

WanderVision: Scenario-Based GUI Testing for Smart TVs via Vision-Language Models

Sihyun Ahn✉

KAIST

Daejeon, Republic of Korea

sihyun.ahn@kaist.ac.kr

Juyeon Yoon

KAIST

Daejeon, Republic of Korea

juyeon.yoon@kaist.ac.kr

Taehyoung Kim

LG Electronics

Seoul, Republic of Korea

taehyoung.kim@lge.com

Eunki Song

LG Electronics

Seoul, Republic of Korea

eunki.song@lge.com

Shin Yoo

KAIST

Daejeon, Republic of Korea

shin.yoo@kaist.ac.kr

Abstract

Agents driven by Large Language Models (LLMs) have been applied to GUI testing to automate the process. The intrinsic text-based nature of LLMs requires the information about the on-screen GUI widgets to be provided in textual formats, which in turn necessitates the dependency on tools like Android Debug Bridge (ADB) that allows extraction of on-screen widget structures. The dependency on structured information, however, raises an important question: what can be done for GUI systems that lack such support? This paper proposes the use of Vision-Language Models (VLMs) for agentic testing of GUI systems that are neither web-based nor Android: for such GUI systems, instead of structured GUI widget information, we use screenshots and VLMs to figure out available on-screen widgets and generate click events to perform test scenarios. We implement a VLM-equipped GUI testing agent by extending DroidAgent, and evaluate its performance against native webOS apps running on LG smart TVs. When tasked with performing manually written test scenarios, the VLM-equipped version of DroidAgent shows 64.10% of success rate, fulfilling 25 out of 39 scenarios against webOS native Netflix and Youtube apps.

CCS Concepts

• **Software and its engineering** → **Operational analysis**; **Software testing and debugging**; • **Computing methodologies** → **Machine learning**.

Keywords

Software testing, GUI testing, Test automation, Agentic artificial intelligence, Large language model, Vision-language model

ACM Reference Format:

Sihyun Ahn, Juyeon Yoon, Taehyoung Kim, Eunki Song, and Shin Yoo. 2026. WanderVision: Scenario-Based GUI Testing for Smart TVs via Vision-Language Models. In *Proceedings of the 8th International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE '26)*, October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3843777.3844529>



This work is licensed under a Creative Commons Attribution 4.0 International License. ASYDE '26, Munich, Germany

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2989-8/2026/10

<https://doi.org/10.1145/3843777.3844529>

1 Introduction

Large Language Models (LLMs) and LLM-driven agents have been rapidly adopted for software engineering [5], particularly for test automation [15]. Since LLMs are fundamentally generative models for textual data, their applications in software testing naturally lean towards generating the test code, e.g., automated bug reproduction [2, 10] or test case generation [11, 13]. However, there is a growing body of work that aims to automate GUI testing using LLM-driven agents [6, 12, 17].

For LLM-driven agents to interact with GUI applications, such as Android apps, it is imperative that the agents can perceive the current state of the GUI on screen and react accordingly. Liu et al. reverse-engineer information about each screen from Android app binaries in order to generate prompts that ask for the next action appropriate for the current screen [12], whereas DroidAgent [17] and AdbGPT [6] rely on Android Debug Bridge (ADB) to access the current screen layout in XML format. The common factor in existing approach is the assumption that some structural or textual information about GUI screens is available, either via static analysis or via ADB. However, there are GUI systems that do not readily provide such information, such as native GUI systems embedded in cars or consumer-level services running on non-Android platforms.

We propose the use of Vision-Language Models (VLMs) to tackle the challenge of automating GUI testing without the support from structured GUI information. VLMs combine Vision Transformers (ViTs) [8] and pre-trained LLMs to receive both visual and textual input and perform reasoning about the given images. Recently announced VLMs include various GUI elements in their training data, to support agents performing screen understanding.¹ The availability of such VLMs can lower the threshold of agent-based GUI testing: instead of structured information about the current screen, we may only need to be able to take screenshots of the current screen – the VLM can figure out on-screen widgets from the screenshot image to generate the next GUI event.

To evaluate a VLM based approach, we introduce WanderVision² by extending an existing Android GUI testing agent, DroidAgent [17], to perform scenario-based vision-grounded testing for webOS applications for smart TVs. Unlike cases in Android where textual representation of GUI elements is available, applications

¹<https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Flash-Model-Card.pdf>

²WanderVision is publicly available from <https://github.com/coinse/wandervision>.

in webOS lack support for GUI widget information. We replace the ADB-based actor-observer of the original DroidAgent with VLM and a native TV remote input generator that can mimic on-screen clicks. Subsequently, we task WanderVision with executing scenario-based testing. By instructing the agent to re-enact the manually created test scenarios, we can precisely evaluate the accuracy of VLM-based perception. The result of our empirical evaluation using a real-world consumer apps as well as a smart TV environment shows that WanderVision can successfully perform 25 out of 39 test scenarios in total, showing potential to automate basic scripted GUI testing. We also compare the performance of different VLMs, and analyze failure cases.

The rest of the paper is organized as follows. Section 2 provides the background information about the underlying framework and targeting environment. Section 3 describes the architecture of WanderVision, focusing on the key differentiators from the foundation framework, DroidAgent. Section 4 describes the settings of our empirical evaluation, and the results are presented in Section 5. Section 6 discusses the limitation and scalability of our work, with presenting the related work in Section 7. Finally Section 8 concludes.

