Using Vision LLMs For UI Testing

Lawrence Tan, David Dai, Jiuyang Lyu
University of Washington
Paul G. Allen School of Computer Science and Engineering
Seattle, United States
{lawtan,kun02,jiuyal2}@cs.washington.edu

ABSTRACT

Automated UI testing is a critical component of frontend software development that helps ensure that key functionality works and is retained after code changes. Traditional UI tests have dependencies on code structure or specific screen presentations that make them brittle and prone to breaking after code changes, even if the user-facing functionality the developer is testing still works. We hypothesize that Large Language Models (LLMs) with image processing capabilities could execute UI tests, reducing brittleness by removing dependencies on code structure and exact element positioning. We develop a LLM UI testing system, evaluate its performance, and present insights for future applications of LLMs for UI testing and other computer-use tasks.

1 INTRODUCTION

Automated UI testing is a crucial component of modern software development to ensure that key user interactions function correctly. Traditional UI tests are typically hardcoded scripts that depend on the source code and interact directly with the front-end. Unfortunately, UI test suites often fail to test all features of front-ends because they are tedious to write and prone to stop working after changes to the front-end being tested. This is because their implementation involves hardcoding element names or display coordinates in the test, effectively coupling the test to the code being tested. An ideal testing system would not depend on hardcoded references to the underlying code or UI layout coordinates, but instead be able to execute common user flows in the app as the human end-users of the UI would.

In this paper, we present an LLM-based UI testing system that uses screenshots of the UI instead of requiring code or hardcoded coordinates as inputs, in order to reduce test brittleness caused by changes in the implementation or presentation of the UI. Our research question is:

RQ1 Can image LLMs execute UI tests without being coupled to the code or hardcoded screen coordinates?

The rest of this work will be organized as follows: Section 2 will summarize prior works and contrast this work against them for the task of creating automated UI tests that are robust against UI code changes. Section 3 will discuss the algorithm our testing framework will follow to execute automated UI tests. Section 4 will detail how our framework is implemented. Sections 5, 6, 7, and 8 will discuss our evaluation techniques, results, and summarize our final insights and contributions.

2 RELATED WORK

There exists a plethora of approaches targeted towards addressing the brittleness of existing UI testing frameworks due to their programmatic specification's dependence on the UI's hierarchy. These approaches typically addresses this issue through either decoupling testing from code using computer vision, or allowing the test framework to adapt to changes in UI hierarchy using LLMs.

Prior to the rise of LLMs, Chang et. al [1] devised a computer vision based approach where developers first pre-specified a series of buttons to interact with in the forms of images, and during test execution, a computer vision model would match the pre-specified button images to the locations of buttons on screen that it should click. This approach decoupled UI tests from the source code itself reducing brittleness, but is still reliant on human users to manually screenshot the buttons they wish to press and chain them into a sequence to test, making it time consuming to create tests.

Given that LLMs have been demonstrated to excel at sequential planning reasoning problems ([6], [2]), there have been recent works that began to utilize LLMs to tackle UI testing due to the task's sequential nature. Prominently, GPT4Droid featured an approach that converted UI widgets and components to text using the UI hierarchy files found in Android apps, then prompted LLMs for actions to perform in order to test for desired behaviors [4]. Guardian builds on top of this idea by adding refinement to the list of potential UI interactions extracted from the UI hierarchy, so that the LLM only needs to search over a limited number of actions, improving its performance [5]. However, both models rely on converting a given app into text by leveraging the fact that Android app package (APK) files have their UI hierarchy readily extractable. This reliance on an application to reveal its internal hierarchy by converting code to text results in some coupling of the test to the code and makes the testing frameworks less portable to other platforms such as iOS where the hierarchy is not easily accessible.

