TuriX: A Modular Vision-Language Framework for Autonomous GUI Task Execution on MacOS

Tongyu Yan* Yankai Pei† Zhixuan Lin‡ Wenjie Dong§ Pinran Wang¶

July 21, 2025

Abstract

Background. Transformer-based large language models (LLMs) such as *ChatGPT* and the recently-released *Grok-4* have set new state-of-the-art levels for reasoning and question answering, but they remain tools that amplify human productivity rather than agents that autonomously *do the work* for us. 2025 has seen the rise of *API-first* agent frameworks—e.g. Google A2A, and Anthropic MCP—which let LLMs invoke registered functions, yet these systems are bounded by the availability and granularity of the underlying APIs.**Objective.** We introduce **TuriX**, the modular GUI vision–language agent that executes arbitrary desktop workflows by imitating human GUI interactions rather than calling predefined APIs, thereby escaping API bottlenecks and enabling "unlimited" autonomy. **Methods.** TuriX decomposes long-horizon tasks into four cooperating roles: *Planner*, *Executor*, *Evaluator*, and *Supervisor*. This architecture is **TuriX Parallelum**. The Executor is a vision-language model (VLM) trained with (i) a region-aware, loss for pixel-accurate grounding and (ii) auxiliary textual context derived from the macOS `AXUIElement` accessibility tree. **Results.** On the public showdown-clicks and private MacClick benchmarks, TuriX attains **64.38%** click-location accuracy—an absolute 0.11% winner than the OpenAI CUA model on showdown-clicks, but a 10% relative improvement over the UI-TARS-72B-SFT model, and achieves end-to-end pass@5 task success rates **68%** in real-world MacOS laptop task execution. **Conclusions.** Our study demonstrates that (1) GUI-level control lifts the ceiling imposed by API-only agents and (2) a role-specialised, multi-model architecture materially improves both robustness and precision for desktop automation.

1 Introduction

1.1 From LLM Tools to Autonomous Agents

Transformer architectures have catalysed a step-change in natural-language technology, culminating in chat systems such as ChatGPT and Grok-4 that routinely outperform expert humans on reasoning benchmarks.Yet these models remain fundamentally reactive: they generate information but do not *act*. Recent "API agents"—Manus AI in China, OpenAI's Operator model, and others—address this gap by allowing an LLM to call external tools via protocols like MCP or A2A to complete multi-step jobs.While successful, API agents inherit a hard constraint: an action is only possible if somebody first exposed a suitable API endpoint. Many real-world processes, especially on legacy desktop software, offer no such endpoints.

*TuriX AI Lab — tongyu.yan@turix.ai
†TuriX AI Lab — yankai.pei@turix.ai
‡TuriX AI Lab — zhixuan.lin@turix.ai
§TuriX AI Lab — wenjie.dong@turix.ai
¶TuriX AI Lab — pinran.wang@turix.ai

1.2 Why a GUI Agent for macOS?

Desktop GUIs remain the lingua franca of professional workflows. A GUI-centric agent that can perceive widgets, decide a sequence of clicks/keystrokes, and monitor outcomes would be able to automate *any* software, circumventing API scarcity entirely. macOS is an ideal first target because its Accessibility framework (`AXUIElement`) exposes a programmatic UI tree that can be fused with visual context to improve grounding accuracy.

1.3 Challenges in Vision-Language GUI Control

Two technical hurdles limit prior attempts:

1. **Prompt complexity and context length.** Packing every intermediate state into a single prompt causes either omission (when too short) or distraction (when too long). Merging all responsibilities into one giant prompt/model further degrades agent performance, as observed in multi-agent orchestration studies.

2. **Pixel-level precision.** Standard VLMs regress coordinates as discrete tokens, yielding low-IoU predictions and missed clicks. Fine-tuning a VLM with region-aware losses and rewards can boost GUI clicking accuracy.

1.4 TuriX Parallelum: A Modular Vision–Language Framework

We therefore design **TuriX Parallelum** with four specialised roles:

- **Planner** parses natural-language instructions into a high-level step by step plan.
- **Executor** (fine-tuned VLM or any VLM API) predicts precise actions, receiving both the screenshot and a textual UI tree to disambiguate visually similar elements.
- **Evaluator** verifies post-action state and decides success or failure, and whether to call for supervisor for replanning.
- **Supervisor** intervenes when Evaluator detects consecutive failures, revising the plan to escape from the loop.

