Element-Aware Fine-Tuning of Vision-Language Models for Cost-Efficient GUI Testing in an Industrial Setting

Mengzhou Wu*
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
wmz@stu.pku.edu.cn

Yuzhe Guo*
School of Computer Science & Technology
Beijing Jiaotong University
Beijing, China
yuzhe.guo@bjtu.edu.cn

Yuan Cao
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
cao_yuan21@stu.pku.edu.cn

Haochuan Lu
Tencent Inc.
Guangzhou, China
hudsonhclu@tencent.com

Hengyu Zhang
Tencent Inc.
Guangzhou, China
lockerzhang@tencent.com

Xia Zeng
Tencent Inc.
Guangzhou, China
xiazeng@tencent.com

Liangchao Yao
Tencent Inc.
Guangzhou, China
clarkyao@tencent.com

Yuetang Deng
Tencent Inc.
Guangzhou, China
yuetangdeng@tencent.com

Dezhi Ran†
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
dezhiran@pku.edu.cn

Wei Yang
Department of Computer Science
University of Texas at Dallas
Dallas, USA
wei.yang@utdallas.edu

Tao Xie†
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
taoxie@pku.edu.cn

***Abstract*—User Interface (UI) testing is crucial for quality assurance of industrial mobile applications, and yet it remains labor-intensive and challenging to automate effectively. Recent advances in Vision-Language Models (VLMs) present a promising solution for automating GUI testing by mapping natural language instructions to pixel-level actions, significantly reducing the manual effort required for writing test scripts and even designing test cases. While numerous VLMs have been proposed and evaluated for GUI testing, they often fail to meet two critical industrial requirements: (1) effectiveness when handling complex, multi-step workflows in industrial applications, and (2) efficiency for large-scale, high-frequency testing environments typical in industrial settings. Toward addressing the preceding industrial requirements, in this paper, we report our experiences in developing and deploying REPEEK, a novel approach employing a unified three-stage pipeline for both training and inference, enables a VLM to explicitly detect and reason over discrete GUI elements, thereby overcoming the limitations of pixel-based reasoning for both efficiency and effectiveness improvements. In the first stage, REPEEK integrates a lightweight UI-element detector named OmniParser to decompose UI screenshots into a structured element list. In the second stage, REPEEK adopts the vision encoder of the VLM to generate the embedding for each element. In the third stage, REPEEK fuses these element embeddings with the textual instruction to reason and perform classification directly on the UI elements, empowering efficient small models to achieve superior performance against expensive large models. Comprehensive evaluations on public benchmarks and deployment at WeChat show that REPEEK consistently achieves superior accuracy and efficiency compared to state-of-the-art VLMs. Specifically, REPEEK enables a fine-tuned Qwen2.5-VL-3B model to outperform a 72B model with 75% less training data, validating the effectiveness of incorporating domain knowledge into VLM-based GUI testing. We conclude by summarizing three key lessons from developing and deploying REPEEK, offering insights for both researchers and practitioners working on industrial-strength UI testing.**

***Index Terms*—Vision-Language Model, GUI Testing, Test Generation, Object Detection, Visual Testing**

*Equal contribution.
†Corresponding authors.

I. INTRODUCTION

User Interface (UI) testing plays a crucial role in ensuring mobile application (in short as app) quality [1]–[4], but manual testing is both time-consuming and labor-intensive [5]–[7], especially given the growing complexity and rapid iteration cycles of modern software development. While automated UI testing has been extensively studied [2], [5], [8]–[18], existing approaches primarily focus on crash detection and typically fail to uncover deeper functional defects that significantly impact user experiences. Recent advances in Large Language Models (LLMs) [19]–[23] and Vision-Language Models (VLMs) [24]–[26] present new opportunities to build more robust, flexible, and powerful automated GUI testing approaches [27]–[43]. Based on testing instructions in natural language and the app under test (AUT), VLMs perceive the screenshot, reason about the testing objectives and plan appropriate actions, and execute the planned interactions, thereby ensuring the quality of core features of the AUT with minimal human intervention [43].

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00296

Among the preceding powerful models, VLMs have emerged as particularly promising for GUI testing for two main reasons. First, VLMs are capable of perceiving the GUI in a manner similar to human users, unlike traditional LLM-based approaches that rely heavily on accessibility trees [44], which are often incomplete, inaccurate, or inconsistent. Such capability enables VLMs to recognize and interact with non-standard or visually complex elements [15] that are missing or poorly represented in accessibility trees. Second, VLMs demonstrate strong generalization and cross-platform capabilities. By directly processing visual information as users, VLMs can interpret GUIs across diverse platforms, including Android, iOS, and desktop systems, without requiring knowledge of implementation details from platform-specific APIs [15]. The general applicability of visual testing [15] makes VLMs inherently robust to variations in design patterns, interface structures, and platform-specific conventions.

To unleash the power of VLMs for GUI testing, recent research has explored two major directions. First, one line of work leverages general-purpose VLMs to develop agent systems [27], [37]–[40], [45]–[47] for automated GUI testing. However, such work often encounters inherent limitations of general VLMs such as limited GUI-grounding capabilities [27]. This deficiency lies at the model level, and thus cannot be overcome simply by layering more sophisticated agent logic on top. Second, an emerging trend [28]–[36], [48] focuses on training VLMs specifically for GUI apps. For example, SeeClick [28], Aria-UI [29], and UGround [30] build models to achieve precise grounding of GUI elements based on natural language instructions, while Aguvis [31] leverages the reasoning abilities of VLMs to enable agents to generate chains of thought for executing complex, multi-step interactions. Additionally, UI-TARS [32] employs iterative training and reflection tuning to facilitate continual learning from errors and adapt to unforeseen scenarios.

Despite extensive efforts to improve VLMs for GUI testing, existing work [28]–[32], [48] typically fails to address two major industrial requirements that make a VLM desirable for practical deployment in industrial settings. First, industrial GUI testing requires precise understanding of complex UI structures to support multi-step workflows, where errors in grounding at any individual screen can cascade and invalidate entire test runs. Industrial applications often involve lengthy workflows with complex element hierarchies, requiring VLMs to achieve high accuracy on individual UI screens. Otherwise, accumulated errors across multiple steps can invalidate entire test runs. Our analysis of WeChat's internal test cases reveals an average of 16.8 steps per test case, significantly higher than most public benchmarks [49]–[51]. However, due to the complexity of industrial app interfaces, existing VLMs often suffer from grounding errors. As illustrated in Figure 1, existing VLMs may click on pixels where no rendered element exists, highlighting fundamental limitations in understanding GUI structure and logic.

Second, industrial deployment demands high efficiency and cost-effectiveness due to resource constraints. In practice, GUI testing faces high-frequency requests (200 processing requests per minute) while having limited computational resources, particularly high-performance GPU resources dedicated to testing. Although large VLMs demonstrate reasonable performance, their computational overhead makes them impractical at scale. Smaller VLMs, which are more feasible for real-world deployment, fall far short of meeting industrial requirements for accuracy. As a result, neither large nor small VLMs, in their current form, satisfy the accuracy and efficiency requirements of industrial GUI testing.

Toward satisfying the preceding industrial requirements for VLMs, in this paper, we report our experience of developing and deploying REPEEK, a three-stage approach for practical VLM-based GUI testing based on a core insight that effective GUI automation requires explicit reasoning over structured UI elements on a screen, rather than treating the screen as unstructured pixels. Modern GUI frameworks render UIs as hierarchical collections of discrete elements (buttons, text fields, menus, etc.), and both human testers and end-users interact with apps through these structured UI components, not arbitrary screen coordinates. However, existing VLM-based approaches primarily operate on raw screenshots and generate pixel-level actions, fundamentally misaligning with how GUIs are actually constructed and used. In contrast, REPEEK addresses the misalignment by explicitly detecting and reasoning over individual UI elements, significantly reducing the grounding errors that plague pixel-level approaches.

