# V-Droid: Advancing Mobile GUI Agent Through Generative Verifiers


Gaole Dai*
Nanyang Technological University
gaole001@e.ntu.edu.sg

Shiqi Jiang†
Microsoft Research
shijiang@microsoft.com

Ting Cao
Tsinghua University
tingcao@air.tsinghua.edu.cn

Yuanchun Li
Tsinghua University
liyuanchun@air.tsinghua.edu.cn

Yuqing Yang
Microsoft Research
yuqyang@microsoft.com

Rui Tan
Nanyang Technological University
tanrui@ntu.edu.sg

Mo Li†
HKUST
lim@cse.ust.hk

Lili Qiu
Microsoft Research
liliqiu@microsoft.com



## Abstract

We propose V-Droid, a mobile GUI task automation agent. Unlike previous mobile agents that utilize Large Language Models (LLMs) as generators to directly generate actions at each step, V-Droid employs LLMs as verifiers to evaluate candidate actions before making final decisions. To realize this novel paradigm, we introduce a comprehensive framework for constructing verifier-driven mobile agents: the discretized action space construction coupled with the prefilling-only workflow to accelerate the verification process, the pair-wise progress preference training to significantly enhance the verifier's decision-making capabilities, and the scalable human-agent joint annotation scheme to efficiently collect the necessary data at scale.

V-Droid obtains a substantial task success rate across several public mobile task automation benchmarks: 59.5% on AndroidWorld, 38.3% on AndroidLab, and 49% on MobileAgentBench, surpassing existing agents by 5.2%, 2.1%, and 9%, respectively. Furthermore, V-Droid achieves a remarkably low latency of 4.3s per step, which is 6.1× faster compared with existing mobile agents.



## CCS Concepts

• **Human-centered computing** → **Ubiquitous and mobile computing**; • **Computing methodologies** → **Artificial intelligence**.

## Keywords

Large Language Models, GUI Agent, Generative Verifier

**ACM Reference Format:**
Gaole Dai, Shiqi Jiang, Ting Cao, Yuanchun Li, Yuqing Yang, Rui Tan, Mo Li, and Lili Qiu. 2026. V-Droid: Advancing Mobile GUI Agent Through Generative Verifiers. In *The 32nd Annual International Conference on Mobile Computing and Networking (ACM MobiCom '26), October 26-30, 2026, Austin, Texas, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn


*Work done during internship at Microsoft Research.
†Corresponding authors.



*ACM MobiCom '26, Austin, Texas, USA*



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

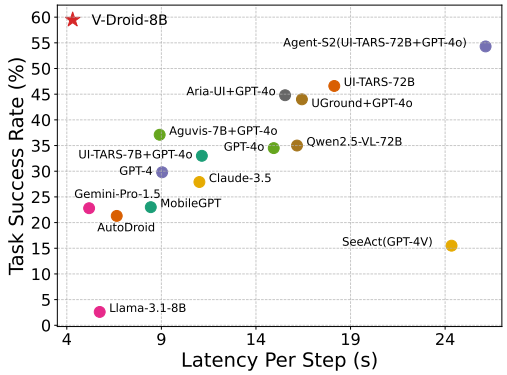




**Figure 1: Task success rate and latency per step of current mobile agents [1] and V-Droid evaluated on AndroidWorld benchmark. The latency of 2B, 7B and 8B agents are measured on 2× Nvidia 4090. For 72B or MoE agents, the latency is measure on 4× Nvidia A100 80G.**

## 1 Introduction

Controlling mobile devices via natural language has been a long-standing aspiration in the mobile domain [11, 23, 31, 32], promising opportunities to automate repetitive tasks and elevate user convenience. Unlike API-based agents that rely on predefined function calls [4, 12], mobile GUI agents simulate human interactions, allowing them to operate across diverse applications through Graphical User Interface (GUI). However, developing such an agent poses significant challenges: it needs to not only interpret on-screen content but also make reasonable decisions to execute multi-step tasks within dynamic and complex GUI environments.

[1] The task success rate achieved by GPT-4 and 4o, Claude, Llama, Gemini and Qwen is measured with the default prompt templates *i.e.,* T3A and M3A, in AndroidWorld benchmark.

In recent years, a variety of LLM-powered mobile GUI agents have been proposed [1, 7, 9, 18, 28, 29, 31, 35, 39]. These agents typically *utilize LLMs as generators*, generating decisions (*e.g.,* reasoning and actions) based on the current task states (*e.g.,* user interfaces, task descriptions), leveraging the contextual understanding and reasoning abilities inherent to LLMs. These agents, however, fall short of meeting practical deployments. Fig.1 presents the performance of state-of-the-art (SOTA) mobile agents in terms of task success rate (SR) and stepwise latency on the AndroidWorld [20]. As shown, the highest SR achieved by existing agents is only 54.3%, significantly lower than the human performance of 80%. Additionally, latency remains a critical challenge. Agent-S2 [1], powered by GPT-4o and UI-TARS-72B [18], takes over 25 seconds for a single step.

The suboptimal SR is predominantly hindered by the limited decision-making capabilities of existing agents when performing mobile tasks. While techniques such as prompt engineering [30, 31] and GUI fine-tuning [7, 18, 35] are commonly employed, these methods primarily focus on enhancing general performance of LLM, such as instruction-following, or refining agents' adaptability to mobile interfaces. However, they fail to adequately address task-specific, multi-step decision-making challenges, particularly in the context of mobile GUI control.

The high latency of existing mobile agents primarily stems from the autoregressive decoding mechanism of LLMs. These agents generate multiple tokens sequentially for each decision at every step. For instance, SeeAct [39] generates around 100 tokens per step for one decision. Moreover, techniques like Chain-of-Thought (CoT) [30] and ReAct [36] are widely employed to enhance reasoning and mitigate hallucination, further increasing output length, worsening latency issues.

The fundamental research problem in developing practical mobile agents lies in enhancing their decision-making capabilities for mobile tasks while simultaneously maintain reasonable latency. To achieve this goal, we introduce V-Droid, which, as illustrated in Fig. 1, achieves a marked improvement in task success rate, refreshing the record to 59.5% on the AndroidWorld benchmark, while achieves 6.1× speed up on the step-wise latency.

The key idea behind V-Droid is *transforming the paradigm of mobile agents by using LLM as verifiers instead of generators*, which is illustrated in Fig. 2: rather than directly generating the final decision, when LLMs are used as verifiers, potential actions are first extracted. The verifier-driven agents then explicitly evaluate each candidate action *e.g.,* by prompting, "*Is action X helpful for completing the task*?" Following these evaluations, the agent selects the action with the highest value, *e.g.,* the likelihood of generating a 'Yes' token.

Essentially, the verifier-driven architecture decouples one action decision-making process into two discrete processes: action extraction and action verification. This decoupling offers substantial advantages for advancing mobile agents: (i) Intuitively, verifying an answer is much easier than generating one from scratch, a phenomenon known as the generation-verification gap [15, 24]. Instead of directly making decisions within an expansive and infinite action space, the verifier-driven agent evaluates actions within an extractable and enumerable space, thereby simplifying the decision-making process. More importantly, for mobile devices, the interactive UI elements is rather limited (see § 2.2) at each task step.

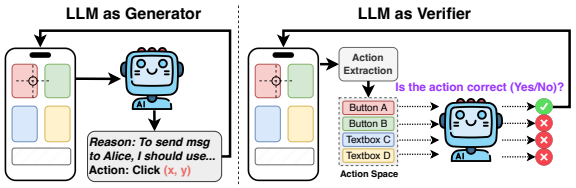



**Figure 2: The key differences in agent architecture between using LLMs as generators and as verifiers for decision-making: rather than directly determine actions based on states, verifier-driven agents explicitly evaluate each action before arriving at the decision.**

(ii) The verification process typically requires generating fewer tokens, such as simple outputs like "Yes" or "No", which dramatically reduces latency. Moreover, actions to be verified can be processed in batches, fully leveraging hardware parallelism and further improving efficiency.

However, to fully unleash the potential of verifier-driven mobile agents, several technical challenges must be addressed: (i) Effectively extracting action from UI and constructing clean and complete action space, while efficiently verifying multiple available actions at each task execution step. (ii) Designing effective training methods for the verifier to improve decision-making capabilities, particularly since directly utilizing pre-trained LLMs as verifiers is inadequate as detailed in § 2.2. (iii) Collecting and organizing the necessary data to support the training process at scale.

To address these challenges, V-Droid introduces holistic designs for building a verifier-driven mobile agent, including:

**Verifier-driven agent workflow.** At each task step, the workflow of V-Droid encompasses three main stages: extracting the action space, scoring with the verifier, and executing the selected action. First, we introduce a lightweight action extractor capable of accurately constructing and augmenting the action space for subsequent verification based on GUI representations, *e.g.,* the Android Accessibility Tree [2]. Next, available actions are verified using a prefilling-only approach, thus eliminating the constraints on LLM decoding. Moreover, multiple actions are verified in batches, leveraging prefix caching to significantly reduce latency. Finally, the action with the highest estimated score is executed.