Our approach seeks to leverage recent advancement in multimodal language models such as GPT-4o to gain the best of both computer vision approaches and natural language model approaches. At a high level, we leverage multi-modal language models by directly feeding our models images of the UI and prompts detailing the test to perform, to allow the model to automatically execute tests using images for context. This will not only decouple our approach's reliance on UI hierarchy that previous language model shared, but

Conference 2025, 1 - 4 January, 2025, City, Country

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

also free it from the computer vision approaches need for developers to micromanage each UI action to perform by passing in screen shots of which button to press.

3 TECHNIQUE

3.1 Automated Testing Loop

As an overview, our approach will take a description of the expected behavior and what steps we need to take to reach the behavior (e.g. "change the app theme by clicking on the settings icon") specified in natural language as input, and automatically execute the UI test by sending the LLM images of the UI and asking it for the appropriate UI action (clicks, scroll, etc.). A pseudocode representation of our algorithm is found in Algorithm 1.

Algorithm 1 Test execution loop

```
prompt ← some user prompt
env ← TestEnvironment()
agent ← Agent(prompt)
oracle ← agent.oracle
curState ← init image
while oracle(curState) is not done do
    action ← agent.getAction(curState)
    curState ← env.takeAction(action)
    if oracle(currState) is failed then
        throw error detected
done!
```

More specifically, our LLM automated testing framework starts by taking in:

- The expected behavior to test for.
- A description of steps we need to take to achieve the behavior (e.g. first we must click on the profile icon, then we need to click on edit).

Where the items above will form our prompt given to the language model we will call *agent*. From there, the main body of the testing loop begins by passing in an image of the current state of the UI which prompts the agent for the next UI action in the format of a click specified by pixel coordinates. The framework will then take the output UI action from the agent and perform the action on the UI. The resulting UI state is captured in an image, and passed to the agent to query for whether we've achieved desired result or a bug is detected (where *bug* is defined in Section **??**). If neither condition is met, the resulting image is passed to the start of the main body loop and the process is repeated until the test is terminated via bug detection or desired result achieved. Thus, the agent acts as the test oracle that determines when the test is completed. As a cost optimization, if the desired state is not reached within a maximum number of UI actions as specified in the expected behavior description, the framework consider this to be a bug and will terminate as well.

3.2 Understanding images

From our preliminary testing, we observed that LLMs struggled with returning the correct pixel location to UI components it identified in the input raw image. We believe that this is because current LLMs lack the capability to understand the relationship between precise pixel coordinates on an image and the identified UI component. Given this limitation, we add an overlay to the images passed to the LLM. At a high level, the overlay is responsible for augmenting the input image with features such as bounding boxes or captions around UI components to help the LLM clearly identify which component it wishes to interact with. In the following subsections, we will elaborate on three overlay approaches.

3.2.1 Grid overlay. To reduce the precision required of the LLM's output, we decided to overlay a grid over the screenshot. Instead of providing exact pixel coordinates, the LLM would simply choose which grid cell it wanted to click, and the Playwright test backend would click the pixel coordinates corresponding to the middle of the grid cell.

This implementation was inspired by an open-source project called GridGPT [1]. Using GridGPT, we first overlay the input image with a grid that evenly divides the image into cells. GridGPT numbers each cell sequentially from left to right, top to bottom, and places this number as text in the middle of each cell. To increase the contrast between the black cell number and the underlying UI, a semi-transparent grey layer is also added on top the image. An example of a GridGPT overlaid image is shown in figure 1.

After adding the grid overlay, the modified image is passed to the LLM to parse. The LLM will return the cell number to indicate which cell it wishes to click on. The cell number will be transformed into the pixel coordinate corresponding to the center of the cell and is returned by the agent.

Figure 1: GridGPT output with numbered labels inside each cell

3.2.2 Bounding Boxes. Throughout our experiments as discussed in Section 6, the key drawback of the grid overlay approach is the need for a hyperparameter specifying the size of grid cells. A large grid cells reduces the amount of noise on the original image due to the grid and cell labels but also decreases the precision for the areas the model is capable is able to interact with. A small grid cell suffers from the opposite effects, increasing precision but also increasing noise. To avoid needing to balance this hyperparameter, we utilize a bounding box overlay instead.

