how many frames reach the vision model, so the text-change filter and the limit *K* are important; without them, all 20 pilot frames would have gone to LLaVA. Using a GPU or a smaller model would make this stage faster.

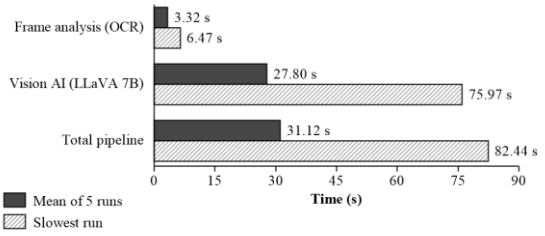


Fig. 3. Stage latency measured over five runs on an Intel Core i5-1335U with CPU-only inference, showing the mean and the slowest run. Change detection, frame selection and action inference each took less than 1 ms and are not shown.

E. Generated Script and Its Execution

Listing 1 shows the main part of the script generated for TC-MANUAL-006, “Verify successful login”. This test case was written by hand in the platform and targets the login page of a public practice website. The script loads the page, fills in the username and password fields found by their labels, presses the submit button found by its accessible role, and then checks for the success URL and the confirmation message. Because it does not use CSS or XPath selectors, a page that is restyled but keeps the same labels would not break the script [24]. We ran the script three times with pytest 9.1.1 and Playwright 1.64.0 on Python 3.12.2 under Windows 11. All runs passed, taking 10.82 s, 13.61 s and 5.16 s (average 9.86 s), and each saved a screenshot of the initial page. This result is for one hand-written case on a site with a matching template; generating and running scripts from recording-derived cases still needs to be tested.

Listing 1. Generated Playwright script, excerpt (TC-MANUAL-006).

```
page.goto(TARGET_URL, wait_until='domcontentloaded')
expect(page.locator('body')).to_be_visible()
page.screenshot(
    path=f'artifacts/{TEST_CASE_ID}_initial.png',
    full_page=True)
page.get_by_label('Username').fill('student')
page.get_by_label('Password').fill('Password123')
page.get_by_role('button', name='Submit').click()
expect(page).to_have_url(
    re.compile(r'.*/logged-in-successfully/?$'))
expect(page.get_by_text(re.compile(
    r'Congratulations|successfully logged in',
    re.IGNORECASE))).to_be_visible()
```

F. Remaining Evaluation

Table V lists the metrics that will require annotated data and the manner in which each will be measured. We give no values for these at this stage, as none have been measured, and with OCR error this high they cannot be reliably inferred from the component results.

TABLE V
METRICS NOT YET MEASURED AND THEIR PROCEDURE

Metric	Planned measurement
Keyframe recall	Label frames with meaningful UI changes; share retained by Eq. (1)
Action accuracy, Eq. (3)	Annotate 10–20 workflows with ground-truth actions
Case correctness, Eq. (3)	Two reviewers score steps and expected results
Scripts from recorded cases	Generate and run scripts for recording-derived cases on more sites; count passes without edits
OCR at scale	Transcribe 50–100 frames, before and after preprocessing
Authoring effort	Time manual writing vs. generation plus review

G. Comparison with Existing Approaches

Table VI puts AutoTestAI alongside existing approaches on the attributes that are important for black-box testing. Record/replay tools capture a script from a live session, but the script is still reliant on DOM locators, which break as the interface changes [1]. VISTA and COLOR are good at fixing locators, but they require an existing script [13], [14], and TestPilot requires source code [4]. Of the approaches listed, only AutoTestAI generates test cases from a recording alone; its scripts currently come from templates, and locator repair with reviewed fixes is planned [30].

TABLE VI
COMPARISON WITH EXISTING APPROACHES

Approach	Code/DOM	Authors	Script	Repairs
Manual scripting	Yes	Manual	Manual	No
Record/replay [1]	Yes	Yes	Yes	No
VISTA [13]	Yes	No	No	Yes
COLOR [14]	Yes	No	No	Yes
TestPilot [4]	Source	Unit	Yes	No
AutoTestAI	**No**	**Yes**	**Template**	Planned

H. Threats to Validity

With one pilot recording, five OCR frames and one hand-written case on one site, the results show that the stages work together and expose the bottlenecks; they do not show system performance across applications. The latency was measured on a single CPU-only laptop and will vary with the hardware. As keyframes are selected from the text, changes that do not alter the text, such as a toggled icon, will go unnoticed, and in a long workflow the limit *K* may drop important steps [31]. LLaVA’s output also varies from run to run, so future measurements should be averaged over several runs.