**Pair-wise process preference ($P^3$) training method.** To improve the decision-making capabilities of V-Droid, we introduce a pair-wise preference training strategy tailored for verifiers. Unlike prior post-training approaches that rely solely on mobile GUI data, our training leverages labeled mobile task trajectories with fine-grained process supervision [15]. It enables the verifier to prioritize correct actions by assigning them higher scores while penalizing incorrect actions at each task step. This approach significantly enhances the task-specific decision-making proficiency of V-Droid.

**Scalable human-agent joint annotation scheme.** To collect the required fine-grained task trajectories, which is absent in existing datasets, we propose a human-agent joint annotation scheme based on the observation: when V-Droid verifies a group of actions at each task step, the entropy of the assigned scores strongly correlates with step-wise correctness. Thus, we utilize a trained

verifier to produce initial annotations, limiting human involvement to correcting only the erroneous annotations identified by the agent. Moreover, the annotation and training process is conducted iteratively, allowing the agent to be progressively trained on increasingly larger datasets, thereby effectively minimizing human labeling efforts while scaling the training process.

The verifier in V-Droid, which uses the small language model (SLM), Llama-3.1-8B, as the backbone, is trained using the $P^3$ method across four iterative training rounds with 110K samples collaboratively annotated by humans and agents. We evaluate V-Droid on three public, realistic task benchmarks: AndroidWorld [20], AndroidLab [34], and the MobileAgentBench [26]. On these benchmarks, V-Droid refreshes the task success rates to 59.5%, 38.3%, and 49%, surpassing previous best-performing agents by absolute margins of 5.2%, 2.1%, and 9.0%, respectively. Compared to other SLM-based agents, V-Droid achieves significant SR improvements of 50%, 22%, and 15% on these benchmarks. In addition, V-Droid delivers a 6.1× speedup over prior SOTA mobile agents.

In summary, we make the following contributions:

- We introduce V-Droid, the first verifier-driven mobile agent framework, accompanied by comprehensive design principles.
- We propose the pair-wise process preference training method, demonstrating its effectiveness in enhancing the decision-making capabilities for mobile GUI task.
- We develop a human-agent joint annotation approach, enabling scalable training of mobile agents.
- V-Droid significantly outperforms previous SOTA in task success rates on multiple public benchmarks while reducing latency by 6.1×.

## 2 Related Work and Motivation

### 2.1 LLM Powered Mobile GUI Agent

Recently, the emergence of numerous LLM-powered mobile agents [5, 14, 14, 25, 27, 31] has been observed. The introduction of large language models (LLMs) has significantly improved the ability of mobile agents to comprehend context and generate effective actions. Existing agents typically employ the following strategies to enhance performance.

Most mobile agents [5, 14, 14, 25, 27, 31, 39] leverage prompt engineering techniques to optimize task execution. For instance, SeeAct [39] employs a ReAct-style [36] prompt to break down tasks into manageable steps, thereby reducing errors and mitigating hallucinations in the LLM's outputs. Additionally, agents like AutoDroid [31] and MobileGPT [11] collect task completion traces during offline preprocessing stages for specific applications. These traces are integrated with the memory of pre-trained LLMs, enabling enhanced performance tailored to particular applications.

Several recent works, such as UGround [7], Aria-UI [35], UI-TARS [18], and Ferret-UI [37], propose training grounding models that leverage pre-trained LLMs to comprehensively interpret and interact with the UI. These approaches exploit the inherent reasoning capabilities of LLMs for mobile tasks. However, despite the rapid advances in LLMs driven by large-scale pre-training, the lack of training corpora specifically tailored to mobile GUI tasks leaves existing models insufficiently prepared for reliable mobile task automation, as demonstrated in Fig. 1.

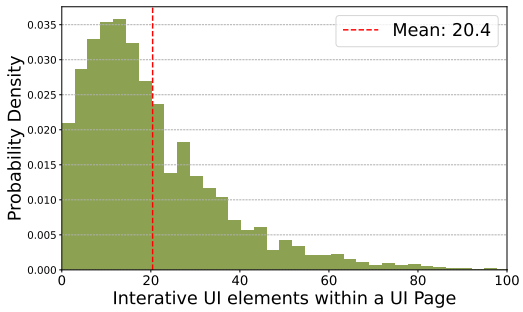



**Figure 3: The distribution of interactive UI elements within each UI page by analyzing around $25,000$ real-world UI screens from the public dataset [13].**

In addition to utilizing pre-trained LLMs for mobile agents, researchers have explored GUI fine-tuning strategies to improve task execution [16, 19]. This involves adapting a pre-trained language model using domain-specific data, such as annotated screenshots and GUI representations. While GUI fine-tuning enhances the model's ability to accurately interpret and interact with GUI elements [6, 38], it remains insufficient for facilitating task-specific reasoning and multi-step decision-making, especially in dynamic and complex GUI environments, as evidenced in Fig. 1.

Apart from the accuracy, few existing studies address latency optimization for mobile agents. MobileGPT [11] attempts to cache execution traces for tasks successfully completed. However, this approach lacks scalability due to the diverse range of tasks and applications. As illustrated in Fig. 1, most existing agents require more than 10 seconds per step, highlighting significant inefficiencies in task execution.

The fundamental challenges of improving decision-making capabilities while reducing decision-making latency for mobile agents remain unresolved.

### 2.2 Opportunity and Challenges

We tackle this challenge by proposing a novel approach: *using LLMs as verifiers instead of generators* for mobile agents.

Fig. 2 illustrates the key difference in agent architecture between using LLMs as generators and using LLMs as verifiers. The feasibility of verifier-driven agents depends on the presence of an enumerable and extractable action space. In the context of GUI automation tasks on mobile devices, this prerequisite is entirely met. Owing to the constrained screen size and the inherent interaction patterns of touchscreen interfaces, both the types of actions and the number of interactive UI elements on a single page are limited.

Fig.3 illustrates the distribution of interactive UI elements within each GUI state. As depicted, the interactive action space on mobile devices is generally constrained to approximately 20 elements on average. Although we have a unique opportunity to build verifier-driven agents for mobile tasks, several technical challenges are not well addressed.

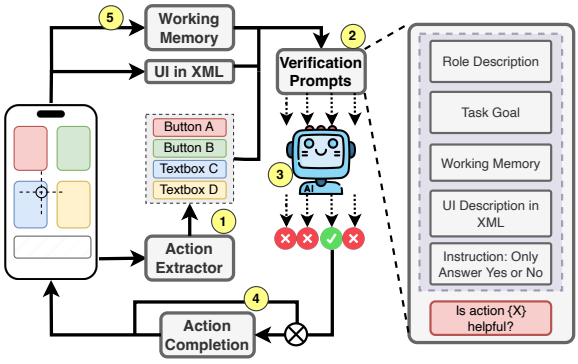



**Figure 4: The Workflow of V-Droid: ① Extracting actions from UI and supplementing default actions; ② Constructing verification prompts with the template for candidate actions; ③ Scoring with the verifier in batch with prefix caching; ④ Completing and executing the selected action; ⑤ Updating the working memory.**

Firstly, the detailed architecture and workflow of the verifier-driven agent remain ambiguous. In particular, constructing a well-defined action space is a nontrivial challenge. The supported action type varies on different UI elements and certain useful actions may not be explicitly visible, *e.g., navigate home*. Furthermore, some actions are inherently continuous rather than discrete, *e.g., type text*. Beyond that, efficiently verifying a set of actions also poses difficulties, especially considering the best utilizing the hardware parallelism.

Secondly, directly utilizing a pre-trained LLM as the verifier is not a feasible solution for mobile agents. For instance, Llama-3.1-8B, fail to complete any tasks within the benchmark (detailed in § 6.4). Even powerful LLMs, *i.e.,* GPT-4 as the verifier, achieve only 34.5% task success rate on the AndroidWorld benchmark. The challenge of effectively fine-tuning such a verifier remains unresolved.

Thirdly, the data necessary to enable effective training, particularly fine-grained labeled task trajectories, is, to the best of our knowledge, absent in publicly available datasets. The collection and annotation of such data at scale present a significant challenge, posing a critical obstacle in the development of verifier-driven agents.

To this end, in V-Droid, we introduce a comprehensive framework that integrates holistic designs to develop verifier-driven mobile agents, encompassing the agent architecture, training methodologies, and data collection strategies. In the following, we present these components in detail, starting with the workflow of V-Droid.

## 3 Workflow of V-Droid

Fig. 4 illustrates the workflow of V-Droid. As a verifier-driven agent, V-Droid requires the enumeration of candidate actions prior to estimating the optimal action at each step. To achieve this, V-Droid employs a rule-based action candidate extractor to obtain actions from the current UI state. Each action candidate is then encapsulated using a predefined prompt template. Following this, the fine-tuned LLM as the verifier is utilized to evaluate and assign scores to these candidates in batch. The action with the highest score is selected and executed. In the following, we provide an in-depth explanation of each module.

