GUIRILLA: A SCALABLE FRAMEWORK FOR AUTOMATED DESKTOP UI EXPLORATION

Sofiya Garkot, Maksym Shamrai, Ivan Synytsia & Mariya Hirna
MacPaw
Kyiv, Ukraine
{sofiyagarkot, mshamrai, sip, maryhirna}@macpaw.com

ABSTRACT

The performance and generalization of foundation models for interactive systems critically depend on the availability of large-scale, realistic training data. While recent advances in large language models (LLMs) have improved GUI understanding, progress in desktop automation remains constrained by the scarcity of high-quality, publicly available desktop interaction data, particularly for macOS. We introduce GUIRILLA[1], a scalable *data crawling framework* for automated exploration of desktop GUIs. GUIRILLA is not an autonomous agent; instead, it systematically collects realistic interaction traces and accessibility metadata intended to support the training, evaluation, and stabilization of downstream foundation models and GUI agents. The framework targets macOS, a largely underrepresented platform in existing resources, and organizes explored interfaces into hierarchical *MacApp Trees* derived from accessibility states and user actions. As part of this work, we release these MacApp Trees as a reusable structural representation of macOS applications, enabling downstream analysis, retrieval, testing, and future agent training. We additionally release macapptree[2], an open-source library for reproducible accessibility-driven GUI data collection, along with the full framework implementation to support open research in desktop autonomy.

1 INTRODUCTION

Understanding user interfaces (UI) through machine learning has emerged as a critical challenge in human–computer interaction. Recent advances in large language models (LLMs) have driven rapid progress in multimodal systems that interact with graphical user interfaces, enabling applications ranging from UI automation and assistive technologies to software testing and interactive agents (Kapoor et al., 2024; Qin et al., 2025; Cheng et al., 2024; Pawlowski et al., 2024). While training models to navigate mobile UIs has been extensively studied (Lee et al., 2024) thanks to the availability of large-scale datasets: RICO(Deka et al., 2017), AITW(Rawles et al., 2023), AutoDroid(Wen et al., 2023), progress in desktop environments remains constrained. Unlike mobile, desktop environments are cluttered and dynamic: small icon-based controls often encode critical meaning for task execution. Moreover, often users face overlapping windows, popups, dialogs, and system widgets. Among others, the macOS GUI presents particular challenges due to different coordinate systems and UI standards compared to other operating systems. As a result, existing multimodal benchmarks expose three structural flaws that currently limit the construction of scalable, reusable datasets and interaction representations for desktop UIs:

1. **Manual annotation does not scale.** Recent benchmarks (Xie et al., 2024; Kapoor et al., 2024; Li et al., 2025) rely on human-designed pipelines where tasks, interactions, or UI elements must be manually demonstrated, recorded, and verified. While such supervision is valuable, it is costly and difficult to scale, limiting dataset coverage across applications, UI states, and interaction flows. This bottleneck affects not only agent training, but also downstream applications such as automated UI testing, retrieval, and software analysis.

[1] https://github.com/MacPaw/GUIrilla
[2] https://github.com/MacPaw/macapptree/

2. **Single-window UIs misrepresent real usage.** Most existing corpora capture clean snapshots of a *single* application window, whereas real desktop usage involves overlapping windows, modal dialogs, background applications, and system widgets. This discrepancy reduces the ecological validity of collected data and limits its applicability for tasks such as automation, testing, trajectory modeling, and long-horizon interaction reasoning across applications.

3. **Automated collection requires platform-specific design.** Constructing diverse, high-quality desktop UI datasets requires OS-specific expertise to handle heterogeneous UI toolkits, event models, and permission systems. Effective automation further demands tailored engineering to reliably parse and interact with each platform. For example, macOS lacks robust virtualization support, significantly complicating large-scale automated crawling compared to Android. As a result, macOS remains severely underrepresented in existing datasets: macOS interfaces comprise only 0.06% of all samples in OS-ATLAS (Wu et al., 2024), and approximately 2.45% among automatically collected desktop UIs overall.

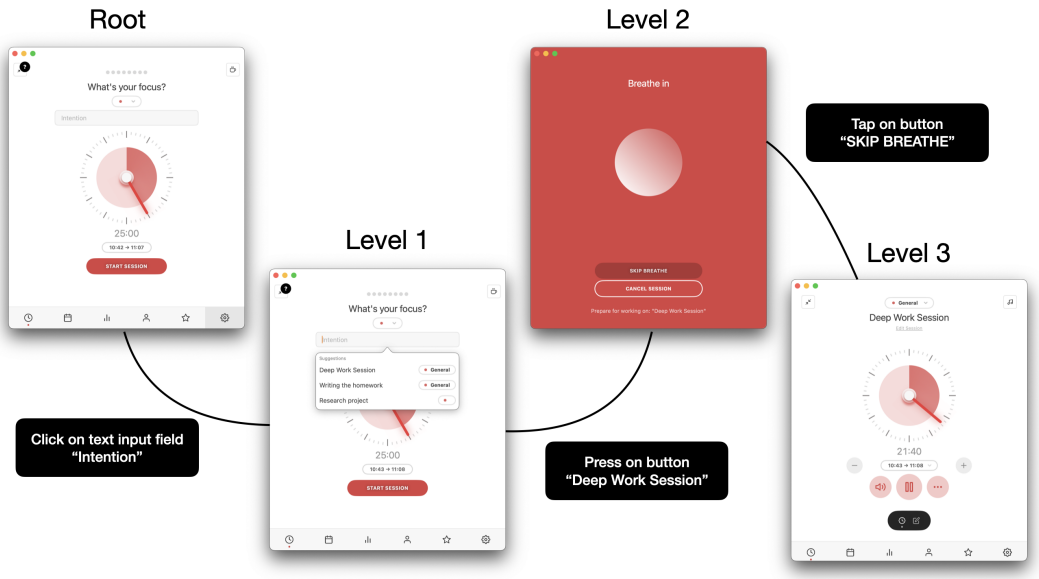


Figure 1: Parsed hierarchical tree structure from the Session application. Each node represents a UI state, containing the full accessibility tree along with a screenshot of the interface, and the edges denote GUIRILLA crawler actions. The hierarchy reflects a sequence of interactions as the agent interacts with application UI, forming the application-specific tree.

Beyond training computer-use agents, large-scale desktop UI interaction data enables a broad range of applications, including automated GUI testing and regression analysis GUI ripping(Memon et al., 2003), DinoDroid(Zhao et al., 2022), software quality assurance and bug reproduction ReCDroid(Zhao et al., 2019), UI retrieval and search GUing (Wei et al., 2025), and trajectory-based reasoning or retrieval-augmented agents (Zhang et al., 2025b; Wen et al., 2023). Structured interaction representations have also been explored in early GUI ripping and event-flow graph methods for automated testing, highlighting the long-standing value of modeling UI behavior as graphs rather than isolated screenshots.

To address these gaps, we introduce GUIRILLA, a fully automated framework that explores macOS GUIs at scale and summarizes them in a hierarchical *application tree* format (see Figure 1). Built on macOS' accessibility API, the GUIRILLA crawler systematically explores applications through simulated user interactions. Optionally, it can be assisted by LLM–based components for safer element ordering, context-aware input generation, and obstacle handling; however, GUIRILLA remains a *data crawler* whose output is intended to train and stabilize downstream models and agents rather than compete with them.

In this work we make the following key contributions:

- **GUIRILLA framework.** The first open-source, automated framework tailored for macOS that constructs detailed full-desktop application trees from Accessibility API snapshots and generates function-centric tasks. Application exploration utilizes specialized interaction handlers and can operate both deterministically
- **GUIrilla MacApp Trees.** We release a large-scale collection of *MacApp Trees*—hierarchical, accessibility-driven representations of macOS applications that capture UI states and user actions across full-desktop environments. These trees provide a reusable structural abstraction of desktop GUI behavior and can be independently used for UI analysis, retrieval, testing, and future agent training.
- **GUIRILLA-TASK dataset.** A macOS, full-desktop corpus of 27,171 tasks across 1,108 applications and 4.2K unique screens. We also release GUIRILLA-GOLD (1,283 human-verified tasks) with a 90.26% human baseline.
- **GUIRILLA-SEE vision–language models.** We release three models: GUIRILLA-SEE (0.7B), GUIRILLA-SEE (3B), and GUIRILLA-SEE (7B), trained on just 4.2K unique macOS full-desktop screens from GUIRILLA-TASK. The models serve as a concrete demonstration that realistic macOS coverage and function-level supervision can yield strong transfer to downstream GUI understanding benchmarks with substantially less training data than large multi-OS synthetic corpora.
- **Open-source reproducible toolkit.** Complete end-to-end implementation including data generation pipeline, model training code, evaluation framework, and the macapptree library for collecting accessibility metadata and screenshots, facilitating reproducible automated data collection efforts on macOS.

2 RELATED WORK

UI understanding has seen rapid progress in *mobile* (Rawles et al., 2023; 2024) and *web* (Liu et al., 2024b;a) domains, driven largely by the availability of structured interface representations (e.g., HTML/XML) and large-scale datasets such as RICO. These resources enable models to jointly reason about visual appearance, textual content, and interaction affordances at scale. In contrast, desktop environments lack a unified structural representation, exhibit greater visual and behavioral heterogeneity, and often require system-level permissions for interaction and inspection. These properties make scalable, automated data collection substantially more challenging.

Several recent efforts have focused on constructing datasets and benchmarks for desktop UI understanding. ScreenSpot Cheng et al. (2024) and its extensions ScreenSpot-v2 and ScreenSpot-Pro Li et al. (2025) introduce task-oriented datasets for evaluating element localization and instruction following on desktop screenshots. Related benchmarks such as WinClick (Hui et al., 2025), UI-Vision (Nayak et al., 2025), GUI-360° (Mu et al., 2025), and MMBench-GUI (Wang et al., 2025b) further expand evaluation coverage across platforms, resolutions, and task granularities. While these benchmarks have advanced evaluation methodology, they typically rely on manually curated task definitions or limited interaction traces, and do not expose the underlying processes used to explore applications or construct interaction trajectories.