Our bounding box overlay uses OmniParser [3] to modify the raw input image. Given an input image, OmniParser will run a YOLO based segmentation model to identify bounding boxes in pixel coordinates for all interactable UI components on the image.

[1] A repository that overlay a given image with labeled grid cells https://github.com/quinny1187/GridGPT

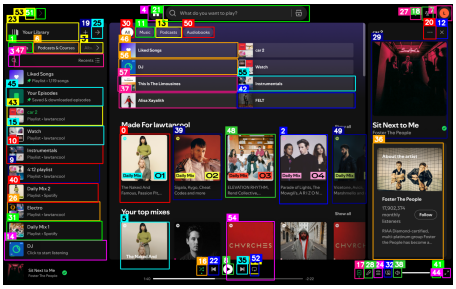

Figure 2: Segmentation Model implementation, with number labels for each identified UI component

From there, we draw the bounding boxes onto the raw image for each identified component, and we label each box with a unique ID. A sample output of our bounding box overlay is shown in figure 2 Finally, like the grid overlay, the modified image is passed to the LLM, and the LLM will return the ID for the bounding box it wishes to click on. The center pixel coordinate of the bounding box will then be returned by the agent.

3.2.3 Bounding Boxes with OCR. Our third approach draws inspiration from how LLMs are trained. Since LLMs are trained on text corpus, it follows naturally that they are best suited to handle textual inputs. Thus, we augment our modified visual data from the previous approach with Optical Character Recognition (OCR) to provide a textual description of each UI component into our overlay step to improve our agent's ability to accurately interact with the UI.

In this approach, we use the EasyOcr library [2] to extract the text present within each bounding box identified by OmniParser. This creates a mapping of caption to bounding box ID that we pass to the LLM in addition to the modified image described in the bounding box approach. A shortened sample of the caption map is shown below.

```
{
    'Liked Songs ': 45,
    'Instrumentals ': 46,
    'This The Limousines ': 47,
    'Liked Songs Playlist 1,119 songs ': 48,
    'car2 ': 49,
    'Follow ': 50,
    'Daily Mix ': 51,
    'Next to Foster The People ': 53,
    None: 76
}
```

We note that the OCR model is incapable of providing accurate captions for UI components without text such as the *None* entry seen above. Thus the LLM is prompted to be aware that the captions may not be accurate.

4 IMPLEMENTATION

In this section, we will detail our Agent to UI interface and our approach to prompt engineering the LLM for test execution.

[2]Optical Character Recognition library EasyOCR https://github.com/JaidedAI/EasyOCR

4.1 Interfacing with the UI

To allow the LLM to execute UI tests, it needs to have access to a test execution system that allows it to launch web UIs, perform clicks and other interactions on the web UI, and take screenshots of the web UI to analyze. To do this, we use the Playwright testing library, which exposes a Python interface for interacting with a web browser in real-time. Using Playwright, we launch the web UI under test and are able to interact with it and capture screenshots of it to send to the LLM.

The LLM will start with receiving a screenshot from Playwright of the initial state of the web UI and text written by the developer describing what the LLM needs to do in order to execute the test. The LLM will reply by outputting the next action as text. We will have code that parses this text and convert it into a Playwright function call and execute this action on the web UI. Playwright will then send a screenshot of the new state of the UI after the action is executed. This loop repeats until the LLM replies by stating that the test has been completed successfully or that a bug was found and the test execution must stop.