### 3.1 Constructing Action Space

A task $\kappa$ is automated through a sequence of steps. At any given step $t$, we extract a set of actions $\mathcal{A}_\kappa(t)$, representing the potential interactions available within the UI state $\mathcal{S}_\kappa(t)$ associated with the current step $t$, collectively defining the action space. This action space generally comprises two categories: UI-dependent and -independent actions, the latter being default actions that are not explicitly visible.

The UI-dependent action space is defined by the basic action types and the interactive UI elements available in the current UI. Owing to the inherent nature of touchscreen interactions on mobile devices, the set of basic action types is relatively limited. Specifically, we identify the following basic actions: *click*, *long-press*, *scroll*, *type text*, and *clear text*.

To identify interactive UI elements, the XML representation of the UI state $\mathcal{S}_\kappa(t)$ is directly analyzed, extracted using the Android Accessibility Service [2]. A rule-based extractor is applied to detect UI elements, such as button, checkbox, and textbox, that are clickable, long-clickable, scrollable, or editable. These identified elements are then mapped to their corresponding basic action types. For example, and editable text box $\beta$ would be mapped to the action *'input {content} to $\beta$'*. To prevent a combinatorial explosion of candidates, the standard on-screen keyboard is disabled, and text input is instead performed directly via Android commands [3].

In addition to the UI-dependent actions, there exist actions that are not visible in the current UI but are essential for device control and task completion. Specifically, we supplement every action space $\mathcal{A}_\kappa(t)$ with the following default actions: *open apps*, *wait*, *navigate home*, *navigate back*, *complete task*, and *answer users' question*.

The analysis presented in § 7 demonstrates that the rule-based extraction employed by V-Droid effectively encompasses the majority of interactions observed in real-world mobile tasks, thereby ensuring comprehensive practical coverage without unnecessarily complicating the action space.

### 3.2 Scoring with Verifier

Once the action space $\mathcal{A}_\kappa(t)$ is determined, each action $\alpha \in \mathcal{A}_\kappa(t)$ is formatted into a predefined prompt template $P$ to construct the corresponding verification prompt $\rho_\kappa^t(\alpha)$. The structure of the prompt template is illustrated in Fig. 4 and includes the following key components of descriptions: role, goal, working memory, UI state, instruction, and the specific question to be verified.

In the role component, we outline the guidelines for operating mobile devices, as in [20]. The goal component specifies the task to be automated. The working memory maintains a record of action histories and trajectories for the current task, and it is updated after the execution of each action. For the UI component, we provide a streamlined description of the UI in XML format, as in [31]. The instruction component delivers explicit directives for the generative verifier, specifying that it must respond strictly with 'Yes' or 'No'. Finally, the prompt concludes with the specific question: *Is the action $\alpha$ helpful for completing the given user task?*

Subsequently, all the formatted verification prompts, each corresponding to an action candidate $\alpha$ within the action space $\mathcal{A}_\kappa(t)$, are fed in batch to the verifier $\mathcal{V}$. The verifier is fine-tuned from a pre-trained LLM, *i.e.,* Llama-3.1-8B. The fine-tuning process enables the verifier to assign a score to each action by analyzing only the first generated token, *e.g.,* the possibility of 'Yes' or 'No'. Ideally, this score reflects the likelihood of successfully completing the task if the corresponding action were executed at the current step $t$. The training details of the verifier are elaborated in Section 4.

Finally, after evaluating action candidates, the action $\alpha_\kappa^y(t)$ assigned the highest score is selected for execution at $t$:

$$\alpha_\kappa^y(t) = \arg\max \mathcal{V}(\rho_\kappa^t(\alpha)), \alpha \in \mathcal{A}_\kappa(t) \quad (1)$$

## 3.3 Accelerating Verifications

Assigning scores to a single action requires only one token generated by the generative verifier, such the pre-filling-only architecture significantly enhances the efficiency of mobile agents. When verifying a set of actions at each step, all verifications can be processed in parallel as a batch. Moreover, it is observed in Fig. 4 that, at a given task step, nearly all components of the verification prompts, apart from the question, remain identical. This design aims to maximize the shared prefix in the prompt, which can be leveraged to further accelerate the inference process.

Prefix caching enables the reuse of key-value (KV) caches across multiple verifications, thereby eliminating the need to recompute costly intermediate results. We adopt the Automatic Prefix Caching (APC) in vLLM [10]. Building on prefix caching, we further group actions to optimize the verification process. Specifically, within each batch of actions, we first perform a warm-up verification. Subsequently, actions of the same type are grouped together for verification. Since verifications of identical action types share a greater portion of their prompt content, this approach increases cache utilization and further reduces latency.

## 3.4 Completing and Executing Action

Before the selected action is exactly executed, three specific types of actions, *open app*, *type text*, and *answer*, require additional completion to specify the target app to open, the content to type, and the response to the user's query. Thus we employ an LLM to generate the necessary content by prompting the current UI state and working memory.

Action completion introduces extra overhead to the prefill-only agent architecture in V-Droid. But the number of actions requiring completion is relatively small. From the measurement on real world 2,000 tasks in [13], only 12.4% actions required completion. Furthermore, the separation of action selection and completion in V-Droid significantly simplifies the action space, enhancing the overall efficiency.

The selected and completed actions are executed by simulating interactions such as clicking, long-clicking, scrolling, and other corresponding operations. This execution results in a new UI state, $\mathcal{S}(t+1)$. Following the reflection framework proposed in [22], a step-level summary is generated by recording the executed action. The working memory is then updated with this summary. Subsequently, V-Droid iteratively performs the extracting actions, scoring with the verifier, and executing workflow until the agent signals task completion by reporting the *complete task* action.

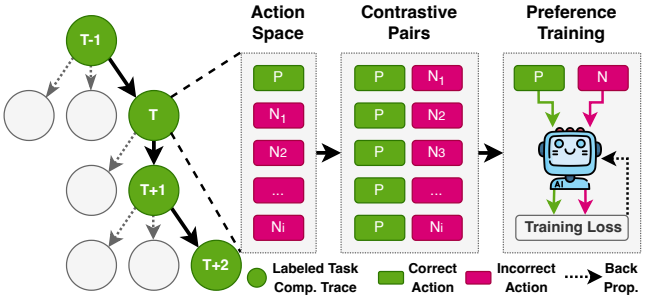



**Figure 5: Illustration of $P^3$ training used in V-Droid.**

# 4 Pairwise Process Preference Training

The generative verifier in V-Droid, which assigns a score to each action, enables the agent to discern which actions are more likely to contribute effectively to completing the given task. A straightforward approach involves directly employing a pre-trained LLM as the verifier without any post-training, using the logits or the probability of the 'Yes' token from the output space as the verification score. However, our measurements (§6.4) indicate that directly leveraging the Llama-3.1-8B as the verifier results in a success rate as low as 0% on AndroidWorld benchmark [20].

The rationale behind the suboptimal performance is twofold. First, for the verifier without post-training, we observe that the scores derived from the token space are insufficiently distinguishable, particularly in scenarios with numerous action candidates. Furthermore, simple fine-tuning on GUI data fails to enhance this indistinguishability. For instance, Qwen2.5-VL-72B [19] fine-tuned with Android GUI data [21] achieves only 35% success rate. Therefore, we believe that, to bridge the gap between the generated token space and the desired scoring (reward) space, additional training is necessary. More importantly, this training must be task-specific rather than relying solely on domain adaptation with GUI data, as the decision-making capabilities of mobile agent cannot be effectively enhanced otherwise. We propose the Pairwise Process Preference ($P^3$) training to significantly enhance the decision-making capabilities of verifier-driven mobile agents.

## 4.1 Decision-Making Training

The objective of $P^3$ training is to maximize the distinction between the action to be selected and the actions to be not selected at each step, guided by the process supervision [15]. To achieve this, training samples are structured into positive-negative action pairs at each step. By utilizing these contrastive pairs, $P^3$ training empowers the verifier to learn to assign higher scores to positive actions and lower scores to negative ones within the given context. Next, we delve into the training details, starting with the format of training data.