Beyond benchmarking, several works explore automated data collection for desktop GUIs. Omni-ACT (Kapoor et al., 2024) presents a manually collected multi-platform dataset spanning macOS, Linux, and Windows, but is limited in scale and application coverage. OS-Atlas (Wu et al., 2024) automates macOS data collection via the Accessibility API, producing large numbers of single-step question–answer pairs. However, its exploration strategies are shallow, rely on raw accessibility labels, and—crucially—the end-to-end crawler implementation is not publicly released, limiting reproducibility and reuse. More recent pipelines such as pyautogui, UI-E2I-Synth (Liu et al., 2025), and OS-Genesis (Sun et al., 2025) leverage LLMs to annotate UI elements or synthesize action trajectories at scale, but focus primarily on generating supervision for grounding or agent training rather than releasing reusable structural representations of application behavior.

A substantial body of work targets GUI grounding and parsing, including OmniParser (Lu et al., 2024), HyperClick (Zhang et al., 2025a), LASER (Wang et al., 2025a), Aria-UI (Yang et al., 2025), and related methods that improve element localization, uncertainty estimation, or active perception. These approaches primarily study model architectures, training strategies, or supervision signals for

grounding benchmarks. While complementary to our work, grounding-focused methods typically assume the availability of curated datasets and do not address the upstream problem of scalable, reproducible desktop UI exploration and representation.

Modeling UI behavior as structured graphs or state-transition systems has a long history in software engineering and automated testing. Early GUI ripping and event-flow graph methods (Memon et al., 2003) aimed to reverse-engineer application behavior to generate test cases. Subsequent work on Android UI testing and QA—including (Yoon et al., 2023), ReCDroid, DinoDroid, and model-based testing frameworks—demonstrated the value of automated exploration for software reliability and regression analysis. More recently, trajectory-based reasoning and retrieval methods have been explored for agentic systems and retrieval-augmented interaction modeling. These lines of work highlight the importance of capturing *interaction structure*, not just isolated screenshots or annotations.

In contrast to prior efforts, GUIRILLA focuses on the scalable, automated construction of *MacApp Trees*—hierarchical, accessibility-driven representations that capture UI states and user actions across full-desktop macOS environments. Our framework emphasizes reproducible exploration, safe interaction handling, and the release of reusable structural artifacts rather than solely task annotations or benchmark scores. The resulting trees and derived datasets can support a broad range of downstream applications, including UI automation, testing, retrieval, accessibility analysis, and future agent training.

3 METHODOLOGY

GUIRILLA introduces a *tree-centric, fullscreen* exploration pipeline for macOS GUIs. Our framework builds on macOS Accessibility API[3], while integrating three specialized agents that interpret accessibility metadata, prioritize interface elements, and generate contextually appropriate actions.

3.1 MACAPPTREE

To navigate between interface states, GUIRILLA leverages the macOS Accessibility API. However, interfacing with this API directly via Python is often cumbersome. To solve this, we developed and open-sourced *macapptree*: a Python package designed to extract an application's accessibility tree and serialize it into a clean, hierarchical JSON format.

As demonstrated in Table 6, Visual Language Models (VLMs) often struggle with precision on UI benchmarks, leading to navigation errors. By utilizing text-based accessibility trees instead of raw screenshots, our approach allows the use of standard Large Language Models (LLMs). This methodology ensures that as long as the application's accessibility metadata is accurate, the agent interacts only with explicitly defined UI elements — guaranteeing that the selected coordinates for actions like clicking are always precise.

Beyond its role in our pipeline, *macapptree* could serve in other use cases:

1. **Accessibility Testing:** Validating whether UI elements are correctly exposed to assistive technologies.
2. **UI Automation:** Providing a reliable, non-visual interface for cross-app interactions.
3. **Visual Debugging:** Allowing developers to inspect exactly how the system interprets the screen.

This versatility makes the tool valuable for both academic research and practical software engineering workflows.

3.2 CRAWLER

The single-app processing pipeline is illustrated in Figure 2 and has the following stages. First, the input bundle undergoes a standard installation routine, and together with user-specified set of

[3] `https://developer.apple.com/documentation/accessibility/accessibility-api`

parameters (such as maximal desired tree depth, and duration of parsing, the full list is available in Appendix A.2), crawler manages each of the windows of the installed app. Upon installation, the crawler attempts to extract an application's accessibility tree according to macOS accessibility framework. This framework enables simpler interaction with UI elements on the screen grouping them into a hierarchical tree structure, where each element contains essential properties such as name, role, description, position, and size. However, application developers must manually annotate or update accessibility metadata. This manual annotation process often results in error-prone accessibility trees with significant limitations: some trees contain UI elements that remain in the tree after disappearing from view, others include components with incorrect role classifications, and inaccurate positioning information.

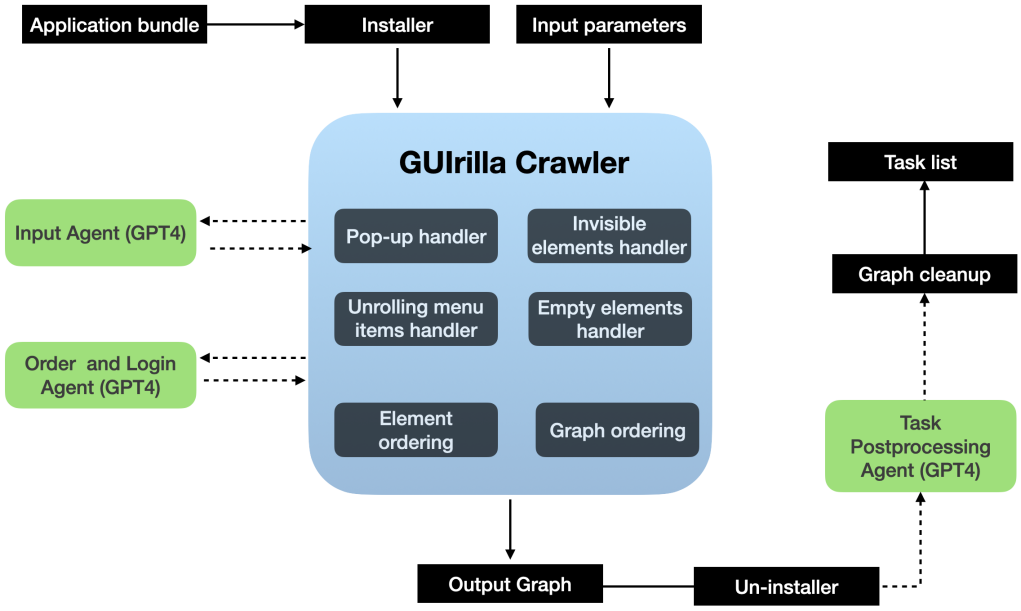


Figure 2: Architecture of the GUIRILLA framework. The GUIRILLA crawler, equipped with various UI handlers, processes an application bundle using input parameters and installer routines. It interacts with autonomous GPT-4 agents (Input, Order, and Login Agents) to navigate the application. The resulting output tree is refined by a Task Postprocessing Agent (GPT-4), which handles uninstallation and tree cleanup, ultimately producing a structured task list. The dashed line denotes the optional usage of LLMs for app exploration.

To handle these edge cases, our GUIRILLA crawler incorporates multiple specialized handlers, as shown in Figure 2. Within the crawler's core (highlighted in blue), multiple handlers address the common parsing challenges: Pop-up handler manages transient modal content, Invisible elements handler filters off-screen components present in accessibility, Unrolling menu items handler processes dynamically generated navigation elements, and Empty elements handler resolves placeholder elements with missing metadata. This multi-handler approach enables robust extraction of actionable interface information despite the underlying data quality issues.

The GUIRILLA crawler performs three types of interactions to explore an application: click, cursor move, type, and press Enter key using *pyautogui* (Sweigart, 2015) library. To enable meaningful interaction with applications, the parsing is supported by three GPT4-based agents (the prompts are available in Appendix A.3):

1. The *Input Agent*: This agent generates contextually appropriate input strings based on the accessibility tree, ensuring relevant text is entered into form fields and search boxes.

2. The *Order and Login Agent*: Given a hierarchical list of on-screen elements, the agent determines an safest interaction sequence starting with elements that cause minimal UI changes and progressing to those with potentially significant effects (e.g., "Delete" buttons). Login pages are treated as a

special case, requiring human input. This agent enhances the security and safety of the exploration process by avoiding random or destructive actions.

3. The *Task Agents*: After the uninstallation phase, these agents refine the resulting output tree, cleaning up duplicates, and transforming the structured data into a readable list of natural language tasks. Their inclusion enables both refinement and generation of more complex and natural language task descriptions.

While our framework leverages GPT-based agents to enable robust and secure interaction, both the application trees and task data can also be collected deterministically without GPT-4 requests by following a fixed element processing order and using default input string values. However, incorporating GPT-based reasoning significantly improves the safety and contextual relevance of interactions. A detailed comparison between deterministic and GPT-guided exploration is provided in Appendix A.8.2.

3.3 TREE STRUCTURE

The application tree collected with GUIRILLA crawler consists of nodes and edges that represent application states and actions, respectively (see Figure 1). All interaction trees are automatically annotated and visualized as accompanying SVG files. Across applications, the trees have an average depth of 3.5, with the deepest tree reaching a depth of 101. Each node corresponds to a specific UI state of an application and contains the following fields:

- *Element*: The accessibility tree of the application window at a state.
- *Image name*: The filename of the full desktop screenshot associated with a state.
- *Actions*: A list of actions that can be executed without causing significant changes to the UI. We define a significant change as the addition or removal of more than 10 UI elements following an interaction.

Each edge captures a possible interaction and includes:

- *Action*: Information about the UI element that triggered the interaction, along with a human-readable action description and a structured dictionary representation that has a 1-to-1 map to *pyautogui* commands.
- *Out vertex*: The resulting UI state after the interaction of the crawler with the GUI.

We release all raw trees as open-source artifacts for broader reuse across research and engineering tasks which includes trees and screenshots for different applications. In total, the dataset comprises 561 GB of compressed data. The examples of tree are provided in Figure 3.

