STOP THINKING WHEN MUSCLES REMEMBER: BOOSTING MOBILE GUI AGENTS WITH MEMOIR

Anonymous authors
Paper under double-blind review

ABSTRACT

Mobile graphical user interface (GUI) agents are gaining wide attention in both academia and industry thanks to their ability to automate everyday tasks. However, they suffer from poor execution efficiency due to the expensive per-step vision-language model (VLM) inference, which prohibits their wide deployment. Existing methods accelerate GUI agents by reducing input information for VLMs, overlooking the high sequential output overhead. In this paper, we identify that the long and redundant thinking tokens in VLM outputs are the key efficiency bottleneck for mobile GUI agents and propose to equip GUI agents with muscle memory to materialize redundant thinking at runtime. Based on the observation, we design **MEMOIR**, a **M**uscle-m**EMO**ry tra**I**ning and **R**ecall framework, to accelerate GUI agents. The framework adopts a domain-specific language (DSL), *i.e.*, ***reduction DSL***, to store muscle memory, an ***offline muscle training pipeline*** to learn muscle memory, and a ***dual-gated memory trigger*** to efficiently and reliably use muscle memory. Our evaluation results on AndroidControl show that MEMOIR accelerates GUI agents by up to $3.25\times$ over state-of-the-art baselines while raising step accuracy from 65.2% to 69.3%.

1 INTRODUCTION

Mobile graphical user interface (GUI) agents are gaining increasing attention as a general way to automate everyday smartphone tasks (Qin et al., 2025; Zhou et al., 2025; Wu et al., 2024). Powered by vision-language models (VLMs), agents can directly operate GUI elements on mobile devices to complete various daily tasks, *e.g.*, booking hotels (Liu et al., 2026) and ordering food (Sun et al., 2026).

However, existing GUI agents suffer from severe efficiency issues, which significantly limit their wide deployment. Specifically, for each task, GUI agents need to execute many steps, each involving the VLM to encode the current screenshot, reason over it, and decide the next action. As a result, existing GUI agents often take about $3\times$ longer than humans to complete a task (Dai et al., 2025).

Many approaches are proposed to accelerate GUI agents, which can be summarized into two categories, *i.e.*, pruning-based methods (Yang et al., 2025; Alvar et al., 2025) and shortcut-based methods (Wang et al., 2025b; 2024; Jiang et al., 2025). Pruning-based methods accelerate each step by removing redundant input patches from GUI screenshots. Shortcut-based methods reduce the number of VLM inferences by summarizing successful executions into reusable shortcuts and conducting detailed planning before execution. However, *both approaches improve efficiency by reducing the information provided to VLMs*, *i.e.*, input tokens. They suffer from a trade-off between accuracy and efficiency since higher efficiency implies that VLMs see less. According to our experiments, the state-of-the-art methods can only accelerate GUI agents by $1.06\times$ within 36.2% accuracy loss.

In this paper, we identify that *output tokens are more important than input tokens in terms of agent execution latency*. We further summarize two key observations to accelerate GUI agents without sacrificing accuracy: ***1) Thinking tokens are critical to execution latency.*** Existing GUI agents typically adopt a chain-of-thought (COT) paradigm where long thinking tokens are generated to improve accuracy. These long thinking tokens are key contributors to per-step latency since the decoding throughput of VLMs is an order of magnitude slower than that of prefill. ***2) Thinking tokens are redundant across tasks.*** Thinking tokens usually contain agents' analysis of the input screenshot and the current task, *e.g.*, important GUI components that can affect next action decisions.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Since input structures of many GUI tasks are similar, it is wasteful to generate these thinking tokens repeatedly during execution.

Based on these observations, we propose to equip GUI agents with ***muscle memory*** to materialize repeated thinking into reflexes. In this way, familiar steps can be resolved with drastically less thinking overhead. Based on this idea, we design MEMOIR, a **M**uscle-m**EMO**ry tra**I**ning and **R**ecall framework, to accelerate GUI agents without compromising accuracy. First, to effectively represent muscle memory, we design a domain-specific language (DSL), *i.e.*, ***reduction DSL***, to formally describe how to compress the input and output of VLMs. Second, to learn effective muscle memory, we design an ***offline muscle training pipeline*** to summarize muscle memory items from past successful execution traces. Finally, to efficiently and reliably trigger muscle memory at runtime, we design a ***dual-gated memory trigger*** that adopts a lightweight machine learning model to identify the suitable memory entries according to the current input.

We evaluate MEMOIR on AndroidControl (Li et al., 2024) with various VLM backbones and compare it with three state-of-the-art pruning-based and shortcut-based approaches, *i.e.*, VisionZip (Yang et al., 2025), DivPrune (Alvar et al., 2025), and AppAgentX (Jiang et al., 2025). Compared with all baselines, MEMOIR reduces both input and output tokens by up to $3\times$, converting to a $3.25\times$ acceleration. Meanwhile, the per-step accuracy is raised from 65.2% to 69.3%.

Our contributions can be summarized as follows: **(1)** We identify redundant thinking as a key efficiency bottleneck of mobile GUI agents. **(2)** We propose the idea of muscle memory that compresses both VLM input and output to accelerate GUI agents. **(3)** We design and implement MEMOIR, a muscle-memory training and recalling framework, and show that it significantly reduces latency and improves accuracy.

2 RELATED WORK

Many approaches are proposed to improve accuracy, *i.e.*, task and step success rates, for GUI agents. Some train new VLMs to improve grounding and reasoning capabilities through pre-training (Qin et al., 2025; Wu et al., 2024; Cheng et al., 2024) and reinforcement learning (Luo et al., 2025). Others design novel multi-agent collaboration frameworks to decompose tasks and reflect actions better (Wang et al., 2024; Zhang et al., 2024). Still others focus on building better agent harnesses to provide richer context to VLMs (Yang et al., 2023; Wen et al., 2024; Xie et al., 2025a).

Different from these works, this paper focuses on improving the efficiency of GUI agents. We categorize existing approaches into two classes, *i.e.*, pruning-based methods and shortcut-based methods, and introduce them accordingly.

Pruning-based Methods. Pruning-based methods improve GUI agent efficiency by removing redundant visual tokens input into VLMs, *i.e.*, tokens with low contributions to accuracy. Based on how redundant tokens are identified, existing methods can be further divided into four categories (Shao et al., 2026), *i.e.*, transformation-based, similarity-based, attention-based, and query-based methods. Transformation-based methods compress visual tokens into compact representations through learned projections (Li et al., 2025). Similarity-based methods remove redundant tokens according to inter-token similarity (Alvar et al., 2025; Lin et al., 2025). Attention-based methods retain tokens that receive high attention scores from visual encoders or VLMs (Chen et al., 2024a; Yang et al., 2025; Ouyang et al., 2026). Finally, query-based methods select the tokens most relevant to the current instruction (Zhang et al., 2025).

Shortcut-based Methods. Shortcut-based methods accelerate agents by summarizing past successful executions into multi-step shortcuts to reduce VLM invocations. Existing shortcuts can be classified into high-level shortcuts and low-level ones. The former summarizes successful executions into general task-solving workflows (Wang et al., 2025b; Xie et al., 2025b). With these workflows, agents can generate better plans to avoid wasting VLM invocations on detoured steps. The latter compiles past executions into executable scripts or macros (Jiang et al., 2025; Chen et al., 2026; Wang et al., 2025a). By directly replaying these scripts, agents can skip some VLM inferences during execution.