**Training data format**. Fig. 5 illustrates the process of organizing the training data. For a given task $\kappa$, $P^3$ training necessitates fine-grained process labels, *i.e.,* the trace for completing task $\kappa$ with the labeled correct action at each step. Specifically, the action

space $\mathcal{A}_\kappa(t)$ is derived from at each step $t$, as described in Section 3. Within the action space $\mathcal{A}_\kappa(t)$, the labeled action is identified as the positive action $\alpha_\kappa^y(t)$, while the remaining actions constitute the set of negative samples $\mathcal{A}'_\kappa(t)$. Subsequently, contrastive training pairs $\Gamma(t)$ are constructed for step $t$ as:

$$\Gamma_\kappa(t) = \{(\rho_\kappa^t(\alpha_\kappa^y(t)), \rho_\kappa^t(\alpha_\kappa^n(t))) \mid \alpha_\kappa^n(t) \in \mathcal{A}'_\kappa(t)\}, \tag{2}$$

where $\rho_\kappa^t$ represents the prompts constructed at step $t$ using the prompt template. These prompts incorporate the positive action $\alpha_s^t$ and the negative action $\alpha_j^t$, respectively, along with the UI state $\mathcal{S}_\kappa(t)$, task description, working memory, and other contextual information, as detailed in Section 3. Finally, we aggregate all the training pairs generated at every step of task $\kappa$ across all labeled tasks to construct the complete training dataset.

It is worth noting that, to the best of our knowledge, no publicly available dataset contains the required fine-grained process labels with contrastive action pairs. Therefore, we propose a novel data collection and synthesis approach, which will be detailed in §5.

**Training loss design**. The following loss is adopted to train the generative verifier at each step in V-Droid:

$$L_\kappa(t) = -E_{(\alpha_\kappa^y(t), \alpha_\kappa^n(t))}[\log \sigma(\tau(\alpha_\kappa^y(t)) - \tau(\alpha_\kappa^n(t)))], \tag{3}$$

where $\alpha_\kappa^n(t) \in \mathcal{A}'_\kappa(t)$ and $\tau$ represents the assigned score for the verified action. Specifically, in V-Droid, a trainable MLP layer is appended to the LLM to project its output token probabilities into the estimated score for each action.

$$\tau(\alpha) = MLP \cdot \mathcal{V}(\rho_\alpha). \tag{4}$$

When the $P^3$ loss $L_\kappa(t)$ at each step $t$ of task $\kappa$ across all labeled tasks is minimized during training, the verifier learns to maximize the score assigned to the positive action while minimizing the score assigned to the negative action.

In $P^3$ training, we introduce three key concepts: *pairwise*, *process*, and *preference*, all of which are crucial for effectively training the verifier. Firstly, preference training offers greater discriminative ability for the verifier to distinguish similar UI or actions which can be otherwise misleading to mobile agents. Secondly, process supervision provides fine-grained supervision signals, which are inherently beneficial for improving the step-wise decision-making ability of the verifier. Furthermore, compared to outcome-based supervision [15], process supervision allows for the construction of a larger number of training samples. For instance, from 100 task traces, we can derive around 55K step-wise process labels. Thirdly, pairwise training can be considered a form of data augmentation. Without such augmentation, only one training sample might be obtained for each task step. In contrast, pairwise training enables the generation of 50× more training samples, which is especially critical when the available training data is limited.

## 4.2 Self-Correcting Training

In $P^3$ training, only the action pairs corresponding to the states on the correct trace are utilized. This raises an intriguing question: can we also harness the data from the states on incorrect traces, as illustrated in Fig. 6.

Such data are valuable in enhancing the *self-correcting* capability of mobile agents. Intuitively, when humans interact with devices, it is possible to enter an erroneous state unrelated to the current

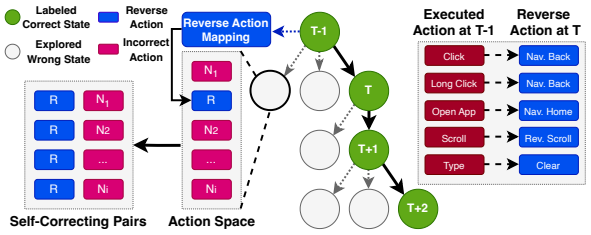



**Figure 6: Illustration of constructing self-correcting pairs and the mapping relationship between executed action and reverse action.**

intent. Instead of abandoning the task and starting over, humans often try to correct such mistakes, *e.g.,* by clicking the *navigate back* button. To the best of our knowledge, no existing mobile agent possesses the self-correcting capability. But the action pairs derived from these erroneous states present an opportunity for us to equip V-Droid with self-correcting ability through $P^3$ training.

Specifically, as illustrated in Fig. 6, given that the state $\mathcal{S}_{t-1}$ at step $t-1$ lies on the correct trace, the state $\mathcal{S}_t$ can be reached by executing the labeled correct action $\alpha_\kappa^y(t-1)$. Conversely, executing any incorrect action $\alpha_\kappa^n(t-1) \in \mathcal{A}'_\kappa(t-1)$ would lead to an erroneous state for the given task $\kappa$, noted as $\mathcal{S}'(t)$.

On the erroneous state $\mathcal{S}'(t)$, none of the actions from the action space may be applicable. Instead, a *reverse action* $\alpha_\kappa^r(t)$ is introduced as the correct action, *e.g.,* pressing the navigate back button. Specifically, the reverse action is derived based on the executed action at the previous step $t-1$ through a predefined mapping relationship, as illustrated in Fig. 6. For example, when V-Droid enters incorrect content by executing the action *type text*, the corresponding reverse action is to clear the previously input content.

Based on the identified reverse actions, we further construct the contrastive training pairs on the erroneous states for self-correcting training as follows:

$$\Gamma'_\kappa(t) = \{(\rho_\kappa^t(\alpha_\kappa^r(t)), \rho_\kappa^t(\alpha_\kappa(t))) \mid \alpha_\kappa(t) \in \mathcal{A}_\kappa(t)\}. \tag{5}$$

The $\alpha_\kappa^r(t)$ represents the derived reverse action,

$$\alpha_\kappa^r(t) = \mathcal{M}(\alpha_\kappa^y(t-1)), \tag{6}$$

where $\mathcal{M}$ represents the mapping function shown in Fig. 6. Ultimately, the self-correcting training pairs are combined with the contrastive training pairs obtained in Section 4.1 to form the final training set for V-Droid.

It is worth noting that the number of erroneous states exceeds the number of correct states for a given task. We find training with excessive self-correcting pairs would lead to collapsed performance by continuing output the *navigating back* action. Therefore, we randomly sample the erroneous states to construct the self-correcting training pairs, which account for approximately 2.5% of the entire training set.

# 5 Human-Agent-Joint-Annotation

$P^3$ training enhances the capabilities of mobile agents. However, to the best of our knowledge, the necessary training data, particularly fine-grained process labels that capture both correct and erroneous

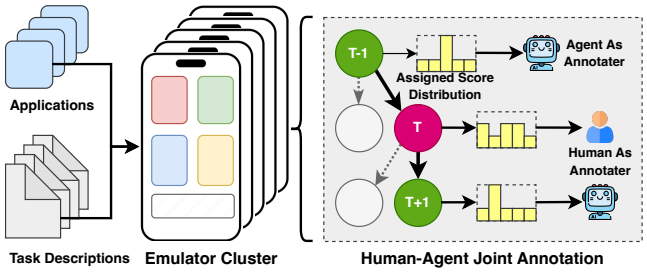
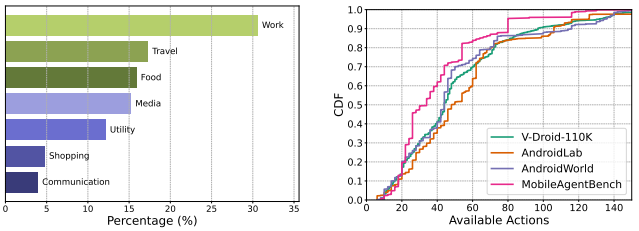



**Figure 7: Illustration of human-agent joint annotation.**




**(a) Distribution of training data across app. categories**

**(b) Distribution of available actions per step**

**Figure 8: Statistics of the collected datasets for training.**

states, remains unavailable in public repositories. To this end, we propose a novel data synthesis methodology, as illusrated in Fig. 7. First, we collect task instructions for Android Apps from public datasets [13, 21]. Additionally, we leverage LLMs to synthesize additional task instructions based on the Apps' descriptions available on the App Store, followed by meticulous manual verification. Subsequently, we generate the training data by actually executing the targeted tasks using the corresponding applications on our cluster of Android emulators. Human and agents jointly annotate the correct actions required at each step towards successful task completion.

The key idea of the human-agent joint annotation is to leverage the trained V-Droid to perform initial annotations, with human involvement only to correcting incorrect annotations, as illustrated in Fig. 7. Furthermore, we divide the annotation and training process into multiple iterations, allowing the agent to be progressively trained with larger datasets, thereby reducing its annotation errors over time. This iterative approach enables us to constrain human labeling efforts while effectively scaling the training process.

## 5.1 Verifier-As-Annotator

To facilitate effective agent-human collaborative annotation, the critical challenge is determining when and at which specific step the agent is likely to produce incorrect annotations, thereby necessitating human intervention.