4 RESULTS

4.1 DERIVED TASK DATASET FROM MACAPP TREES

As an application of the collected MacApp Trees, we derive a task-oriented dataset, GUIRILLA-TASK, by extracting function-centric interactions from tree edges. We deployed the GUIRILLA crawler on 12,298 macOS applications using an open Mac App Store dataset (Sergii Kryvoblotskyi, 2025). Out of these, 1,108 applications supported the macOS Accessibility framework and yielded valid interaction trees.

The resulting task dataset spans a wide range of domains, including productivity, creative tools, system utilities, and developer software, ensuring diverse coverage of real-world desktop UI paradigms. In total, GUIRILLA-TASK contains 27,171 tasks across 23 application genres, derived from approximately 4.2K unique full-desktop screens.

Each task corresponds to a concrete user interaction (mouse click or keyboard input) paired with a full-desktop screenshot and the associated accessibility tree state. Tasks range from simple navigational actions (e.g., “open settings”) to higher-level functional instructions (e.g., “change your working hours to end at 18:00”). Tasks are annotated with task type (e.g., navigation, settings) and target element category (e.g., button, menu, input field).

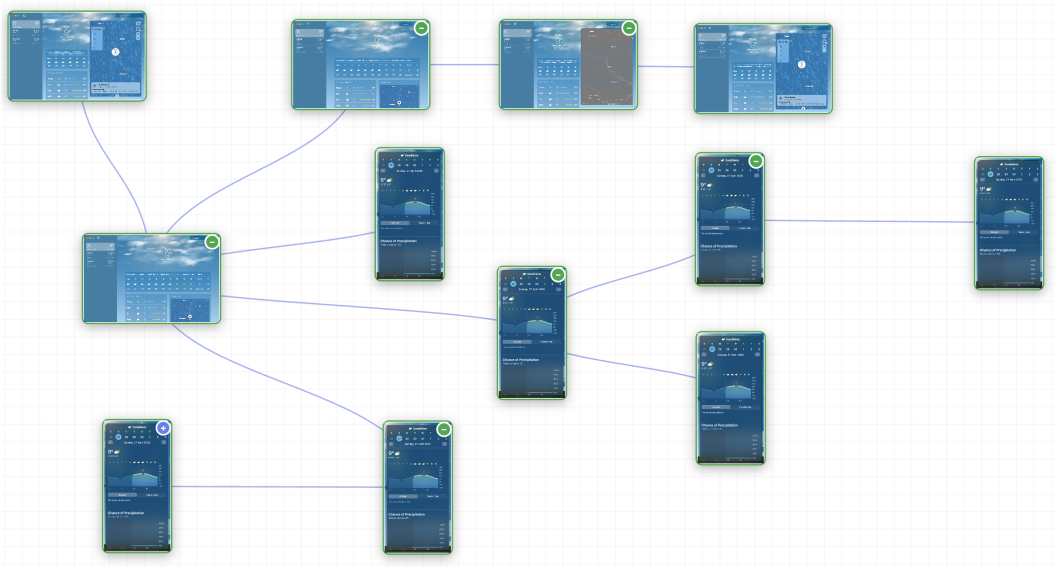


Figure 3: Example of trees based on GUIRILLA crawler

Additional dataset statistics, representative samples, and entry attributes are provided in Appendix A.1. Data collection was performed on a cluster of four M1 Mac Mini machines (16 GB RAM) running macOS 14.7.5, as well as two MacBook Pros, with multiple parallel exploration environments per host.

4.2 DOWNSTREAM AGENT EVALUATION

Models. To illustrate the utility of GUIRILLA-derived data for downstream applications, we evaluate a range of vision–language models (VLMs) on tasks extracted from MacApp Trees. These include proprietary systems like OpenAI Computer Use (OpenAI, 2025) and Claude Computer Use (Anthropic, 2024) as well as open-source models: UI-TARS 1.5 (7B), UI-TARS (2B) (Qin et al., 2025), Qwen 2.5 VL (7B, 3B) (Bai et al., 2025), CogAgent 9B (Hong et al., 2024), and OS-Atlas Pro 7B (Wu et al., 2024).

Metrics. We report task success rates based on action accuracy. For click tasks, success requires the predicted coordinates to fall within the target element's bounding box. Input tasks additionally require exact text matches.

Results. Without task-specific fine-tuning, models struggle particularly with text-input interactions, reflecting the complexity of long-horizon desktop behavior. Proprietary systems achieve the strongest performance overall, while open-source models benefit substantially from training on realistic macOS interaction data. Full quantitative results are reported in Appendix A.6.3.

4.3 ABLATION STUDY

Impact of Accessibility Handlers on Exploration Coverage. Native accessibility annotations vary inconsistently across applications, creating barriers to systematic exploration. The accessibility handlers anticipate UI changes and execute meaningful interactions beyond basic clicking. Testing on three macOS applications (Stocks, Maps, Weather) across tree depth, duplicate rate, task diversity, and process time shows handlers increase task discovery by 5× in Stocks and 3× in Maps while reducing duplicates and processing time (Appendix A.8.1). These handlers target platform-agnostic problems: inconsistent element labeling, hidden components, and dynamic content. The logic transfers to other operating systems facing similar accessibility inconsistencies.

Generative Task Agents. We compare two training approaches: (1) deterministic accessibility metadata (`name`, `role`, `role_description`, `value`), and (2) GPT-4 task descriptions from screenshots and element crops (Table 10). Accessibility metadata often reduces to generic labels like "button" or "text" without capturing functional intent. In contrast, GPT-generated descriptions understand visual context and explicit purpose. When UI screens contain similar elements, accessibility labels create ambiguous supervision signals that hinder target identification. Florence-0.7B achieved 53.55% accuracy on GPT-generated tasks versus 40.35% on accessibility-based tasks—a 13-point gap demonstrating that functional supervision outperforms surface-level properties for training UI agents.

5 Impact, Limitations, and Ethics

Broader Impact. This work has significant potential to advance accessibility technology development, directly benefiting users with disabilities who rely on assistive technologies. By systematically collecting UI interaction data, our framework can improve screen readers, voice-controlled interfaces, and other adaptive technologies that help users navigate complex desktop environments.

Technical Limitations. Our approach is primarily constrained by dependence on developer-provided accessibility metadata, which exhibits considerable variation in quality across applications. While currently implemented for macOS, the methodology can be adapted to other platforms such as Windows[4], Linux[5], and Android[6] by leveraging their existing accessibility infrastructures, though this requires platform-specific engineering. Additionally, solutions like OmniParser or Screen2AX (Muryn et al., 2025) can be used to remove full reliance on accessibility metadata.

5.1 Ethical Considerations

We acknowledge potential risks including privacy violations, security circumvention, and malicious automation. To mitigate these concerns, we implement technical safeguards:

- **Sandboxed Environments**: We strongly recommend conducting data collection in dedicated environments with anonymized profiles to prevent accidental data leakage
- **Local-Only Operation**: All collection, replay, and annotation occur entirely locally without requiring data transmission to third parties
- **Deterministic Handlers**: Rule-based handlers enable fully offline, privacy-preserving automation without external API dependencies
- **Limited API Access**: Framework operates strictly via public macOS Accessibility APIs with no privileged system calls
- **Security-Critical Exclusion**: We explicitly avoid interaction with authentication, payment, or CAPTCHA-related interfaces

Responsible Use Guidelines. We explicitly discourage malicious use through clear documentation and recommend: (1) running crawlers only in controlled environments with synthetic inputs, (2) applying data filtering to remove sensitive content, and (3) using deterministic handlers for regulated data. Acceptable use cases include academic Human-Computer-Interaction research, accessibility technology development, and educational applications in controlled environments. Prohibited uses include automation of financial/healthcare systems, security circumvention, unauthorized personal data collection, and creation of tools for harassment or illegal activities. We remain committed to community oversight and transparent release practices, maintaining openness to policy revisions based on feedback to ensure responsible deployment of UI automation capabilities.

Use of Large Language Models. Portions of this manuscript were refined with the assistance of large language models (LLMs) for grammar and style.

[4] `https://learn.microsoft.com/en-us/dotnet/framework/ui-automation/`

[5] `https://gnome.pages.gitlab.gnome.org/at-spi2-core/libatspi/`

[6] `https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo`

6 Conclusions and Future Directions

We introduce GUIrilla, a fully automated framework that addresses the data scarcity challenge in desktop UI research by enabling scalable, reproducible exploration of macOS applications. By leveraging accessibility-driven crawling and structured interaction handling, GUIrilla constructs hierarchical MacApp Trees that capture UI states and user actions across full-desktop, multi-application environments.

Unlike prior work that emphasizes task annotation or benchmark performance, our contribution centers on releasing reusable structural representations of desktop UI behavior, along with an open-source crawling framework and tooling. These artifacts support a wide range of downstream applications, including UI automation, software testing, accessibility analysis, retrieval, and agent training, and can be reused independently of any specific model architecture.

Future Work. While GUIrilla currently leverages native accessibility APIs on macOS, future work could integrate image-to-accessibility or vision-based UI parsing techniques to enable crawling in environments where accessibility metadata is incomplete or unavailable. Another promising direction is the development of local vision–language model (VLM) agents that actively explore applications using reinforcement learning or related approaches, allowing more adaptive exploration strategies and improved coverage of complex interaction patterns. Beyond macOS, the framework can be extended to additional operating systems by adapting platform-specific accessibility interfaces, and its continuous crawling pipeline enables long-term data collection as applications evolve. We hope this work lays the foundation for future research on structured desktop UI representations and serves as an open, extensible resource for the community to build upon as interactive systems continue to advance.

Acknowledgments

We thank the Armed Forces of Ukraine for providing security to complete this work. We thank Yaroslav Tereshchenko and Victor Muryn for their support, insightful discussions, and assistance in framing the paper and releasing the dataset as open source.

References

Anthropic. Introducing computer use, a new claude 3.5 sonnet, and claude 3.5 haiku. https://docs.anthropic.com/en/docs/agents-and-tools/computer-use, October 2024. Accessed: 2025-05-14.

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, et al. Qwen2. 5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025.

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. doi: 10.48550/ARXIV.2401.10935. URL https://arxiv.org/abs/2401.10935.

Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pp. 845–854, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349819. doi: 10.1145/3126594.3126651. URL https://doi.org/10.1145/3126594.3126651.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a Generalist Agent for the Web. 2023. doi: 10.48550/ARXIV.2306.06070. URL https://arxiv.org/abs/2306.06070.

Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. Navigating the digital world as humans do: Universal visual grounding for gui agents, 2025. URL https://arxiv.org/abs/2410.05243.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxuan Zhang, Juanzi Li, Bin Xu, Yuxiao Dong, Ming Ding, and Jie Tang. Cogagent: A visual language model for gui agents, 2024. URL https://arxiv.org/abs/2312.08914.

Zheng Hui, Yinheng Li, Dan zhao, Tianyi Chen, Colby Banbury, and Kazuhito Koishida. Winclick: Gui grounding with multimodal large language models, 2025. URL https://arxiv.org/abs/2503.04730.

Raghav Kapoor, Yash Parag Butala, Melisa Russak, Jing Yu Koh, Kiran Kamble, Waseem Alshikh, and Ruslan Salakhutdinov. Omniact: A dataset and benchmark for enabling multimodal generalist autonomous agents for desktop and web, 2024. URL https://arxiv.org/abs/2402.17553.

Sunjae Lee, Junyoung Choi, Jungjae Lee, Munim Hasan Wasi, Hojun Choi, Steven Y. Ko, Sangeun Oh, and Insik Shin. Explore, select, derive, and recall: Augmenting llm with human-like memory for mobile task automation, 2024. URL https://arxiv.org/abs/2312.03003.

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. doi: 10.48550/ARXIV.2504.07981. URL https://arxiv.org/abs/2504.07981.

Junpeng Liu, Tianyue Ou, Yifan Song, Yuxiao Qu, Wai Lam, Chenyan Xiong, Wenhu Chen, Graham Neubig, and Xiang Yue. Harnessing webpage uis for text-rich visual understanding, 2024a. URL https://arxiv.org/abs/2410.13824.

Junpeng Liu, Yifan Song, Bill Yuchen Lin, Wai Lam, Graham Neubig, Yuanzhi Li, and Xiang Yue. Visualwebbench: How far have multimodal llms evolved in web page understanding and grounding?, 2024b. URL https://arxiv.org/abs/2404.05955.

Xinyi Liu, Xiaoyi Zhang, Ziyun Zhang, and Yan Lu. Ui-e2i-synth: Advancing gui grounding with large-scale instruction synthesis, 2025. URL https://arxiv.org/abs/2504.11257.

Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. Omniparser for pure vision based gui agent, 2024. URL https://arxiv.org/abs/2408.00203.

A. Memon, I. Banerjee, and A. Nagarajan. Gui ripping: reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, pp. 260–269. IEEE, 2003. doi: 10.1109/WCRE.2003.1287256.

Jian Mu, Chaoyun Zhang, Chiming Ni, Lu Wang, Bo Qiao, Kartik Mathur, Qianhui Wu, Yuhang Xie, Xiaojun Ma, Mengyu Zhou, Si Qin, Liqun Li, Yu Kang, Minghua Ma, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Gui-360°: A comprehensive dataset and benchmark for computer-using agents, 2025. URL https://arxiv.org/abs/2511.04307.

Viktor Muryn, Marta Sumyk, Mariya Hirna, Sofiya Garkot, and Maksym Shamrai. Screen2ax: Vision-based approach for automatic macos accessibility generation. *arXiv preprint arXiv:2507.16704*, 2025.

Shravan Nayak, Xiangru Jian, Kevin Qinghong Lin, Juan A. Rodriguez, Montek Kalsi, Rabiul Awal, Nicolas Chapados, M. Tamer Özsu, Aishwarya Agrawal, David Vazquez, Christopher Pal, Perouz Taslakian, Spandana Gella, and Sai Rajeswar. Ui-vision: A desktop-centric gui benchmark for visual perception and interaction, 2025. URL https://arxiv.org/abs/2503.15661.

OpenAI. Computer-using agent, January 2025. URL https://openai.com/index/computer-using-agent/. Accessed: 2025-05-15.

Pawel Pawlowski, Krystian Zawistowski, Wojciech Lapacz, Marcin Skorupa, Adam Wiacek, Sebastien Postansque, and Jakub Hoscilowicz. Tinyclick: Single-turn agent for empowering gui automation, 2024. URL https://arxiv.org/abs/2410.11871.

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.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Android in the Wild: A Large-Scale Dataset for Android Device Control. 2023. doi: 10.48550/ARXIV.2307.10088. URL https://arxiv.org/abs/2307.10088.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. Androidworld: A Dynamic Benchmarking Environment for Autonomous Agents. 2024. doi: 10.48550/ARXIV.2405.14573. URL https://arxiv.org/abs/2405.14573.

Nataliia Stulova Sergii Kryvoblotskyi. Collecting a Dataset of macOS Apps: Pains, Gains, Lessons Learned. Part 1, May 2025. URL https://research.macpaw.com/publications/macos-app-dataset. [Online; accessed 16. May 2025].

Qiushi Sun, Kanzhi Cheng, Zichen Ding, Chuanyang Jin, Yian Wang, Fangzhi Xu, Zhenyu Wu, Chengyou Jia, Liheng Chen, Zhoumianze Liu, Ben Kao, Guohao Li, Junxian He, Yu Qiao, and Zhiyong Wu. Os-genesis: Automating gui agent trajectory construction via reverse task synthesis, 2025. URL https://arxiv.org/abs/2412.19723.

Al Sweigart. Pyautogui: A cross-platform gui automation python module for human beings. https://github.com/asweigart/pyautogui, 2015. Accessed: 2025-05-15.

Wanfu Wang, Qipeng Huang, Guangquan Xue, Xiaobo Liang, and Juntao Li. Learning active perception via self-evolving preference optimization for gui grounding, 2025a. URL https://arxiv.org/abs/2509.04243.

Xuehui Wang, Zhenyu Wu, JingJing Xie, Zichen Ding, Bowen Yang, Zehao Li, Zhaoyang Liu, Qingyun Li, Xuan Dong, Zhe Chen, Weiyun Wang, Xiangyu Zhao, Jixuan Chen, Haodong Duan, Tianbao Xie, Chenyu Yang, Shiqian Su, Yue Yu, Yuan Huang, Yiqian Liu, Xiao Zhang, Yanting Zhang, Xiangyu Yue, Weijie Su, Xizhou Zhu, Wei Shen, Jifeng Dai, and Wenhai Wang. Mmbench-gui: Hierarchical multi-platform evaluation framework for gui agents, 2025b. URL https://arxiv.org/abs/2507.19478.

Jialiang Wei, Anne-Lise Courbis, Thomas Lambolais, Binbin Xu, Pierre Louis Bernard, Gérard Dray, and Walid Maalej. Guing: A mobile gui search engine using a vision-language model. *ACM Transactions on Software Engineering and Methodology*, 34(4):1–30, April 2025. ISSN 1557-7392. doi: 10.1145/3702993. URL http://dx.doi.org/10.1145/3702993.

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. *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pp. 543–557, 2023. doi: 10.48550/ARXIV.2308.15272. URL https://arxiv.org/abs/2308.15272.

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.

Bin Xiao, Haiping Wu, Weijian Xu, Xiyang Dai, Houdong Hu, Yumao Lu, Michael Zeng, Ce Liu, and Lu Yuan. Florence-2: Advancing a unified representation for a variety of vision tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4818–4829, 2024.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio

Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024. URL https://arxiv.org/abs/2404.07972.

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.

Juyeon Yoon, Robert Feldt, and Shin Yoo. Autonomous large language model agents enabling intent-driven mobile gui testing, 2023. URL https://arxiv.org/abs/2311.08649.

Shaojie Zhang, Pei Fu, Ruoceng Zhang, Jiahui Yang, Anan Du, Xiuwen Xi, Shaokang Wang, Ying Huang, Bin Qin, Zhenbo Luo, and Jian Luan. Hyperclick: Advancing reliable gui grounding via uncertainty calibration, 2025a. URL https://arxiv.org/abs/2510.27266.

Xuan Zhang, Ziyan Jiang, Rui Meng, Yifei Leng, Zhenbang Xiao, Zora Zhiruo Wang, Yanyi Shang, and Dehan Kong. Universal retrieval for multimodal trajectory modeling, 2025b. URL https://arxiv.org/abs/2506.22056.

Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G.J. Halfond. Recdroid: Automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 128–139, 2019. doi: 10.1109/ICSE.2019.00030.

Yu Zhao, Brent Harrison, and Tingting Yu. Dinodroid: Testing android apps using deep q-networks, 2022. URL https://arxiv.org/abs/2210.06307.

A Appendix

A.1 Dataset statistics

The collected tasks were split into train and test subsets, such that the applications in test did not appear in train, and test applications contained larger, more complicated accessibility trees. There are 881 applications with $25,606$ entries in train and 227 applications with $1,565$ task entries in test.

A.1.1 Representative sample from the dataset

Each sample in the dataset includes the following structured fields:

- **Screen ID**: Unique identifier for the UI screen.
- **App Name**: Bundle identifier of the application.
- **Task**: Natural language description of the agent's objective.
- **Raw Action**: Deterministic textual representation of the user action.
- **Action**: Structured action format, e.g., "`left click, (x, y)`".
- **Element Data**: JSON metadata of the target UI element extracted from the accessibility tree.
- **Scaling Factor**: Display scaling factor for the specific screen.
- **Original Task**: Boolean indicating whether the task was derived directly from the original interaction tree.
- **A11y Path**: Full accessibility tree before the action was taken.
- **Image**: Full-screen desktop screenshot, stored as a binary image.
- **Cropped Image**: Subregion of the full screenshot containing the target application (variable dimensions).
- **Segmented Image**: Screenshot of the application window with segmented UI regions.
- **Task Category**: One of 22 predefined task categories (e.g., *Search & Information*, *Files*).
- **Element Category**: One of 16 UI element types (e.g., *Slider*, *Button*).