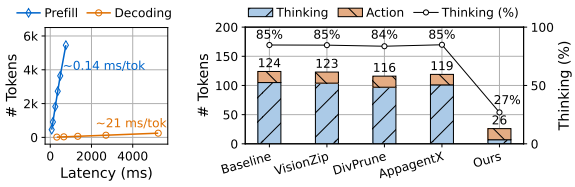
Limitations. Both approaches still have limited efficiency gains in GUI agents since they improve efficiency mainly by reducing input information. Specifically, for pruning-based methods, GUI elements are pruned before VLM invocations. For shortcut-based methods, intermediate GUI changes are ignored if a shortcut is invoked to perform multiple actions. Both approaches adopt CoT-based agents that generate long thinking tokens in the output, which, according to our experiments, is the key efficiency bottleneck. Different from all existing approaches, MEMOIR cooperatively reduces input and output tokens on familiar tasks with muscle memory, resulting in a higher efficiency boost without accuracy loss.

3 PRELIMINARY ANALYSIS

In this section, we conduct preliminary experiments to understand the efficiency bottleneck of existing GUI agents and motivate the design of MEMOIR.

3.1 ANALYSIS ON THE THINKING OVERHEAD

Existing VLMs follow the autoregressive paradigm, where input tokens are processed in parallel during prefill and output tokens are generated sequentially during decoding (Vaswani et al., 2017). The cost of generating output tokens is significantly higher than that of parallel input processing. We measure the prefill and decoding latency under different numbers of tokens with Qwen3-VL 8B on an H20 GPU. As shown in Figure 1a, within the time that prefill processes $2,000$ input tokens, decoding generates only about 13 output tokens, resulting in a $153\times$ throughput gap.



(a) P/D efficiency. (b) Output token breakdown.

Figure 1: Experiments on the thinking overhead.

We further find that *the output tokens of existing GUI agents are dominated by thinking tokens*. GUI agents usually adopt COT-style reasoning to improve accuracy (Wei et al., 2022; Luo et al., 2025; Yao et al., 2023). Specifically, they first generate long thinking tokens to accumulate context about the current screenshot and task, and then emit a short action based on the thinking. Figure 1b shows the output token breakdown of existing approaches on AndroidControl (Li et al., 2024). Thinking tokens account for more than 84% of the output tokens across all approaches. Moreover, compared with the baseline COT-based agent on Qwen3-VL 8B, neither state-of-the-art pruning-based methods (Yang et al., 2025; Alvar et al., 2025) nor shortcut-based methods (Jiang et al., 2025) reduce the number of output tokens, which limits their end-to-end acceleration.

Observation 1. *Output tokens are more expensive than input tokens, and thinking tokens dominate the output.*

3.2 ANALYSIS ON THE THINKING CONTENT

We further analyze what agents are actually thinking about during execution. As shown in Figure 2, the thinking mainly analyzes which GUI elements in the input screenshot are important for the current step, *e.g.*, locating the search bar and checking whether the keyboard is available. To quantify this behavior, we use DeepSeek-V4-Pro to classify the thinking content of the baseline COT-based agent on all execution traces. The results show that 38.0% of the thinking traces are devoted to analyzing important GUI elements.

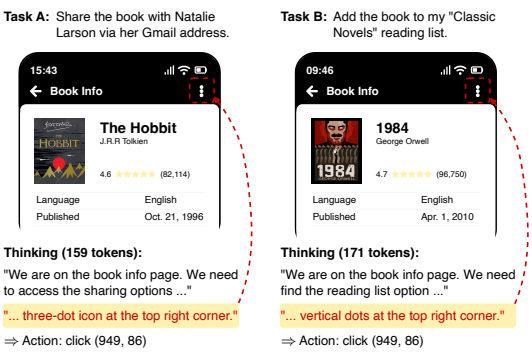


Figure 2: Thinking traces of two different tasks.

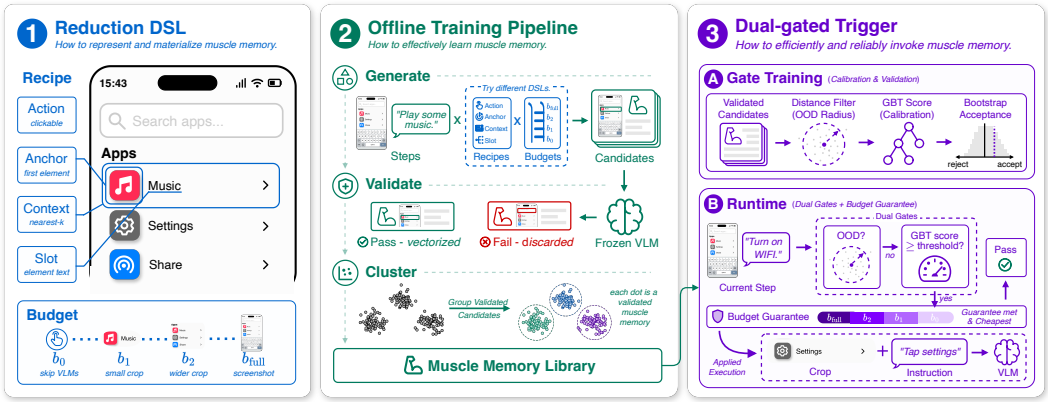


Figure 3: Overview of MEMOIR.

Moreover, the element analyses in thinking tokens are highly redundant across tasks. As shown in Figure 2, two different GUI tasks can land on similarly structured screens and attend to the same type of elements, *e.g.*, the search bar and the keyboard. According to our analysis on AndroidControl, 18.4% of click actions target the top-center and 16.6% target the top-right region of the screen. Therefore, it is wasteful to generate thinking tokens to repeatedly identify and analyze GUI elements during execution.

Observation 2. *Thinking tokens analyze the importance of GUI elements and are redundant across executions.*

4 METHOD

Based on these observations, we propose to empower GUI agents with ***muscle memory*** to materialize the thinking process and react during runtime. However, the following three challenges have to be addressed to make this idea practical: **(1)** How to *represent and materialize* muscle memory. **(2)** How to *effectively learn* muscle memory. **(3)** How to *efficiently and reliably invoke* muscle memory.

We design MEMOIR, a muscle memory training and recall framework, to address the above challenges. Figure 3 shows the overview of MEMOIR. First, we design a novel domain-specific language (DSL), *i.e.*, ***reduction DSL***, to represent muscle memory entries and materialize them at very low storage cost (§4.1). Second, we employ an ***offline muscle training pipeline*** that learns reusable thinking processes in past successful executions and represents them as muscle memory entries with the reduction DSL (§4.2). Finally, we design a ***dual-gated memory trigger*** that invokes muscle memory efficiently and reliably during runtime (§4.3). We detail these three designs below.

4.1 REDUCTION DSL

Ideally, a muscle memory entry should satisfy three requirements. First, it needs to localize relevant regions on screenshots for similar steps to avoid thinking (***R1***). Second, it needs to adaptively adjust the thinking effort during runtime according to the difficulty of the current step (***R2***). Finally, it must be compact in storage so that entries can be retrieved efficiently during runtime (***R3***).

The ***reduction DSL*** satisfies all the above requirements. It has two parts, *i.e.*, a reduction recipe and a budget constraint. First, the reduction recipe localizes relevant GUI elements by defining a *four-stage selection pipeline* in the DSL. We can apply the described pipeline to any new input to get the relevant regions (***R1***). Second, the budget constraint defines four different input and thinking reduction levels. Agents can adjust their input details and thinking efforts by dynamically selecting a budget level (***R2***). Finally, since both parts are formally defined by the DSL, each muscle memory entry can be serialized into text with about 100 bytes, enabling efficient storage and retrieval at runtime (***R3***).

Reduction Recipe. The reduction recipe is a DSL that defines element selection over the input as a four-stage pipeline, *i.e.*, action filtering, anchor identification, context policy, and slot policy. Each reduction recipe is a four-field tuple $(action, anchor, context, slot)$, where each field can be instantiated with a pre-defined primitive in the DSL. Table 1 summarizes two example primitives for each stage. The complete primitive list is shown in the supplementary material.