VI. CONCLUSION AND FUTURE SCOPE

AutoTestAI creates reviewable draft test cases from a screen recording and generates Playwright scripts from saved cases, all without needing source code, the DOM or an API. Keyframes are selected based on changes in OCR text, and a locally hosted vision-language model uses this text to recover the workflow screens. Rules and templates then turn these into structured cases for testers to review in the dashboard. In the pilot, the vision model processed 3 out of 20 frames and three test cases were stored. The pipeline averaged 31.12 s per run on a CPU-only laptop, with 89% of the time spent in the vision

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 11 , November - 2026

model. A generated login script passed all three runs. OCR error was high (CER 32.04%), partly due to reading order. An annotated evaluation over 10–20 workflows will measure action-recognition accuracy and test-case correctness. Next steps include improving OCR with preprocessing and layout-aware reading, using a YOLO detector fine-tuned for interface widgets [7] to handle controls without text, generating scripts beyond fixed templates, adding case approval and execution inside the platform, failure analysis, and locator repair using visual techniques [13], [14]. Other possibilities are CI/CD integration, test prioritisation [17] and synthetic input generation [15].

ACKNOWLEDGMENT

We thank the Department of Computer Science and Engineering, Prof. Ram Meghe Institute of Technology and Research, Badnera, for providing facilities and support. An AI writing assistant helped draft and edit this manuscript; all content was reviewed by the authors, who take full responsibility for it.

REFERENCES

[1] M. Hammoudi, G. Rothermel, and P. Tonella, "Why do record/replay tests of web applications break?" in *Proc. IEEE Int. Conf. Software Testing, Verification and Validation (ICST)*, Chicago, IL, USA, 2016, pp. 180–190.

[2] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Visual vs. DOM-based web locators: An empirical study," in *Proc. 14th Int. Conf. Web Engineering (ICWE)*, 2014, pp. 322–340.

[3] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Softw. Eng.*, vol. 50, no. 4, pp. 911–936, Apr. 2024.

[4] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Trans. Softw. Eng.*, vol. 50, no. 1, pp. 85–105, Jan. 2024.

[5] R. Smith, "An overview of the Tesseract OCR engine," in *Proc. 9th Int. Conf. Document Analysis and Recognition (ICDAR)*, Curitiba, Brazil, 2007, pp. 629–633.

[6] H. Liu, C. Li, Q. Wu, and Y. J. Lee, "Visual instruction tuning," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 36, 2023.

[7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.

[8] J. Chen, M. Xie, Z. Xing, C. Chen, X. Xu, L. Zhu, and G. Li, "Object detection for graphical user interface: Old fashioned or deep learning or a combination?" in *Proc. 28th ACM Joint European Software Engineering Conf. and Symp. Foundations of Software Engineering (ESEC/FSE)*, 2020.

[9] T. D. White, G. Fraser, and G. J. Brown, "Improving random GUI testing with image-based widget detection," in *Proc. 28th ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*, 2019, pp. 307–317.

[10] Z. Wang, W. Chen, L. Yang, S. Zhou, S. Zhao, H. Zhan, J. Jin, L. Li, Z. Shao, and J. Bu, "MP-GUI: Modality perception with MLLMs for GUI understanding," in *Proc. IEEE/CVF Conf. Computer Vision and Pattern Recognition (CVPR)*, 2025, pp. 29711–29721.

[11] W. Hu, Y. Xu, Y. Li, W. Li, Z. Chen, and Z. Tu, "BLIVA: A simple multimodal LLM for better handling of text-rich visual questions," in *Proc. AAAI Conf. Artificial Intelligence*, vol. 38, no. 3, 2024, pp. 2256–2264.