To instantiate the preceding insight, REPEEK reframes VLM-based GUI testing from the traditional coordinate prediction into element selection with a unified inference-training pipeline consisting of three stages. First, REPEEK constructs a structured representation of the GUI by applying OmniParser [52] to detect interactive elements and extract their bounding boxes. Second, REPEEK encodes visual context by processing the full screenshot with a vision encoder and generating spatially grounded embeddings for each detected element, thereby bridging raw visual input with the GUI hierarchy. Third, REPEEK performs element-level grounding by fusing element embeddings with textual instruction representations and classifying over the candidate element set. Rather than regressing to pixel coordinates, which are prone to minor misalignments that can cause grounding failures, REPEEK directly outputs the index of the target element, ensuring that actions remain constrained to valid UI components. This classification-based approach significantly improves robustness by constraining actions to valid interactive elements.

To enable the preceding element-centric approach cost-effectively, we fine-tune Qwen2.5-VL-3B [24], an efficient VLM backbone, using curated datasets where each sample contains a screenshot, an instruction, and a ground-truth target element index. The fine-tuning strategy simultaneously addresses our industrial requirements: the efficient 3B model ensures deployment efficiency and cost-effectiveness, while the element-centric fine-tuning of REPEEK significantly improves grounding accuracy compared to pixel-based fine-tuning employed by previous work [32], [48].

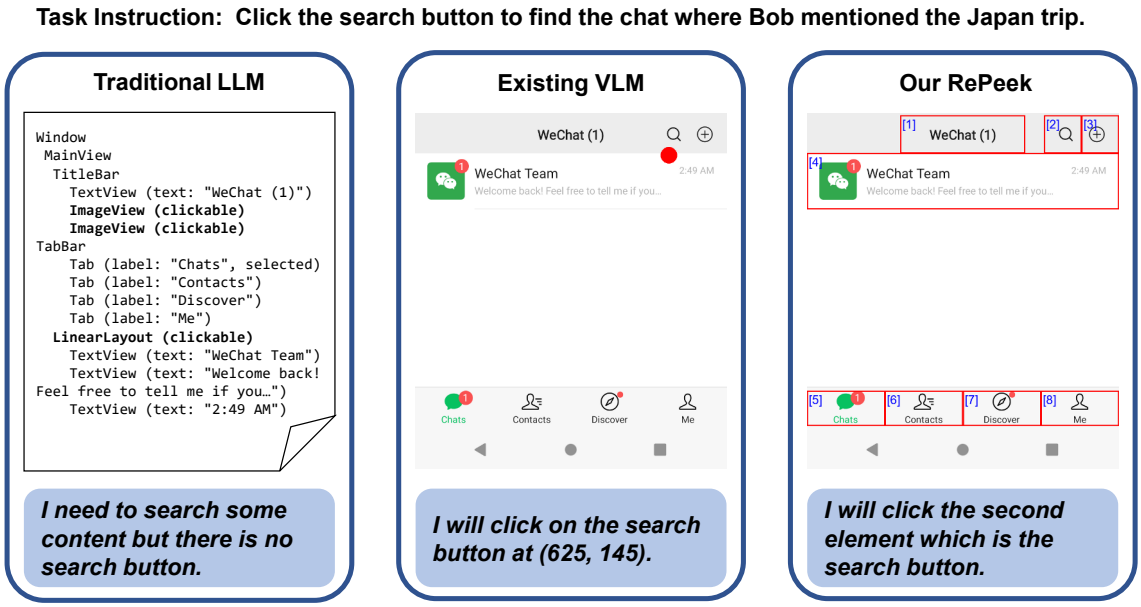

Fig. 1: A motivating example illustrating how REPEEK transforms traditional VLM-based GUI testing from pixel-level coordinate prediction to element-level classification, inspired by the structural composition of UI screenshots.

To evaluate the efficacy of REPEEK, we conduct extensive experiments against state-of-the-art VLMs across popular GUI automation benchmarks and in large-scale deployment within WeChat, a highly popular app with over one billion monthly active users. Evaluation results demonstrate that REPEEK consistently matches or surpasses existing VLMs, both in public benchmarks and industrial deployment. Despite using only a lightweight 3B model trained on approximately 0.7B tokens, REPEEK outperforms 72B models on ScreenSpot [28]. Deployed at WeChat, the model fine-tuned by REPEEK achieves 6.9% higher accuracy than the state-of-the-art model UI-TARS-7B [32] while reducing inference time by 30%, confirming the efficacy of REPEEK and the usefulness of incorporating domain knowledge with fine-tuning VLMs.

In summary, this paper makes the following major contributions:

- REPEEK, a three-stage approach that transforms GUI automation from pixel-level to element-level reasoning for practical VLM-based testing. We open-source the model architecture and parameters of REPEEK [53] to foster further research and application.
- Extensive evaluations on multiple popular benchmarks and deployment at WeChat demonstrating the efficacy of REPEEK.
- Experience and lessons learned from developing and deploying REPEEK at WeChat.

II. MOTIVATING EXAMPLE

In this section, we present a motivating example to illustrate the limitations of existing LLMs and VLMs in GUI testing.

Figure 1 presents a GUI testing task "Click the search button to find the chat where Bob mentioned the Japan trip" on WeChat. Despite intuitive for human testers, this task exposes fundamental limitations in existing LLMs and VLMs. First, LLMs rely on structured text representations of GUI hierarchies, but suffer from information loss during the text conversion process. As shown in the left panel, the search button is rendered as an ImageView without accessible text content, making it completely invisible to text-only models. Such visual elements are common in modern mobile apps, limiting the applicability of LLM-based solutions. Second, VLMs address the visibility issue by processing raw screenshots directly. However, while GUI interactions inherently operate at the semantic element level (buttons, text fields, and icons), existing VLMs reason at the raw pixel level. This disconnect manifests as imprecise coordinate predictions that often target non-interactive screen regions, as illustrated in the middle panel where the model clicks adjacent pixels rather than the actual button element.

REPEEK addresses these limitations by recognizing that GUIs are element-driven structures rather than unorganized pixel arrangements. Instead of forcing GUI understanding into pixel-level reasoning, REPEEK leverages the inherent structured nature of GUI applications. This insight leads to an element-aware approach that better aligns with how GUI interactions actually work, achieving higher precision and efficiency compared to existing coordinate-based approaches.

III. APPROACH

Motivated by the limitations discussed in Section II, we propose REPEEK, an element-aware approach for VLM-based

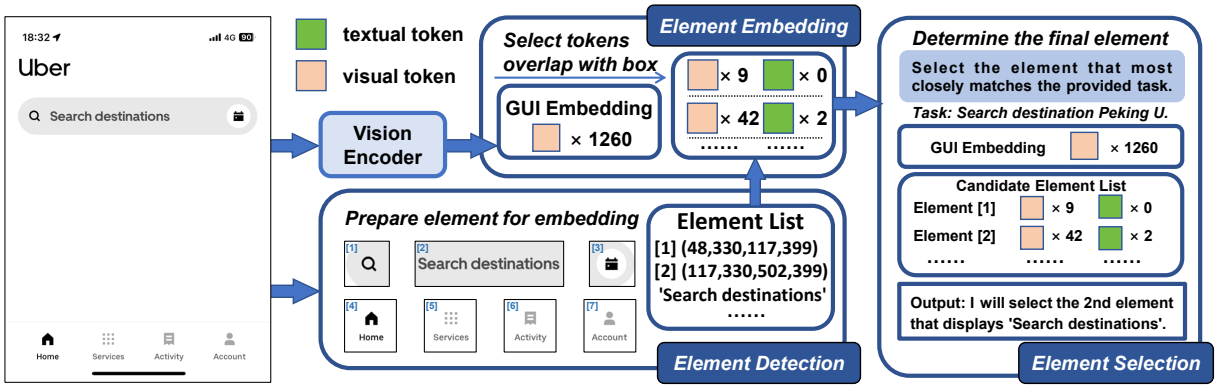

Fig. 2: Overview of REPEEK. First, REPEEK employs a detection module to identify all candidate UI elements by extracting their bounding boxes and corresponding textual content. Second, REPEEK encodes the screenshot with a vision encoder, where each element's visual tokens are fused with its textual tokens to form a unified representation. Finally, REPEEK incorporates the task instruction and processes all fused element representations to select a UI element to achieve the task.