The four-stage pipeline takes two inputs, *i.e.*, the accessibility (a11y) tree of the input screenshot and the instruction text, and works as follows.

First, the action field restricts the action the recipe can apply to, and filters out the a11y tree elements that cannot receive these actions. Second, the anchor policy identifies the most important element among the remaining candidates, *i.e.*, the anchor element. The context policy then specifies which elements around the anchor should also be considered by VLM. Finally, slot policies apply to text-related actions, *e.g.*, type, and specify how to extract the required text from the input. With the recipe, MEMOIR crops the screenshot to contain all required elements and inputs them to VLM to generate actions without thinking.

Table 1: Example primitives of the four recipe stages.

Stage	Primitive	Description
Action	`clickable`	Keep clickable elements
	`editable`	Keep editable elements
Anchor	`first`	First element on the tree
	`history`	Closest element to the last action
Context	`anchor`	Elements overlapping the anchor
	`nearest_k`	k nearest elements to the anchor
Slot	`instruction`	Quoted text from the instruction
	`element`	Text of an on-screen element

Budget Constraints. Budget constraints define how much input and thinking effort the agent should spend when using muscle memory. Each budget is a three-field tuple, which caps the size of the screenshot, the number of candidate elements in the a11y tree, and the number of output tokens. MEMOIR pre-defines four budget levels with increasing costs, as summarized in Table 2. The lowest budget b_0 skips VLM invocation when actions can be directly performed and require no coordinate output, *e.g.*, wait and navigation. The intermediate budgets b_1 and b_2 crop the screenshot around the anchor element under different numbers of pixel constraints and let VLM directly emit the action invocation without or at a low thinking budget. Finally, the full budget b_{full} falls back to the COT-based agent, covering cases when no muscle memory entry can be applied to the current step.

Table 2: Budget constraints adopted by MEMOIR.

Budget	Pixels	#Candidate	#Output Token
b_0	0	1	0
b_1	120 K	8	8
b_2	350 K	16	16
b_{full}	All	All	512

Extensibility. The reduction DSL is not limited to the pre-defined primitives and budget levels. Users can extend a new primitive by writing a Python function, *e.g.*, a new anchor policy for a customized UI layout. New budget levels can also be defined on demand. The extension effort is low with the help of coding agents, *e.g.*, CodeX.

4.2 OFFLINE MUSCLE TRAINING PIPELINE

The offline muscle training pipeline is designed to learn muscle memory entries from past successful executions. The goal is to remember the set of elements that are critical to generating the correct action for different steps from past successful steps. The key challenge is that the important elements are identified in thinking tokens with natural language, which results in a gap between the vague language and the specific GUI elements on the screenshot and a11y tree.

We attack this challenge by deriving important elements through trial and error. Specifically, instead of parsing what the thinking says, we directly test which reductions can still produce correct actions in a calibration set. The pipeline consists of three stages. First, we enumerate primitives in the reduction DSL to generate a large set of potential muscle memory entries. We then apply each potential entry with the VLM on past successful steps and only keep the entries that still yield the correct actions. Finally, we cluster the validated entries so that each cluster forms one muscle memory entry that can be applied to similar steps.

Stage 1: Generating Potential Memory Entries. For each step in the calibration set, MEMOIR enumerates all possible combinations of the reduction DSL as candidate entries, *e.g.*, the (`clickable`, `history`, `anchor`, `none`) recipe with the b_0 budget, and the (`editable`, `first`, `nearest_k`, `instruction`) recipe with the b_1 budget. We filter out candidate entries whose reduction recipes no longer contain the ground-truth target element and rank the remaining entries according to their reduced input size. Only the top 16 entries are selected as potential memory entries to keep validation overhead under control. **Stage 2: Validating Potential Memory Entries.** For each step in the calibration set, we apply the potential entries to generate reduced inputs. We take the reduced input into VLM with different text instructions, screenshot size, and output tokens, according to their budgets. Entries that generate correct outputs are preserved as valid memory entries. **Stage 3: Clustering Valid Memory Entries.** We further cluster valid memory entries to summarize general muscle memory. We start by grouping valid entries that share the same reduction recipe. Within each group, we use reduced inputs generated by entries to vectorize entries so that entries for the same problem, *i.e.*, input, are clustered together. Specifically, we vectorize each reduced input with a structural fingerprint to capture the types and numbers of remaining GUI elements and the instruction text. We then adopt density-based clustering on the entry vectors that connects entries when their cosine similarity exceeds a threshold. Each cluster is a general memory entry represented by its centroid vector, and all clusters form a muscle memory library.

4.3 DUAL-GATED MEMORY TRIGGER

At runtime, MEMOIR needs to quickly retrieve suitable muscle memory entries from the library. Achieving this is challenging for two reasons. On one hand, evaluating the fitness of all entries is too expensive since it locates on the critical path. On the other hand, triggering a wrong entry can mislead the agent and hurt execution accuracy.

The ***dual-gated memory trigger*** addresses both challenges with two cascading selection criteria, *i.e.*, a distance-based filtering gate and a machine-learning (ML)-based scoring gate. The former quickly filters out muscle memory entries that are obviously irrelevant. The latter further scores the remaining entries and selects the most suitable one.

Distance-Based Filtering Gate. The distance-based filtering gate filters out irrelevant muscle memory entries using cosine distance. During the offline phase, for each muscle memory entry, we use an additional calibration set to get its effective distance region, *i.e.*, out-of-distribution (OOD) distance. Specifically, for a memory entry m with centroid vector v_m, we compare the VLM output with and without using m for each step in the calibration set. Only successful steps, *i.e.*, ones that generate the same output, are reserved. An entry with successful steps less than a threshold s is dropped due to its low support in the calibration distribution. If m is not dropped, we apply it on successful steps to get reduced inputs and apply the structural fingerprint (§4.2) to vectorize the reduced inputs. An OOD distance d_m for m is then calculated as the 95th percentile cosine distance between v_m and the input vectors.

At runtime, we first vectorize the current step v_s by applying muscle memory entries and conducting the structural fingerprint on the reduced input. We filter out a muscle memory entry m if $d_m < 1 - \cos(v_s, v_m)$. This gate guarantees that memory entries will not be mistakenly applied to steps with very different inputs.

ML-Based Scoring Gate. The ML-based scoring gate is responsible for scoring and ranking all remaining memory entries after the filtering gate. In the offline phase, we train a lightweight gradient boosting model (Friedman, 2001) for each memory entry to predict the success rate of applying this entry to a step input. The model training process enforces two reliability guarantees.

First, we statistically guarantee accuracy and efficiency of applying an entry. For an entry, we compare the per-step VLM outputs with and without the entry on the calibration set and calculate their difference in step accuracy and token count. To be robust against the calibration samples, we resample the calibration set $1,000$ times and obtain a statistical distribution of the accuracy difference. We drop an entry if 1) the q-th percentile of its accuracy drop distribution is higher than ϵ, or 2) the entry does not reduce input and output tokens. By default, we set q as 0.75 and ϵ as 0.05 to ensure at most 5% accuracy drop on the calibration distribution.

Second, after training the model, we decide a threshold for applying each budget level to enable adaptive budget selection according to the step difficulty. Specifically, we use the trained model to predict the success rate of all steps in the calibration set. A threshold on the success rate is chosen for each budget to keep the accuracy drop within ϵ.

At runtime, we predict the success rate for all remaining muscle memory entries. We then decide their budget levels according to their success rates, rank them according to their budget levels and success rates, and try them in order. Most steps succeed within the first try, so the trigger only adds marginal overhead to execution.