4.2 Instructing the language model

To make the LLM understand the task and execute UI tests effectively, we design setup prompt to provide context information for the UI testing and input/output format LLM should follow. The setup prompt includes following information:

- **Context:** Statement asking LLM to act as a testing agent. Description of the functionality of app to be tested.
- **Interaction Flow:** Description of what LLM is expected to do.
- **Input/Output format:** Description of input and output(response) format, bug types, and valid UI actions. The input format for each UI testing step is a screenshot of the UI after last step's UI action has been executed. The output of LLM should be text that contains: expected previous UI action reaction, actual previous UI action reaction, bug/bug-free statement, next UI action, and expected next UI action reaction.
- **Testing Workflow:** The workflow of the UI test. Natural language description of what to test for each step and the expected reaction of the app. Relative position of buttons and UI components (e.g., "the submit button is located below the username field") can be provided as hints to the LLM to help it understanding the spatial layout of the UI.

After receiving the setup prompt, the LLMs begin executing the test, and screenshots of UI will be provided to it following each UI testing action it takes.

5 EVALUATION SETUP

5.1 Web UIs

To evaluate the performance of LLMs in UI testing, we will have LLMs run UI tests on three common real-world web UIs: BBC News, Google Maps, and Spotify. We chose these three apps as they offer a range of different UI complexity levels. BBC News is the simplest, with fewer, clearly labelled buttons in a more uniform layout, while Spotify is the most complex, with many different buttons with different shapes, some of which do not have text labels. As we do

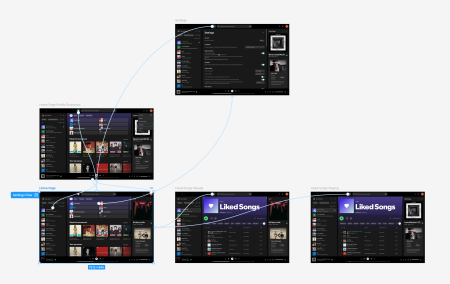

Figure 3: Example of the Spotify user flow recreated as an interactive Figma prototype, with arrows showing the connections between different app states

not have access to the source code of these apps, we recreate their user flows in Figma so that we can modify the behavior of these existing web UIs and introduce bugs for testing without needing to modify the code of the web UIs directly. A user flow is a set of actions and screens that lead to the user's execution of a specific app functionality. For example, Figure 1 shows the user flow of navigating to various screens in the Spotify app. Figma is a design tool for the creation of high-fidelity interactive prototypes and will allow us to easily modify the appearance and behavior of real-world closed-source web UIs.

5.2 Inserting UI Bugs

We will evaluate the LLMs ability to execute test cases on both correct (no known bugs) UIs and incorrect UIs (bugs intentionally inserted). We insert two types of bugs: unresponsive buttons and buttons that lead to the wrong app state when clicked. Since this project focuses on evaluating the ability of LLM to detect UI bugs in user flows, we will not consider the code coverage of our tests. We will instead evaluate the LLMs on the number and types of flows that it can successfully execute and detect bugs in.

5.3 LLM Performance Evaluation Metrics

We will evaluate our system by running the LLM on known bug-free and buggy web UIs. For the bug-free test cases, we will evaluate performance based on the number of times the LLM can successfully execute all steps in the test case and correctly determines that there is no bug present. For the buggy test cases, we will evaluate performance based on the number of times the LLM can execute the test cases to the point where the bug is exposed and correctly report the bug.

6 RESULTS

6.1 Grid Overlay Results

Initial test runs of the Grid LLM interface showed that the language model was capable of identifying and clicking on large features on screen. This is particularly prominent when the model was able to identify and click on larger icons such as liked playlist or the profile icon, but failed to identify the smaller icons such as the mute button. Through experimenting with various grid sizes, we identified that there exists a trade off between larger and smaller grids. Larger grids on the magnitude of 60 pixels by 60 pixels or above allows the model to more easily observe the UI content being overlaid, but is too coarse for the model to be able to identify smaller icons exactly. Smaller grids (lower than 60x60) offers more precise locations to click on, but the grid overlay introduces too much noise to the image, making it difficult for the model to correctly identify the right buttons.