2 Background

We present the background of this work, including the base architecture, DroidAgent, and the target environment, webOS.

2.1 DroidAgent

DroidAgent [17] is a GUI testing multi-agent system for Android that enables semantic, intent-driven automation of GUI testing. Four main agents (Planner, Actor, Observer, and Reflector) and three memory modules (short-term, spatial, and long-term memory) interact with each other to realize its key features. The architecture of DroidAgent is illustrated in Figure 1a.

The Planner is responsible for establishing the semantic direction of exploration. Based on the user instruction and the previously recorded history written in the long-term memory, it generates high-level tasks to be achieved by the following agents. The Actor and Observer together exercise the App Under Test (AUT) to meet the high-level goal set by the Planner. Given the task-wise memory and spatial knowledge discovered in previous steps, the two agents enter a loop in which the Actor selects and executes the next action, and the Observer summarizes its outcome, which is provided to the Actor in the next step. Meanwhile, self-critique mechanism within the Actor is activated periodically to ensure that these two agents are proceeding in a proper direction. It evaluates if it is okay to continue as is, and suggests a workaround plan if needed. The Actor-Observer loop terminates when the task is achieved or the loop reaches the maximum number of iterations. The Reflector is activated once a task execution round finishes. It describes the result of the operation, including a short summary and whether the goal was achieved. The authors additionally instruct the Reflector to generate reflections that support more effective planning of subsequent tasks. This generated reflection along with the task execution history is registered in the long-term memory and is later referenced by the Planner to promote diverse and productive subsequent task generation. The Planner-to-Reflector cycle repeats until the testing budget is consumed.

Throughout the testing process, the agents interact with the AUT by utilizing ADB as the communication pipeline to the device shell. Through this channel, the agents use UIAutomator [7], a framework operating within Android that utilizes the system's Accessibility Service. UIAutomator acts as the execution engine responsible for both perception and actuation. It extracts the on-screen GUI widget elements into a structured, XML-based textual hierarchy, and directly executes physical interactions on the device, such as clicking, scrolling, or text input.

Throughout the work, the authors report that DroidAgent autonomously pursues semantic test objectives through multiple agents each focusing on a specific role, and that this in turn brings about high coverage of the AUT. However, the reliance on ADB and UIAutomator limits its applicability to Android apps, as other platforms may not provide a similar channel for perceiving the on-screen GUI in a structured textual form.

2.2 webOS

webOS is a Linux-based operating system installed on LG smart TVs. Our work targets the commercial LG smart TVs running webOS TV [4]. The platform architecture is publicly documented through the webOS Open Source Edition (OSE) [3] materials released by LGE. While the commercial and open-source editions may differ in detail, they share the same core mechanisms.

One notable difference between webOS and other mobile operating systems is the support for standardized and test-oriented GUI representation. Android and iOS expose a structured element tree programmatically through their accessibility services, accessed via UIAutomator [7] and XCUIAutomation [9] respectively. The current webOS, on the other hand, supports two types of applications, web apps and native apps, and neither offers such system-level access to the textual GUI hierarchy. The web apps contains a DOM internally; however, the extracted DOM contains thousands of lines of rendering-oriented markups per screen, making automated distillation of relevant information at each testing step infeasible. Native apps do not provide structural representation of the GUI either. This motivates our approach based on VLMs, which is outlined in Section 3.

While access to GUI hierarchy is not available, webOS does provide a channel for actuation, i.e., generating GUI events. The Luna Service Bus (LS2) is an inter-process communication mechanism among webOS components, through which a client can access system functionality by calling the registered methods of services. Using LS2 via Luna Service API,³ we inject inputs such as clicks, scrolls, and launch or termination of specific applications.

In summary, webOS exposes a channel for actuation, but not for perception. This asymmetry motivates our design: we rely on vision-language models to perceive the screen, while using LS2 to manipulate the device.

3 Approach

We extend DroidAgent to propose **WanderVision**, a VLM based multi-agent system for webOS. DroidAgent targets the Android environment, where elements are readily accessible as text through the accessibility tree. In contrast, due to limitations described in

³<https://webostv.developer.lge.com/develop/references/luna-service-introduction>