This role separation shortens individual prompts, keeps token windows tractable, and lets each model specialised.

1. A *GUI-first* autonomy paradigm that removes API dependence by operating directly on macOS desktops.
2. A role-based agent architecture that empirically increases success rate by **5**% to **15**% on long-horizon tasks versus monolithic baselines.
3. A region-aware training recipe that delivers **38.4**% absolute gains in click accuracy.

2 Brief Review of Existing GUI Agents

2.1 Agents

The earliest software "agents" were Robotic Process Automation (RPA) systems [1]. Operating in highly structured environments, these platforms executed hand-crafted rules and depended on explicit APIs; they neither adapted to novel stimuli nor accumulated knowledge, and any change in workflow demanded manual reprogramming. The release of ChatGPT [2] demonstrated that large language models (LLMs) can reason autonomously, leading to agent frameworks such as AutoGPT, SeeAct, and MobileExperts [3–5]. These systems reduce human supervision by decomposing tasks into step-wise plans that the agent then executes. Nevertheless, workflow-based agents remain *fragile*, impose significant *maintenance overhead*, separate *learning paradigms*, require *rigid control flows*, and often suffer from *module incompatibilities* [6]. Consequently, there is growing interest in agents that perceive, interpret, and act directly within GUIs [7].

2.2 GUI Agents

Early GUI agents represented the interface state using structured text—HTML or DOM trees—and generated textual actions accordingly [8, 9]. This approach incurs high annotation cost, struggles to capture the *actual* visual state, and generalises poorly [10]. With the rapid progress of vision–language models (VLMs), recent work has shifted to *vision-centric* interaction [11, 12]. To benchmark the challenging task of **GUI grounding** under diverse visual layouts, several datasets have been released [12–14], accelerating research momentum in this space.

2.3 Frameworks

Designing architectures that can tackle multi-step GUI tasks is an active research area:

1. **AgentS2** [11] splits complex tasks into sub-goals using a *planning expert*; a Mixture-of-Grounding-Experts (MoG) then carries out each sub-goal, after which the planner updates the remaining goal list.

2. **UI-TARS** [6] inserts an explicit *reasoning* step before every action. Unlike AgentS2, it dispenses with a global planner and instead selects actions directly from a predefined space, conditioning on the current screen, the task instruction, and memory from previous steps.

3. **GTA1** [15] introduces a two-stage paradigm. Stage 1 enumerates candidate actions via a planner; Stage 2 employs a judge to pick the best action. Non-grounding actions are executed immediately, whereas grounding actions rely on a self-trained visual model to predict coordinates.

2.4 Performance Improvement

Irrespective of framework, *GUI grounding accuracy* is the primary performance bottleneck. Early work relied on supervised fine-tuning (SFT) [14, 16], but SFT alone struggles on high-resolution, visually complex UIs [12]. Inspired by DeepSeek [17], researchers have adopted reinforcement learning. Many leverage GRPO [18]—for example, GTA1 [15]— and have explored test-time RL as well. **ZeroGUI** [7] combines GRPO with online adaptation, demonstrating that reinforcement-based training substantially advances grounding precision.

3 Methodology

Our approach is divided into two complementary parts: the *agent framework* (Section 3.1) and the *model-training pipeline* (Section 3.2). The former explains how TuriX Parallelum orchestrates multi-role cooperation for reliable desktop control; the latter details how we train a vision–language Executor that is both GUI-aware and pixel-accurate.

3.1 Agent Framework

As mentioned in the section 2, the overall task success rate of Agent S2 increases significantly from 26% to 34.5% on the OSWorld benchmark, demonstrating the effectiveness of multi-role cooperation framework. Therefore, a modular agent framework is designed to improve the task success rate, execution efficiency and the overall stability of the agent.

3.1.1 Overall Architecture

TuriX Parallelum follows a four-role design—**Planner**, **Executor**, **Evaluator**, and **Supervisor**—connected in a closed feedback loop. A natural-language instruction is first decomposed by the Planner into a step-by-step plan. The current step is executed by the Executor, verified by the Evaluator, and, if necessary, corrected or replanned by the Supervisor. This modular decomposition shortens individual prompts and lets each model specialise, mitigating the "context dilution" observed in monolithic agents.