Ultimately, the success rate of the grid overlay was quite poor across our three web UIs, as while it often could correctly click on large buttons, it would not be able to click on smaller buttons and therefore would not be able to progress to the end of the test cases. It was only able to succeed in detecting unresponsive button bugs, but this may be because it is not able to click on the buttons at all, not because it can distinguish between a working and non-working button.

6.2 Bounding Box Results

As seen in Table 1, our grid approach's ability to even identify the correct button to press was heavily dependent on the size of the grid and was unable to identify the correct location to press for less than half of the time. Thus we utilize OmniParser, a UI segmentation model, to identify and label each interactable UI component on screen with numerical IDs [3]. In Table 1, we observed that the segmentation model was able to both outperform existing grid approaches, but also eliminate the need to tune hyperparameters such as grid sizes. We believe that the segmentation model enjoyed higher success rate compared to the grid approaches due to two reasons. The first was that by eliminating a grid on the screen shot and highlighting only the interactable components, the segmentation model is able to provide a more limited yet functional selection of IDs for the LLM to select from. The second reason is the fact that now when the LLM attempt to select a specific ID, it is guaranteed to hit the button being highlighted by the segmentation model, whereas the grid approach suffered from the fact that the grid may not perfectly overlap with a button the LLM wants to press.

6.3 Bounding Box with OCR Results

We observed that Bounding Box with OCR performed better compared to its Bounding box only counterpart. For tasks that involved interacting with UI components that features text only, the Bounding box with OCR approach was capable of clicking on the correct UI component almost every time. However, The OCR approach did not show any improvement over the other two improvements. We believe this is due to the fact that the OCR model we are using is unable to provide accurate captioning for icons without text. Thus, the OCR overlay in turn provides no useful textual label to the LLM. On the other hand, the fact that we see a significant improvement in the LLM's capability to interact with text labeled UI components demonstrated that the OCR augmentation does provide useful additions to our overlay. Quantitatively, as seen from Table 3, the OCR approach does not demonstrate clear performance improvement over the other two model. This is largely due to the fact that our experiment features a mix of UI components where some do not have text labels. Finally, from Table 3, we observed that the OCR approach was unable to enable the LLM to become a reliable oracle in determining cases where the test has ran into a bug. We believe this is due to the same reason as other approaches where the LLM

lacks both a clear idea what state it was previously on as well as the issue that the LLM tends to hallucinate the state its currently on.

7 DISCUSSION AND FUTURE WORK

From our experience developing our LLM testing system and observing the LLM's actions as it executed the tests, we offer some insights into the limitations of LLMs for UI testing, methods we used to mitigate them in our work, and areas of future work that could help address these key challenges.

7.1 Vision LLMs Require Spatial Awareness Augmentations

We found that LLMs, despite their ability to accept images as input, have very poor spatial awareness of the absolute position of objects in the image. For example, in our preliminary explorations, the LLM was not able to output accurate pixel locations of UI elements, necessitating the implementation of various image overlays to assist the model. Even after adding overlays, there were still spatial awareness challenges. When using the grid overlay, the LLM often selected the wrong grid cell to click, especially when the grid cells became smaller and there were more cells in the image, probably because the LLM is not able to process the small visual details of the grid numbers in addition to the rest of the underlying UI. When using OmniParser without OCR, we found that the LLM was better able to select the correct buttons to click but still sometimes incorrectly clicked a nearby button instead, further suggesting that the LLM cannot consistently process small visual details. We had our best results when combining OmniParser with an OCR model that provided text descriptions of each button, showing that today's LLMs, despite their new vision capabilities, are still optimized for text input. We believe that future systems should focus on using text descriptions to augment the LLM's understanding of visual inputs, instead of trying to add more visual overlays that increase the level of visual detail beyond what the LLM is able to process.