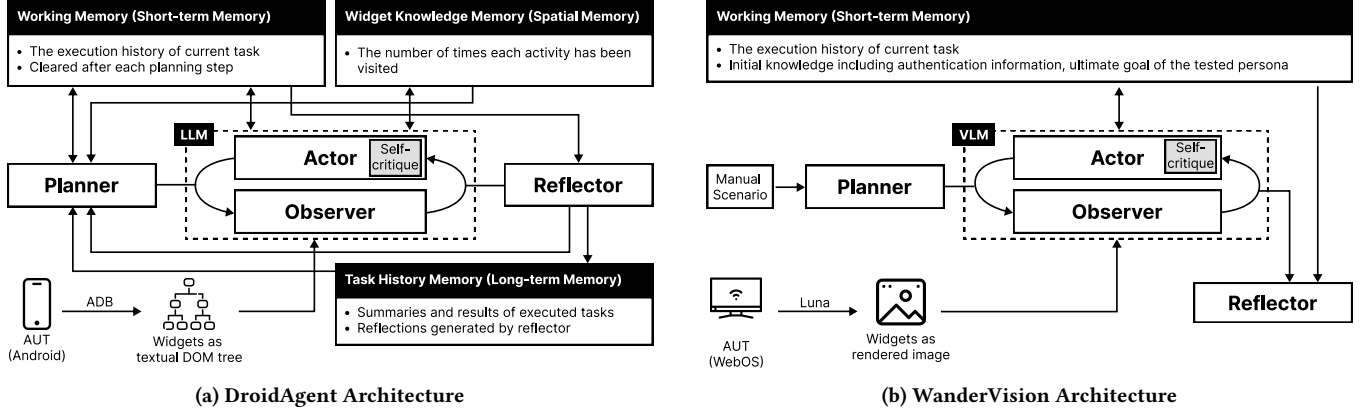


Figure 1: The architectures of the two pipelines, DroidAgent and WanderVision

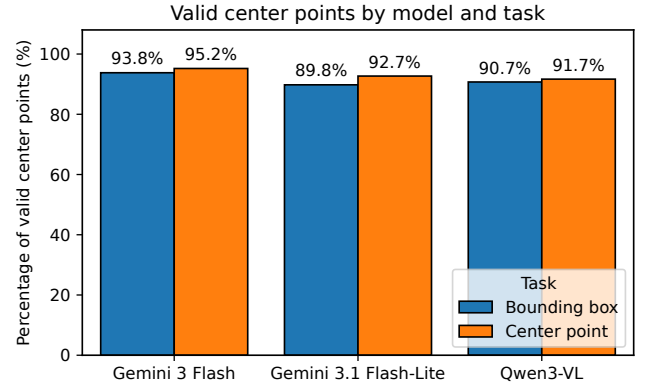
Section 2.2, WanderVision does not have access to such reliable view hierarchy and therefore cannot easily extract widget information in textual forms. This key difference – a vision-only environment – shapes how to perceive its surroundings: we replace the LLM-based Actor and Observer of DroidAgent with VLM-based counterparts, as illustrated in Figure 1b. The remaining changes concern the control flow and are described in Section 4.3.

While the Planner and Reflector play key roles in continuously exercising diverse tasks in the original DroidAgent, we flatten this loop and confine their roles for this work. As a first step toward validating WanderVision for industrial application, we focus on whether it can reproduce the scenarios that were previously executed by human testers, rather than on autonomously generating new ones. Accordingly, the Planner is limited to interpreting a given manual scenario, and the Reflector is limited to evaluating the resulting execution, rather than generating reflections to guide further exploration.

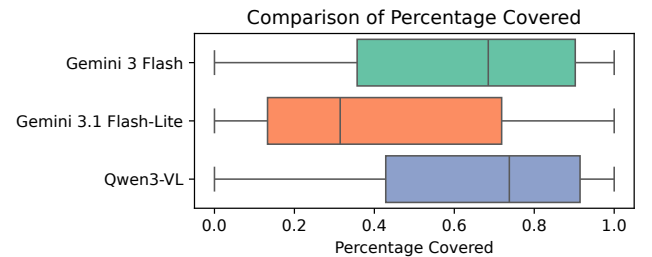
3.1 VLM Selection

To identify the most suitable VLM for the Actor and Observer roles, we conduct a preliminary experiment with several candidate models. As both agents operate exclusively within GUI environments, they do not require real-world visual perception. Instead, they must reliably perceive and localize widgets from screen observations. To assess this capability, we evaluate candidate models on the ScreenSpot benchmark [1], which targets GUI grounding tasks given natural language instructions. The benchmark consists of 1,272 items, each evaluated under both the bounding box and center point detection tasks. We include Gemini 3 Flash as a state-of-the-art proprietary model, Gemini 3.1 Flash-Lite as a cost-efficient proprietary alternative, and Qwen3-VL-30B-A3B-Instruct (hereafter Qwen3-VL) as a representative open-source model.

Figure 2a shows the percentage of valid predictions for the three models. Across both tasks, all tested models achieve validity rates around 90% or higher, with center point accuracy consistently higher than bounding box accuracy for every model. Among the three, Gemini 3 Flash achieves the best performance on center point detection at 95.2%. Figure 2b further examines bounding box quality, measuring how much of the ground-truth area is covered by the



(a) Percentage of valid predictions under the bounding box and center point detection tasks



(b) Percentage of the ground-truth bounding box area covered by the predicted region

Figure 2: Preliminary comparison of candidate VLMs on the ScreenSpot benchmark for VLM selection

predicted region. This shows that Gemini 3 Flash again ranks first, with Qwen3-VL following closely.

Since the Actor ultimately requires a single coordinate for a click event, rather than a region, and center point detection is both more direct and more accurate, we apply center point detection for the Actor. Based on its overall performance, we adopt Gemini 3 Flash as the main configuration for both the Actor and Observer agents.