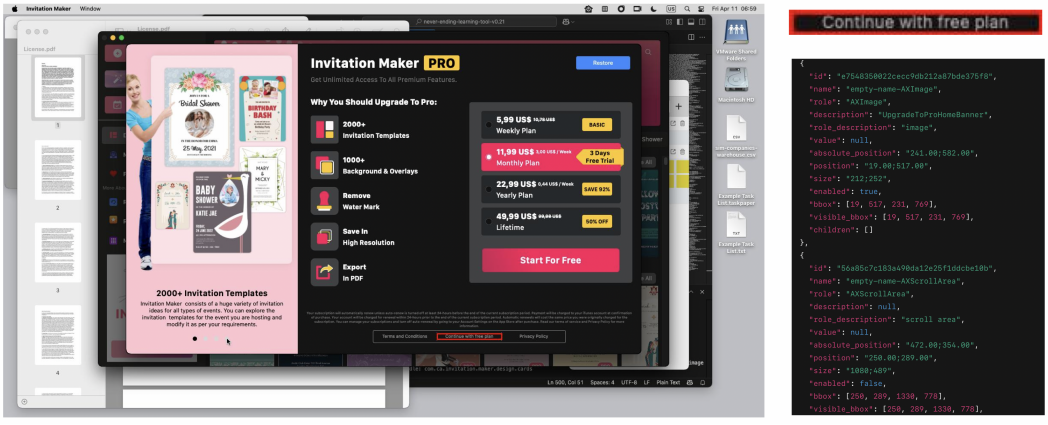


Figure 4: Example sample from our dataset. Left: screenshot of the macOS desktop interface. Upper right: target element cropped. Lower right: a segment of the accessibility tree.

A.1.2 COLLECTION STATISTICS

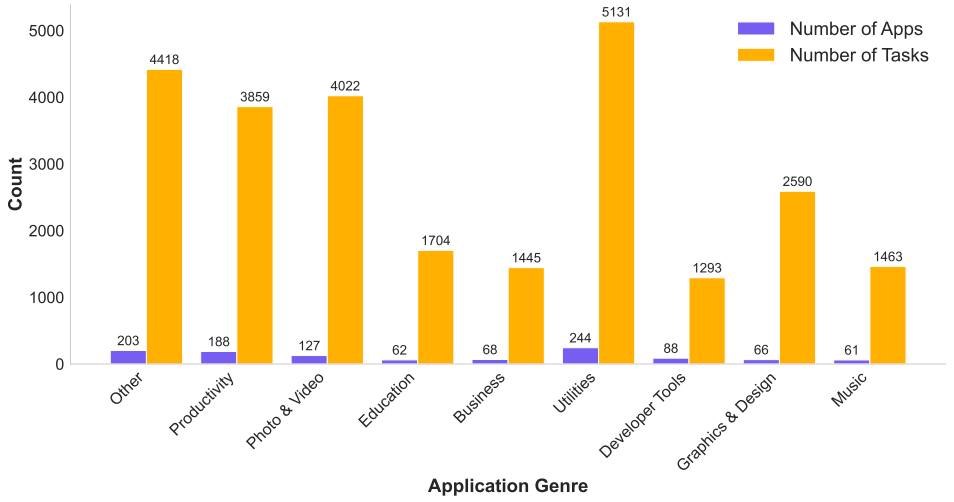
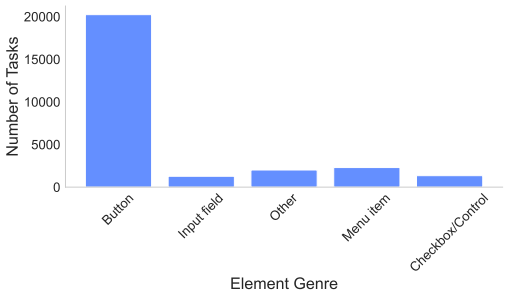
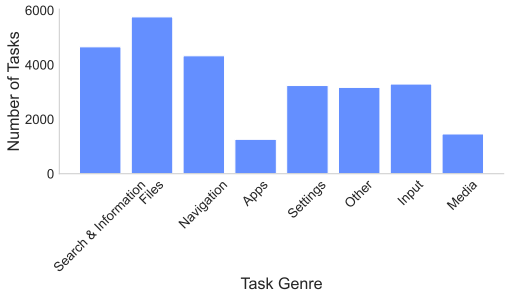


Figure 5: Number of apps and associated tasks per genre. For each genre, the left bar shows the number of apps, and the right bar shows the number of tasks. Colors distinguish between the two quantities. The figure highlights disparities between app availability and task density across categories.



(a) Distribution of tasks per element type. The most prevalent category is buttons.



(b) Distribution of tasks per task type.

Figure 6: Distributions of tasks across element types (left) and task types (right).

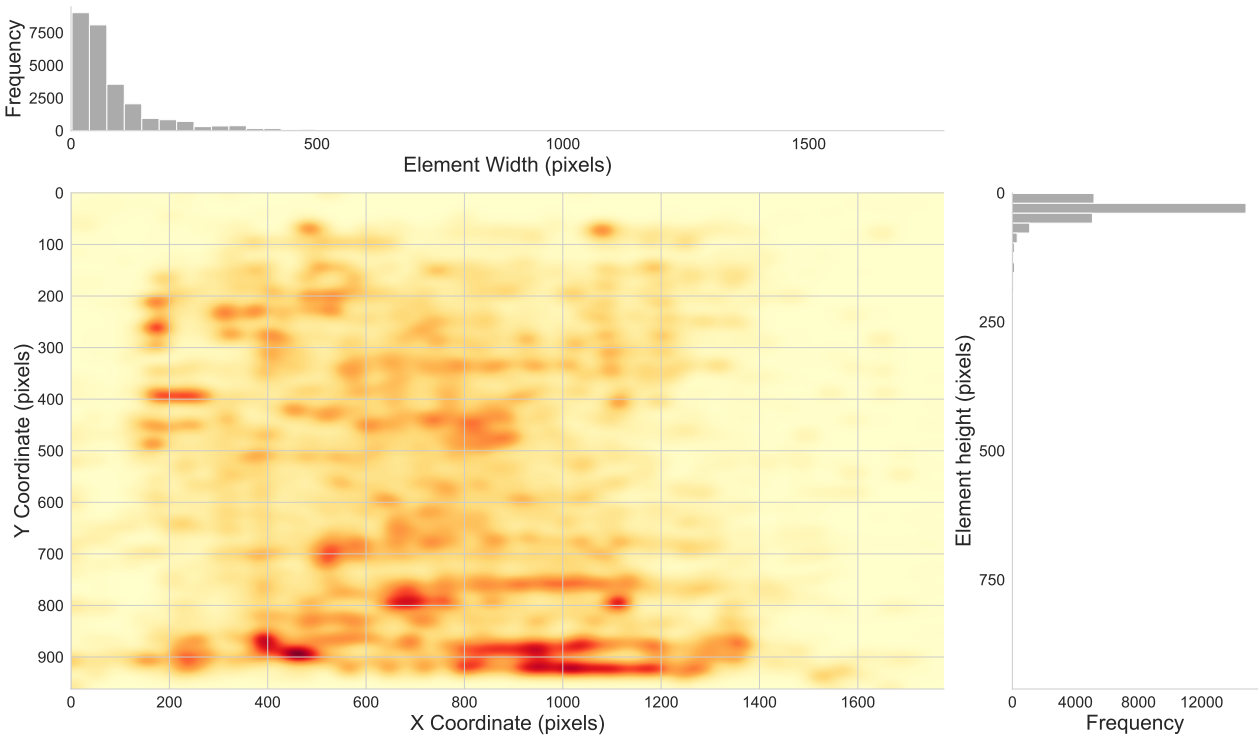


Figure 7: Top-left: Distribution of target element widths. Bottom-left: Distribution of target element center locations, showing that most target elements are positioned near the bottom of the desktop interface. Bottom-right: Distribution of target element heights.

Many interaction targets are located in peripheral regions (e.g., toolbars, corners), and a large proportion are visually small, with limited surface area (Figure 7).

A.2 PARAMETERS

Parameter	**Description**
Maximum parsing duration	Specified in minutes, default is 2 hours
Deterministic text input	Default string is `'DEFAULT'`
Maximum parsing tree depth	Default is 25
Cursor move before click	Defaults to `False`
Agent usage	Set to `True` by default. Can be enabled if an OpenAI API key is provided in a separate file
Task collection	Defaults to `True`. If set to `False`, trees can be collected without their associated tasks

Table 1: Configuration Parameters for the Crawler

A.3 PROMPTS

Input Agent Instructions

Analyze the given macOS application accessibility screen information and follow these steps:

1. Determine the type and purpose of the application based on the provided elements and descriptions.
2. Identify all `AXTextField` elements present in the structure.
3. For each `AXTextField`:
   (a) Infer its specific purpose within the application context.
   (b) Consider what a user would input in this field based on accessibility cues and typical behavior.
   (c) Generate an example input relevant to the field's likely function and the app's overall purpose.

Output: A JSON object where:

- **Keys** = integer IDs of the `AXTextField` elements
- **Values** = realistic example inputs, based on screen context

Only return the JSON object—no additional explanations.
Examples:

- `{7: "Yellow Submarine"}` // Music app search
- `{12: "John", 15: "Smith", 21: "07580198241"}` // Contacts app
- `{8: "main"}` // IDE project file search

Note: Ensure that inputs are app-appropriate and reflect common human interactions.

Order Agent Instructions

Given accessibility screen info, organize UI elements in logical interaction order. Consider irreversible actions and screen transitions.
Output: JSON with nested groups (max 8), each containing element IDs:

- Prioritize elements in popovers, content switches, and window controls.
- Derive element type from description if needed.
- Include **ALL element IDs** from input.

Grouping Rules:

- `dynamic_TYPE` — dynamic lists (emails, notes, etc.)
- `repeated_TYPE` — options where only one is needed (date, category, etc.)
- Avoid grouping unrelated or static UI items together.