[12] K. Cheng, Q. Sun, Y. Chu, F. Xu, Y. Li, J. Zhang, and Z. Wu, "SeeClick: Harnessing GUI grounding for advanced visual GUI agents," in *Proc. 62nd Annu. Meeting Assoc. Computational Linguistics (ACL)*, Bangkok, Thailand, 2024, pp. 9313–9332.

[13] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in *Proc. 26th ACM Joint European Software Engineering Conf. and Symp. Foundations of Software Engineering (ESEC/FSE)*, 2018, pp. 503–514.

[14] H. Kirinuki, H. Tanno, and K. Natsukawa, "COLOR: Correct locator recommender for broken test scripts using various clues in web application," in *Proc. IEEE 26th Int. Conf. Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 310–320.

[15] X. Guo, H. Okamura, and T. Dohi, "Automated software test data generation with generative adversarial networks," *IEEE Access*, vol. 10, pp. 20690–20700, 2022.

[16] C. Ni, K. Yang, X. Xia, D. Lo, X. Chen, and X. Yang, "Defect identification, categorization, and repair: Better together," arXiv preprint arXiv:2204.04856, 2022.

[17] S. I. Khaleel and R. Anan, "A review paper: Optimal test cases for regression testing using artificial intelligent techniques," *Int. J. Electr. Comput. Eng. (IJECE)*, vol. 13, pp. 1803–1816, 2023.

[18] Z. He, W. Liu, W. Guo, J. Qin, Y. Zhang, Y. Hu, and R. Tang, "A survey on user behavior modeling in recommender systems," arXiv preprint arXiv:2302.11087, 2023.

[19] S. Ramírez, "FastAPI," [Online]. Available: https://fastapi.tiangolo.com, accessed: Oct. 24, 2026.

[20] G. Bradski, "The OpenCV library," *Dr. Dobb's J. Software Tools*, vol. 25, no. 11, pp. 120–125, 2000.

[21] Ollama, "Ollama: Run large language models locally," [Online]. Available: https://ollama.com, accessed: Oct. 24, 2026.

[22] Microsoft, "Playwright: Fast and reliable end-to-end testing for modern web apps," [Online]. Available: https://playwright.dev, accessed: Oct. 24, 2026.

[23] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A detailed investigation of the effectiveness of whole test suite generation," *Empirical Softw. Eng.*, vol. 22, pp. 852–893, 2017, doi: 10.1007/s10664-015-9424-2.

[24] N. Le, Q. M. Bui, M. N. Nguyen, H. Nguyen, T. Vo, S. T. Luu, S. Nomura, and M. L. Nguyen, "Automated web application testing: End-to-end test case generation with large language models and screen transition graphs," arXiv preprint arXiv:2506.02529, 2025.

[25] T. Hegghammer, "OCR with Tesseract, Amazon Textract, and Google Document AI: A benchmarking experiment," *J. Comput. Soc. Sci.*, vol. 5, pp. 861–882, 2022, doi: 10.1007/s42001-021-00149-1.

[26] M. Arrafi, "Oculon Cast: Vision-AI test generator," 2024. [Online]. Available: https://naradipta.com/projects/oculon-cast

[27] Y. Zhao, S. Talebipour, K. Baral, H. Park, L. Yee, S. A. Khan, Y. Brun, N. Medvidovic, and K. Moran, "Avgust: Automating usage-based test generation from videos of app executions," arXiv preprint arXiv:2209.02577, 2022.

[28] "Effects of different filters on text extractions from videos using Tesseract," *J. Comput. Sci. Technol.*, vol. 36, no. 3, pp. 567–578, 2021, doi: 10.1007/s11390-021-1367-5.

[29] "Investigating the robustness of locators in template-based web application testing using a GUI change classification model," *J. Syst. Softw.*, vol. 210, Art. no. 111932, 2024, doi: 10.1016/j.jss.2023.111932.

[30] AutoTestX, "Self-healing: Suggested fixes you review," AutoTestX Docs, 2024. [Online]. Available: https://docs.autotestx.com/maintain-tests/self-healing/

[31] X. Tang, J. Qiu, L. Xie, Y. Tian, J. Jiao, and Q. Ye, "Adaptive keyframe sampling for long video understanding," arXiv preprint arXiv:2502.21271, 2025.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)