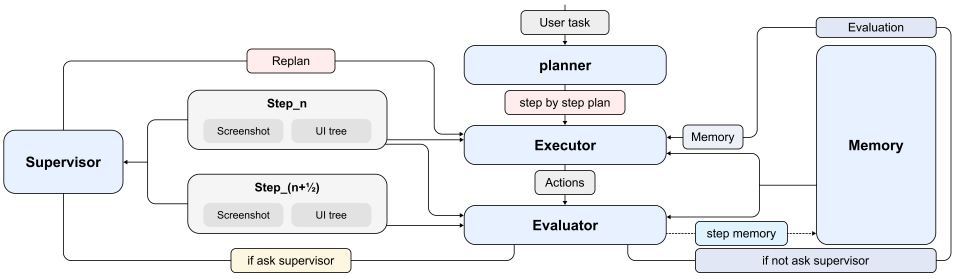

Figure 1: This is the highlevel overview of the TuriX agent framework. The Planner decomposes the task into a step-by-step plan, which is executed by the Executor. The Evaluator verifies the success of the action, and the Supervisor intervenes if necessary. The supervisor accepts all the screenshots, state, and whole history and replans the remaining steps if necessary.

Figure 1 illustrates the TuriX agent framework, which consists of the following components:

Planner. The Planner parses natural-language instructions into a high-level step by step plan. It is a common fact that most of the user tends to provide short prompts, which is not a great instruction for the Executor to follow. Therefore, a well-engineered planner is essential to understand the user's intention and decompose the task into smaller steps. This can stabilises the Executor's performance and improve the overall task success rate, proved by the Agent S2.

Executor. The Executor is a vision-language model (VLM), either an API endpoint like Google Gemini or a fine-tuned model such as our TuriX model. It should understand the step by step

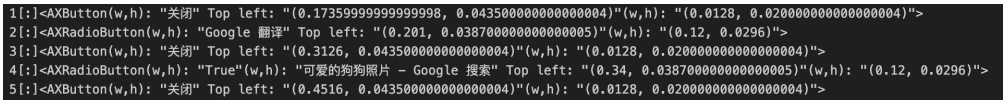

Figure 2: A sample UI sheet. The UI sheet contains the accessibility tree information, including the bounding boxes, roles, and labels of the elements on the screen.

plan, make decisions based on the current state (screenshot and UI tree), and predict precise actions (clicks, scrolls, etc.) to execute the current step.

Evaluator. The Evaluator verifies whether the Executor's action succeeded base on the screenshots before and after the action, and the action being executed. It judges success or failure, and returns a brief diagnostic to guide the Executor's retry if a failure is detected. If the action fails repeatedly, the Evaluator calls for the Supervisor to replan the remaining steps.

Supervisor. The Supervisor is responsible for high-level oversight and intervention. If the Evaluator detects persistent failures, the Supervisor analyzes the situation by looking at the screenshots, state, and action history, and may reassign roles, adjust the plan, or provide additional context to the Executor. This ensures that the agent can adapt to unforeseen challenges and maintain progress toward the goal.

Fine-grained Control Loop. We adopt a *per-step* verification scheme rather than waiting for an entire sub-task to finish (as in Agent S2). After every GUI action, the Evaluator independently judges success and, on failure, returns a brief diagnostic that guides the Executor's retry. Persistent failure triggers the Supervisor, which replans the remaining steps to escape local loops, thus preventing unproductive retries and improving overall robustness.

3.1.2 Framework Optimization

Parallel Execution. The Executor and Evaluator run in a pipelined fashion: while the Executor works on step n, the Evaluator assesses step $n-1$. When the Executor model is generating the actions of step n by assuming the previous step is successful, the Evaluator can simultaneously generate the evaluation for step $n-1$. The generated step n actions can only be sent to the controller after the Evaluator has approved the actions of step $n-1$. If the evaluation of step $n-1$ fails, the Executor will not send the actions of step n to the controller. Instead, it will generate alternative actions based on the current state and the evaluation guide. This overlap keeps latency close to an Executor-only baseline when accuracy is high, yet provides rapid correction when errors emerge.