GUI testing that interacts with the GUI by identifying and reasoning over semantically meaningful elements such as buttons, text fields, and images. Inspired by how real users perceive and interact with an app through distinct UI components, REPEEK leverages VLM reasoning on individual interactive elements rather than treating the GUI as a pixel-based grid and operating over low-level coordinate positions. The following subsections first introduce the inference pipeline and subsequently detail the corresponding fine-tuning strategy.

A. Element-Aware Inference Workflow

As illustrated in Figure 2, the inference workflow of REPEEK consists of three sequential stages. Given a screenshot and a natural language instruction as inputs, REPEEK processes them as follows.

GUI Element Detection. First, REPEEK utilizes a detection module (Section III-B) to identify candidate interactive elements within the screenshot. The module extracts the visual regions along with the textual content of each detected element, forming a structured representation of the GUI.

GUI Element Embedding. Next, an embedding module (Section III-C) efficiently generates a joint vision-language representation for each detected element. The key to this efficiency is avoiding the computational overhead of encoding each element's image patch individually. Specifically, the module constructs a representation for each element by combining its tokenized text with visual features cropped from the entire image's feature map using the element's bounding box. These element-specific representations are then concatenated with the tokenized natural language instruction to create a unified multimodal token sequence.

GUI Element Selection. Finally, REPEEK processes the unified multimodal token sequence to predict the index of the element that best matches the given instruction. This element-centric prediction enables precise and targeted interaction with the selected GUI component.

B. GUI Element Detection

The first step in REPEEK is to detect interactive elements from a GUI screenshot. Each GUI element typically has a clearly defined spatial location (usually represented by a bounding box) and corresponding textual content indicating its functionality. Accurately and effectively extracting both the positional bounding box and textual content of these elements is crucial for GUI automation.

REPEEK provides two techniques for GUI element detection: visual element detection using OmniParser [52], and structured parsing using the accessibility (a11y) tree.

OmniParser is a lightweight visual GUI parsing tool built upon YOLOv8, trained specifically for tasks of parsing GUI screens. Given a GUI screenshot I, OmniParser performs a single forward pass to output the bounding boxes of all candidate interactive elements along with their corresponding textual content.

In addition, REPEEK supports parsing interactive elements based on the a11y tree. However, annotations derived from the a11y tree often contain noise, including imprecise and overlapping bounding boxes. To address this issue, we propose a heuristic-based filtering strategy to extract reliable interactive elements.

The procedure begins by filtering all interactive nodes according to their interactivity attributes (such as 'clickable' and 'scrollable'), These nodes are then sorted by bounding box area in ascending order. Next, for each node, we compute the total overlapping area with all smaller candidate elements. If this overlapping area exceeds the node's own bounding box area, the node is deemed mostly covered by smaller elements and is discarded as redundant. Otherwise, the node is retained in the candidate element list, and its textual attributes (e.g., 'text', 'content-desc') are extracted as textual content.

By employing the preceding techniques, we efficiently extract all potential interactive elements from either a GUI screenshot I or the a11y tree. For each detected element e_i, we

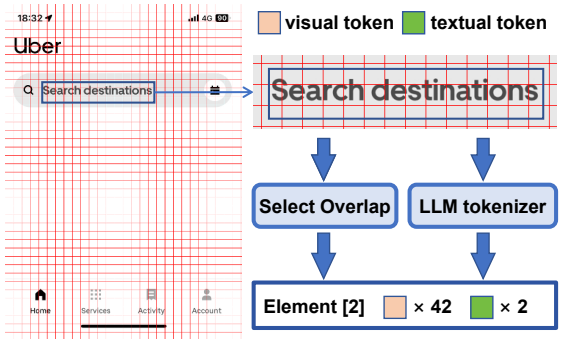

Fig. 3: Details of GUI Element Embedding. Each red grid cell denotes a visual token. The "Search destinations" element is represented by selecting overlapping visual tokens (orange) and textual tokens (green) from the LLM tokenizer, resulting in a multimodal embedding.

obtain its spatial location b_i (i.e., bounding box) and the textual content t_i within the region, thereby forming the complete element set:

$$E = \{e_1, e_2, \ldots, e_N\} \qquad e_i = (b_i, t_i)$$

The resulting element set E thus contains both spatial locations and semantic content of all candidate elements, and is passed to the embedding module for further processing.

C. GUI Element Embedding

A primary challenge in applying a VLM to GUI automation is their inability to directly associate bounding boxes with corresponding regions in an image. Consequently, providing the raw coordinates b_i as input yields limited benefit.

To address this challenge, we propose a novel element embedding technique that transforms each detected GUI element into a representation that enables the VLM to interpret both visual and textual information effectively. Given the set of detected elements $\mathcal{E} = \{(b_i, t_i)\}_{i=1}^{N}$, we construct a joint vision-language input representation for each element:

$$e_i^{\text{input}} = (v_i,\ l_i), \quad i = 1, \ldots, N$$

where v_i denotes the visual representation extracted by the vision encoder from the image patch corresponding to the bounding box b_i in the GUI screenshot I, and l_i denotes the textual representation obtained by processing the associated text t_i.

To improve computational efficiency, we avoid individually cropping and encoding the image region for each element. Instead, we process the entire screenshot once using the vision encoder and then select only the patch tokens that overlap with the element's bounding box.

Specifically, the entire GUI screenshot I is first processed by the vision encoder of the VLM, which partitions the image into a grid of fixed-size patches and generates each patch into a visual token. For the i-th element, we extract the subset of patch tokens whose corresponding image region overlaps with the bounding box b_i. These selected tokens jointly serve as the visual representation of the element:

$$v_i = \text{SelectOverlap}(\text{VisualTokens}(I),\ b_i), \quad i = 1, \ldots, N$$

where $\text{VisualTokens}(I)$ denotes all the patch tokens for the screenshot, and SelectOverlap returns the patch tokens whose corresponding image region overlaps with the bounding box b_i.

For the textual representation l_i, we directly tokenize the associated text t_i using the tokenizer of the VLM. As the VLM natively processes natural language, no further transformations are required. The resulting tokens capture the semantic content of the element, such as its label, text, and functional description.

For example, in Figure 3, the set of elements is $E = \{(b_1, \text{“”}), (b_2, \text{“Search destinations”}), (b_3, \text{“”}), (b_4, \text{“Home”}), (b_5, \text{“Services”}), (b_6, \text{“Activity”}), (b_7, \text{“Account”})\}$. The original screenshot and the element set E are simultaneously processed by the vision encoder. After encoding by the vision encoder, element e_2 selects the patch tokens whose corresponding region overlaps with its bounding box b_2, resulting in a total of $3 \times 14 = 42$ visual tokens. The associated text "Search destinations" is tokenized using the tokenizer to obtain 2 textual tokens. These visual and textual tokens are then combined to form the joint input representation e_1^{input} of element e_1.

The multimodal representation e_i^{input} enables the VLM to consider each candidate element in context and determine the most appropriate target for interaction given the user instruction.

D. Data Collection

In the open-source communities, GUI datasets are typically divided into two categories.

The first category centers on GUI element grounding, which can be formalized as a set of triples: <referring expression, screenshot, element coordinates>. The element coordinates are represented either as a point or as a bounding box.

The second category targets GUI navigation, usually consisting of triples for each high-level natural language task: <natural language instruction, screenshot sequence, action sequence>. The action sequence is composed of a series of operations, such as click, swipe, and text input.

To enhance the capabilities of REPEEK in both GUI element grounding and complex navigation tasks, we create a collection of high-quality training datasets.