3.2 Prompting Strategy

Both the Actor and the Observer must produce structured output so that their responses can be reliably parsed and consumed by the rest of the system. The Actor’s action decision, for instance, has to be within a fixed set of predefined actions. To enforce this, we leverage the structured output feature of Gemini, which accepts a response schema through the system prompt and constraints the generated output to conform to it. We define each schema in JSON, specifying the type of every field as a string, number, or boolean. A dedicated schema is defined for each agent according to its expected output. The Actor outputs an action type together with its parameters (e.g., the target coordinate or scroll direction), whereas the Observer outputs a textual description of the action’s outcome. By constraining the VLM outputs, we obtain consistent, well-formed responses that the agent system can process without additional parsing or error handling.

4 Experimental Setup

We present our experimental setup in this section.

4.1 Research Questions

Our primary objectives are to assess whether WanderVision can automate scenario-based GUI testing in the webOS environment, and to analyze the factors that affect its effectiveness.

- **RQ1. Effectiveness** How effectively does WanderVision execute manual test scenarios, when given either the high-level goal alone or the full step-by-step procedure? To address this, we extract the goal and procedure of each scenario, run WanderVision on it, and check whether the agent reaches the expected outcome of the scenario.
- **RQ2. Automated Evaluation** Can WanderVision judge by itself whether the executed run achieves the intended outcome of a scenario? With RQ2, we investigate whether the Reflector agent can serve as an evaluator. Given the executed trajectory together with the scenario’s intent and expected result, the Reflector judges whether the scenario is fulfilled. We then compare its judgements against human-annotated labels to measure their agreement.
- **RQ3. VLM Dependency** How much does the overall performance of WanderVision depend on the choice of VLM? Unlike DroidAgent, WanderVision utilizes VLM to perceive the screen and decide the next action. To assess the impact the VLM has on the performance of WanderVision, we run the full pipeline with different VLMs of varying capabilities and compare the resulting testing performance.

4.2 Dataset

Our evaluation dataset consists of testing scenarios for YouTube and Netflix, created and used in practice by the LGE webOS team. Each scenario contains three main parts: a *goal* describing the high-level category of the test, a *procedure* listing the concrete steps a tester would follow, and an *expected result* specifying the state that indicates success. This structure lets us evaluate WanderVision at two levels of guidance: providing both the goal and the procedure, to test whether it can follow a specified sequence to the expected

result; or providing only the goal, to test whether it can be fulfilled by WanderVision on its own.

Table 1: Statistics of testing scenarios

	YouTube	Netflix
Total scenarios	47	92
w/ qualitative evaluation	35	20
w/ unsupported input source	5	40
Available scenarios	7	32

The full dataset contains 47 YouTube and 92 Netflix scenarios, covering a range of tasks such as authentication, video playback, content searching, preference settings, and context switching. However, we exclude two categories of scenarios from evaluation. First, scenarios requiring qualitative judgements that we cannot reliably assess, such as checking for the absence of contour noise, correct HDR behavior, or audio-video synchronization, are ignored, as these properties are perceptual and lack a clear rubric for human scoring. Second, scenarios demanding inputs beyond what WanderVision supports (click, scroll, go back, go home, and exit) are also ruled out. Consequently, scenarios depending on inputs such as voice commands or dedicated remote-control shortcuts are excluded. After filtering, our evaluation set comprises 7 YouTube and 32 Netflix scenarios, spanning 6 distinct YouTube goals and 12 distinct Netflix goals, as summarized in Table 1.⁴

4.3 Implementation

WanderVision aims to operate on manually defined scenarios on behalf of human testers, unlike the original DroidAgent which explores the AUT autonomously in an open-ended manner. This shift removes the need for the repetitive task-generation cycle. Rather than iteratively proposing and achieving tasks, each run is a single pass consisting of one Planner step, an Actor-Observer loop, and a final Reflector step. The Planner no longer generates high-level tasks, but interprets the given scenario and produces a rough plan for it. The long-term memory, which previously accumulated history across tasks, is therefore unused. The Reflector is also repurposed. Instead of producing reflections for subsequent steps, it acts as an evaluator for the execution trajectory to judge whether the scenario was successfully completed.

To prevent non-terminating runs, we cap the maximum number of actions per scenario. For scenarios that require logging in, the cap is set to 80 actions. For the others, we perform the login manually beforehand and set the cap to 20. We allow a higher budget for authentication because it requires expensive text input events. As WanderVision lacks a primitive action that fills in a sequence of characters at once, such as Android’s `setText`, the VLM enters the user credentials character by character through a series of click events on the virtual keyboard, each of which costs one Actor step and one Observer step. In practice, authentication completes within roughly 60 action steps for our tested account. We re-initialize the target application only for scenarios that explicitly include

⁴The final, filtered test scenarios are also included in the replication package, which is available from <https://github.com/coinse/wandervision>.

an authentication step. For the remaining scenarios, we keep the logged-in session rather than clearing the app state.⁵

4.4 Models

WanderVision uses Gemini 3 Flash as the Vision-Language Model (VLM), based on the results of our preliminary analysis (Section 3.1), while using Gemini 2.5 Flash-Lite for the purely text-based components (Planner, Reflector, and Critique). Additionally, for RQ3, which compares the performance of different VLMs, we test Gemini 3.1 Flash-Lite and Qwen3-VL as in the preliminary analysis.

4.5 Evaluation Metrics

Using goal, procedure, and expected result fields of our evaluation data, we evaluate WanderVision under two settings that differ in how much information is provided as input and in how the outcome is scored. Both settings serve as the basis for measuring effectiveness in RQ1.