We observe that when our trained verifier evaluates a set of actions at a given step, there exists a strong correlation between the ambiguity of the score distribution and the step-wise correctness of the agent. Intuitively, when the verifier assigns a single high score to one action while assigning significantly lower scores to the others, the action with the highest score is likely correct. However, if the score distribution is inconsistent, with multiple actions receiving similar scores, the decision at this step is more likely to be incorrect.

Particularly, we employ the entropy to measure the ambiguity of the score distribution $\mathcal{E}$ at the step $t$,

$$\mathcal{E}_\kappa(t) = -\sum \tau(\alpha_\kappa(t)) \log(\tau(\alpha_\kappa(t)))|\alpha_\kappa(t) \in \mathcal{A}_\kappa(t), \quad (7)$$

The entropy $\mathcal{E}$ serves as an effective metric for assessing the step-wise correctness of V-Droid. A conservative threshold is introduced to filter out steps where the agent is likely to perform incorrect actions. Table. 1 shows the prediction accuracy on the newly collected out-of-domain data can be as high as 76%. In such cases, a human annotator reviews the surrounding steps near $t$ and provides corrective actions as needed. The agent then resumes execution using the supplied actions until the task is successfully completed.

**Table 1: Truth table based on entropies of scores. The threshold is set as the median value of the entropies.**

| Iterations | Data | TP | TN | FP | FN | Accuracy |
|---|---|---|---|---|---|---|
| Iter. 2 | 27k | 0.11 | 0.40 | 0.39 | 0.10 | 0.51 |
| Iter. 3 | 55k | 0.42 | 0.34 | 0.08 | 0.16 | 0.76 |
| Iter. 4 | 110k | 0.40 | 0.31 | 0.10 | 0.19 | 0.71 |

## 5.2 Iterative Annotation and Training

Leveraging human–agent joint annotation, we iteratively optimize the verifier using $P^3$ training and deploy it to collect new data on out-of-domain tasks and apps over four rounds. To address the cold-start problem in the first round, we manually labeled data without agent assistance. Human annotators selected the correct actions from the extracted action lists provided by V-Droid, requiring approximately four minutes per task and about five hours in total to collect the initial 9K data pairs. In subsequent rounds, the agent was deployed to execute tasks and mark stepwise correctness, enabling progressive dataset expansion from 9K to 27K, then to 55K, and finally to 110K after four iterations. Human annotators primarily corrected errors in the trajectories, which reduced the average annotation time to around one minute per task. As shown in Fig. 8, the collected dataset spans more than 90 applications across seven common categories.

As data volume grows, the verifier demonstrates continuous performance improvements in both decision-making capability and annotation accuracy. Table 1 presents the truth table across different iterations, comparing verifier predictions based on entropy with step-wise ground truth. The true positive ratio increases from 0.11 to approximately 0.40, while annotation accuracy improves from 0.51 to over 0.70. These improvements indicate that as the verifier's scoring ability strengthens, the agent correctly annotates more steps, further reducing human annotation effort and enhancing training efficiency.

## 5.3 Training Details

In V-Droid, the generative verifier is built based on Llama-3.1-8B-Instruct-4-bit with one MLP layer attached to project the token

logits into the action scores. We use Q-LoRa training with rank 16. The learning rate is set to 1e-4 with 10 warm-up steps and then gradually reduced to 1e-6 until the end of the training. The training epoch is set to 20 for the 4th iteration, which is larger than the usual LoRa training epoch. The reason is that longer training enlarge the score gap between the correct and rejected actions, which is observed to be more robust during the test time. The 4th iteration of training is conducted on around 110k data pairs for 90 hours on 16×Nvidia A100 40G GPUs.

# 6 Evaluation

In the section, we evaluate V-Droid to highlight 1) the performance of V-Droid compared to SOTA mobile agents in terms of improved task success rate and reduced latency; 2) the training scaling law and the savings of annotation overhead in multiple iterations of annotation and training; and 3) the effectiveness of the training designs and the inference optimizations in V-Droid.

## 6.1 Experiment Settings

*6.1.1 Benchmark.* V-Droid is evaluated on three widely-used public benchmarks that cross varied types of mobile phones, system versions, applications and user instructions.

**AndroidWorld** [20] includes 20 APPs, 116 tasks, infinite task instructions that support random initialization to benchmark mobile agents abilities in real-execution environment. It includes challenging tasks that take more than 30 steps to complete and involves multi-app interactions.

**AndroidLab** [34] is a emulation environment that includes 138 tasks in nine kinds of daily used APPs, such as map, calendar, books, music. The agents are required to manage events, edit notes, and check information, etc.

**MobileAgentBench** [26] provides a phone usage environment built on 10 open-source applications and 100 tasks.

In contrast to static benchmarks, *e.g.,* DroidTask [31] and AndroidControl [13], the realistic and dynamic Android environment we evaluate V-Droid can better reflect its ability. Among the three benchmarks, AndroidWorld is designated as the in-the-domain test set, whereas AndroidLab and MobileAgentBench are designated as out-of-domain (OOD) test sets. During the training data collection process, we rigorously excluded applications utilized in these two benchmarks to ensure unbiased evaluation. The differences in data distributions between the training set and the test benchmarks are illustrated in Fig. 8b.

*6.1.2 Baselines.* V-Droid is compared with a wide range of mainstream mobile agents, including text-only agents (T3A [20], AutoDroid [31], AutoDroid-V2 [32], MobileGPT [11], Ponder&Press [29]), multimodal agents (M3A [20], SeeAct [39], AndroidArena [33], CogAgent [9], APPAgent [14], MobileAgent [25]), and agents with grounding models (Agent-S2 [1], Aria-UI [35], UGround [7], and UI-TARS [18]). Advanced LLMs, including GPT-3.5, GPT-4, GPT-4V, GPT-4o, GLM, DeepSeek-R1, Qwen, Llama-3.1, Claude and Gemini, are included as the base models for these agents. All the cloud-source LLMs are prompted using the M3A template in AndroidWorld [20]. Note that the baseline differences across benchmarks are attributable to the fact that some agents [14, 16, 18], were not open-sourced or lacked reproducible implementations, making it infeasible to evaluate them across all benchmarks. Therefore, we followed the practice and used all baselines available in each benchmark's leaderboard to ensure fairness and completeness.

*6.1.3 Metrics.* V-Droid is evaluated against baselines using two key metrics: task success rate (SR) and latency. We directly adopt the SR number reported in the respective baseline papers on the corresponding benchmarks. To assess latency, we employ two measures: the total step-wise latency of the entire agent and the decision-making latency of the LLM. The total step-wise latency primarily comprises the decision-making latency, the working memory update latency, and the execution time. Latency is evaluated using 20 randomly sampled tasks from benchmarks.

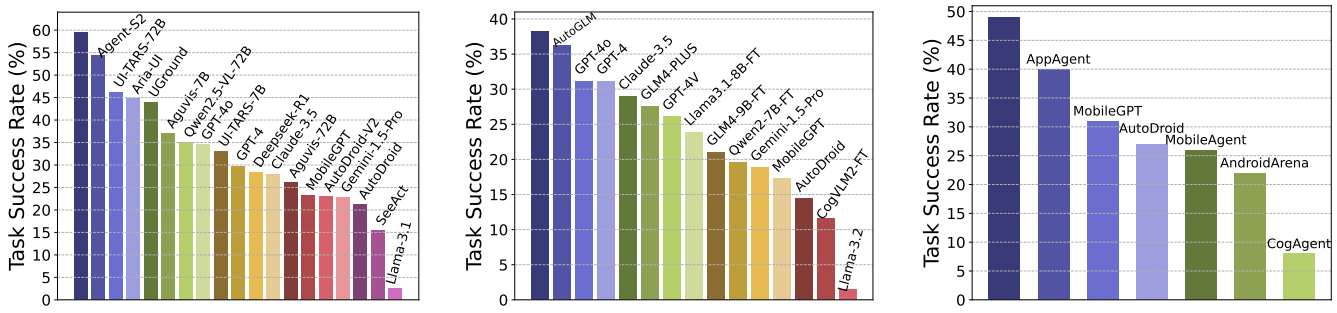
*6.1.4 Evaluation platform.* We evaluate V-Droid across various hardware configurations. The agent system runs within an Android emulator on a PC with an Intel i9-10900X CPU. The LLMs used by V-Droid and baseline models are tested on NVIDIA GPUs, including 4090, A100, A6000. Unless otherwise specified, all time measurements of V-Droid are conducted on a server with two NVIDIA 4090 GPUs.