Flags to include when relevant:

- `"login_page": true`
- `"system_access_required": true`

Example 1 — Complex App:

```
{
  "action_order": [
    {"menu_buttons": [1, 2, 3]},
    {"dynamic_emails": [4, 5, 6, 7]},
    {"repeated_time_selection": [8, ..., 31]},
    {"popover_buttons": [32, 33]}
  ],
```

```
  "login_page": false,
  "system_access_required": false
}
```

Example 2 — Login Page:

```
{
  "action_order": [
    {"login_elements": [1, 2, 3]},
    {"account_settings": [4]}
  ],
  "login_page": true,
  "system_access_required": false
}
```

Click Task Prompt

You are given a UI screenshot, an image of the clicked UI element. The clicked element is highlighted in red. Your task is to describe the action needed to click this element.
Guidelines:
0. If the element is not perfectly selected (ex. partially), the box is strangely located, or no human would do this task - return empty string.
1. The task must describe the function, not the appearance of the element. For example, prefer "Create a new document" over "Click the grey + button." Repeating the element's text is acceptable.
2. The task must be unique to this screen. For example, if there are two buttons labeled "Open," you must specify which "Open" button is meant.
3. The task must consider the app context, but not imagine extra information. For example, if the app is an image editor and the button is "Delete," the better task is "Delete an image", not just a generic "delete."
4. Use the fewest words possible without sacrificing clarity.
5. Write the task in straightforward English only.
6. Select a category for each task. Must be one of Navigation (go back), Settings (adjust volume), Files (save file), Apps (open edge), Search & Information (check weather), Media (play music), Accounts (sign in), Communication (share file), Input (enlarge font), Connectivity (connect wifi), Modes (dark mode), E-commerce (add to cart)
7. Select a category for each element. Must be one of Image, Text, Checkbox/Control, Menu item, Input field, Button, Group, Link.
Important notes:
The click is based on accessibility information. Metadata may be incorrect or the element may not exist. Rely primarily on the images.
The element image should show a single element with a unique function. If the element is obstructed, covered by a window or pop-up, or if multiple cropped elements are shown — return an empty string.
Inspect the red box carefully: if the element is not visible, return an empty string.
If there is no red box - return empty string. Return your answer in JSON format, with no extra text.
Example:

```
{
    "task": "Open the menu to see tutorials",
    "task_category": "Search & Information",
    "element_category": "Button"}
```

Input Task Prompt

You are given:

- An original task description for a UI interaction: `{task_string}`
- A screenshot showing the full interface with a red-highlighted element
- A cropped view focusing on just the highlighted element

Your goal: Change the task into a natural-language instruction **fully in English** that involves only inputting text. Output an action as `"type"` + the exact text to input. If not clearly solvable from the task, revise it.
Key Principles:

- Make it sound like a real instruction a person would give
- Use exact input (no placeholders); don't interpret content—be explicit
- Focus on real-world intent and what a user is likely trying to do

Requirements:

- Instruction must be clear, natural, and concise
- Action must start with `type` and include exact text
- Both fields must be fully in English
- No placeholders like "your name" or "email"
- Avoid click/press/select – only typing
- Must be obvious what to type from the instruction
- Never add phrases like "by typing it"

Output Format (JSON):

```
{"task": "Use john.doe@example.com as your login email",
 "action": "type john.doe@example.com"}
```

Bad vs Good Examples:

- "Enter coded message" → "Enter 1234 as your coded message"
- "Save your converted files…" → "Use /Users/yourname/Desktop as your destination folder"
- "Check the box labeled…" → "Select Include borders and shadings as your option"

Avoid These Mistakes:

- Placeholder text: "your name" → "Maria Garcia"
- Mechanical: "password in field" → "Use TrustNo1 as your password"
- UI-only focus: "Fill search box" → "Find information about electric cars"
- Vague: "Type the code" → "Enter 8294 as your verification code"
- Impersonal: "Input required" → "Add your birthday as 03/15/1988"

A.4 Training Setup

We training 3 GUI agents on the collected dataset.

A.4.1 GUIrilla-See-0.7B

GUIrilla-See-0.7B is built on Florence 2-Large (≈ 0.7 B parameters) and fine-tuned via supervised fine-tuning for *open-vocabulary detection* in GUI screenshots. Given an image and a free-form textual query, the model predicts either a bounding box or a polygon mask that encloses the best-matching UI element.

LoRA configuration. Fine-tuning uses Low-Rank Adaptation with RSLoRA initialisation:

- rank $r = 8$
- scaling $\alpha = 16$
- dropout $= 0.05$
- bias $=$ *none*
- target modules $= \{$`q_proj`, `o_proj`, `k_proj`, `v_proj`, `linear`, `Conv2d`, `lm_head`, `fc2`$\}$
- weight init *Gaussian*

Training setup.

- Hardware: 1 × NVIDIA A100 40 GB.
- Batch size: 8, mixed precision.

- Optimiser: AdamW, learning rate 2×10^{-5}. Cosine decay schedule with a 5% warm-up fraction.
- Epochs: 4; total wall-clock time $\approx$ 10 hours.

A.4.2 GUIRILLA-SEE-3B

GUIrilla-See-3B starts from QWEN-2.5-VL-3B-INSTRUCT (3 B parameters) and is fine-tuned with supervised fine-tuning (SFT) to localise macOS GUI elements. Given a full-desktop screenshot and a natural-language instruction, the model outputs a single coordinate (x, y) that lies at (or very close to) the centre of the referenced region.

LoRA configuration. Fine-tuning uses Low-Rank Adaptation (LoRA) in `bfloat16` mixed precision:

- rank $r = 32$
- scaling $\alpha = 16$
- dropout $= 0.1$
- bias $=$ *none*
- target modules $= \{$`down_proj`, `o_proj`, `k_proj`, `q_proj`, `gate_proj`, `up_proj`, `v_proj`$\}$
- weight init *Gaussian*

Training setup.

- Hardware: **2 $\times$ NVIDIA H100 80 GB**.
- Global batch size: 16
- Optimiser: AdamW with $\beta_1 = 0.9,\ \beta_2 = 0.95$.
- Learning rate: 2×10^{-5}, cosine decay schedule, warm-up ratio 0.05.
- Attention kernel: **FlashAttention-2** for memory-efficient training.
- Epochs: 2; total wall-clock time $\approx$ 5 hours.

A.4.3 FINE-TUNING IMPROVEMENT ON BASE MODELS.

Model	Base Acc. (%)	Tuned Acc (%)
Florence Large (0.7B)	8.31	53.55
Qwen 2.5 VL (3B)	18.40	73.48
Qwen 2.5 VL (7B)	35.78	**75.59**

Table 2: Accuracy improvements after fine-tuning on GUIRILLA-TASK.

A.4.4 GUIRILLA-SEE-7B

We also train a larger model that starts from QWEN-2.5-VL-7B-INSTRUCT (7B parameters). All LoRA, optimiser, and scheduler settings are kept *identical* to the 3B run. Using the same 2 $\times$ H100 80 GB configuration with FlashAttention-2, training finishes in roughly 6–7 hours.

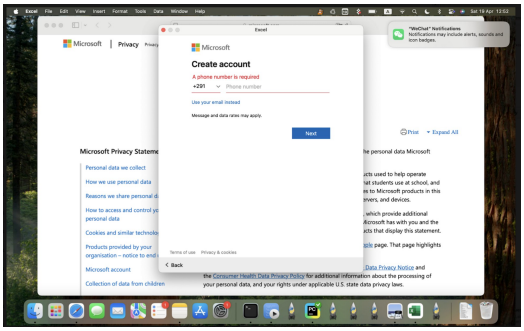
A.5 SCREENSPOT DETAILS

A.5.1 DATA LEAKAGE ANALYSIS ON SCREENSPOT

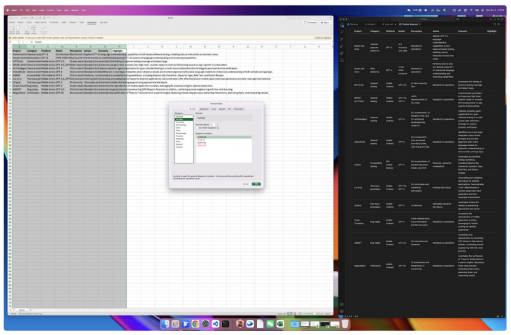
We manually screened overlaps by bundle IDs and application names to ensure no data leakage happened during training for both ScreenSpot-v2 and ScreenSpot-Pro benchmarks. As ScreenSpot-v2 doesn't provide this information, we manually labeled the apps there.

We discovered the following overlaps out of 881 applications in our train dataset and ScreenSpot test sets: OneNote appears in train data (macOS app) and in ScreenSpot-v2 (Windows). This app

has 1 task in the benchmark, and the login screen looks identical, so the leakage may have affected the result. We adjusted the score to account for it from 90.41% -> 90.33%. This doesn't influence the ranking, yet we adjusted the score for fairness. Microsoft Excel appears in the train dataset and in ScreenSpot-Pro. Here we manually looked at every screen (screen ids 4650-4659) and found that our data only includes a login flow and never actually opens the main app and its functionality. In ScreenSpot-Pro on the other hand, all tasks focus on Excel functions as part of the multi screen window. So, we assume that no major leakage was done here.



(a) GUIrilla: Excel login page.



(b) ScreenSpot-Pro: Main app with table manipulation.

Figure 8: Side-by-side comparison of Excel app data across datasets.

Model	Development	Creative	CAD	Scientific	Office	Overall Acc
UI-TARS-72B	40.8	39.6	17.2	45.7	54.8	38.1
UI-TARS-7B	36.1	32.8	18.0	50.0	53.5	35.7
GUIrilla-See-7B	30.10	31.96	26.82	47.64	53.04	35.36
GUIrilla-See-3B	24.08	25.51	24.52	39.37	47.39	29.35
UI-TARS-2B	26.4	27.6	14.6	39.8	42.6	27.7
Qwen 2.5 VL (7B)	24.7	24.9	14.6	30.3	47.0	27.1
OS-Atlas-7B	17.7	17.9	10.3	24.4	27.4	18.9
UGround-7B	14.7	17.0	11.1	19.3	27.0	16.5
Qwen 2.5 VL (3B)	10.7	17.3	7.3	24.4	28.7	15.9
CogAgent (18B)	8.0	5.6	6.1	13.4	10.0	7.7
ShowUI (2B)	9.4	5.3	1.9	10.6	13.5	7.7
GUIrilla-See-0.7B	6.69	6.45	3.83	12.20	8.26	7.34
OS-Atlas-4B	3.7	2.3	1.5	7.5	4.8	3.7
Florence-2 (0.7B)	0.0	0.0	0.0	0.4	0.0	0.12

Table 3: Task category performance per app category on ScreenSpot-Pro.