In the procedure-based setting, WanderVision is given a goal and a step-by-step procedure. It is expected to follow the assigned procedure and reach the expected result. As the success criterion is explicit and objective, each scenario is scored simply as success or failure, a binary outcome determined by the authors.

In the goal-based setting, WanderVision recognizes a goal and should exercise the AUT following the intent of the goal. Since success depends on how well the execution reflects that intent rather than on a single fixed outcome, the assessment is inherently subjective. Four human evaluators, two from academia and the other two from our industry partner representing practitioners, independently rate the fulfillment of each scenario on a 5-point scale (1–5), and the average serves as the final score. To confirm that these ratings are sufficiently consistent across evaluators, we report inter-rater reliability using the Intraclass Correlation Coefficient (ICC) with absolute agreement. As the four evaluators are treated as a sample of a larger rater population, we use a two-way random-effects model, reporting average measures $ICC(2, k)$ as our primary metric alongside single measures $ICC(2, 1)$.

These two scoring methods form our core effectiveness metric for RQ1, and the same two settings are reused to measure agreement in RQ2. Because the procedure-based setting produces binary labels, we use Cohen’s κ to measure the agreement between the Reflector’s and the authors’ judgements. As the goal-based setting produces continuous ratings, agreement is measured using the ICC with absolute agreement between the mean of the four human ratings and the Reflector’s rating. The Reflector, being a fixed method rather than a sampled rater, calls for a two-way mixed-effects model. We report single measures, $ICC(3, 1)$, and average measures, $ICC(3, k)$, as the primary and supplementary metrics, respectively. In RQ3, we focus on the binary success/failure outcomes of the procedure-based setting and compare the success rates across different VLMs.

5 Results

We present our evaluation results in this section.

⁵Account-level state persists on the server after login, so clearing the local app data does not guarantee a consistent starting state across runs.

5.1 RQ1. Effectiveness

Table 2 shows the procedure-based evaluation results. WanderVision successfully follows 64.10% of the manually defined procedures. We highlight that this performance is achieved without background knowledge of webOS or task-specific calibration, relying instead on the same prompts across all scenarios. Figure 3 demonstrates a successful case where WanderVision follows the given procedure, “Select content → View detail page → Open Episodes & More”, and gets to the expected result. It shows the sequence of screenshots captured during the execution annotated with the coordinate clicked by WanderVision at each step. The final screenshot confirms that the expected result is reached.

Table 2: Results of procedure-based evaluation

	# scenarios	# passing scenarios	Pass rate (%)
Youtube	7	3	42.86
Netflix	32	22	68.75
Total	39	25	64.10

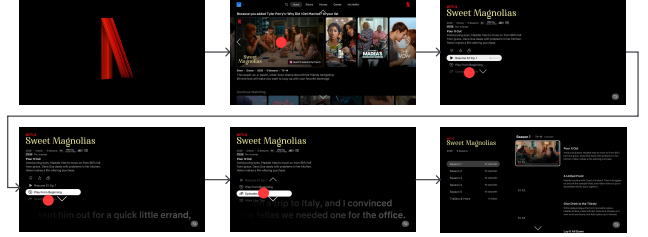


Figure 3: Successful case where the given procedure is “Select content → View detail page → Open Episodes & More”

Table 3 displays 5-point scale scores given by four human labelers. The average score across all labelers is 3.67 where 66.67% of the scenarios got scores more than or equal to 4. That is, WanderVision can successfully test AUT following the given purpose. We report intraclass correlation coefficient (ICC) using a two-way random-effects model with absolute agreement as in Table 4. We use average measures, $ICC(2, k)$ with $k = 4$, as our primary metric since the final score is the mean of four raters, and additionally report single measures, $ICC(2, 1)$. The correlation is high enough that we can assume no critical disagreement has happened within different labelers. $ICC(2, k)$ is 0.953, and $ICC(2, 1)$ is 0.870, both of which indicate strong agreement between the four humans.

Table 3: Results of goal-based evaluation

	Evaluator 1	Evaluator 2	Evaluator 3	Evaluator 4
YouTube	2.67	3.00	2.67	3.33
Netflix	3.92	4.00	4.08	4.17
Total	3.50	3.33	3.61	3.89
Scenarios ≥ 4	61.11%	66.67%	66.67%	72.22%

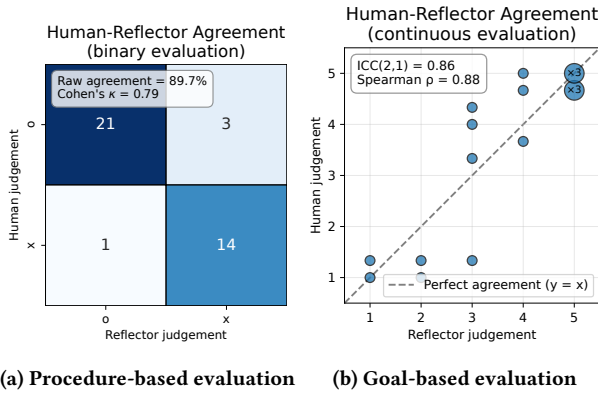
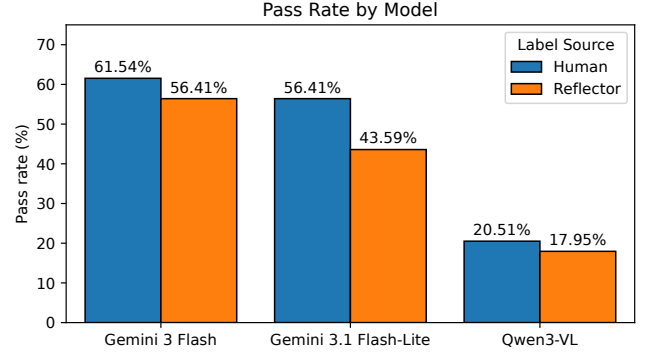
Table 4: Agreement across four human evaluators on goal-based evaluation