## 6.2 Performance Improvement

**Improved task success rate**. Fig. 9 demonstrates that V-Droid outperforms existing mobile agents across three realistic MobileAgent-Benchmarks, AndroidWorld, AndroidLab, and MobileAgentBench, achieving success rate (SR) improvements of 5.2%, 2.1%, and 9.0%, respectively. Compared to cloud-based LLM-powered agents (*e.g.,* GPT-4, GPT-4o, DeepSeek-R1, Gemini-1.5-Pro, Claude-3.5), V-Droid achieves 25.0% and 7.13% higher SR on AndroidWorld and AndroidLab, respectively. Against advanced mobile agent frameworks (*e.g.,* Agent-S2, AutoDroid, AppAgent), V-Droid improves SR by 5.2% on AndroidWorld and 9.0% on the MobileAgentBench. Compared to models that decompose decision-making into reasoning and grounding (*e.g.,* UI-TARS, Aria-UI, UGround, Aguvis), V-Droid demonstrates a 14.3% SR improvement on the AndroidWorld. Against other fine-tuned SLMs (*e.g.,* Qwen-VL-7B-FT, Llama-3.1-8B-FT), V-Droid achieves a notable 14.3% improvement on AndroidLab. Compared with memory-driven agents like AutoDroid and MobileGPT, V-Droid achieves over 36.2%, 21.0%, and 18.0% SR improvement on the three benchmarks. Note that on MobileAgentBench, baselines such as AppAgent [14] already outperform cloud LLMs so those results are not included.

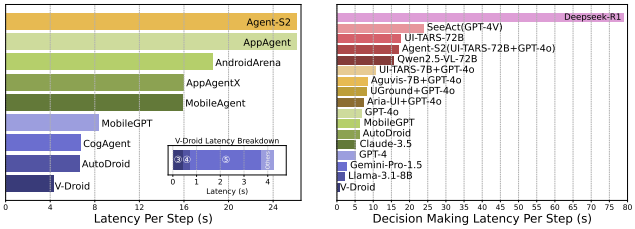
Unlike existing generation-based GUI agents that operate in continuous UI spaces, V-Droid simplifies decision-making by mapping UI states to a finite action space and decomposing the process into verification and completion, making it more tractable for SLMs. In addition, the verifier-driven workflow embeds UI-specific knowledge directly into the action candidates, thereby reducing the complexity of decision-making. Moreover, the $P^3$ training framework, built on large-scale data, further enhances the verifier's ability to distinguish between similar actions, self-correct errors, and maintain awareness of task progress.

**Reduced Latency.** Fig. 10 highlights the significant speed advantage of V-Droid over SOTA mobile agents in both step-wise and decision-making latency. While SOTA agents typically take over 20 seconds per step, V-Droid takes just 4.3 seconds per step,




**(a) AndroidWorld** **(b) AndroidLab** **(c) MobileAgentBench**

**Figure 9: Task success rate achieved by V-Droid compared to a range of mobile agents on three public benchmarks. The leftmost bar corresponds to V-Droid.**




**(a) Step-wise latency** **(b) Decision-making latency**

**Figure 10: The step-wise latency and decision-making latency of V-Droid compared to typical mobile agents.**

**Table 2: Decision-making latency of GPT-4o, UI-TARS-7B [18] with and without CoT, compared to V-Droid.**

| Agent | GPT-4o | | UI-TARS-7B | | V-Droid |
|---|---|---|---|---|---|
| Decision | w/ CoT | w/o CoT | w/ CoT | w/o CoT | Verifier |
| SR (%) | 41.3 | 39.2 | 33.2 | 29.3 | 59.5 |
| Latency (s) | 6.70 | 5.74 | 10.6 | 0.97 | 0.74 |
| Input Tokens | 6.2K | 6.2K | 8.9K | 2.9K | 2.6K |
| Output Tokens | 63.4 | 19.6 | 75.9 | 14.6 | 54.3 |

which is 6.1× faster. Specifically, it takes 0.44s for the verification (stage ③ in Fig. 4), 0.30s for the action completion (stage ④), and 3.03s for the working memory (stage ⑤). The action space and the prompt construction takes less than 1ms (stage ①-②) and the photo transition time takes 0.54s on average.

This efficiency gain stems from V-Droid 's verifier-driven workflow for decision-making, which transforms traditional auto-regressive decoding into a parallelized, prefilling-only scoring process. As shown in Fig. 10b, V-Droid achieves up to a 32.1× speedup compared to grounding-based agents that decompose decision-making into reasoning and grounding (e.g., UI-TARS, Aria-UI, UGround, Aguvis). Against UI-TARS [18] and DeepSeek-R1 with System-2 reasoning [8], V-Droid achieves 25.1× and 112.2× speed improvements, respectively. To the best of our knowledge, V-Droid is the first mobile GUI agent capable of near-real-time decision-making. We further discuss design alternatives for optimizing working memory in Section 6.4.

The action completion in V-Droid is only needed in 12.4% cases. In some input heavy tasks, the stepwise latency of V-Droid could be longer. For instance, in calendar-related tasks, there are 25.5% actions needs completion, which increases the average latency to 0.71s for the completion and 4.7s per step. In contrast, selection-heavy tasks that do not require text input (such as opening Wi-Fi) achieve 4.0s per step.

**Comparison with CoT-Disabled Agents.** The advantages of the verification-driven workflow in V-Droid are further demonstrated in Table 2 on AndroidWorld. We compare V-Droid with the best agents using cloud-based GPT-4o and open-sourced local-served UI-TARS-7B [18] (similar size with V-Droid) that support execution with or without CoT. Other agents (e.g., Agent-S2 [1]) rely on mandatory CoT or perform even worse, thus are excluded for this comparison. Disabling CoT reasoning in GPT-4o and UI-TARS leads to reduced decision-making latency. However, this comes at the cost of decreased SR. Despite removing CoT, both GPT-4o and UI-TARS still exhibit higher latency than V-Droid, as they autoregressively generate full actions. In contrast, V-Droid achieves a substantially higher SR with lower decision-making latency. V-Droid requires far fewer input tokens because it operates on simplified XML, lightweight system instructions, and compressed memory, whereas GPT-4o and UI-TARS depend on screenshots and excessive historical context. Besides, GPT-4o's latency does not scale proportionally with output token reductions, which might due to the internet latency dominate its runtime. These results further underscore the generation–verification gap in GUI agents and highlight the effectiveness of verifier-driven workflow.

**Self-correction showcase.** In Fig. 11, V-Droid explores one reasonable action "Click Textfile.txt" but then realizes itself in a wrong status that deviates from the goal. Later, it selects to navigate back and long-press the button to reveal the file properties. Similar cases are observed on the other two benchmarks. Notably, the self-correction training improves the SR of V-Droid from 52.2% to 59.5% on AndroidWorld and slightly increases the average trajectory length from 10.3 to 11.6 steps due to additional explorations. This

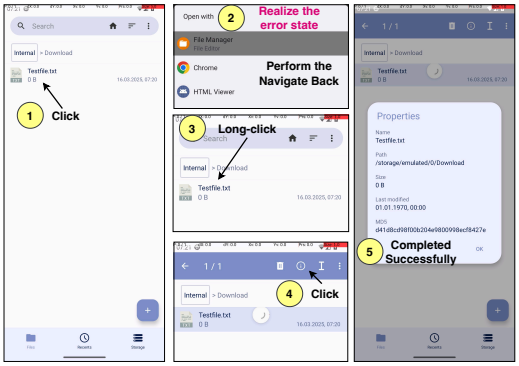
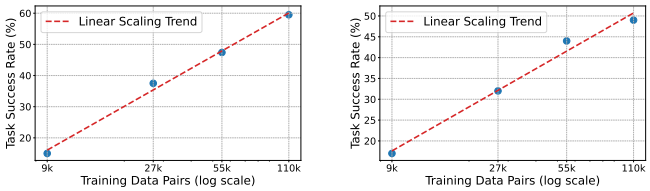



**Figure 11: A showcase of V-Droid automating the task "*Check the file property of Textfile.txt*" on MobileAgentBench. Note that both this task and application were excluded from the training process**




**(a) AndroidWorld** **(b) MobileAgentBench**

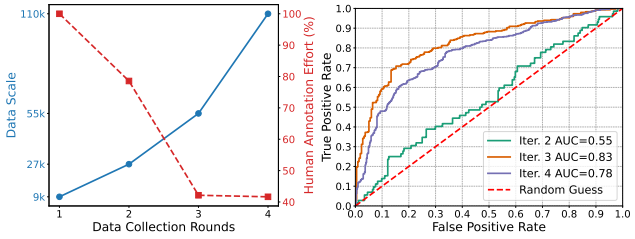
**Figure 12: Training time scaling law of the generative verifier. As the data pairs scales from 9k to 110k, the performance of the generative verifier boosts.**

modest increase in trajectory length is a reasonable trade-off given the substantial gain in overall success rate.

## 6.3 Training Scaling Law in V-Droid

V-Droid **improves with more training data.** Fig.12(a) and Fig.12(b) illustrate how the V-Droid's performance scales with increasing training data. On AndroidWorld, the SR improves from 15.0% to 37.5%, then further scales to 47.4% and 59.5% as the number of training data pairs increases from 9K (Human-Annotated) to 110K (Human-Agent Joint-Annotated). Similarly, on the MobileAgentBench, the SR increases from 17.0% to 49.0% as more training data becomes available. This continuous improvement highlights that V-Droid benefits from diverse app environments, instructions, and execution trajectories, which gradually expand throughout the training. The increasing dataset variety enhances the agent's generalization ability and robustness, leading to more effective decision-making across different tasks.