7.2 Test Prompts Must Be Carefully Written to Prevent Deviation From Test Instructions

Before we began our testing, we hypothesized that the advanced text understanding of an LLM would allow UI test cases to be written without many constraints on the specific language used for the instructions. While the LLMs were indeed able to understand test case instructions written in natural language, we found that it was important to be very specific with the content of the instructions. For example, it often helped the LLM if the instructions contained a hint for the position of the button it should click on screen and a visual description of the button (ex. "Click the hamburger menu icon on the top left corner of the screen" instead of "Navigate to the menu"). We also found that we needed to prompt the LLM to verify that the previous step had successfully changed the state of the app, otherwise the LLM would simply execute each step in order without any regard to whether the steps were actually working and would report no bugs even if the app did not respond to the interactions. This shows that LLMs do not inherently have reasoning capabilities for UI testing, and shows the potential for future work to investigate improved prompting as a method for improving LLM UI testing performance.

7.3 LLMs Struggle to Be Accurate Test Oracles

When the LLM failed to successfully execute the test case and correctly report the presence of bugs using the OmniParser overlay with OCR, we found that it was not because the LLM could not visually detect the correct button to select, but rather because the LLM had hallucinated the current state of the app and was therefore executing the wrong steps, choosing to select the wrong buttons, or outputting an incorrect bug report. We describe the two forms of state hallucination below:

7.3.1 False test fail. In this case, the LLM would stop execution and report that a bug was present, even if there was no bug. This usually happened when the LLM accidentally selected the wrong button to press in the previous step, leading to the app state changing in a way that the LLM did not expect. The LLM would sometimes be able to undo its previous action (ex. closing a pop-up that it accidentally triggered) and try clicking a different button to follow the test instructions, but other times it would simply stop execution, reasoning that the unexpected result was due to a bug and not an incorrect action. Future work could address this problem by explicitly prompting the LLM to attempt to undo its actions if it encounters an unexpected app state change instead of immediately reporting it as a bug, or by giving the LLM textual descriptions of what the different buttons on the screen do so that the LLM knows that the state change was an expected behavior of clicking the wrong button instead of a bug.

7.3.2 False test pass. While we added prompts to prevent this from happening, we found that in rare cases the LLM would proceed with the next testing steps even if the app did not respond correctly to the previous testing steps. For example, in our Spotify tests, the LLM would sometimes state that it had successfully navigated to the Liked Songs page even if the app was still on the Home page due to a bug or incorrect action. The LLM would then proceed with the next step of returning to the Home screen, which it would interpret as a successful action as the app was already on the Home screen to begin with. It would then report that the test succeeded with no bugs, despite not being able to navigate to the Liked Songs page. We believe that this could be caused by a combination of the LLM not visually understanding that the screen has not changed and the prompting not being robust enough to make the LLM verify that its actions were successful instead of simply assuming they are successful. Future work could focus on improving prompts to help the LLM ignore the fact that it attempted to click on a button and verify that the click action was actually successful. We also believe that there is potential, with newer LLMs having larger context windows, to pass in multiple past images of the UI instead of only the current image so that the LLM can compare them to determine whether the correct state change has occurred or not.

7.4 Interactive Backend and Recovery Mechanisms

One key limitation we observed during testing was that the LLM often forgets the overall testing workflow and starts executing

	Grid overlay	GridGPT (Large Grid)	GridGPT (Small Grid)	OmniParser	OmniParser with OCR
Home Button	0	0	0	2	2
Liked Songs	1	3	1	3	3
Profile Button	0	0	0	0	0
Play Button	0	0	0	0	1
Search Bar	0	0	0	0	3

Table 1: LLM Button detection. Out of 3 runs each

	OmniParser	OmniParser with OCR
Liked Songs	5	5
Liked Songs, Home	2	3
Liked Songs, Play	0	0
Liked Songs, Play, Pause, Home	0	0

Table 2: LLM Multi-step performance. Out of 5 runs each

BBC.com	**GridGPT**	**OmniParser**	**OmniParser with OCR**
Positive	0	0	0
Non-responsive	2	0	0
Wrong link	0	0	1
Google Maps	**GridGPT**	**OmniParser**	**OmniParser with OCR**
Positive	0	0	2
Non-responsive	3	2	2
Wrong link	1	0	0
Spotify	**GridGPT**	**OmniParser**	**OmniParser with OCR**
Positive	0	0	0
Non-responsive	3	0	0
Wrong link	0	0	0