For GUI element grounding tasks, we utilize the WidgetCaption [54] and AMEX [55] Level II datasets. WidgetCaption includes over 60,000 functionally-annotated GUI elements across more than 20,000 mobile screenshots. We reformulate the original task by using the functional description of each element as a natural language instruction and the coordinates of each element as the target location. AMEX Level II contains over 700,000 functionally labeled GUI elements from more than 100,000 mobile screenshots.

TABLE I: Number of Training Samples and Tokens Used by Different GUI Models

Model	Training Samples	Training Tokens
SeeClick-9.6B	1 M	-
UGround-7B	10 M	-
ShowUI-2B	2.7 M	-
Aguvis	1.3 M	-
OS-Atlas	13.6 M	-
UI-TARS	-	50 B
Tong-UI	1.3 M	-
REPEEK	**0.32 M**	**0.7 B**

For GUI navigation tasks, we select the GUIAct-smartphone [56] and AMEX [55] Level III datasets. The GUIAct-smartphone dataset comprises approximately 10,000 multi-step tasks with 70,000 screenshots. The AMEX Level III dataset contains 3,000 task instructions spanning approximately 50 applications.

As shown in Table I, most existing GUI models are trained on datasets containing several billion tokens, while REPEEK is trained on only 0.3 million samples and 0.7 billion tokens. The comparison highlights the data efficiency of REPEEK across both GUI grounding and navigation tasks. Despite training on fewer tokens, experiments (Section IV) show that REPEEK demonstrates strong performance across diverse scenarios.

E. Fine-Tuning Strategy

The fine-tuning process for REPEEK consists of two main stages based on the preceding task categories: (1) GUI element grounding fine-tuning: REPEEK is fine-tuned to enhance general understanding of GUI screenshots and ability to locate elements, while also learning its element embedding mechanism; (2) GUI navigation fine-tuning: REPEEK is further fine-tuned to transfer grounding capabilities to navigation tasks, thereby improving overall performance on general GUI navigation.

Additionally, REPEEK can be fine-tuned on downstream tasks to learn task-specific actions and adapt to app-specific requirements.

GUI element grounding task fine-tuning. Both the WidgetCaption and AMEX Level II datasets provide bounding boxes for target elements. We use OmniParser for element detection in GUI screenshots. From the sample, we select the detected element with the highest IoU (Intersection over Union) relative to the target bounding box for training. If all detected elements have IoU < 0.5 with the target, the sample is discarded.

We employ full-parameter supervised fine-tuning (SFT) on all modules of Qwen2.5-VL. Training is conducted on 8 Nvidia H20 GPUs with NVLink [57] using the DeepSpeed framework [58]. We use the AdamW optimizer [59], set the learning rate to 3×10^{-6}, and set the gradient accumulation steps to 8. In total, approximately 200,000 samples are trained over 15 hours, resulting in REPEEK-Base.

GUI navigation task fine-tuning. Due to variations in action spaces across datasets, directly combining them for training could introduce action conflicts and degrade performance [34]. To address this challenge, we translate all actions to a unified action space following previous work [31]–[34].

The training setup for GUI navigation matches that of element grounding. Building upon REPEEK-Base, we train around 120,000 samples over 10 hours to obtain REPEEK-Navigation.

Downstream task fine-tuning. During downstream task fine-tuning, REPEEK-Navigation is further fine-tuned on each task-specific training set to precisely align action spaces and distributions. Given the relative simplicity of downstream tasks, only the language model is fine-tuned, while the vision encoder remains frozen.

We train roughly 120,000, 40,000, and 120,000 samples on the AndroidControl, AITZ, and GUI-Odyssey datasets, respectively.

IV. EVALUATIONS

To validate the effectiveness of REPEEK on mobile device GUI tasks, we conduct comprehensive evaluations on two tasks: GUI element grounding and GUI navigation. Our experiments are designed to address the following research questions (RQs):

- **RQ1:** How does REPEEK perform on the GUI element grounding task?
- **RQ2:** How does REPEEK perform on the GUI navigation task?
- **RQ3:** What is the impact of the fine-tuning strategy and the different components of REPEEK on its final performance?

A. RQ1: GUI Element Grounding Performance

1) Experimental Setup: **Benchmarks.** We select ScreenSpot [28] and ScreenSpot-v2 [34] as our evaluation benchmarks. These benchmarks consist of six subsets spanning two types (icons and text) and three platforms (mobile, desktop, and web). Each test instance provides a unique GUI image paired with human-annotated natural language instructions. ScreenSpot-v2 updates 11.32% annotation from ScreenSpot while maintaining the same sample size. Since our training dataset utilizes only mobile data, the evaluations are conducted exclusively on the mobile subsets—mobile icon and mobile text.

Baselines. We compare against state-of-the-art baselines, which include three main categories: domain-specific GUI models [29]–[34], [48], general-purpose VLM Qwen2.5-VL [24], and OmniParser [52] that leverages Set-of-Mark (SoM) [60] prompting with GPT-4V [61] for GUI understanding. We reproduce the results of Qwen2.5-VL with a maximum image token length of 1344. Results for other models are quoted from their original publications. Since prior work has shown that general-purpose VLMs such as GPT-4o lack pixel-level GUI grounding capabilities [34], we omit them from comparison.

TABLE II: Performance Evaluation of Different Models on GUI Grounding Tasks

Model Name	Model Size	ScreenSpot		ScreenSpot-v2	
		Icon	Text	Icon	Text
Qwen2.5-VL	3B	52.8	78.8	-	-
Qwen2.5-VL	7B	77.7	94.9	-	-
OmniParser	-	57.0	93.9	-	-
OS-Atlas-Base	4B	58.5	85.7	59.7	87.2
OS-Atlas-Base	7B	72.9	93.0	75.8	95.2
Show-UI	2B	75.5	92.3	-	-
Aria-UI	25B-A4B	73.8	92.3	-	-
UGround	7B	60.3	82.8	-	-
Aguvis	7B	77.7	95.6	-	-
Aguvis	72B	85.2	94.5	-	-
UI-TARS	2B	75.5	93.0	79.1	95.2
UI-TARS	7B	**85.2**	94.5	**89.1**	96.9
UI-TARS	72B	82.5	94.9	86.3	94.8
TongUI	3B	77.7	92.6	79.6	94.4
TongUI	7B	79.5	91.9	81.5	93.1
REPEEK	3B	<u>83.8</u>	**96.3**	<u>82.5</u>	**97.6**

Underlined values indicate the best performance among models smaller than 3B parameters.

Metrics. Following existing work, we adopt accuracy as the evaluation metric. For pixel-based models, a prediction is correct if the predicted pixel falls inside the ground-truth bounding box. For element-based models, a prediction is correct if the midpoint of the predicted element falls inside the ground-truth bounding box.

2) Experimental Results: As shown in Table II, REPEEK achieves high accuracy on mobile GUI element grounding, reaching 83.8% on the ScreenSpot mobile icon subset and 96.3% on the mobile text subset, outperforming mainstream baselines at similar scale. Based on the results, we have two major observations. **First, REPEEK significantly improves upon its base model, Qwen2.5-VL.** At the same 3B scale, it boosts performance by 31.0% on mobile icons and 17.5% on mobile text, demonstrating the effectiveness of our domain-specific fine-tuning. **Second, REPEEK achieves state-of-the-art performance that is highly competitive with much larger domain-specific models.** Despite its efficient 3B-parameter scale and 0.7B-token training data, REPEEK excels on key benchmarks. On the ScreenSpot mobile icon task, it achieves 83.8% accuracy, surpassing the best-performing 3B model by 6.1% and closing the gap with much larger 7B and 72B models. Notably, on the mobile text subset, its performance is even more striking: at 96.3% accuracy, REPEEK outperforms leading 72B model such as UI-TARS-72B and Aguvis-72B. These outstanding results validate that our element-aware domain-knowledge integration is highly effective for GUI grounding tasks.