A.6 Additional Evaluation Details

A.6.1 Grounding

A detailed quantitative comparison with existing datasets is provided in Table 4.

Dataset	#Apps	#Tasks	#Unique UIs	Collection	Desktop	macOS	Fullscreen	Grounding	Agentic
OSWorld (Xie et al., 2024)	10	369	-	Manual	✓	×	✓	×	✓
OmniACT (Kapoor et al., 2024)	60	9802	-	Manual	✓	✓	✓	×	×
ScreenSpot-V2 (Wu et al., 2024)	6	324	187	Manual	✓	×	×	✓	×
ScreenSpot-Pro (Li et al., 2025)	23	1581 (511)	-	Manual	✓	✓	✓	✓	×
Mind2Web (Deng et al., 2023)	×	2350	2350	Manual	×	×	×	✓	✓
OSAtlas (Wu et al., 2024)	-	-	2.2M (1339)	Automated	✓	✓	✓	✓	×
Web-Hybrid (Gou et al., 2025)	×	-	773K	Automated	×	×	×	✓	×
GUIrilla-Task	**1108**	**27171**	**4179**	Automated	✓	✓	✓	✓	✓

The number in brackets denotes the reported quantity for macOS.

Table 4: Comparison of Existing Datasets for Task Automation

We fine-tune and release three GUIrilla-See agents of varying parameter scales on our GUIrilla-Task dataset: GUIrilla-See (0.7B) (based on Florence-2-large (Xiao et al., 2024)), GUIrilla-See (3B) and GUIrilla-See (7B) (based on Qwen-2.5-VL-Instruct (Bai et al., 2025)). All models are trained exclusively on GUIrilla-Task dataset. For training details, see Appendix A.4.

macOS Grounding Evaluation. We evaluate element localization by functional category on the GUIrilla-Task test set and compare against representative multi-OS baselines (UI-TARS, OS-Atlas, UGround). Rather than positioning this as a race for state-of-the-art, we use these results to illustrate how GUIrilla-Task provides practical, function-level supervision for realistic macOS desktop scenarios. We find particularly strong gains in Settings (+8.7), Connectivity (+26.3), Files (+7.5), and Input (+8.7), as shown in Table 5. Importantly, improvements are spread across element types as well (buttons, input fields, dialogs), the full table can be found in Appendix, Table 7.

Model	Communication	Files	Navigation	Search & Information	E-commerce	Accounts	Input	Apps	Media	Settings	Connectivity	Total
UI-TARS 2B	27.6%	45.6%	53.3%	49.5%	52.2%	61.9%	31.3%	50.0%	35.3%	50.3%	42.1%	47.53%
UI-TARS 1.5 7B	48.3%	67.0%	63.9%	74.7%	72.6%	**81.0%**	56.5%	68.8%	54.9%	80.9%	68.4%	69.07%
OS-Atlas 7B	48.3%	64.9%	59.9%	68.3%	70.8%	**81.0%**	53.9%	66.7%	**60.8%**	66.5%	63.2%	64.86%
UGround 2B	51.7%	63.0%	60.8%	70.0%	69.0%	**81.0%**	45.2%	62.5%	56.9%	67.6%	68.4%	64.03%
UGround 7B	62.1%	67.4%	68.7%	75.4%	69.0%	**81.0%**	54.8%	70.8%	52.9%	78.6%	52.6%	69.46%
GUIrilla-See 3B	51.7%	74.7%	68.7%	77.8%	76.1%	**81.0%**	57.4%	**72.9%**	**60.8%**	82.7%	73.7%	73.48%
GUIrilla-See 7B	**65.5%**	**74.9%**	**70.5%**	**79.2%**	**78.8%**	**81.0%**	**65.2%**	70.8%	**60.8%**	**87.3%**	**78.9%**	**75.59%**

Table 5: Performance breakdown across task categories on GUIrilla-Task test set. Best performance per category shown in **bold**.

ScreenSpot Evaluation. Table 6 reports grounding accuracy on ScreenSpot-v2 (Li et al., 2025) and ScreenSpot-Pro. While absolute comparisons are influenced by differences in architectures, training pipelines, and closed-source data, the results provide context on how *dataset composition and platform realism* affect transfer. Notably, GUIrilla-See (7B), trained on only 4.2K macOS full-desktop screens, achieves comparable macOS-specific accuracy to much larger multi-OS training regimes, despite using orders of magnitude fewer images. This supports the central point of the paper: GUIrilla-Task is a compact but high-utility macOS dataset that can be reused to train and stabilize downstream models. To ensure fairness in evaluation, we made sure that there is no data leakage, details can be found in Appendix A.5.1.

Additionally, we see that full-screen supervision (compared to (Gou et al., 2025)) as well as task formulation on a function level (compared to description-only as in (Wu et al., 2024)) can enable better contextual grounding in realistic GUI settings. Additional analysis of model robustness across different decoding strategies and confidence intervals are provided in Appendix A.10.

In grounding evaluations (Table 7), GUIrilla-See (7B) achieved the highest overall accuracy at 75.59%. GUIrilla-See (7B) showed particularly strong results on buttons, input fields, and "Other" elements such as icons and links, demonstrating robust performance across varied UI components. Interestingly, UGround shows marginal advantages in menu-heavy tasks, and we found their data to contain 400× more menu samples, reflecting the limits of scale without functional task diversity.

Model	Platform	Data	# Images	ScreenSpot-v2	ScreenSpot-Pro	ScreenSpot-Pro(macOS)
Florence-2 (0.7B)	Multi-OS	Real + Synthetic	-	1.8%	0.12%	0.16%
Qwen 2.5 VL (3B)	Multi-OS	Real + Synthetic	-	62.34%	15.93%	18.37%
Qwen 2.5 VL (7B)	Multi-OS	Real + Synthetic	-	87.5%	27.13%	34.27%
CogAgent (18B)	Multi-OS	Real + Synthetic	40M	52.8%	7.7%	4.6%
UGround (7B)	Web + Android	Real	1.3M	76.3%	16.5%	12.3%
ShowUI (2B)	Multi-OS	Real + Synthetic	256K	77.3%	7.7%	10.8%
OS-Atlas (7B)	Multi-OS	Synthetic	2.2M	83.3%	18.9%	20.0%
UI-TARS (2B)	Multi-OS	Real + Synthetic	∼20M (est.)	84.7%	27.7%	15.4%
UI-TARS (72B)	Multi-OS	Real + Synthetic	∼20M (est.)	90.3%	**38.1%**	40.0%
UI-TARS (7B)	Multi-OS	Real + Synthetic	∼20M (est.)	91.6%	35.7%	27.7%
GUIrilla-See (0.7B)	macOS	Synthetic	**4.2K**	53.55%	7.34%	7.95%
GUIrilla-See (3B)	macOS	Synthetic	**4.2K**	89.54%	29.35%	32.62%
GUIrilla-See (7B)	macOS	Synthetic	**4.2K**	**94.73%**	35.36%	**41.39%**

Table 6: Grounding accuracy comparison on ScreenSpot-v2 and ScreenSpot-Pro (full and macOS subset). **Bold** is used to highlight the best result, underline to hightlight second best result.

	Grounding Accuracy (%)					
Model	Button	Input	Menu	Checkbox	Other	Overall
Qwen 2.5 3B	8.0	0.0	0.0	-	–	-
Qwen 2.5 7B	36.49	30.36	46.0	44.68	24.74	35.78
UI TARS 2B	50.56	25.0	52.67	59.57	36.84	47.53
UGround v1 2B	64.26	48.21	79.33	68.09	58.95	64.03
OS-Atlas-Base-7B	65.76	53.57	72.67	57.45	62.11	64.86
UI TARS 1.5 7B	68.57	64.29	86.67	**78.72**	58.42	69.07
UGround v1 7B	68.67	56.25	**88.67**	**78.72**	64.21	69.46
GUIrilla-See (3B)	74.48	61.61	81.33	**78.72**	**66.84**	73.48
GUIrilla-See (7B)	**76.55**	**66.07**	86.67	76.6	**66.84**	**75.59**

Table 7: Grounding accuracy across models and element categories on GUIRILLA-TASK.

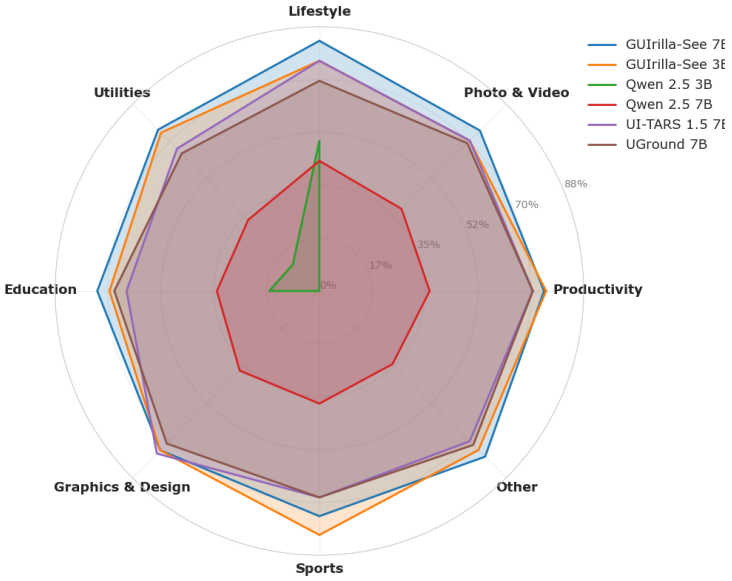


Figure 9: Comparative Performance on GUIrilla-Task (Grounding) of Vision Language Models Across Application Domains. Larger models (7B) generally outperform smaller ones, with the biggest gains in Developer Tools, Productivity, and Graphics & Design. GUIrilla-See 3B shows strong performance relative to 7B models, indicating effective domain specialization.