Table 3: Various overlay's performance on bugless case (positive), non-responsive bug case, and wrong link bug case. Out of 3 runs each.

steps autonomously and deviating from the intended workflow. Also, there are pop up windows and advertisement in the tested websites. While LLM attempts basic recovery steps and succeeds on recovering from simple advertisement by click the "close" or "x" button, it fails to recover from unexpected status caused by incorrect previous steps, "sign in/sign up" pop window, and other complex status. This results in incomplete tests, incorrect operations, or failure to recover from unexpected app status. We believe that an interactive backend and a recovery guide in prompt could be developed in future system.

- **Interactive backend:** Interactive backend defines structured objectives for the LLM to complete. Instead of instructing the LLM to recall the entire testing workflow, the backend will provided one objective and basic instructions at a time, and the LLM will come up steps to fulfill the objective,. Once the LLM determines that the objective is complete, it will send a completion signal to the backend, which will then assign the next objective. LLM only need to focus on the current objective and not required to remember the whole workflow, potentially reducing errors caused by forgetting previous steps.
- **Recovery guide in prompt:** A recovery section could be added to the prompt to explicitly instruct the LLM to recognize unexpected UI elements and attempt recovery. When app transitions to an incorrect state, the LLM is instructed to navigate back to the correct state rather than blindly coming up next steps or following the objective. When a UI element does not appear as expected, the LLM will be instructed to retry the action.

8 CONTRIBUTION

In this work, we introduce a novel UI testing approach that reduces brittleness by using multimodal large language models with image processing capabilities. Our contributions are as follows.

- **Decoupling UI tests from UI hierarchy and screen coordinates:** By relying on image-based inputs and predefined UI testing workflow description, our method eliminates the dependency on UI hierarchy and hardcoded screen coordinates, making UI testing more adaptable to front-end changes.
- **Evaluating LLM performance in UI interaction:** We systematically evaluate LLMs' ability to interpret UI states, execute interactions, navigate user flows, and detect potential UI flaws based on visual inputs and natural language instructions. Through empirical evaluation, we evaluate the feasibility of using LLMs for automated UI testing and find scenarios where LLMs struggle.
- **Foundation for broader applications:** Our findings contribute insights that can be applied to future applications of

LLMs in UI testing and other LLM computer-use tasks, such as automated game testing and agentic computer interaction tasks.

REFERENCES

[1] Tsung-Hsiang Chang, Tom Yeh, and Rob Miller. 2010. GUI testing using computer vision. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2010). https://api.semanticscholar.org/CorpusID:9870881

[2] Difei Gao, Lei Ji, Luowei Zhou, Kevin Qinghong Lin, Joya Chen, Zihan Fan, and Mike Zheng Shou. 2023. AssistGPT: A General Multi-modal Assistant that can Plan, Execute, Inspect, and Learn. arXiv:2306.08640 [cs.CV] https://arxiv.org/abs/2306.08640

[3] Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. 2024. OmniParser for Pure Vision Based GUI Agent. arXiv:2408.00203 [cs.CV] https://arxiv.org/abs/2408.00203

[4] Dezhi Ran, Hao Wang, Zihe Song, Mengzhou Wu, Yuan Cao, Ying Zhang, Wei Yang, and Tao Xie. 2024. Guardian: A Runtime Framework for LLM-Based UI Exploration. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 958–970. https://doi.org/10.1145/3650212.3680334

[5] Hao Wen, Hongming Wang, Jiaxuan Liu, and Yuanchun Li. 2024. DroidBot-GPT: GPT-powered UI Automation for Android. arXiv:2304.07061 [cs.SE] https://arxiv.org/abs/2304.07061

[6] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] https://arxiv.org/abs/2210.03629