UI-Tree–Augmented Perception. Figure 2 shows a sample UI sheet. MacOS exposes a hierarchical accessibility tree that lists on-screen widgets, bounding boxes, roles, and labels.We serialise this tree into a compact textual "UI sheet" and feed it to the Executor and Evaluator alongside the screenshots. Two complementary click modes are thereby enabled:

1. **Element-index click**: the model outputs the index of a desired node (e.g. `element_41`). Accuracy is high when the UI tree information is complete but degrades if elements are missing or element descriptions are missing.

2. **Coordinate click**: the model predicts pixel coordinates $[x, y]$, guided by both the screenshot and the UI sheet. This remains viable when the tree is sparse but demands fine localisation skill.

A lightweight decision heuristic is to select the element mode when the tree does not miss many elements, else the coordinate mode is used. This hybrid strategy seems to be clever. In practice, the model first tries to use the element mode, no matter the UI is sparse or not, to click the wrong element, and then it tries the coordinate mode to click the right element. The issue here is that the first failure can be costly, as it may lead to a wrong click that confuses the model and leads to a failure in the next step. In addition, models without grounding training fail to click the right element in coordinate mode with a high probability.

Memory Budgeting. A high quality screenshot is around a few thousand tokens, and the UI sheet is typically a few thousand to several tens of thousands of tokens. Most of the opensource VLMs have a context length of 128k tokens, which is sufficient for single-turn conversation, but insufficient for longer conversation. Tasks that require more than 6 steps are not feasible with a multi-turn conversation framework, therefore a single-turn conversation framework with advanced memory and context management is used in this paper.

3.2 Executor Model Training Pipeline

Grounding is the key to precise GUI control, while most of the existing VLMs been tested are not capable of pixel-level precision. We therefore fine-tune a vision–language model (VLM) to achieve this level of accuracy, and correctly execute GUI tasks with minimum number of steps. Our Executor model TuriX is based on the Qwen2.5-VL-72B model.

3.2.1 Data Sources & Curriculum

Training combines (i) public grounding corpora such as Aria-UI, Uipad and Wave-UI, (ii) 3,000 hours of self-recorded MacOS workflows spanning daily, productivity, creative, and developer apps, and (iii) thousands of human labeled DPO data. Each stage has different objective or reward function to guide the model to learn different skills. We employ a four-stage curriculum:

1. **Stage 0: Background Supervised Fine Tuning (SFT).** Initialize from Qwen2.5-VL-72B and teach basic GUI concepts (*icons*, *text elements*, *images*).

2. **Stage 1: Grounding + Task Mix.** Mix element-box grounding with general multi-step tasks to align perception and smart action.

3. **Stage 2: Group Relative Policy Optimization (GRPO).** Guided RLHF with GRPO maximises a composite reward: behave like a human expert on grounding and task execution strategy. multi-action tasks.

4. **Stage 3: DPO Refinement.** Direct Preference Optimisation corrects residual failure cases and stabilises the policy before deployment.

This four-stage curriculum is designed to progressively teach the model the necessary knowledge and skills for GUI task execution.

3.2.2 Stage 0

The stage 0 is neccessary when finetuning a Qwen2.5-VL-72B model, as it is a general VLM model that is not specifically trained for GUI tasks. In addition, the training dataset of Qwen2.5-VL-72B contain only a small amount of GUI tasks, which is not sufficient for the model to learn the GUI concepts. Therefore, a background SFT stage can help the model to learn the basic GUI concepts, such as icons, text elements, and images.

3.2.3 Stage 1

This is the first time the model learn to execute GUI tasks. The training data for this stage consists of a mixture of element-box grounding and general multi-step tasks. This helps the model to align its perception with the actions it needs to take in a GUI environment.

3.2.4 Stage 2

In this stage, we use Group Relative Policy Optimization (GRPO) to train the model. GRPO is a guided RLHF method that maximises a composite reward, which is designed to guide the model to think before it acts. However, training a GUI agent is different from training a general model when doing GRPO. GUI agent requires fast execution, which is not the case for general models. Therefore, we use a modified GRPO method that is designed for GUI agents. The objective of this stage is to let the model to think and understand how to executue the task so that it can behave like a human expert. This stage consolidate the knowledge and skills learned in the previous stages, and prepares the model for the final refinement stage.