Measure	ICC	95% CI
ICC (2, k), k=4	0.953	[0.89, 0.98]
ICC (2, 1)	0.870	[0.74, 0.95]

A case-by-case analysis of the failures reveals two failure groups, both stemming from design constraints of WanderVision. The first group contains scenarios that require background knowledge of the webOS environment beyond what the given context or trajectory provides. For instance, procedures referring to webOS-specific terms like “Analog/DTV Caption”, without comprehensive explanation of what they are, are unlikely to be achieved as the agent cannot ground them in the observed screen. The second group contains scenarios that require an indefinite number of steps to resolve. Verifying video playback termination is one of the typical examples. WanderVision operates within a finite step budget, while such features demand a larger or flexible number of steps. We note that both failure groups do not stem from the perception or decision quality of WanderVision, but rather from structural constraints of the current pipeline, suggesting that they could be mitigated by supplying domain knowledge, or by relaxing the fixed step limit.

5.2 RQ2. Automated Evaluation

Figure 4 shows the alignment between evaluation from human and evaluation from the Reflector, where Figure 4a corresponds to the procedure-based setting and Figure 4b to the goal-based setting. In the procedure-based setting, the Reflector’s binary judgments directly match the human labels on 34 out of 39 scenarios, which is 87.2% of raw agreement rate. Along with Cohen’s κ of 0.73, these values indicate the strong agreement between the human evaluators and the Reflector. In the goal-based setting, a clear positive trend is observed between the mean of the four human ratings and the Reflector’s rating. We measure ICC using a two-way mixed-effects model with single measures, obtaining 0.86, and with average measures obtaining 0.92. The Spearman correlation is 0.86, again indicating strong agreement between the two.

**Figure 4: Agreement between evaluations from the Reflector and human evaluators****Figure 5: Performance comparison of different VLMs on procedure-based evaluation**

Examining the disagreements, we find that the Reflector yields stricter and more conservative assessments than the human evaluators. Most mismatches arise when the Reflector marks a scenario as failed while humans judge it successful, rather than the reverse. This means the Reflector rarely over-accepts a failed execution, introducing few false positives at the cost of occasionally rejecting a genuinely successful run, which indicates that automated evaluation using the Reflector is feasible and can reduce the effort required for manual assessment of test outcomes. In practice, human evaluators could only inspect the cases that the Reflector classifies as failures.

5.3 RQ3. VLM Dependency

Figure 5 shows the performance comparison of different VLMs on the procedure-based evaluation. Each VLM is plugged into the Actor and Observer agents of WanderVision and drives the whole pipeline. The results show that the performance of WanderVision heavily depends on the choice of VLM. Gemini 3 Flash, which is selected based on our preliminary analysis, achieves the best result with a pass ratio of 64.10%. Gemini 3.1 Flash-Lite, a cost-efficient alternative, reaches 56.41%, while the open-source Qwen3-VL reaches only 20.51%.

5.3.1 Gap Between the Benchmark and the Application. The results diverge from our preliminary results on ScreenSpot benchmark shown in Figure 2, where the three tested models showed little performance gap. However, in the full pipeline, Qwen3-VL shows a significant performance drop of 41.03 percentage points relative to Gemini 3 Flash. Upon manual examination, both Gemini 3.1 Flash-Lite and Qwen3-VL completely failed on the authentication task. As illustrated in Figure 6, this failure stems from the fact that it requires constant reference to the account information and precise localization of virtual keyboard keys. This indicates that the performance of VLMs on ScreenSpot benchmark does not necessarily reflect their performance in real-world GUI testing.

We attribute this gap to the fact that Actor demands much more complicated functions than the ScreenSpot does. While ScreenSpot asks the VLM to output a single coordinate for a given user instruction, the Actor must select an action type and its arguments, based on a high-level goal and the execution trajectory rather than