Cross-OS transfer. Despite macOS-only training, GUIRILLA-SEE (7B) reaches 32.03% on Windows (ScreenSpot-Pro) and 41.39% on macOS, exceeding OS-Atlas 7B (12.3% / 20%) and UGround 7B

(14.9% Windows). Thus, single-platform, function-level supervision does not preclude transfer and can outperform larger mixed-OS synthetic sets on challenging full-desktop scenes.

A.6.2 Qualitative Analysis

Analysis of 1,565 tasks across 227 applications reveals that macOS-specific training yields consistent improvements across fundamental UI interaction patterns. We identified 79 tasks where GUIrilla succeeds while all baselines fail, demonstrating strong understanding of macOS-specific paradigms: Finder-style dialogs ("Browse for movie destination folder"), System Preferences ("Edit advanced output settings"), and window management ("Close the Chat-with-Erix panel"). These success patterns validate our function-oriented approach, showing models learn what UI elements do rather than where they appear or their visual description.

Failure Mode Analysis. ScreenSpot-Pro evaluation reveals two key weaknesses: (1) icon-dense engineering tools such as Vivado, where tasks like “click group by repository button” or “open TCL console” fail due to limited representation of compact, icon-heavy UIs in the dataset; and (2) creative software like Illustrator and DaVinci Resolve, where canvas-focused actions such as “draw a circle” or “select brush tool” expose insufficient coverage of creative workflows(Table 3). The model performs well on office applications and system-level tasks, suggesting macOS-focused training generalizes across typical desktop environments but requires targeted data collection for specialized professional domains. This can be mitigated by extending crawling to more creative apps and using accounts with pre-filled user-generated content in the future work, that allow for more content manipulation.

A.6.3 Agentic

	Success Rate (%)		
Model	Input	Click	Overall
OpenAI Computer Use	8.04	**68.75**	**64.41**
Claude Computer Use	8.93	65.59	61.53
OS-Atlas-Pro-7B	7.14	62.84	58.85
UI TARS 1.5 7B	1.79	54.65	50.86
UI TARS 2B	7.14	50.24	47.16
Qwen 2.5 VL 3B	**12.5**	42.95	40.77
Qwen 2.5 VL 7B	2.68	39.16	36.55
CogAgent 9B	3.57	15.83	14.95

Table 8: Success rate across models and interaction categories on GUIrilla-Task (agentic)

A.7 Impact of Backbone Model

We further examined how the choice of backbone model influences performance. When trained on our dataset, a Qwen2-VL 7B backbone already surpasses OS-Atlas, despite both using the same underlying model. Notably, OS-Atlas was trained on nearly **300×** more data, yet our model achieves higher accuracy: +1.34% in average and +2.02% on the macOS subset of ScreenSpot-Pro. These results highlight the *data efficiency* of our approach.

A.8 Ablation Details

A.8.1 Influence of Handlers

Application	Metric	Handler-Supported Crawler	Random Crawler
Stocks	Tree Depth	14	16
	Number of Tasks	162	32
	Duplicate rate	0.08	0.14
	Parse Time (hh:mm:ss)	00:20:51	00:29:35
Maps	Tree Depth	6	9
	Number of Tasks	107	36
	Duplicate rate	0.1	0.2
	Parse Time (hh:mm:ss)	00:21:00	00:25:35
Weather	Tree Depth	6	7
	Number of Tasks	73	73
	Duplicate rate	0.0	0.01
	Parse Time (hh:mm:ss)	00:10:56	01:05:48

Table 9: Comparison of Handler-Supported Crawler vs Random Crawler Across Applications

A.8.2 Comparing Deterministic and GPT-Refined Task Descriptions

Table 10: Examples of Task Agent and Deterministic Instructions by App

App	Task Agent	Deterministic
Prayer Notes	Access Prayer Notes support page	button
GoProPlayer	Open a media file	button Open_Media...
Fax	Add files or images to the fax	ADD FILES OR IMAGES button

A.9 GUIrilla-Gold Dataset Annotation Guidelines

A.9.1 Overview

To assess the reliability of macOS accessibility (AX) metadata and the quality of GPT-generated task strings, we hired 5 annotators, who were given the test split data. Annotators with accessibility expertise reviewed each data entry along five dimensions: (1) task feasibility; (2) task instruction clarity and editing for ambiguity; (3) manual task execution; (4) accessibility tree quality rating (Good/Medium/Bad scale); and (5) element-level verification of semantic properties (role, description, value) and bounding-box accuracy.

Task Quality. From the 1319 original English language-based tasks, 84.3% of tasks were marked as DOABLE after manual verification. Comparing GPT strings to human edits, 91% required no change. The 109 edited cases showed 97% similarity to originals (Ratcliff/Obershelp), confirming minor edits. We release manully edited dataset as GUIrilla-Gold [7].

AX Quality. Accessibility metadata quality varies significantly: 64% of screens received GOOD ratings, 24% MEDIUM, and 12% BAD. At the element level, only 40% have correct role and description pairs, while 49% contain role information only, and 11% are mislabeled. Bounding boxes are accurate for 80% of elements, though 10% extend outside the visible window. This metadata sparsity and noise make accessibility-only task generation unreliable. We therefore recommend combining accessibility trees with screenshots and applying vision-based semantic adjustment to generate more precise, function-oriented, visually grounded tasks.

This document provides instructions for annotators to evaluate and improve UI task datasets, with focus on accessibility principles.

[7] https://huggingface.co/datasets/GUIrilla/GUIrilla-Gold/

A.9.2 TASK STRING FEASIBILITY EVALUATION

Evaluation Steps:

- **Step 1:** Evaluate clarity and readability of the task string. Edit if ambiguous or poorly phrased.
- **Step 2:** Assess executability. Mark as DOABLE if the task is clear and the required element is visible. Mark as NOT DOABLE if the element is not visible or does not exist.

DOABLE Examples: "Click the Submit button", "Type 'hello world' in the search field"
NOT DOABLE Examples: "Click the button" (when multiple buttons present), "Enter your password" (if no password input visible)

A.9.3 TASK EXECUTION GUIDELINES

Attempt to execute the task exactly once to verify correctness:

- **Click Actions:** Locate the correct element and click once within its bounding box
- **Type Actions:** Find the input field and type the exact text provided (case-sensitive)
- **Multi-step Tasks:** Mark as NOT DOABLE if requiring multiple distinct actions

Constraints: Attempt only once, no retries. Do not fabricate actions not in the task string.

A.9.4 ACCESSIBILITY QUALITY RATING (1–3 SCALE)

Score 1 – BAD: Critical issues severely impact assistive technology—missing labels, incorrect roles, invisible elements, broken grouping, no logical structure.

Score 2 – MEDIUM: Moderate issues present—incomplete/generic labels, occasional role mismatches, partial grouping, minor positioning issues.

Score 3 – GOOD: Accessibility tree accurately represents visual UI—descriptive labels, accurate roles, proper grouping, logical structure, complete state information.

A.9.5 LABEL AND ROLE VERIFICATION

Review each element's semantic description and role. Uncheck "Semantic" if the meaning or AX role is incorrect. Uncheck "BBox" if the bounding box doesn't match the visible element.

A.9.6 QUALITY ASSURANCE CHECKLIST

- ☐ Task string evaluated and edited for clarity
- ☐ NOT DOABLE marked for invisible elements
- ☐ Task execution attempted once
- ☐ Accessibility score reflects usability
- ☐ Labels and roles verified
- ☐ Checkboxes unchecked for mismatches

A.10 CONFIDENCE INTERVALS AND SENSITIVITY TO DECODING STRATEGIES

To provide uncertainty estimates and strengthen the reliability of model comparisons, we conducted additional experiments examining performance variability across different decoding strategies.

A.10.1 EXPERIMENTAL SETUP

Our primary results were obtained using greedy decoding with the following generation parameters: `num_beams=3`, `do_sample=False`, `temperature=None`, `top_p=None`, `top_k=None`.

Model	**Greedy (%)**	**Sampled (T=0.3) (%)**
Florence Base	10.73	10.64 ± 0.10
Florence Large	8.31	8.08 ± 0.10
Qwen 2.5 VL 3B	17.96	18.06 ± 0.22
Qwen 2.5 VL 7B	35.78	35.40 ± 0.45
UI-TARS 2B	47.54	18.43 ± 0.22
UI-TARS 1.5 7B	69.07	52.17 ± 0.42
UGround v1 2B	64.03	64.41 ± 0.00
UGround v1 7B	69.46	69.84 ± 0.06
OS-Atlas-Base-7B	64.86	61.82 ± 0.10
GUIrilla-See-0.7B	53.55	53.48 ± 0.32
GUIrilla-See-3B	73.48	73.90 ± 0.03
GUIrilla-See-7B	75.59	75.85 ± 0.06

Table 11: Performance comparison between greedy and stochastic decoding strategies. Models fine-tuned on GUIrilla-Task (bottom section) show consistent performance with minimal variance, while some pretrained models exhibit notable degradation under stochastic decoding.

To assess performance variability, we re-evaluated all models using stochastic decoding with parameters: `num_beams=3`, `do_sample=True`, `temperature=0.3`. For each model, we conducted three independent runs and computed mean ± standard deviation of success rates.

A.10.2 RESULTS

Table 11 presents results for both decoding strategies on the GUIrilla-Task test set.

The results reveal distinct patterns in decoding sensitivity. Fine-tuned GUIrilla-See models demonstrate remarkable consistency across decoding strategies, with standard deviations below 0.32% in all cases. This stability suggests robust learning of UI interaction patterns.

In contrast, several pretrained models show significant performance degradation under stochastic decoding, most notably UI-TARS models which experience drops of 16-29 percentage points. This sensitivity highlights the importance of decoding strategy selection and suggests that some models may be overfitting to specific generation patterns during pretraining.

The minimal variance observed in our fine-tuned models provides confidence in the reported performance gains and demonstrates the robustness of the GUIrilla training approach across different inference conditions.