3.2.5 Stage 3

In the final stage, we use Direct Preference Optimization (DPO) to refine the model. This is the fastest stage that human expert can teach the model how to do a combination of actions in one steps, and what combination of actions are not acceptable. DPO is a method that corrects residual failure cases and stabilises the policy before deployment.

Each stage has different objective or reward function to guide the model to learn different skills, and the performance keeps improving as the model goes through the stages as shown in the next section. The final model is a TuriX model that is capable of executing GUI tasks with high accuracy and efficiency.

4 Experiments and Results

This section evaluates both the full **TuriX** agent and its underlying Executor on macOS. Because existing public suites emphasise Windows-centric or API-level tasks, we introduce a 40-task *macOS Workflow Benchmark* that stresses cross-application operation, long horizons, and execution efficiency.

4.1 Evaluation Framework

Benchmark design. Each task spans $\geq$2 applications and is annotated with four complementary metrics:

- **Task Success (pass@5).** Fraction of tasks completed within five independent trials (best trial retained). Comparable to OSWorld's success rate but with richer task diversity.

- **Completion Proportion.** Human-scored progress ratio (e.g. 10/15 steps = 67%). Captures partial credit that matters for user experience.
- **Task Completion Time.** Wall-clock time of the fastest successful trial, normalised by equal-length task design.
- **Single-Action Latency.** Mean latency per atomic or macro-action (total time / #actions).

Tasks are partitioned into two 20-task subsets: *Easy* (general models plausibly solve) and *Full* (requires GUI-specific training).

4.2 Whole-Agent Performance

20-Task Easy Suite				
Model	**Pass@5**	**Comp.%**	**Time (s)**	**Act. Time (s)**
TuriX	**85.0**	**92**	76	6
Gemini 2.5-Pro (Full)	55	79	249	25
Gemini 2.5-Pro (w/o E&S)	50	74	147	18
Gemini 2.5-Pro (w/o UI)	45	68	204	33
Gemini 2.5-Pro (P-only)	30	60	161	21
Grok-4 (w/o E&S)	45	70	327	70
Kimi-1.5 (w/o E&S)	0	2	–	18
GLM-PC	70	86	97	12
UI-TARS-1.5-7B	65	76	147	13
ACE	40	60	**30**	**3**

Table 1: Performance on the 20 easy GUI tasks. Full = UI-tree + Planner + Evaluator + Supervisor; w/o E&S = no Evaluator/Supervisor; w/o UI = no UI-tree; P-only = Planner only.

Key observations.

- **TuriX dominates mainstream LLMs.** Its 85 % pass@5 is a 30-point absolute gain over Gemini 2.5-Pro (Full) while being 3× faster per action.
- **Evaluator and Supervisor matter.** Removing them from Gemini drops success by 5 pp and halves completion proportion, validating the multi-role design.
- **Structured UI context helps, but is not sufficient.** Gemini with UI-tree but without higher-level roles underperforms TuriX by 40 pp, underscoring the benefit of role specialisation and VLM fine-tuning.

Full 40-Task Suite				
Model	**Pass@5**	**Comp.%**	**Time (s)**	**Act. Time (s)**
TuriX	**67.5**	**88**	71	6
GLM-PC	57.5	78	87	11
UI-TARS-1.5-7B	50	70	167	15
ACE	37.5	67	**28**	**3**

Table 2: End-to-end results on the full 40-task benchmark. TuriX and UI-TARS were served on 4 × H20 GPUs; extrapolation suggests latency halves on 4 × H100/H200 nodes.

On the comprehensive suite TuriX retains a commanding 10 pp margin over the next best model while maintaining industry-competitive latency.

4.3 Executor-Only Analysis

Precise clicking is a prerequisite for downstream success. We isolate this capability with two dedicated datasets:

- **Showdown-Click** (557 labelled clicks).
- **MacClick** (private, hundreds of clicks across diverse apps).