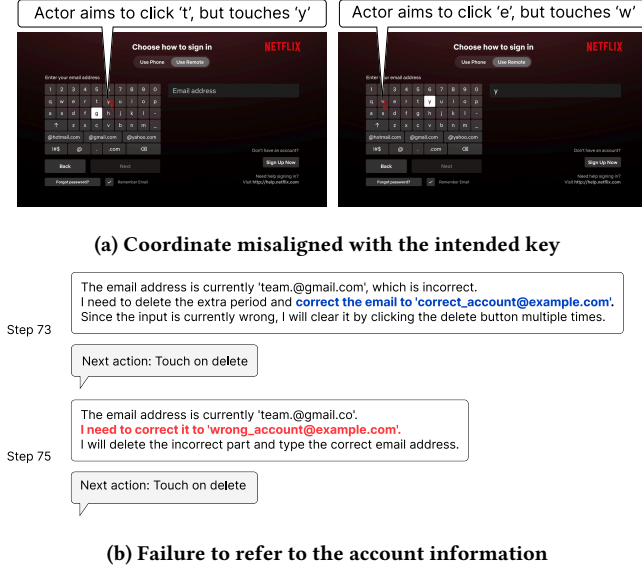


Figure 6: Representative failure cases with Gemini 3.1 Flash-Lite

a concise instruction. This added complexity separates benchmark grounding from operating an agent in a realistic environment.

5.3.2 Gap Between Human and Reflector Evaluation. We also observe a notable gap between the human- and Reflector-assessed success rates specifically for Gemini 3.1 Flash-Lite. The success rate under human assessment is 56.41%, while under Reflector assessment it is 43.59%, a gap of 12.8 percentage points. We manually examine the disagreeing cases to understand this gap.

In cases where a human labels a scenario as successful but the Reflector labels it as a failure, which accounts for 69% of disagreement, the main root cause is noisy steps within the Actor-Observer loop. For example, in a scenario asking whether the virtual keyboard actually inputs text on the login screen, WanderVision successfully reaches the login screen and exercises the virtual keyboard, fulfilling the goal of this scenario. However, it does not terminate afterward and instead continues attempting to log in with incorrect characters. Since the Reflector evaluates the entire execution trace rather than the goal-relevant steps alone, such irrelevant noisy steps negatively affect its final judgement.

The opposite pattern, where humans label a scenario as a failure but the Reflector labels it as a success, is mainly caused by VLM hallucination. The primary root cause is the Actor’s widget detection failure. For instance, the Actor reports clicking the “More Episodes” button, but in fact misdetects its coordinate, so the intended click never takes place. Normally, the Observer is responsible for verifying whether an action actually occurs by comparing the screenshots before and after execution. However, in this case, the Observer also fails to catch the disagreement and reports the action as successful, describing the outcome as though the click had been performed, initiating the reflection with “After tapping ‘More Episodes’”. The Reflector relies solely on this textual log, such as “Touch on More Episodes”, without access to the actual execution outcome, and

cannot verify it directly against the screen. As a result, both the Actor’s detection error and the Observer’s failure to catch it result in judging the task as successful.

To sum up, the overall performance of WanderVision depends heavily on the choice of VLM, and this dependence extends beyond the action decisions made by the Actor. Weaker VLMs not only produced less accurate actions but also generated less reliable Observer reports, and these errors propagated further into the Reflector’s evaluation. This shows that the effect of VLM capability is not confined to a single component but compounds across the pipeline, and that grounding performance on a benchmark such as ScreenSpot alone does not fully capture this cascading effect.

6 Discussion

We analyze the cost limitations of our work and discuss potential future directions in this section.

6.1 Computational and Monetary Cost

Every core step of the pipeline, such as perceiving the screen and deciding the next action, relies on a VLM or LLM call. As a result, both the performance and the operational cost of WanderVision heavily depends on the underlying models. The measurements below are reported for our main configuration, which uses Gemini 3 Flash as the VLM and Gemini 2.5 Flash-Lite as the LLM. Two main limitations arise from this characteristic. First, high latency is observed through the pipeline. Across all scenarios in the procedure-based setting, the average execution time is 276 seconds per scenario. This includes an average cumulative API response time of 205 seconds across all model calls, accounting for about 74% of the total execution time. Since model inference dominates the runtime, the responsiveness of WanderVision is largely bounded by the latency of the underlying APIs. Another limitation is the API cost of the model calls. As Table 5 describes, the average cost is about \$0.099 per scenario, almost all of which comes from the VLM. The VLM costs \$0.0984 per scenario on average, roughly a hundred times the LLM cost of \$0.0009. This cost scales along two axes: the number of tested scenarios, and, within each scenario, the number of Actor and Observer steps required to complete the task. Consequently, applying WanderVision to a large scenario suite, or to scenarios that demand long action sequences, can become expensive.

Table 5: Average API cost per scenario under main config.

Component	Model	Avg cost (\$)	Share (%)
VLM	Gemini 3 Flash	0.098444	99.1
LLM	Gemini 2.5 Flash-Lite	0.000910	0.9
Total	—	0.099354	100.0

6.2 Future Work

WanderVision leaves a few directions for improvement. First, its action space can be extended. WanderVision is currently limited to click, scroll, home, and back actions: the limited set restricts the scenarios it can handle. This can be broadened with more primitive or composite high-level actions. We also see early evidence for this direction. Section 5.1 discusses a scenario asking the agent to

“enter Netflix while ATV/DTV caption is operating, and check how the native caption reacts”, which WanderVision never completes under the current action set. However, in a separate trial, when we additionally gave WanderVision a text-based intent input that mimics the role of a voice command on the remote, it successfully turned on ATV/DTV caption by issuing “open accessibility settings” and completed the scenario. This suggests that enriching the action space is a promising way to broaden the range of testable scenarios.