**Human annotation overhead decreases with better verifier.** As shown in Fig. 13a, the human annotation effort, measured as the ratio of data pairs collected by human annotators, gradually decreases across multiple iterative annotation and training cycles. After training with 27K and 55K data pairs, the AUC improves from 0.55 to approximately 0.8, demonstrating a significant enhancement in the verifier's ability to predict decision correctness. This




**(a) Annotation overhead** **(b) ROC of entropy.**

**Figure 13: Leveraging Verifier-as-annotator saves the annotation overhead. (a) As the agent ability improves, the annotation overhead of the human annotators decrease. (b) The verifier is an accurate decision classifier after two iterations of training.**

**Table 3: Comparing V-Droid with the design alternatives of agent architectures and corresponding training approaches with training data from 1.1K task steps.**

| Methods | Training | Base Model | SR (%) |
|---|---|---|---|
| V-Droid | $P^3$ | Llama-3.1-8B | 47.4 |
| Selector | SFT | Llama-3.1-8B | 35.8 |
| Generator | SFT | Llama-3.1-8B | 27.4 |
| LLM-as-a-Judge | – | Llama-3.1-8B | 0 |
| LLM-as-a-Judge | – | GPT-4 | 34.5 |
| Generator | – | GPT-4 | 29.6 |

improved accuracy enables human annotators to quickly rectify errors and recover the data collection process with minimal overhead. To further enhance the verifier's capabilities, we continue to scale the dataset using the human-agent joint annotation scheme in V-Droid, ensuring progressive improvements in annotation efficiency and model performance.

## 6.4 Comparison with Design Alternatives

**Architecture and Training Alternatives.** We craft three baselines to justify the designs in V-Droid, including *LLM-as-a-Judge*, *Selector* and *Generator*. LLM-as-a-Judge follows the architecture of V-Droid, but uses GPT-4 as the verifier. The action score is obtained by extracting the output logits value of "Yes". Selector is presented with XML descriptions and the extracted action lists and is fine-tuned to output the correct action number. Generator follows the architectures of T3A [20] but replaces the policy agent with a fine-tuned LLama-3.1-8B that is trained to output thought chains followed by the chosen actions. We adopt supervised fine-tuning for both designs.

Table 3 highlights the limitations of LLM-As-A-Judge using Llama-3.1-8B without training, which fails to assign accurate action scores, resulting in a 0% success rate. While GPT-4 performs better due to stronger reasoning and instruction-following abilities, it also outperforms its own generator-based approach, reinforcing the generation-verification gap. However, it is observed that GPT-4 often assigns high scores to multiple actions or low scores to all

**Table 4: Decision Alternatives on Working Memory.**

| Design Alter. | SR | Latency (s) |
|---|---|---|
| LLM-based | 59.5% | 3.03 |
| Rule-based | 46.1% | 1e-5 |
| Actions History | 40.0% | – |

**Table 5: Decision-making latency with and without the prefix caching on different instances of 4090, A100 and A6000 GPUs.**

| GPU | A100 | | | | 4090 | | | A6000 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Num. | 4 | 2 | 1 | 1 | 2 | 1 | 1 | 4 | 2 | 1 | 1 |
| P.C. | W/ | W/ | W/ | W/O | W/ | W/ | W/O | W/ | W/ | W/ | W/O |
| Latecy (s) | 0.717 | 0.932 | 1.446 | 7.116 | 0.744 | 1.034 | 7.847 | 0.808 | 1.027 | 1.555 | 11.36 |

actions, exposing a misalignment between token space and action space. Training Llama-3.1-8B as either a selector or generator improves success rates to 35.8% and 27.4%, respectively. However, these models still underperform compared with V-Droid, which leverages a verifier-driven workflow with $P^3$ training. The advantage of $P^3$ training comes from its pair-wise learning structure, where each step with $N$ available actions generates $N-1$ training pairs, effectively amplifying the training data by $N-1$ times. Additionally, pair-wise training enhances the model's ability to distinguish similar UI elements and actions, a crucial factor for accurate action selection.

**Working memory alternatives**. We observe that the step-wise latency of V-Droid is primarily constrained by the time required to construct working memory using an LLM, from 0.7 seconds to 3.8 seconds. It is because we use GPT-4 to update the working memory in current implementation. To mitigate this bottleneck, we explore two alternative designs at test time aimed at reducing latency: 1) **Action History Only** – This approach retains only a sequential log of past actions as the working memory. 2) **Rule-based Memory** – This method generates concise, structured descriptions of past actions and UI changes by applying rule-based heuristics. It extracts and compares UI content descriptions before and after an action, enabling a high-level summary. For instance, *Clicked the 'Save' button. Now an 'OK' text box appears, indicating that the action likely succeeded.*

As shown in Table 4, using only action history reduces the SR on AndroidWorld to 40.0%, as the agent struggles to retain key contextual information, leading to repeated actions and suboptimal decisions. Incorporating rule-based memory improves SR to 46.1%, demonstrating the benefit of structured summaries. However, the SR remains significantly lower than when using LLM-based memory construction, underscoring the importance of high-level action impact summarization and the ability to retain crucial contextual information across steps.

## 6.5 System Overhead

Running the verifier of V-Droid on GPUs requires approximately 16GB of memory, including the KV cache. Across the evaluated benchmarks, V-Droid verifies an average of 50.3 actions per task step.

Such verifications could be significantly accelerated with the batching inference and prefix caching. We highlight the decision-making latency on 4× NVIDIA GTX A100 80G, 4× NVIDIA GTX A6000, and 2× NVIDIA GTX 4090.

As shown in Table 5, given the optimization of the prompt format in V-Droid that maximizes the length of the shared prefix, a 10× speed up is obtained from the prefix caching across different actions, steps, and tasks. Furthermore, V-Droid turns the auto-regressive decision making scheme of LLM agents into the parallel pre-filling only verification scheme, which can be conducted in batch on multiple GPUs. The parallelism further decreases the selection time to 0.42s, 0.44s, and 0.52s with 4× A100, 2× 4090, and 4× A6000.

We also try to infer the overhead running the verifier on mobile devices. Taking the shared prefix into account, the total input token count for all action verifications per step is approximately 1.1K tokens. Given a prefill speed of 450 tokens per second on the Qualcomm Snapdragon 8 Gen 3 NPU [17], the decision-making latency is around 2.5 seconds.

## 6.6 Failure Study

We conducted a manual analysis of the failure cases for V-Droid and MobileGPT across more than 300 tasks spanning three benchmarks. We identify four categories of failure: 1) Hallucinated decisions, where erroneous actions are taken despite the agent possessing accurate memory and full observational input; 2) Inaccurate memory, where erroneous contextual memory leads the agent to make wrong decisions; 3) Incomplete actions, wherein available action types are insufficient to complete the task; and 4) Modality limitations, where tasks that demands vision capabilities cannot be accomplished by text-only LLM agents. MobileAgentBench tasks inherently require minimal perception and memory, leading to a high percentage of failures in the hallucinated decision category.

As illustrated in Fig. 14, failure cases of MobileGPT primarily stem from suboptimal decision-making. For example, given the task instruction *Add one expense ($307.01, in Health Care categories) to APP Expense Pro*, MobileGPT erroneously inputs the wrong amount and neglects to select the *Health Care* category, whereas V-Droid accurately provides all required information. There are still a large portion of failure cases of V-Droid in hallucinated decisions. For example, when required to share a file, V-Droid presses the file to read it instead of long-pressing it to reveal the *sharing* button, which might due to the lack of functional understanding of some UI elements. This observation further underscores the necessity of a larger scale training on more diverse tasks. Additional sources of failure of V-Droid include an incomplete action space caused by inaccurate information from the accessibility service and the lack of visual processing required for interpreting images and videos, which are further discussed in § 7.

# 7 Discussion

**Multimodality**. Although V-Droid is currently text-only, the proposed approach can also be extended to train multimodal mobile agents. For instance, V-Droid can be integrated with a grounding model with vision capability, which generate the initial actions for V-Droid instead of relying on accessibility services, followed by the action verification and completion process. We also plan to

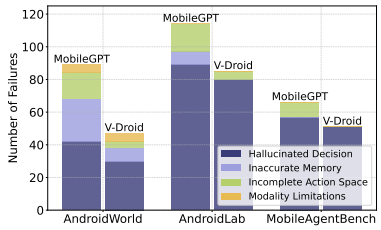



**Figure 14: Failure Analysis of MobileGPT and V-Droid.**

further train a vision-language model as the verifier as the future work, which assigns scores to actions based on the vision and text information jointly.