B. RQ2: GUI Navigation Performance

1) Experimental Setup: **Benchmarks.** We evaluate on three mobile GUI navigation datasets: AITZ [50], AndroidControl [49], and GUI-Odyssey [62]. AITZ is a refined extension of AITW [51], adding annotations to improve data quality. AndroidControl assesses planning and execution abilities, with two task types: *high-level*, requiring multi-step planning and execution, and *low-level*, where the model selects GUI actions based on human-annotated single-step instructions. We report results for both. GUI-Odyssey focuses on cross-app, complex navigation tasks, with each task averaging over 15 steps. Following prior work [30], [49], we report the average success rate over splits by randomness, app, device, and task for GUI-Odyssey, and randomly sample 500 tasks from each dataset for evaluation.

Baselines. We adopt state-of-the-art baselines, including Qwen2.5-VL [24] and the domain-specific model UI-TARS [32]. Prompts are carefully designed following their official papers and available code to ensure fair comparison.

Implementation details. We evaluate the performance of REPEEK-Navigation after fine-tuning on each dataset. All models are set with a minimum image-token length of 1200, and a maximum image-token length of 1344. By default, all models are prompted with action history and current screen. For UI-TARS, which uses an interleaved image-action prompt, we additionally report results when the model is prompted with action history and the last two images. For AndroidControl, which provides a11y tree element bounding boxes, we report results for both REPEEK with OmniParser and with a11y tree parsing.

Metrics. We evaluate model performance using step-level success rate. For the AITZ and AndroidControl datasets, we adopt the evaluation protocol from Qwen2.5-VL's open-source implementation, where success is determined by matching the predicted action type with the reference action and verifying that the predicted interaction pixel falls within the ground-truth bounding box. For GUI-Odyssey, since element bounding boxes are not originally provided, we manually annotate them for our sampled tasks and apply the same comparison criteria. Given that REPEEK outputs only the target element for interaction rather than specific pixel coordinates, we use the midpoint of the identified element as the evaluation coordinate for a fair comparison across all baselines.

2) Experimental Results: Table III presents the performance comparison between REPEEK and baseline models on mobile GUI navigation tasks. The evaluation reveals three key findings.

REPEEK establishes clear superiority over both large-scale general-purpose VLMs and specialized domain-specific models. REPEEK-OmniParser achieves an average success rate of 77.3%, outperforming Qwen2.5-VL-7B (68.7%) and UI-TARS-7B-1image (70.3%). Even when UI-TARS-7B is provided with increased historical context through two images, resulting in a richer and longer visual token sequence, it reaches only a 74.7% success rate. This result is still below REPEEK's performance, demonstrating REPEEK's

TABLE III: Performance Evaluation of Different Models on GUI Navigation Tasks

	Model	AITZ	AndroidControl-low	AndroidControl-high	GUI-Odyssey	Average
Zero-Shot	REPEEK-a11y	-	74.0	48.8	-	-
	REPEEK-Omniparser	56.4	72.0	51.4	47.9	56.9
Fine-Tuned	Qwen2.5-VL-3B	57.5	90.6	56.0	39.5	60.9
	Qwen2.5-VL-7B	60.5	91.6	60.8	61.7	68.7
	UI-TARS-2B-1image	55.4	84.2	62.8	61.3	65.9
	UI-TARS-2B-2image	62.0	87.6	69.0	71.4	72.5
	UI-TARS-7B-1image	57.7	87.6	66.8	68.9	70.3
	UI-TARS-7B-2image	64.7	90.0	71.4	72.7	74.7
	REPEEK-a11y	-	93.8	**72.0**	-	-
	REPEEK-Omniparser	**67.0**	**94.2**	**72.0**	**75.8**	**77.3**

strong capability for generalization in multi-step planning and execution.

REPEEK exhibits remarkable robustness across different element detection techniques. Whether using a11y tree or OmniParser, REPEEK shows consistent and high performance. For instance, on AndroidControl-low, REPEEK-a11y and REPEEK-OmniParser achieve 93.8% and 94.2%, respectively, indicating insensitivity to the input parsing technique and strong transferability.

REPEEK exhibits strong zero-shot generalization. Even under zero-shot settings, when trained on GUIAct and AMEX (which differ in observation and action space distributions), REPEEK still achieves promising results, demonstrating the generalization of REPEEK.

C. RQ3: Ablation Studies

1) Experimental Setup: **Benchmarks and metrics.** We employ identical benchmarks and metrics from RQ1 and RQ2 to evaluate the contributions of REPEEK's multistage fine-tuning strategies and element-aware fine-tuning approach to overall performance.

Multistage fine-tuning ablation. We examine the effectiveness of REPEEK's multistage fine-tuning strategy through two comparative configurations. First, the REPEEK-Only2 configuration involves fine-tuning Qwen2.5-VL-3B exclusively on the second stage dataset using the REPEEK approach, allowing us to determine whether initial grounding fine-tuning enhances navigation task performance. Second, the REPEEK-Only3 configuration applies direct fine-tuning of Qwen2.5-VL-3B on downstream navigation datasets using the REPEEK approach, enabling evaluation of whether both grounding fine-tuning in Stage 1 and general navigation fine-tuning in Stage 2 meaningfully contribute to downstream navigation capabilities.

Fine-tuning approach ablation. We conduct additional analysis to assess the specific contributions of our fine-tuning approach. For GUI grounding evaluation, we utilize TongUI [48] as our ablation baseline, given that TongUI shares the same Qwen2.5-VL backbone architecture as REPEEK, providing a controlled comparison for isolating the impact of our fine-tuning approach. For GUI navigation tasks, we establish a comparison against Qwen2.5-VL-3B fine-tuned using coordinate-level prompts, designated as Qwen2.5-VL-3B-Coord.

TABLE IV: Ablation Results on GUI Grounding Tasks

Model Name	Model Size	ScreenSpot		ScreenSpot-v2	
		Icon	Text	Icon	Text
Qwen2.5-VL	3B	52.8	78.8	-	-
Qwen2.5-VL	7B	77.7	94.9	-	-
TongUI	3B	77.7	92.6	79.6	94.4
TongUI	7B	79.5	91.9	81.5	93.1
REPEEK	3B	**83.8**	**96.3**	**82.5**	**97.6**

2) Experimental Results: Tables IV and V present the ablation study results for REPEEK across GUI element grounding and navigation tasks, examining the contributions of multistage fine-tuning and element-aware fine-tuning approach.

The ablation results demonstrate the clear effectiveness of the multistage fine-tuning strategy. Both REPEEK-Only2 and REPEEK-Only3 configurations, which skip either the grounding stage or the complete two-stage training, respectively, exhibit degraded performance compared to the full multistage fine-tuning pipeline on navigation tasks. Specifically, in zero-shot settings, REPEEK-Omniparser-Only2 achieves 55.7%, lower than the full pipeline's 56.9%. REPEEK-Omniparser-Only3 obtains 75.8%, compared to 77.3% for the complete approach. These findings confirm that the first-stage grounding-focused fine-tuning establishes a robust foundation for subsequent complex navigation reasoning capabilities. Furthermore, the second-stage general navigation fine-tuning exposes the model to diverse interaction patterns, effectively enhancing its generalization ability for downstream tasks.