Second, the Planner can be repurposed to actively guide the entire testing process. In the current setting, the Planner only interprets a given scenario, whereas the original DroidAgent iteratively generates high-level tasks. By restoring this loop, WanderVision becomes not just a predefined scenario reproducer, but an autonomous explorer of the AUT. Such an exploratory mode would complement our current scenario-driven design, potentially uncovering issues that predefined scenarios do not cover.

7 Related Work

7.1 Agentic GUI Testing

Agentic GUI testing approaches leverage LLMs to move beyond low-level random or model-based exploration toward semantic, scenario-driven testing. GPTDroid [12] demonstrates that LLMs can generate human-like testing behaviors by reasoning over textual descriptions of GUI pages. DroidAgent [17] further introduces a multi-agent testing framework that decomposes planning, acting, observing, and reflection to improve exploration effectiveness. More recently, LLMdroid [14] incorporates LLM-guided reasoning into automated mobile GUI testing to enhance exploration coverage. Despite their differences in architecture, these approaches predominantly rely on textual GUI representations, such as extracted widgets, layouts, accessibility metadata, or serialized page information. This assumption does not hold on platforms such as webOS, where text layouts are either missing or incomplete. Our work aims to overcome such limitations by replacing textual observation with VLM-based screen perception.

7.2 Screen Understanding with VLMs

Recent VLM-based GUI agents have demonstrated that screenshots can serve as the primary observation modality for UI understanding and interaction. SeeClick [1] shows that GUI agents can operate directly from screenshots without relying on structured UI representations, highlighting GUI grounding as a key capability for visual interaction. OSWorld [16] establishes a realistic benchmark for evaluating multimodal agents on long-horizon computer-use tasks, accelerating research on screenshot-based GUI agents. However, these studies primarily focus on general-purpose GUI automation and grounding benchmarks rather than GUI testing. In contrast, our work investigates whether VLM-only agents can effectively support GUI testing and systematically analyzes how different VLMs perform in both GUI grounding and action generation.

8 Conclusion

We present WanderVision, a GUI testing framework for webOS built upon DroidAgent. WanderVision employs VLMs to perceive the testing environment directly from the screen, making it applicable to other environments where a text-based layout representation

is unavailable. We evaluate WanderVision on manually defined test scenarios that are actually used in industry. Our evaluation shows that WanderVision completes 64.10% of the scenarios without task-specific calibration, and that its Reflector agent can evaluate outcomes on its own, reaching high agreement with human evaluators. We expect further improvement through expanding the action space, providing additional background knowledge via prompting, and enabling fully automated task generation.

References

- [1] Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Li YanTao, Jianbing Zhang, and Zhiyong Wu. 2024. SeeClick: Harnessing GUI grounding for advanced visual GUI agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 9313–9332. doi:10.18653/v1/2024.acl-long.505
- [2] Runxiang Cheng, Michele Tufano, Jürgen Cito, José Cambronero, Pat Rondon, Renyao Wei, Aaron Sun, and Satish Chandra. 2025. Agentic Bug Reproduction for Effective Automated Program Repair at Google. arXiv:2502.01821 [cs.SE] doi:10.48550/arXiv.2502.01821
- [3] LG Electronics. 2026. webOS Open Source Edition. <https://www.webosose.org/>.
- [4] LG Electronics. 2026. webOS TV Developer. <https://webostv.developer.lge.com/>.
- [5] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008
- [6] Sidong Feng and Chunyang Chen. 2024. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 803–815. doi:10.1145/3597503.3608137
- [7] Google. 2026. UI Automator. <https://developer.android.com/training/testing/other-components/ui-automator>.
- [8] Kai Han, Yunhe Wang, HanTing Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, et al. 2022. A survey on vision transformer. *IEEE transactions on pattern analysis and machine intelligence* 45, 1 (2022), 87–110. doi:10.1109/TPAMI.2022.3152247
- [9] Apple Inc. 2026. XCUIAutomation. <https://developer.apple.com/documentation/xcuiautomation>.
- [10] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023)*. 2312–2323. doi:10.1109/ICSE48619.2023.00194
- [11] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 919–931. doi:10.1109/ICSE48619.2023.00085
- [12] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make LLM a testing expert: Bringing human-like interaction to mobile GUI testing via functionality-aware decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639180
- [13] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-aware prompting: A study of coverage-guided test generation in regression setting using LLM. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 951–971. doi:10.1145/3643769
- [14] Chenxu Wang, Tianming Liu, Yanjie Zhao, Minghui Yang, and Haoyu Wang. 2025. LLMdroid: Enhancing automated mobile app gui testing coverage with large language model guidance. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 1001–1022. doi:10.1145/3715763
- [15] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936. doi:10.1109/TSE.2024.3368208
- [16] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. 2024. OS-WORLD: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems* 37 (2024), 52040–52094. doi:10.52202/079017-1650
- [17] Juyeon Yoon, Robert Feldt, and Shin Yoo. 2024. Intent-Driven Mobile GUI Testing with Autonomous Large Language Model Agents. In *Proceedings of the 16th IEEE International Conference on Software Testing, Verification and Validation (ICST 2024)*. 129–139. doi:10.1109/ICST60714.2024.00020

Received 2026-07-17; accepted 2026-08-23