5 EXPERIMENTS

5.1 EXPERIMENT SETTINGS

All our experiments are conducted on 8 H20 GPUs, representing GUI agents use APIs to query models on the cloud. The detailed settings are listed as follows:

Baselines. We compare MEMOIR with four baselines, *i.e.*, a COT-based GUI agent, two pruning-based baselines, and a shortcut-based baseline. For the COT-based baseline, we employ the same prompt in MAIUI (Zhou et al., 2025) to guide agent thinking. For pruning-based baselines, we adopt VisionZip (Yang et al., 2025) and DivPrune (Alvar et al., 2025), which are two state-of-the-art approaches. For the shortcut-based baseline, we adopt AppAgentX (Jiang et al., 2025), which uses VLMs to summarize past successful experiences into low-level reusable shortcuts.

Models. We compare MEMOIR on three VLMs, *i.e.*, Qwen3-VL 8B (Team, 2025), InternVL2.5 8B (Chen et al., 2024b), and LLaVA-OneVision-1.5 8B (An et al., 2025). For VisionZip and DivPrune, we directly adapt their open-source code to the new models. We evaluate DivPrune under two different token retention ratios, *i.e.*, 9.8% and 17%, to show the trade-off between accuracy and latency. For AppAgentX, we learn shortcuts with Qwen3-VL 8B and adopt the learned shortcuts on the other two models. Finally, for MEMOIR, we conduct offline muscle training on Qwen3-VL 8B and reuse the learned muscle memory on the other two VLMs to show the generality of the learned muscle memory.

Benchmark & Metrics. We adopt AndroidControl (Li et al., 2024) as our major benchmark. We follow the official train/validation/test splits, with 1,543 episodes and 8,444 steps in the test set. For each GUI automation task, it provides both high-level and low-level instructions on each step for each action. We report three types of metrics over the benchmark, *i.e.*, accuracy, latency, and token count. For accuracy, we report both low-level accuracy, *i.e.*, the accuracy of each step, and high-level accuracy, *i.e.*, the success rate of tasks. For latency, all methods are executed under batch size 1, and the per-step latency is the time between the VLM generating two actions. Finally, we also report the number of input and output tokens to show the token reduction of all approaches.

5.2 MAIN RESULTS

Table 3 compares MEMOIR with four baselines across three VLM backbones on AndroidControl. Across all VLMs, MEMOIR leverages muscle memory on up to 92.4% of steps, cuts end-to-end latency by $3.25\times$, and raises low-level accuracy by 6.2 points. Next, we analyze the improvement in accuracy and latency, respectively.

Latency. MEMOIR achieves the highest speedup among all approaches on all three VLMs. Compared with pruning-based methods, the latency improvement of MEMOIR comes from the significant reduction of output tokens, *e.g.*, up to $4.7\times$ reduction on Qwen3-VL. Figure 4 shows the latency breakdown of all approaches on Qwen3-VL. The reduction in output tokens directly translates to $4.2\times$ reduction in decoding latency, whereas VisionZip and DivPrune have comparable decoding latency as the COT-based agent.

Table 3: Main results on AndroidControl. Best result within each VLM backbone are highlighted in bold.

VLM	Method	Accuracy (%)		Tokens / step				Latency / step	
		Low-Level	High-Level	Visual	Input	Output	Total	ms	Speedup
Qwen3-VL 8B (Team, 2025)	COT-Based Agent	65.2	10.8	997	2008	124	2131	3314	1.00×
	+VisionZip	65.2	11.1	546	1711	123	1834	3330	1.00×
	+DivPrune (9.8%)	29.0	1.0	**85**	1118	116	1234	3127	1.06×
	+DivPrune (17%)	38.5	2.1	168	1189	116	1305	3971	0.84×
	+AppAgentX	60.3	10.6	987	2016	132	2148	10247	0.32×
	+MEMOIR (Ours)	**69.3**	**11.3**	382	**680**	**26**	**706**	**1019**	**3.25**×
InternVL2.5 8B (Chen et al., 2024b)	COT-Based Agent	40.1	2.7	2304	3657	59	3716	3058	1.00×
	+VisionZip	30.4	1.7	1602	2955	62	3017	3405	0.90×
	+DivPrune (9.8%)	16.8	0.8	**225**	**1578**	59	1637	2991	1.02×
	+DivPrune (17%)	17.5	0.9	396	1749	59	1808	3033	1.01×
	+AppAgentX	34.3	2.4	2806	3696	55	3751	8192	0.37×
	+MEMOIR (Ours)	**46.3**	**3.2**	1038	1599	**17**	**1616**	**1146**	**2.67**×
LLaVA-OV-1.5 8B (An et al., 2025)	COT-Based Agent	33.0	1.4	1284	2484	75	2559	1999	1.00×
	+VisionZip	29.5	1.4	636	1836	75	1910	1934	1.03×
	+DivPrune (9.8%)	23.5	0.7	**125**	1337	79	1416	1977	1.01×
	+DivPrune (17%)	24.5	0.8	216	1428	79	1506	1952	1.02×
	+AppAgentX	26.7	0.6	1272	3039	83	3122	8866	0.23×
	+MEMOIR (Ours)	**35.4**	**1.5**	465	**846**	**24**	**870**	**820**	**2.44**×

Compared with shortcut-based methods, the latency improvement of MEMOIR comes from the efficient dual-gated memory trigger. AppAgentX invokes multiple calls to large models at each step to match usable shortcuts and generate invocation arguments, making it even slower than the CoT-based agent. In contrast, the dual-gated trigger of MEMOIR locates suitable memory entries with a distance metric and lightweight ML models, adding no extra large model invocation to execution.

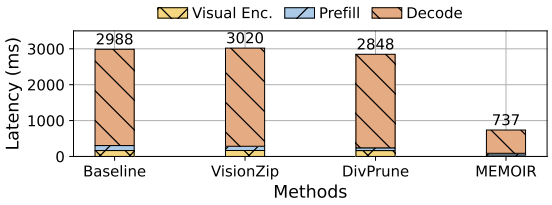


Figure 4: Latency decomposition.

Accuracy. MEMOIR has the highest accuracy on all VLMs. Compared with the COT-based agent, the accuracy is higher since the muscle memory reduces distraction in the VLM input. Compared with pruning-based methods, the accuracy increases since the reduced GUI elements in MEMOIR are verified as useless during the offline muscle training pipeline, resulting in less information loss. Finally, MEMOIR also outperforms AppAgentX. This is because the multi-step shortcuts in AppAgentX have severe cascading errors when a shortcut is mistakenly invoked.

5.3 ABLATION STUDY

We ablate three sources of MEMOIR's gains, *i.e.*, the muscle-memory mechanism itself, the budget levels, and the calibration strength of the dual-gated trigger.

Table 4: MEMOIR under various thinking efforts.

Method	LL Acc. (%)	HL Acc. (%)	Lat. (ms)
Full Thinking	65.2	10.8	3,314
No Thinking	59.7	7.1	**710**
MEMOIR	**69.3**	**11.3**	1,019

Effect of Muscle Memory. A natural question is whether simply removing the thinking trace already captures most of the gain. Table 4 shows the accuracy and latency of MEMOIR under different thinking effort configurations. As shown in the table, the No Thinking variant has the lowest latency due to its smallest number of output tokens. However, compared with Full Thinking, removing thinking reduces low-level (LL) and high-level (HL) accuracies by 5.5 and 3.7 points, respectively. MEMOIR instead fixes the recurring thinking into a muscle-memory reflex

and reduces the number of both input and output tokens. Consequently, both accuracy and execution efficiency can be improved compared with the Full Thinking variant.