Click Accuracy (no UI-tree)		
Model	Showdown	MacClick
TuriX	64.38	64.31
Ace-Control-medium	**77.56**	–
OpenAI CUA	64.27	–
UI-TARS-72B-SFT	54.40	–
Gemini-2.0-flash	33.40	27.00
Qwen2.5-VL-72B-Instruct	24.78	25.00
GPT-4o	5.21	–

Table 3: Pixel-level click accuracy (%). Higher is better. TuriX matches proprietary OpenAI CUA and surpasses all open-source counterparts except Ace-Control (trained expressly for Showdown-Click).

TuriX Ablations			
Model Variant	Showdown	MacClick	GUI Tasks
TuriX (final)	64.38	64.31	95.44
TuriX + UI-tree	–	**67.12**	95.32
TuriX-ui (UI-tree during train)	58.07	61.80	83.36
SFT-only	61.07	63.80	88.80
GRPO stage	63.09	64.20	94.23

Table 4: Effect of training stages and UI-tree usage. Curriculum culminates in a 39.6-point gain over the Qwen-base initialisation, while retaining robustness when UI metadata is absent.

Curricular ablation. Insights.

- **GRPO → DPO adds real value.** The jump from SFT to GRPO yields $\sim 2\%$ accuracy lift and +5 pp on imitating human, indicating superior strategic reasoning.
- **UI-tree is optional, not mandatory.** At inference time, adding a tree gives a modest +2.8 pp on MacClick; crucially, performance without it remains strong, suiting apps with sparse accessibility data.

5 Discussion

5.1 Trade-offs Between Success Rate, Latency, and Compute

The four-role TuriX Parallelum architecture delivers a clear Pareto frontier. Packing every role into one monolithic prompt yields the lowest latency but sacrifices success (*pass@5*) because critical context is forgotten. Conversely, our pipelined Planner–Executor–Evaluator–Supervisor loop raises the success rate from 50 % to 55 % and 30 % to 45 % on MacOS 20-easy-task at a modest latency cost (Table 1). Parallel execution hides much of that cost, yet does consume extra GPU cycles because the Executor sometimes generates actions that are subsequently vetoed by the Evaluator. Practitioners must therefore choose a point on the speed–accuracy–compute curve that matches their deployment constraints.

5.2 Model Fine-tuning Strategy

Fine-tuning a general VLM for GUI control is non-trivial. Naïve supervised fine-tuning (SFT) boosts clicking accuracy but erodes the model's broad knowledge. Our curriculum—SFT → grounding mix → GRPO → DPO—preserves general capabilities while lifting Showdown-Click accuracy from 24.78 % to 64.38 % (Table 3). GRPO is especially valuable: although slower to train, it teaches the model to *think before acting* (even not outputing thinking process), raising both click precision and downstream task completion.

5.3 Role of UI-Tree Information

UI-tree context is a double-edged sword. When the accessibility tree is rich and complete, element-index clicks are highly reliable; when it is sparse or absent, coordinate mode is indispensable. Our ablation in Table 4 shows that TuriX without UI-tree still attains 64.38% accuracy, but gains a further +2.8% on MacClick when the tree is available. This suggests shipping a tree-agnostic default and opportunistically leveraging structured UI data when present.

5.4 Limitations

- **Evaluator feedback.** The Executor is not yet conditioned on real evaluator signals during training, so it sometimes ignores corrective hints at run-time.
- **Model capacity.** Qwen2.5-VL-72B is small for the breadth of skills required; larger checkpoints may improve long-horizon reasoning.
- **UI-tree dependency.** Some macOS applications expose partial or noisy accessibility trees, limiting element-index mode.

6 Future Work

1. **Training with evaluator signals.** Collect executor–evaluator interaction traces and incorporate them through multi-agent RLHF, enabling the Executor to internalise revision cues. Evaluator will also be trained to provide more accurate and informative feedback to the Executor.
2. **Scaling model size.** Port the curriculum to larger VLMs when availible.

3. **Robust error recovery.** Develop learned rollback policies that can detect and undo destructive clicks without human intervention, further boosting *pass@1*.

4. **Cross-platform generalisation.** Extend TuriX to Windows and Linux by replacing `AXUIElement` features with analogue accessibility APIs, evaluating zero-shot transfer ability.