The element-aware fine-tuning approach proves superior to coordinate-based alternatives. On grounding tasks, when compared against TongUI-3B, which shares the same Qwen2.5-VL-3B backbone architecture, REPEEK achieves significant performance improvements on both icon and text subsets (83.8% vs 77.7% for icons and 96.3% vs 92.6% for text on ScreenSpot). This advantage successfully transfers to

TABLE V: Ablation Results on GUI Navigation Tasks

	Model	AITZ	AndroidControl-low	AndroidControl-high	GUI-Odyssey	Average
Zero-Shot	REPEEK-a11y-Only2	-	72.0	49.0	-	-
	REPEEK-Omniparser-Only2	55.2	70.4	50.2	47.1	55.7
	REPEEK-a11y	-	**74.0**	49.2	-	-
	REPEEK-Omniparser	**56.4**	72.0	**51.4**	**47.9**	**56.9**
Fine-Tuned	Qwen2.5-VL-3B	57.5	90.6	56.0	39.5	60.9
	Qwen2.5-VL-7B	60.5	91.6	60.8	61.7	68.7
	Qwen2.5-VL-3B-Coord	65.4	93.4	71.4	75.3	76.4
	REPEEK-a11y-Only3	-	92.8	70.0	-	-
	REPEEK-Omniparser-Only3	64.5	93.2	70.0	75.4	75.8
	REPEEK-a11y	-	93.8	**72.0**	-	-
	REPEEK-Omniparser	**67.0**	**94.2**	**72.0**	**75.8**	**77.3**

navigation tasks, where training with coordinate-level prompts (Qwen2.5-VL-3B-Coord) produces inferior results compared to REPEEK, with average performance of 76.4% vs 77.3%. These results demonstrate the effectiveness of our element-aware fine-tuning approach for mobile GUI scenarios and validate that element-level understanding outperforms pixel-level coordinate prediction while generalizing well across different GUI tasks.

V. DEPLOYMENT AND LESSONS LEARNED

A. Deployment of REPEEK

REPEEK has been integrated into WeChat's testing infrastructure to support two kinds of testing: (1) *exploratory testing*, where natural language instructions are dynamically converted into executable test cases [12], [63], and (2) *robust test automation* [64], where predefined test cases written in natural language are interpreted by REPEEK and executed across different builds, configurations, and devices to validate app behavior. We deploy REPEEK as an API service leveraging the OpenAI-compatible interface [65], supporting both OmniParser-based element detection and user-provided element lists to facilitate flexible integration with existing workflows. The integration enables both manual testers and automated testing systems to efficiently validate complex user scenarios while adapting to rapidly evolving product requirements through intuitive natural language commands. To optimize REPEEK's performance within WeChat's Chinese-language environment and domain-specific UI patterns, we construct a specialized dataset containing 20,000 screenshot-instruction pairs extracted from WeChat's historical testing automation logs and use a Qwen-2.5-VL-72B model to generate high-quality instruction annotations to adapt to WeChat's UIs and testing terminology.

Deployment and efficacy evaluation. To assess both the effectiveness and efficiency of REPEEK at WeChat, We conduct a comprehensive evaluation compared to state-of-the-art VLMs. As shown in Table VI, REPEEK achieves substantial accuracy improvements (94.2%), surpassing state-of-the-art baselines including UI-TARS-7B [32] (87.3%) and pixel-level fine-tuned Qwen2.5-VL-3B [24] (93.1%). For efficiency evaluation, REPEEK uses 30% less inference time compared to the state-of-the-art open-source UI model UI-TARS-7B. Interestingly, although the element embedding module (Section III-C) introduces additional visual tokens compared to Qwen-VL-Coord, REPEEK still improves inference speed by 31%. Our investigation reveals that in typical model serving deployments, decoding (the stage where a model generates responses) is inherently slower than prefilling (the stage where a model processes the prompts) due to the auto-regressive nature. Compared to state-of-the-art VLMs that generate coordinates [24] along with chain-of-thought reasoning [31], [32], REPEEK directly generates indices of grounded elements, substantially improving efficiency.

TABLE VI: Deployment Efficacy of REPEEK at WeChat

Model Name	Model Size	Accuracy	Efficiency
Qwen2.5-VL	3B	79.9	0.74s
Qwen2.5-VL	7B	85.4	0.77s
Qwen2.5-VL-Coord	3B	93.1	0.74s
UI-TARS-2B	2B	81.3	0.58s
UI-TARS-7B	7B	87.3	0.73s
REPEEK	3B	**94.2**	**0.51s**

Notes: Efficiency is measured by average inference time.

B. Lessons Learned

We next summarize three major lessons learned during our development and deployment of REPEEK at WeChat.

There is a need to derive optimizations considering infrastructure constraints. Industrial GUI testing systems must handle high-frequency requests while operating under strict computational budget constraints, particularly regarding expensive GPU resources dedicated to testing infrastructure. In such environments, large-parameter models and API calls to expensive proprietary models are both prohibitively costly. Rather than simply accepting these constraints, we actively derive optimizations from them. First, these budget limitations guide us to deliberately select Qwen2.5-VL-3B as our

backbone, a lightweight model that balances capability with efficiency. Remarkably, our experiments demonstrate that this compact model, when properly fine-tuned for GUI-specific tasks, can surpass much larger state-of-the-art models on relevant benchmarks while maintaining the computational efficiency required for large-scale industrial deployment. More importantly, we discover that modern LLM serving infrastructure exhibits significant performance differences between prefilling and decoding phases, with decoding being substantially slower. This characteristic inspires us to design REPEEK to generate concise element indices rather than verbose coordinate predictions or chain-of-thought reasoning. This design choice is therefore a direct application of our principle: by considering the infrastructure's primary bottleneck, we accelerate inference speed.

There is a need to incorporate domain knowledge with general-purpose AI models. Throughout the development of REPEEK, we discover that leveraging domain-specific knowledge is crucial for achieving both high accuracy and robustness in GUI testing. Unlike natural images, GUIs are structured and rendered based on a hierarchy of interactive elements rather than unorganized pixels. Although traditional LLM-based GUI testing approaches that rely solely on accessibility trees have limitations because accessibility trees are often fragile and incomplete, these approaches still offer valuable insights into the underlying structure of GUI applications. This domain knowledge, particularly the understanding that GUIs are element-driven rather than pixel-based, guides us to explicitly detect UI elements and perform reasoning at the element level. By adopting element-level classification instead of direct pixel coordinate prediction, REPEEK better aligns its behavior with the actual logic and structure of GUIs, substantially improving both performance and efficiency.

Task-specific fine-tuning is essential for optimal performance. Our experience demonstrates that fine-tuning is crucial for achieving optimal performance in GUI automation tasks. While general-purpose VLMs exhibit impressive capabilities across diverse domains, our experiments show their suboptimal performance on GUI testing, especially for challenging testing tasks at WeChat. UI-specific training significantly addresses this limitation by teaching models to understand GUI-specific visual elements, layout conventions, and interaction semantics that differ substantially from natural image understanding. Furthermore, when transferring our REPEEK-Base model to the WeChat deployment scenario, we discover that domain-specific fine-tuning on WeChat's interface patterns yields substantial performance improvements.

VI. RELATED WORK

A. Automated GUI Testing with LLMs and VLMs

Recent advances in LLMs [19]–[21], [23], [66] and VLMs [24]–[26] have opened new avenues for automated GUI testing. Existing work can be broadly categorized into three approaches. The first category focuses on improving test coverage through enhanced exploration strategies. The work [45], [47], [67]–[70] in this category leverages LLMs/VLMs to guide testing tools toward unexplored functionalities, achieving higher code coverage and activity coverage compared to traditional automated testing approaches. The second category emphasizes functional testing through natural language instructions, where systems such as Guardian [43], AutoDroid [41], and DroidAgent [70] enable intent-driven testing by either following the given test objectives or autonomously generating realistic testing tasks. A distinct third approach is represented by GUIPilot [46], which addresses application-specific bug detection by automatically comparing design mock-ups with actual implementations. GUIPilot uses VLMs to detect both screen inconsistencies and GUI transition inconsistencies between designs and implementations.

B. VLMs for GUI Automation and Testing

Early work [27], [37], [39] employed Set-of-Mark [60] prompting to design complex workflow agents for GUI automation. Due to the limited GUI domain capabilities [27] of general-purpose VLMs, recent researchers have focused on enhancing GUI-specific abilities through specialized training approaches. One line of research [28]–[30], [34], [36] builds models to achieve precise grounding of GUI elements based on natural language instructions. These research efforts construct comprehensive data collection pipelines to gather large-scale grounding datasets that cover diverse GUI scenarios and interactions. By training on such specialized datasets, these models demonstrate improved capabilities in understanding and localizing specific GUI components when given textual descriptions. Another research direction explores native agent models for multi-step GUI automation [31]–[35]. Representative work includes Aguvis [31], which enhances planning with an inner monologue mechanism, and UI-TARS [32], which employs System 2 reasoning for complex workflows. It is worth noting that ShowUI [33] attempts to perceive structured information from UI screenshots by creating UI connection graphs through RGB values. However, their connection graphs are primarily used for efficiency improvements while still fundamentally employing pixel-level reasoning. In contrast to existing approaches, our work is the first to explicitly detect and reason over discrete GUI elements rather than relying solely on holistic image understanding.