Table 5: Performance decomposition of MEMOIR under different budget configurations

Variants	Step Coverage (%)	#In Tok.	#Out Tok.	Lat. (ms)	Low-Level Acc. (%)	High-Level Acc. (%)
b_2-only	90.1	770	30	1119	66.3	10.8
b_1-only	84.3	**589**	37	1293	62.9	7.5
b_0-only	6.4	1879	116	3088	**70.0**	11.1
MEMOIR (full)	**92.4**	680	**26**	**1019**	69.3	**11.3**

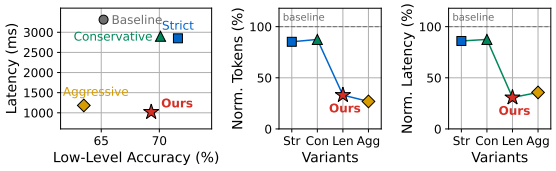
Effect of Budget Levels. Table 5 shows the effect of different budget levels. Compared with only using one budget level, MEMOIR applies muscle memory to the highest number of steps since it has more options in budgets when triggering muscle memory entries. Moreover, the savings in the output tokens and the latency reduction are also the highest in MEMOIR. An interesting fact is that the accuracy is higher when only using the most strict b_0 budget, which skips VLM invocations when a muscle memory entry is triggered. This implies that 1) MEMOIR can effectively summarize deterministic rules from execution history and 2) these deterministic rules are key to improve accuracy.

Effect of Calibration Strength. Calibration strength controls how strictly muscle memory entries are filtered in the dual-gated memory trigger. There are three tunable parameters, *i.e.*, the minimum number of successful steps s, the tolerated accuracy drop ϵ, and the percentile used in the confidence bound q. A stricter calibration requires more supporting steps, smaller accuracy drops, and a more conservative percentile, resulting in fewer high-confidence muscle memory entries. We evaluate MEMOIR under four different levels of calibration strengths summarized in Table 6.

Table 6: Parameters of the four calibration strengths.

Strength	Steps (s)	Acc. Drop (ϵ)	Percentile (q)
Strict	50	0.00	0.95
Conservative	25	0.00	0.90
Aggressive	5	0.10	0.60
Lenient (Ours)	**8**	**0.05**	**0.75**

As shown in Figure 5, looser calibration generally leads to lower latency and lower accuracy, *e.g.*, from strict to lenient, the speedup grows from $1.16\times$ to $3.25\times$ while the low-level accuracy drops from 71.6% to 69.3%. Meanwhile, over-relaxation compromises accuracy, *i.e.*, the aggressive level retains 83 muscle memory entries but collapses accuracy to 63.5%. We adopt the lenient calibration strength by default to achieve a better trade-off between accuracy and latency.



(a) Accuracy and latency. (b) Tokens. (c) Latency.

Figure 5: MEMOIR under different calibration strengths.

6 CONCLUSION

In this paper, we identify that the long and redundant thinking tokens of mobile GUI agents are their key efficiency bottleneck. We propose to equip GUI agents with muscle memory to fix repeated thinking into reflexes and reduce their execution latency. Therefore, we design and implement MEMOIR that represents muscle memory with a reduction DSL, learns muscle memory entries through an offline muscle training pipeline, and invokes them efficiently and reliably with a dual-gated memory trigger. Experimental results show that MEMOIR accelerates GUI agents by up to $3.25\times$ while improving low-level step accuracy by 4.1 points.

AI USE STATEMENT

DeepSeek-V4-Pro was used to classify the thinking content of the baseline COT-based agent, as described in Section 3. Generative AI was also used for manuscript polishing, literature search for related work, and code implementation.

REPRODUCIBILITY STATEMENT

We provide an anonymized code package as supplementary material to reproduce the reported experiments. Section 4 specifies the reduction DSL, offline muscle training pipeline, and dual-gated memory trigger. Section 5 describes the experimental settings, model backbones, baselines, and evaluation metrics. Appendices A, B, and C provide the complete reduction recipe inventory, reduced-model prompts and decoding settings, and feature-vector construction, respectively.

REFERENCES

Saeed Ranjbar Alvar, Gursimran Singh, Mohammad Akbari, and Yong Zhang. Divprune: Diversity-based visual token pruning for large multimodal models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2025, Nashville, TN, USA, June 11-15, 2025*, pp. 9392–9401. Computer Vision Foundation / IEEE, 2025. doi: 10.1109/CVPR52734.2025.00877. URL https://openaccess.thecvf.com/content/CVPR2025/html/Alvar_DivPrune_Diversity-based_Visual_Token_Pruning_for_Large_Multimodal_Models_CVPR_2025_paper.html.

Xiang An, Yin Xie, Kaicheng Yang, Wenkang Zhang, Xiuwei Zhao, Zheng Cheng, Yirui Wang, Songcen Xu, Changrui Chen, Chunsheng Wu, Huajie Tan, Chunyuan Li, Jing Yang, Jie Yu, Xiyao Wang, Bin Qin, Yumeng Wang, Zizhen Yan, Ziyong Feng, Ziwei Liu, Bo Li, and Jiankang Deng. Llava-onevision-1.5: Fully open framework for democratized multimodal training. *CoRR*, abs/2509.23661, 2025. doi: 10.48550/ARXIV.2509.23661. URL https://doi.org/10.48550/arXiv.2509.23661.

Liang Chen, Haozhe Zhao, Tianyu Liu, Shuai Bai, Junyang Lin, Chang Zhou, and Baobao Chang. An image is worth 1/2 tokens after layer 2: Plug-and-play inference acceleration for large vision-language models. In Ales Leonardis, Elisa Ricci, Stefan Roth, Olga Russakovsky, Torsten Sattler, and Gül Varol (eds.), *Computer Vision - ECCV 2024 - 18th European Conference, Milan, Italy, September 29-October 4, 2024, Proceedings, Part LXXXI*, volume 15139 of *Lecture Notes in Computer Science*, pp. 19–35. Springer, 2024a. doi: 10.1007/978-3-031-73004-7_2. URL https://doi.org/10.1007/978-3-031-73004-7_2.

Qijia Chen, Andrea Bellucci, Zhida Sun, and Giulio Jacucci. Skilldroid: Compile once, reuse forever. *CoRR*, abs/2604.14872, 2026. doi: 10.48550/ARXIV.2604.14872. URL https://doi.org/10.48550/arXiv.2604.14872.

Zhe Chen, Weiyun Wang, Yue Cao, Yangzhou Liu, Zhangwei Gao, Erfei Cui, Jinguo Zhu, Shenglong Ye, Hao Tian, Zhaoyang Liu, Lixin Gu, Xuehui Wang, Qingyun Li, Yimin Ren, Zixuan Chen, Jiapeng Luo, Jiahao Wang, Tan Jiang, Bo Wang, Conghui He, Botian Shi, Xingcheng Zhang, Han Lv, Yi Wang, Wenqi Shao, Pei Chu, Zhongying Tu, Tong He, Zhiyong Wu, Huipeng Deng, Jiaye Ge, Kai Chen, Min Dou, Lewei Lu, Xizhou Zhu, Tong Lu, Dahua Lin, Yu Qiao, Jifeng Dai, and Wenhai Wang. Expanding performance boundaries of open-source multimodal models with model, data, and test-time scaling. *CoRR*, abs/2412.05271, 2024b. doi: 10.48550/ARXIV.2412.05271. URL https://doi.org/10.48550/arXiv.2412.05271.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing GUI grounding for advanced visual GUI agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pp. 9313–9332. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.505. URL https://doi.org/10.18653/v1/2024.acl-long.505.

Gaole Dai, Shiqi Jiang, Ting Cao, Yuanchun Li, Yuqing Yang, Rui Tan, Mo Li, and Lili Qiu. Advancing mobile GUI agents: A verifier-driven approach to practical deployment. *CoRR*, abs/2503.15937, 2025. doi: 10.48550/ARXIV.2503.15937. URL https://doi.org/10.48550/arXiv.2503.15937.

Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pp. 1189–1232, 2001.

Wenjia Jiang, Yangyang Zhuang, Chenxi Song, Xu Yang, and Chi Zhang. Appagentx: Evolving GUI agents as proficient smartphone users. *CoRR*, abs/2503.02268, 2025. doi: 10.48550/ARXIV.2503.02268. URL https://doi.org/10.48550/arXiv.2503.02268.

Wei Li, William E. Bishop, Alice Li, Christopher Rawles, Folawiyo Campbell-Ajala, Divya Tyamagundlu, and Oriana Riva. On the effects of data scale on UI control agents. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/a79f3ef3b445fd4659f44648f7ea8ffd-Abstract-Datasets_and_Benchmarks_Track.html.

Wentong Li, Yuqian Yuan, Jian Liu, Dongqi Tang, Song Wang, Jie Qin, Jianke Zhu, and Lei Zhang. Tokenpacker: Efficient visual projector for multimodal LLM. *Int. J. Comput. Vis.*, 133(10): 6794–6812, 2025. doi: 10.1007/S11263-025-02491-7. URL https://doi.org/10.1007/s11263-025-02491-7.

Kevin Qinghong Lin, Linjie Li, Difei Gao, Zhengyuan Yang, Shiwei Wu, Zechen Bai, Stan Weixian Lei, Lijuan Wang, and Mike Zheng Shou. Showui: One vision-language-action model for GUI visual agent. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2025, Nashville, TN, USA, June 11-15, 2025*, pp. 19498–19508. Computer Vision Foundation / IEEE, 2025. doi: 10.1109/CVPR52734.2025.01816. URL https://openaccess.thecvf.com/content/CVPR2025/html/Lin_ShowUI_One_Vision-Language-Action_Model_for_GUI_Visual_Agent_CVPR_2025_paper.html.

Guangyi Liu, Pengxiang Zhao, Liang Liu, Zhiming Chen, Yuxiang Chai, Yaozhen Liang, Wenhao Wang, Siheng Chen, Zhengxi Lu, Shuai Ren, Hao Wang, Shibo He, Yong Liu, and Wenchao Meng. Learnact: Few-shot mobile GUI agent with a unified demonstration benchmark. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics, ACL 2026, San Diego, California, United States, July 2-7, 2026*, pp. 29820–29843. Association for Computational Linguistics, 2026. URL https://aclanthology.org/2026.findings-acl.1491/.

Run Luo, Lu Wang, Wanwei He, and Xiaobo Xia. GUI-R1 : A generalist r1-style vision-language action model for GUI agents. *CoRR*, abs/2504.10458, 2025. doi: 10.48550/ARXIV.2504.10458. URL https://doi.org/10.48550/arXiv.2504.10458.

Mingyu Ouyang, Kevin Qinghong Lin, Mike Zheng Shou, and Hwee Tou Ng. Focusui: Efficient UI grounding via position-preserving visual token selection. *CoRR*, abs/2601.03928, 2026. doi: 10.48550/ARXIV.2601.03928. URL https://doi.org/10.48550/arXiv.2601.03928.

Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, Wanjun Zhong, Kuanye Li, Jiale Yang, Yu Miao, Woyu Lin, Longxiang Liu, Xu Jiang, Qianli Ma, Jingyu Li, Xiaojun Xiao, Kai Cai, Chuang Li, Yaowei Zheng, Chaolin Jin, Chen Li, Xiao Zhou, Minchao Wang, Haoli Chen, Zhaojian Li, Haihua Yang, Haifeng Liu, Feng Lin, Tao Peng, Xin Liu, and Guang Shi. UI-TARS: pioneering automated GUI interaction with native agents. *CoRR*, abs/2501.12326, 2025. doi: 10.48550/ARXIV.2501.12326. URL https://doi.org/10.48550/arXiv.2501.12326.

Kele Shao, Keda Tao, Kejia Zhang, Sicheng Feng, Mu Cai, Yuzhang Shang, Haoxuan You, Can Qin, Yang Sui, and Huan Wang. A survey of token compression for efficient multimodal large language models. *Trans. Mach. Learn. Res.*, 2026, 2026. URL https://openreview.net/forum?id=G2od9JVHkE.

Jiazheng Sun, Mingxuan Li, Yingying Zhang, Jiayang Niu, Yachen Wu, Ruihan Jin, Shuyu Lei, Pengrongrui Tan, Zongyu Zhang, Ruoyi Wang, Jiachen Yang, Boyu Yang, Jiacheng Liu, and Xin Peng. Ambibench: Benchmarking mobile GUI agents beyond one-shot instructions in the wild. *CoRR*, abs/2602.11750, 2026. doi: 10.48550/ARXIV.2602.11750. URL https://doi.org/10.48550/arXiv.2602.11750.

Qwen Team. Qwen3-vl technical report. *CoRR*, abs/2511.21631, 2025. doi: 10.48550/ARXIV.2511.21631. URL https://doi.org/10.48550/arXiv.2511.21631.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-agent: Autonomous multi-modal mobile device agent with visual perception. *CoRR*, abs/2401.16158, 2024. doi: 10.48550/ARXIV.2401.16158. URL https://doi.org/10.48550/arXiv.2401.16158.

Zhenhailong Wang, Haiyang Xu, Junyang Wang, Xi Zhang, Ming Yan, Ji Zhang, Fei Huang, and Heng Ji. Mobile-agent-e: Self-evolving mobile assistant for complex tasks. *CoRR*, abs/2501.11733, 2025a. doi: 10.48550/ARXIV.2501.11733. URL https://doi.org/10.48550/arXiv.2501.11733.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net, 2025b. URL https://proceedings.mlr.press/v267/wang25bx.html.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. Autodroid: Llm-powered task automation in android. In Weisong Shi, Deepak Ganesan, and Nicholas D. Lane (eds.), *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking, ACM MobiCom 2024, Washington D.C., DC, USA, November 18-22, 2024*, pp. 543–557. ACM, 2024. doi: 10.1145/3636534.3649379. URL https://doi.org/10.1145/3636534.3649379.

Zhiyong Wu, Zhenyu Wu, Fangzhi Xu, Yian Wang, Qiushi Sun, Chengyou Jia, Kanzhi Cheng, Zichen Ding, Liheng Chen, Paul Pu Liang, and Yu Qiao. OS-ATLAS: A foundation action model for generalist GUI agents. *CoRR*, abs/2410.23218, 2024. doi: 10.48550/ARXIV.2410.23218. URL https://doi.org/10.48550/arXiv.2410.23218.

Bin Xie, Rui Shao, Gongwei Chen, Kaiwen Zhou, Yinchuan Li, Jie Liu, Min Zhang, and Liqiang Nie. Gui-explorer: Autonomous exploration and mining of transition-aware knowledge for GUI agent. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pp. 5650–5667. Association for Computational Linguistics, 2025a. doi: 10.18653/V1/2025.ACL-LONG.282. URL https://doi.org/10.18653/v1/2025.acl-long.282.

Yuquan Xie, Zaijing Li, Rui Shao, Gongwei Chen, Kaiwen Zhou, Yinchuan Li, Dongmei Jiang, and Liqiang Nie. Mirage-1: Augmenting and updating GUI agent with hierarchical multimodal skills. *CoRR*, abs/2506.10387, 2025b. doi: 10.48550/ARXIV.2506.10387. URL https://doi.org/10.48550/arXiv.2506.10387.

Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. Set-of-mark prompting unleashes extraordinary visual grounding in GPT-4V. *CoRR*, abs/2310.11441, 2023. doi: 10.48550/ARXIV.2310.11441. URL https://doi.org/10.48550/arXiv.2310.11441.