**Security and privacy**. In V-Droid, we adopt methods similar to those proposed in [31] to conduct security and privacy checks for actions prior to execution. Additionally, $P^3$ training offers a unique opportunity to train V-Droid to adhere to security guidelines, which will be explored as part of our future work.

## 8 Conclusion

For the first time, V-Droid demonstrates near-real-time, effective decision-making for mobile agents. It discretizes the decision space for mobile interactions into a finite set of action candidates, transforming the autoregressive process of generator-based agents into a parallelized, prefilling-only scoring mechanism using a generative verifier. $P^3$ training and the scalable human-agent joint annotation framework, significantly enhances the verifier's decision-making capabilities. Experimental results showcase the training scaling law of the verifier-driven architecture for mobile task automation.

## Acknowledgments

This research is partially supported by Singapore Ministry of Education under its AcRF Tier 1 grant RT14/22, the Global STEM Professorship Scheme of Hong Kong, the HKUST start up grant, and the Research Grants Council (RGC) General Research Fund (GRF) 16210425.

## References

[1] Saaket Agashe, Jiuzhou Han, Shuyu Gan, Jiachen Yang, Ang Li, and Xin Eric Wang. 2025. Agent S: An Open Agentic Framework that Uses Computers Like a Human. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2410.08164

[2] Android Developers. 2025. AccessibilityWindowInfo. https://developer.android.com/reference/android/view/accessibility/AccessibilityWindowInfo. Accessed: 2025-03-05.

[3] Android Developers. 2025. Android Debug Bridge. https://developer.android.com/tools/adb. Accessed: 2025-03-05.

[4] Wei Chen and Zhiyuan Li. 2024. Octopus v2: On-device language model for super agent. *arXiv preprint arXiv:2404.01744* (2024).

[5] Tinghe Ding. 2024. MobileAgent: Enhancing Mobile Control via Human-Machine Interaction and SOP Integration. arXiv:2401.04124 [cs]

[6] Longxi Gao, Li Zhang, Shihe Wang, Shangguang Wang, Yuanchun Li, and Mengwei Xu. 2024. MobileViews: A Large-Scale Mobile GUI Dataset. arXiv:2409.14337 [cs.HC] https://arxiv.org/abs/2409.14337

[7] Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. 2025. Navigating the Digital World as Humans Do: Universal Visual Grounding for GUI Agents. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=kxnoqaisCT

[8] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[9] Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. 2024. Cogagent: A visual language model for gui agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 14281–14290.

[10] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. arXiv:2309.06180 [cs.LG] https://arxiv.org/abs/2309.06180

[11] Sunjae Lee, Junyoung Choi, Jungjae Lee, Munim Hasan Wasi, Hojun Choi, Steve Ko, Sangeun Oh, and Insik Shin. 2024. Mobilegpt: Augmenting llm with human-like app memory for mobile task automation. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 1119–1133.

[12] Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244* (2023).

[13] Wei Li, William Bishop, Alice Li, Chris Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. 2024. On the Effects of Data Scale on Computer Control Agents. *arXiv preprint arXiv:2406.03679* (2024).

[14] Yanda Li, Chi Zhang, Wanqi Yang, Bin Fu, Pei Cheng, Xin Chen, Ling Chen, and Yunchao Wei. 2024. Appagent v2: Advanced agent for flexible mobile interactions. *arXiv preprint arXiv:2408.11824* (2024).

[15] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's Verify Step by Step. arXiv:2305.20050

[16] Xiao Liu, Bo Qin, Dongzhu Liang, Guang Dong, Hanyu Lai, Hanchen Zhang, Hanlin Zhao, Iat Long Iong, Jiadai Sun, Jiaqi Wang, Junjie Gao, Junjun Shan, Kangning Liu, Shudan Zhang, Shuntian Yao, Siyi Cheng, Wentao Yao, Wenyi Zhao, Xinghan Liu, Xinyi Liu, Xinying Chen, Xinyue Yang, Yang Yang, Yifan Xu, Yu Yang, Yujia Wang, Yulin Xu, Zehan Qi, Yuxiao Dong, and Jie Tang. 2024. AutoGLM: Autonomous Foundation Agents for GUIs. arXiv:2411.00820 [cs.HC] https://arxiv.org/abs/2411.00820

[17] PowerServe Project. 2024. PowerServe: Efficient LLM Serving. https://github.com/powerserve-project/PowerServe. Accessed: March 19, 2025.

[18] Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. 2025. UI-TARS: Pioneering Automated GUI Interaction with Native Agents. *arXiv preprint arXiv:2501.12326* (2025).

[19] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] https://arxiv.org/abs/2412.15115

[20] Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, et al. 2024. Androidworld: A dynamic benchmarking environment for autonomous agents. *arXiv preprint arXiv:2405.14573* (2024).

[21] Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. 2023. Android in the Wild: A Large-Scale Dataset for Android Device Control. arXiv:2307.10088 [cs]

[22] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2023), 8634–8652.

[23] Maayan Shvo, Zhiming Hu, Rodrigo Toro Icarte, Iqbal Mohomed, Allan Jepson, and Sheila A. McIlraith. 2021. AppBuddy: Learning to Accomplish Tasks in Mobile Apps via Reinforcement Learning. arXiv:2106.00133 [cs]

[24] Yuda Song, Hanlin Zhang, Carson Eisenach, Sham Kakade, Dean Foster, and Udaya Ghai. 2025. Mind the Gap: Examining the Self-Improvement Capabilities of Large Language Models. arXiv:2412.02674 [cs.CL] https://arxiv.org/abs/2412.02674

[25] Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-Agent: Autonomous Multi-Modal Mobile Device Agent with Visual Perception. *arXiv preprint arXiv:2401.16158* (2024).

[26] Luyuan Wang, Yongyu Deng, Yiwei Zha, Guodong Mao, Qinmin Wang, Tianchen Min, Wei Chen, and Shoufa Chen. 2024. MobileAgentBench: An Efficient and User-Friendly Benchmark for Mobile LLM Agents. *arXiv preprint arXiv:2406.08184* (2024).

[27] Luyuan Wang, Yongyu Deng, Yiwei Zha, Guodong Mao, Qinmin Wang, Tianchen Min, Wei Chen, and Shoufa Chen. 2024. MobileAgentBench: An Efficient and User-Friendly Benchmark for Mobile LLM Agents. arXiv:2406.08184 [cs]

[28] Taiyi Wang, Zhihao Wu, Jianheng Liu, Jianye Hao, Jun Wang, and Kun Shao. 2024. Distrl: An asynchronous distributed reinforcement learning framework for on-device control agents. *arXiv preprint arXiv:2410.14803* (2024).

[29] Yiqin Wang, Haoji Zhang, Jingqi Tian, and Yansong Tang. 2024. Ponder & press: Advancing visual gui agent towards general computer control. *arXiv preprint arXiv:2412.01268* (2024).

[30] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[31] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 543–557.

[32] Hao Wen, Shizuo Tian, Borislav Pavlov, Wenjie Du, Yixuan Li, Ge Chang, Shanhui Zhao, Jiacheng Liu, Yunxin Liu, Ya-Qin Zhang, and Yuanchun Li. 2024. AutoDroid-V2: Boosting SLM-based GUI Agents via Code Generation. arXiv:2412.18116 [cs] doi:10.48550/arXiv.2412.18116

[33] Mingzhe Xing, Rongkai Zhang, Hui Xue, Qi Chen, Fan Yang, and Zhen Xiao. 2024. Understanding the Weakness of Large Language Model Agents within a Complex Android Environment. *arXiv preprint arXiv:2402.06596* (2024).

[34] Yifan Xu, Xiao Liu, Xueqiao Sun, Siyi Cheng, Hao Yu, Hanyu Lai, Shudan Zhang, Dan Zhang, Jie Tang, and Yuxiao Dong. 2024. Androidlab: Training and systematic benchmarking of android autonomous agents. *arXiv preprint arXiv:2410.24024* (2024).

[35] Yuhao Yang, Yue Wang, Dongxu Li, Ziyang Luo, Bei Chen, Chao Huang, and Junnan Li. 2024. Aria-UI: Visual Grounding for GUI Instructions. *arXiv preprint arXiv:2412.16256* (2024).

[36] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

[37] Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. 2024. Ferret-UI: Grounded Mobile UI Understanding with Multimodal LLMs. arXiv:2404.05719 [cs]

[38] Li Zhang, Shihe Wang, Xianqing Jia, Zhihan Zheng, Yunhe Yan, Longxi Gao, Yuanchun Li, and Mengwei Xu. 2024. LlamaTouch: A Faithful and Scalable Testbed for Mobile UI Automation Task Evaluation. arXiv:2404.16054 [cs]

[39] Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. 2024. GPT-4V(ision) is a Generalist Web Agent, if Grounded. In *Forty-first International Conference on Machine Learning*. https://openreview.net/forum?id=piecKJ2DlB