VII. CONCLUSION

In this paper, we have reported our experience of developing and deploying REPEEK, a novel approach to fine-tuning efficient VLMs for large-scale, industrial GUI testing at WeChat. In contrast to existing VLMs for GUI testing that rely on pixel-based coordinate prediction over raw screenshots, REPEEK introduces an element-aware paradigm that transforms the task into element-level classification, satisfying the effectiveness and efficiency requirements for industrial settings. Comprehensive experiments on public benchmarks and deployment demonstrate the effectiveness, efficiency, and cost-effectiveness of REPEEK. We have summarized lessons learned from developing and deploying REPEEK at WeChat.

ACKNOWLEDGMENTS

Tao Xie is with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China; Fudan University Institute of Systems for Advanced Computing, Shanghai, China; Shanghai Institute of Systems for Open Computing, Shanghai, China. Dezhi Ran and Tao Xie are partially supported by the National Natural Science Foundation of China under Grant Nos. 623B2006 and 92464301. Dezhi Ran is partially supported by a Hunyuan Scholar Award. Wei Yang is supported by an Amazon Research Award.

REFERENCES

[1] J.-W. Lin, N. Salehnamadi, and S. Malek, "Test automation in open-source Android apps: A large-scale empirical study," in *ASE*, 2020, pp. 1078–1089.

[2] D. Ran, H. Wang, W. Wang, and T. Xie, "Badge: prioritizing UI events with hierarchical multi-armed bandits for automated UI testing," in *ICSE*, 2023, pp. 894–905.

[3] J.-W. Lin, N. Salehnamadi, and S. Malek, "Route: Roads not taken in UI testing," *TOSEM*, vol. 32, no. 3, pp. 1–25, 2023.

[4] D. Ran, Y. Fu, Y. He, T. Chen, X. Tang, and T. Xie, "Path toward elderly friendly mobile apps," *Computer*, vol. 57, no. 06, pp. 29–39, June 2024.

[5] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra, "Automating test automation," in *ICSE*, 2012, pp. 881–891.

[6] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *STVR*, vol. 12, no. 4, pp. 219–249, 2002.

[7] M. Viggiato, D. Paas, C. Buzon, and C.-P. Bezemer, "Using natural language processing techniques to improve manual test case descriptions," in *ICSE-SEIP*, 2022, pp. 311–320.

[8] C. Hu and I. Neamtiu, "Automating GUI testing for Android applications," in *ICSE-AST*, 2011, pp. 77–83.

[9] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *FASE*. Springer, 2013, pp. 250–265.

[10] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight UI-guided test input generator for Android," in *ICSE-C*, 2017, pp. 23–26.

[11] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *OOPSLA*, 2013, pp. 641–660.

[12] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for Android: Towards getting there in an industrial case," in *ICSE-SEIP*, 2017, pp. 253–262.

[13] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su, "Practical GUI testing of Android applications via model abstraction and refinement," in *ICSE*, 2019, pp. 269–280.

[14] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of Android applications," in *ISSTA*, 2020, pp. 153–164.

[15] D. Ran, Z. Li, C. Liu, W. Wang, W. Meng, X. Wu, H. Jin, J. Cui, X. Tang, and T. Xie, "Automated visual testing for mobile apps in an industrial setting," in *ICSE-SEIP*, 2022, pp. 55–64.

[16] E. Alégroth, R. Feldt, and L. Ryrholm, "Visual GUI testing in practice: challenges, problems and limitations," *Empirical Software Engineering*, vol. 20, no. 3, pp. 694–744, 2015.

[17] R. Coppola, M. Morisio, and M. Torchiano, "Mobile GUI testing fragility: a study on open-source Android applications," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 67–90, 2018.

[18] D. Ran, Z. Song, W. Wang, W. Yang, and T. Xie, "TaOPT: Tool-agnostic optimization of parallelized automated mobile UI testing," in *ASPLOS*, 2025, pp. 1251–1265.

[19] OpenAI, "GPT-4 technical report," 2023.

[20] A. Yang, B. Yang, B. Hui, B. Zheng, B. Yu, C. Zhou, C. Li, C. Li, D. Liu, F. Huang *et al.*, "Qwen2 technical report," *arXiv preprint arXiv:2407.10671*, 2024.

[21] A. Liu, B. Feng, B. Wang, B. Wang, B. Liu, C. Zhao, C. Dengr, C. Ruan, D. Dai, D. Guo *et al.*, "DeepSeek-v2: A strong, economical, and efficient mixture-of-experts language model," *arXiv preprint arXiv:2405.04434*, 2024.

[22] D. Ran, M. Wu, W. Yang, and T. Xie, "Foundation model engineering: engineering foundation models just as engineering software," *TOSEM*, vol. 34, no. 5, pp. 1–18, 2025.

[23] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[24] S. Bai, K. Chen, X. Liu, J. Wang, W. Ge, S. Song, K. Dang, P. Wang, S. Wang, J. Tang, H. Zhong, Y. Zhu, M. Yang, Z. Li, J. Wan, P. Wang, W. Ding, Z. Fu, Y. Xu, J. Ye, X. Zhang, T. Xie, Z. Cheng, H. Zhang, Z. Yang, H. Xu, and J. Lin, "Qwen2.5-vl technical report," *arXiv preprint arXiv:2502.13923*, 2025.

[25] H. Liu, C. Li, Q. Wu, and Y. J. Lee, "Visual instruction tuning," *NeurIPS*, vol. 36, pp. 34 892–34 916, 2023.

[26] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford *et al.*, "GPT-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.

[27] B. Zheng, B. Gou, J. Kil, H. Sun, and Y. Su, "GPT-4V(ision) is a generalist web agent, if grounded," *arXiv preprint arXiv:2401.01614*, 2024.

[28] K. Cheng, Q. Sun, Y. Chu, F. Xu, Y. Li, J. Zhang, and Z. Wu, "Seeclick: Harnessing GUI grounding for advanced visual GUI agents," *arXiv preprint arXiv:2401.10935*, 2024.

[29] Y. Yang, Y. Wang, D. Li, Z. Luo, B. Chen, C. Huang, and J. Li, "Aria-UI: Visual grounding for GUI instructions," *arXiv preprint arXiv:2412.16256*, 2024.

[30] B. Gou, R. Wang, B. Zheng, Y. Xie, C. Chang, Y. Shu, H. Sun, and Y. Su, "Navigating the digital world as humans do: Universal visual grounding for GUI agents," *arXiv preprint arXiv:2410.05243*, 2024.

[31] Y. Xu, Z. Wang, J. Wang, D. Lu, T. Xie, A. Saha, D. Sahoo, T. Yu, and C. Xiong, "Aguvis: Unified pure vision agents for autonomous GUI interaction," in *ICML*, 2025.

[32] Y. Qin, Y. Ye, J. Fang, H. Wang, S. Liang, S. Tian, J. Zhang, J. Li, Y. Li, S. Huang *et al.*, "UI-tars: Pioneering automated GUI interaction with native agents," *arXiv preprint arXiv:2501.12326*, 2025.

[33] K. Q. Lin, L. Li, D. Gao, Z. Yang, S. Wu, Z. Bai, S. W. Lei, L. Wang, and M. Z. Shou, "ShowUI: One vision-language-action model for GUI visual agent," in *CVPR*, 2025, pp. 19 498–19 508.