7 Conclusion

We have introduced **TuriX Parallelum**, a modular vision–language agent that achieves state-of-the-art GUI autonomy on macOS. By decomposing long-horizon tasks into Planner, Executor, Evaluator, and Supervisor roles, and by fine-tuning the Executor with a region-aware, curriculum-based pipeline, TuriX reaches 67.5 % *pass@5* on a 40-task benchmark—exceeding prior VLM baselines while maintaining competitive latency. Our study confirms that (i) GUI-level control removes the ceiling imposed by API-only agents, and (ii) role specialisation, combined with rigorous grounding training, materially improves both robustness and precision in real-world desktop automation. By decomposing long-horizon tasks into Planner, Executor, Evaluator, and Supervisor roles, and by fine-tuning the Executor with a region-aware, curriculum-based pipeline, TuriX reaches 67.5 % *pass@5* on a 40-task benchmark—exceeding prior VLM baselines while maintaining competitive latency. Our study confirms that (i) GUI-level control removes the ceiling imposed by API-only agents, and (ii) role specialisation, combined with rigorous grounding training, materially improves both robustness and precision in real-world desktop automation.

References

[1] Liliana Dobrica. Robotic process automation platform uipath. *Commun. ACM*, 65(4):42–43, March 2022. ISSN 0001-0782. doi: 10.1145/3511667. URL `https://doi.org/10.1145/3511667`.

[2] OpenAI. GPT-3.5 Turbo Model Documentation. `https://platform.openai.com/docs/models#gpt-3-5-turbo`, 2022.

[3] Hui Yang, Sifu Yue, and Yunzhong He. Auto-gpt for online decision making: Benchmarks and additional opinions, 2023. URL `https://arxiv.org/abs/2306.02224`.

[4] Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. Gpt-4v(ision) is a generalist web agent, if grounded, 2024. URL `https://arxiv.org/abs/2401.01614`.

[5] Jiayi Zhang, Chuang Zhao, Yihan Zhao, Zhaoyang Yu, Ming He, and Jianping Fan. Mobileexperts: A dynamic tool-enabled agent team in mobile devices, 2024. URL `https://arxiv.org/abs/2407.03913`.

[6] 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, 2025. URL `https://arxiv.org/abs/2501.12326`.

[7] Chenyu Yang, Shiqian Su, Shi Liu, Xuan Dong, Yue Yu, Weijie Su, Xuehui Wang, Zhaoyang Liu, Jinguo Zhu, Hao Li, Wenhai Wang, Yu Qiao, Xizhou Zhu, and Jifeng Dai. Zerogui: Automating online gui learning at zero human cost, 2025. URL https://arxiv.org/abs/2505.23762.

[8] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

[9] 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, 2024. URL https://arxiv.org/abs/2308.15272.

[10] Zihe Yan and Zhuosheng Zhang. Lasm: Layer-wise scaling mechanism for defending pop-up attack on gui agents, 2025. URL https://arxiv.org/abs/2507.10610.

[11] Saaket Agashe, Kyle Wong, Vincent Tu, Jiachen Yang, Ang Li, and Xin Eric Wang. Agent s2: A compositional generalist-specialist framework for computer use agents, 2025. URL https://arxiv.org/abs/2504.00906.

[12] Kaixin Li, Ziyang Meng, Hongzhan Lin, Ziyang Luo, Yuchen Tian, Jing Ma, Zhiyong Huang, and Tat-Seng Chua. Screenspot-pro: Gui grounding for professional high-resolution computer use, 2025. URL https://arxiv.org/abs/2504.07981.

[13] 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, 2024. URL https://arxiv.org/abs/2410.23218.

[14] Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing gui grounding for advanced visual gui agents, 2024. URL https://arxiv.org/abs/2401.10935.

[15] Yan Yang, Dongxu Li, Yutong Dai, Yuhao Yang, Ziyang Luo, Zirui Zhao, Zhiyuan Hu, Junzhe Huang, Amrita Saha, Zeyuan Chen, Ran Xu, Liyuan Pan, Caiming Xiong, and Junnan Li. Gta1: Gui test-time scaling agent, 2025. URL https://arxiv.org/abs/2507.05791.

[16] Yuhao Yang, Yue Wang, Dongxu Li, Ziyang Luo, Bei Chen, Chao Huang, and Junnan Li. Aria-ui: Visual grounding for gui instructions, 2025. URL https://arxiv.org/abs/2412.16256.

[17] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong

Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

[18] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.