Senqiao Yang, Yukang Chen, Zhuotao Tian, Chengyao Wang, Jingyao Li, Bei Yu, and Jiaya Jia. Visionzip: Longer is better but not necessary in vision language models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2025, Nashville, TN, USA, June 11-15, 2025*, pp. 19792–19802. Computer Vision Foundation / IEEE, 2025. doi: 10.1109/CVPR52734.2025.01843. URL https://openaccess.thecvf.com/content/CVPR2025/html/Yang_VisionZip_Longer_is_Better_but_Not_Necessary_in_Vision_Language_CVPR_2025_paper.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Jiayi Zhang, Chuang Zhao, Yihan Zhao, Zhaoyang Yu, Ming He, and Jianping Fan. Mobileexperts: A dynamic tool-enabled agent team in mobile devices. *CoRR*, abs/2407.03913, 2024. doi: 10.48550/ARXIV.2407.03913. URL https://doi.org/10.48550/arXiv.2407.03913.

Yuan Zhang, Chun-Kai Fan, Junpeng Ma, Wenzhao Zheng, Tao Huang, Kuan Cheng, Denis A. Gudovskiy, Tomoyuki Okuno, Yohei Nakata, Kurt Keutzer, and Shanghang Zhang. Sparsevlm: Visual token sparsification for efficient vision-language model inference. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/zhang25s.html.

Hanzhang Zhou, Xu Zhang, Panrong Tong, Jianan Zhang, Liangyu Chen, Quyu Kong, Chenglin Cai, Chen Liu, Yue Wang, Jingren Zhou, and Steven Hoi. MAI-UI technical report: Real-world centric foundation GUI agents. *CoRR*, abs/2512.22047, 2025. doi: 10.48550/ARXIV.2512.22047. URL https://doi.org/10.48550/arXiv.2512.22047.

MEMOIR: SUPPLEMENTARY MATERIAL

This supplementary material provides the implementation details of MEMOIR. First, we introduce the complete reduction recipe inventory. Second, we show the prompt used for reduced-model execution. Finally, we describe the structural fingerprint used to vectorize reduced inputs.

A REDUCTION RECIPE INVENTORY

As described in the main paper, a reduction recipe is a four-field tuple

$$r = (action,\ anchor,\ context,\ slot).$$

The four fields form a pipeline rather than four independent labels. First, the action field keeps elements that can receive the required action. Second, the anchor field identifies the element or region around which the input should be reduced. The context field then decides which neighboring elements remain visible. Finally, the slot field copies an exact text argument when the action requires one. Table 7 shows how the implemented primitives are combined. We do not enumerate the full Cartesian product. Instead, one wait recipe, four navigation recipes, and four localized action families paired with nine anchors produce $1 + 4 + 4 \times 9 = 41$ recipes.

Table 7: Construction of the reduction-recipe family. "All nine" refers to the anchor primitives in Tables 8 and 9.

Action	Eligible elements	Anchor set	#
wait	none; deterministic	screen	1
navigate	instruction-derived target	screen/history/ bottom/top bar	4
click	clickable elements	all nine	9
long press	clickable elements	all nine	9
type	editable elements	all nine	9
scroll	scrollable elements	all nine	9
Complete recipe family			**41**

Action Primitives. The action field has six primitives: `clickable`, `long_pressable`, `editable`, `scrollable`, `navigate`, and `wait`. The first two retain clickable a11y elements, while `editable` and `scrollable` retain editable and scrollable elements. `navigate` extracts a target from the current instruction. Its target can be Back, Home, Enter, or an application name from an open-app instruction. `wait` has no visual target. When `navigate` yields one target or `wait` applies, b_0 directly issues the action without invoking the VLM. The other action primitives require a reduced-model call.

Table 8: Instruction- and element-based anchor primitives.

Anchor	Selection rule and applicability
`instruction`	Merge elements whose text, role, class, or resource identifier overlaps an instruction token; invalid if none overlaps.
`editable`	Select the first editable element in a11y-tree order (`first` after editable action filtering); invalid if none exists.
`clickable-text`	Rank clickable text elements by instruction overlap, break ties by screen order, and select the first; invalid if none has text.
`scrollable`	Select the first scrollable container in a11y-tree order; invalid if none exists.

Anchor Primitives. Tables 8 and 9 give the complete anchor inventory and its materialization rules. For every non-screen anchor, we merge the selected a11y bboxes, expand the resulting region by 15% of its width and height, and clip it to the screenshot boundary. This padding retains immediate visual context without restoring the full screen. The `screen` anchor retains the full input for each localized action family.

Table 9: Layout- and history-based anchor primitives. Together with Table 8, these form the complete anchor inventory.

Anchor	Selection rule and applicability
`top-bar`	Merge elements intersecting the top 18% of the screen; invalid if the region is empty.
`bottom-nav`	Search the bottom 20% and prioritize short clickable labels, tabs, and navigation controls; invalid if empty.
`history`	Select the current action-eligible element closest to the last action.
`keyboard`	Merge keyboard-like elements in the lower 45%; if keys are absent but the keyboard is visible, use the lower-screen keyboard region.
`screen`	Use the complete screen; always valid and retained as the full-context member.

Recipe Materialization. During offline muscle training, we discard a recipe for a step if its action-compatible set is empty, its anchor cannot be grounded, or its slot cannot be copied unambiguously. For coordinate actions, the target center must also lie inside the materialized region when the recipe is ranked during discovery. Among valid recipes, a smaller region receives a higher reduction score, and only the top candidates are replayed under the budget levels in the main paper. We use ground-truth information only for this offline ranking. At runtime, the stored recipe is reapplied to the current instruction, history, and a11y tree without access to the ground-truth action.

Table 10: Context and slot primitives. Slot values are copied rather than generated.

Stage	Primitive	Materialization rule
Context	`anchor`	Keep elements overlapping the padded anchor region.
	`nearest_k`	Add the k nearest eligible elements by bbox-center distance; k is capped by the selected budget.
	`screen`	Keep the complete screenshot and all eligible elements.
Slot	`none`	Bind no textual argument.
	`instruction`	Copy an exact quoted or syntactically delimited span from the instruction.
	`element`	Copy exact visible-node text when the instruction requests an on-screen value.
	`history`	Copy the exact argument of a recent type action.

Context and Slot Primitives. Table 10 completes the primitive list. The context policy specifies the final region around the anchor. The slot policy is used only when an action contains a text argument. For a type action, we first check an exact visible-node source when the instruction asks to copy an on-screen value, then check the instruction, and finally check the recent history. Every slot stores its source and exact span. We reject the recipe before model execution if several observable values conflict or the runtime value does not match the source pattern learned during muscle training.

B Prompt for Reduced-Model Execution

We use the same frozen VLM backbone for the reduced model and the full agent. The difference is the decision problem presented to it. We first crop the screenshot with the selected reduction recipe and resize it under the selected budget. Compared with the full-agent prompt, the reduced prompt removes the screen description, candidate serialization, chain-of-thought request, and tool-call wrapper. It asks the model to emit one action invocation directly. The complete system and user prompts are shown below.

We normalize the output coordinates to the visible crop. After parsing, we map them back to the coordinate system of the original screenshot. The type, open-app, and system-button arguments remain strings, while swipe directions are restricted to `up`, `down`, `left`, or `right`. Decoding is greedy. The deterministic budget b_0 does not construct this prompt and makes no VLM call.