[34] Z. Wu, Z. Wu, F. Xu, Y. Wang, Q. Sun, C. Jia, K. Cheng, Z. Ding, L. Chen, P. P. Liang *et al.*, "OS-ATLAS: Foundation action model for generalist GUI agents," in *ICLR*, 2025.

[35] W. Hong, W. Wang, Q. Lv, J. Xu, W. Yu, J. Ji, Y. Wang, Z. Wang, Y. Dong, M. Ding *et al.*, "Cogagent: A visual language model for GUI agents," in *CVPR*, 2024, pp. 14 281–14 290.

[36] P. Shaw, M. Joshi, J. Cohan, J. Berant, P. Pasupat, H. Hu, U. Khandelwal, K. Lee, and K. N. Toutanova, "From pixels to UI actions: Learning to follow instructions via graphical user interfaces," *NeurIPS*, vol. 36, pp. 34 354–34 370, 2023.

[37] Y. Li, C. Zhang, W. Yang, B. Fu, P. Cheng, X. Chen, L. Chen, and Y. Wei, "Appagent v2: Advanced agent for flexible mobile interactions," *arXiv preprint arXiv:2408.11824*, 2024.

[38] C. Zhang, Z. Yang, J. Liu, Y. Li, Y. Han, X. Chen, Z. Huang, B. Fu, and G. Yu, "Appagent: Multimodal agents as smartphone users," in *CHI*, 2025, pp. 1–20.

[39] J. Wang, H. Xu, J. Ye, M. Yan, W. Shen, J. Zhang, F. Huang, and J. Sang, "Mobile-agent: Autonomous multi-modal mobile device agent with visual perception," in *ICLR 2024 Workshop on Large Language Model (LLM) Agents*, 2024.

[40] J. Wang, H. Xu, H. Jia, X. Zhang, M. Yan, W. Shen, J. Zhang, F. Huang, and J. Sang, "Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration," *NeurIPS*, vol. 37, pp. 2686–2710, 2024.

[41] H. Wen, Y. Li, G. Liu, S. Zhao, T. Yu, T. J.-J. Li, S. Jiang, Y. Liu, Y. Zhang, and Y. Liu, "Autodroid: LLM-powered task automation in Android," in *MobiCom*, 2024, pp. 543–557.

[42] D. Ran, M. Wu, Y. Cao, A. Marron, D. Harel, and T. Xie, "An infrastructure software perspective toward computation offloading between executable specifications and foundation models," *Science China Information Sciences*, vol. 68, no. 4, 2025.

[43] D. Ran, H. Wang, Z. Song, M. Wu, Y. Cao, Y. Zhang, W. Yang, and T. Xie, "Guardian: A runtime framework for LLM-based UI exploration," in *ISSTA*, 2024, pp. 958–970.

[44] Android Developers. (2025) Accessibility overview — Android developers. Accessed: 2025-07-31. [Online]. Available: https://developer.android.com/guide/topics/ui/accessibility

[45] B. F. Demissie, Y. N. Tun, L. K. Shar, and M. Ceccato, "VLM-Fuzz: Vision language model assisted recursive depth-first search exploration for effective UI testing of Android Apps," *arXiv preprint arXiv:2504.11675*, 2025.

[46] R. Liu, X. Teoh, Y. Lin, G. Chen, R. Ren, D. Poshyvanyk, and J. S. Dong, "GUIPilot: A consistency-based mobile GUI testing approach for detecting application-specific bugs," *ISSTA*, 2025.

[47] S. Wang, S. Wang, Y. Fan, X. Li, and Y. Liu, "Leveraging large vision-language model for better automatic web GUI testing," in *ICSME*, 2024, pp. 125–137.

[48] B. Zhang, Z. Shang, Z. Gao, W. Zhang, R. Xie, X. Ma, T. Yuan, X. Wu, S.-C. Zhu, and Q. Li, "TongUI: Building generalized GUI agents by learning from multimodal web tutorials," *arXiv preprint arXiv:2504.12679*, 2025.

[49] W. Li, W. E. Bishop, A. Li, C. Rawles, F. Campbell-Ajala, D. Tyamagundlu, and O. Riva, "On the effects of data scale on UI control agents," *NeurIPS*, vol. 37, pp. 92 130–92 154, 2024.

[50] J. Zhang, J. Wu, Y. Teng, M. Liao, N. Xu, X. Xiao, Z. Wei, and D. Tang, "Android in the zoo: Chain-of-action-thought for GUI agents," *arXiv preprint arXiv:2403.02713*, 2024.

[51] C. Rawles, A. Li, D. Rodriguez, O. Riva, and T. Lillicrap, "Androidinthewild: A large-scale dataset for Android device control," *NeurIPS*, vol. 36, pp. 59 708–59 728, 2023.

[52] Y. Lu, J. Yang, Y. Shen, and A. Awadallah, "Omniparser for pure vision based GUI agent," *arXiv preprint arXiv:2408.00203*, 2024.

[53] PKU-ASE-RISE, "Repeek," https://github.com/PKU-ASE-RISE/Repeek, 2025, GitHub repository.

[54] Y. Li, G. Li, L. He, J. Zheng, H. Li, and Z. Guan, "Widget captioning: Generating natural language description for mobile user interface elements," *arXiv preprint arXiv:2010.04295*, 2020.

[55] Y. Chai, S. Huang, Y. Niu, H. Xiao, L. Liu, D. Zhang, S. Ren, and H. Li, "Amex: Android multi-annotation expo dataset for mobile GUI agents," *arXiv preprint arXiv:2407.17490*, 2024.

[56] W. Chen, J. Cui, J. Hu, Y. Qin, J. Fang, Y. Zhao, C. Wang, J. Liu, G. Chen, Y. Huo *et al.*, "Guicourse: From general vision language models to versatile GUI agents," *arXiv preprint arXiv:2406.11317*, 2024.

[57] D. Foley and J. Danskin, "Ultra-performance Pascal GPU and NVLink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[58] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters," in *KDD*, 2020, pp. 3505–3506.

[59] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[60] J. Yang, H. Zhang, F. Li, X. Zou, C. Li, and J. Gao, "Set-of-mark prompting unleashes extraordinary visual grounding in GPT-4V," *arXiv preprint arXiv:2310.11441*, 2023.

[61] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[62] Q. Lu, W. Shao, Z. Liu, F. Meng, B. Li, B. Chen, S. Huang, K. Zhang, Y. Qiao, and P. Luo, "GUI odyssey: A comprehensive dataset for cross-app GUI navigation on mobile devices," *arXiv preprint arXiv:2406.08451*, 2024.

[63] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for Android: Are we really there yet in an industrial case?" in *FSE*, 2016, pp. 987–992.

[64] W. Lam, Z. Wu, D. Li, W. Wang, H. Zheng, H. Luo, P. Yan, Y. Deng, and T. Xie, "Record and replay for Android: Are we there yet in industrial cases?" in *FSE*, 2017, pp. 854–859.

[65] OpenAI, "OpenAI Python library," 2025, accessed: 2025-08-02. [Online]. Available: https://github.com/openai/openai-python

[66] D. Ran, L. Li, L. Zhu, Y. Cao, L. Zhao, X. Tan, G. Liang, Q. Wang, and T. Xie, "Efficient and robust security-patch localization for disclosed OSS vulnerabilities with fine-tuned LLMs in an industrial setting," in *FSE*, 2025, pp. 262–273.

[67] C. Wang, T. Liu, Y. Zhao, M. Yang, and H. Wang, "LLMDroid: Enhancing automated mobile app GUI testing coverage with large language model guidance," *FSE*, 2025.

[68] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Make LLM a testing expert: Bringing human-like interaction to mobile GUI testing via functionality-aware decisions," in *ICSE*, 2024, pp. 1–13.

[69] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, "Fill in the blank: Context-aware automated text input generation for mobile GUI testing," in *ICSE*, 2023, pp. 1355–1367.

[70] J. Yoon, R. Feldt, and S. Yoo, "Autonomous large language model agents enabling intent-driven mobile GUI testing," *arXiv preprint arXiv:2311.08649*, 2023.