Table 11: Complete prompt for reduced-model execution. The step-instruction and history fields in the user message are omitted when unavailable.

```
You are a GUI agent.  You are given a task and your action history.  You
need to perform the next action to complete the task.

## Output Format
Output ONLY one plain text command.  No thinking, no explanation, no
JSON, no tags:
click(x,y)

## Action Space

click(x,y)
long_press(x,y)
type("text")
swipe(x,y,direction)
open("app_name")
system_button("button_name")
wait()

Coordinates are 0-999 relative to the visible region.
direction is one of:  up, down, left, right.

Use double quotes for string arguments.
```

```
[cropped screenshot]
Task goal:  <task goal>
Instruction for the current step:  <step instruction>
Actions already performed:
<action type>:  <argument, direction, application, or (done)>
<additional history entries, if any>

Output exactly one next action for the current screenshot.
```

C VECTORIZATION OF REDUCED INPUTS

We use the structural fingerprint to turn a reduced input into a fixed-width vector. The fingerprint contains two parts. First, the runtime part is computed from the current instruction, action history, and a11y tree. Second, during offline muscle training, we append the action produced by replay and its agreement across budgets. The runtime part contains only features that are available before model execution.

Table 12: Composition of the vectorized feature. The final six dimensions are discovery-only.

Feature group	**Dim.**	**Coordinates**
Recipe identity	4	Action, anchor, context, and eligible-element rule.
Structural size	6	Instruction length; retained/all candidates; option count; singleton indicator; slot count.
Reduction ratios	3	Text, visual-area, and candidate reduction ratios.
Element topology	3	Counts of clickable, editable, and scrollable elements.
Lexical alignment	2	Maximum and mean instruction–element token overlap.
Hashed signatures	12	$H_4(I)$, $H_4(c^\star)$, $H_2(z)$, and $H_2(q)$.
Runtime structural vector	**30**	**Centroid matching and OOD distance.**
Action signature	4	Hashed action type and normalized argument/direction.
Cross-budget agreement	2	Fraction of budgets producing the same action and its normalized support.
Discovery fingerprint	**36**	**Runtime vector plus discovery-only evidence.**

Runtime Structural Vector. Let $H_d(x)$ be a d-dimensional hashed text signature. We tokenize x, take at most its first 16 tokens, assign each token to one of d buckets, count the assignments, and ℓ_2-normalize the counts. This process is deterministic and requires no learned text encoder or model call. The 30-dimensional runtime vector is

$$v_{\mathrm{rt}}(s) = [e_{\mathrm{act}}, e_{\mathrm{anc}}, e_{\mathrm{ctx}}, e_{\mathrm{gen}}, \ell_I, n_c, n_{\mathrm{all}}, n_{\mathrm{opt}}, \mathbb{1}_{\mathrm{single}}, n_{\mathrm{slot}}, \\ \rho_{\mathrm{text}}, \rho_{\mathrm{vis}}, \rho_{\mathrm{cand}}, n_{\mathrm{click}}, n_{\mathrm{edit}}, n_{\mathrm{scroll}}, o_{\max}, o_{\mathrm{mean}}, H_4(I), H_4(c^\star), H_2(z), H_2(q)]. \quad (1)$$

Here, $e_{\mathrm{act}}, e_{\mathrm{anc}}, e_{\mathrm{ctx}}, e_{\mathrm{gen}}$ are stable scalar encodings of the action, anchor, context, and eligible-element rule. ℓ_I is the instruction length, using the task goal when no step instruction is available. n_c and n_{all} are the numbers of retained and original a11y candidates; n_{opt} is the number of admissible action options; $\mathbb{1}_{\mathrm{single}}$ marks a singleton option set; and n_{slot} is the number of bound text arguments. The three ρ terms are the text, visual-area, and candidate reduction ratios. The next three terms count clickable, editable, and scrollable elements on the full screen. $o_{\max}$ and o_{mean} measure the maximum and mean token overlap between the instruction and retained elements. Finally, I, $c^\star$, z, and q denote the instruction, top retained element text, slot value, and top element role.

Discovery Feature Vector. For a successful replay w, we append a four-dimensional action signature and two agreement statistics:

$$v_{\mathrm{disc}}(w) = [v_{\mathrm{rt}}(w),\ H_4(a_w),\ 2\gamma_w,\ 2u_w] \in \mathbb{R}^{36},$$

where a_w is the normalized action signature, γ_w is the fraction of the recipe's budget replays that produce the same action, and u_w is the corresponding normalized support. We multiply the two agreement statistics by two to give cross-budget consistency comparable weight during cosine clustering. We connect successful steps with the same reduction recipe and compatible slot sources when their cosine similarity exceeds the clustering threshold.

For each cluster, we store the centroid of the 30-dimensional runtime vectors as a muscle memory entry. At runtime, MEMOIR reapplies the entry's reduction recipe, computes $v_{\mathrm{rt}}(s)$, and compares it with the stored centroid by cosine distance. The action signature and agreement terms are not available and are not needed. The ML-based gate then reuses the observable scalars in Table 12, together with the cosine similarity and candidate budget, to score the remaining entries. The structural fingerprint adds only deterministic text hashing, counting, and arithmetic to the critical path.

Normalization and Missing Values. We encode the four recipe identifiers with a deterministic scalar function $E(x) = (\sum_{c \in x} \mathrm{ord}(c) \bmod 997)/997$. For a text field x, we use $H_d(x)$ to count the first 16 tokens in d deterministic buckets and divide the resulting vector by its ℓ_2 norm. An absent instruction, candidate text, slot, or role contributes an all-zero signature. Element and token counts remain non-negative scalars, while all reduction ratios are clipped to $[0, 1]$. We do not store learned embeddings, vocabularies, or normalization statistics. We apply the same feature extractor during muscle training, calibration, and runtime matching.

Example. For example, consider the instruction `Type "Boston" in the search box`. The action primitive retains editable elements, an editable or instruction anchor localizes the search box, and the context policy adds its nearby elements. The slot policy copies `Boston` from the instruction and records one bound slot. The runtime vector records one slot, the number of editable elements on the screen, the instruction–element overlap, the crop and text reduction ratios, and hashed signatures for the instruction, selected element, copied value, and element role. During offline muscle training, we append the reduced model's normalized type-action signature and its agreement across the tested budgets. At runtime, we omit these six discovery-only features and compare the 30 runtime features with the stored centroid vector.

Feature Use at Runtime. During offline muscle training, we use the 36-dimensional vector only to cluster successful steps. After clustering, we use the first 30 dimensions of all cluster members to calculate the centroid vector. We also calculate the OOD distance as the 95th-percentile cosine distance between the successful steps and the centroid vector. At runtime, we apply the recipe and extract the same 30 features from the current input. This process does not use screenshot pixels, the ground-truth action, the full-agent output, any future step, or any success label. The distance-based filtering gate then compares the current runtime vector with the stored centroid vector.

D LIMITATIONS AND DISCUSSIONS

High-Level Success Rate. The gains of MEMOIR concentrate on low-level step accuracy. The high-level success rate is preserved but not substantially improved. This is because muscle memory accelerates the execution of familiar steps, whereas task failures mostly come from planning steps that require full context. Combining MEMOIR with planning-oriented approaches, *e.g.*, high-level shortcuts, is a promising direction since the two are complementary.

Offline Training Overhead. MEMOIR requires an offline muscle training pipeline over past successful traces, which takes several GPU-hours to train. The training process can be conducted periodically without affecting agent execution. Besides, the training is incremental by design, *i.e.*, newly collected traces can be clustered into the existing memory store without retraining from scratch.

Generality. The current design of MEMOIR targets GUI agents with screenshot inputs. The reduction recipes rely on the a11y tree to locate elements, and the budget levels reduce inputs through image cropping. Therefore, MEMOIR does not directly apply to agents without structured visual inputs, *e.g.*, software engineering agents whose steps demand broad and task-specific textual context. We leave extending muscle memory to general agents as our future work.