A11y-Compressor: A Framework for Enhancing the Efficiency of GUI Agent Observations through Visual Context Reconstruction and Redundancy Reduction

Michito Takeshita Takuro Kawada Takumi Ohashi
Shunsuke Kitada Hitoshi Iyatomi
Hosei University, Tokyo, Japan
Correspondence: michito.takeshita.4t@stu.hosei.ac.jp, iyatomi@hosei.ac.jp

Abstract

AI agents that interact with graphical user interfaces (GUIs) require effective observation representations for reliable grounding. The accessibility tree is a commonly used text-based format that encodes UI element attributes, but it suffers from redundancy and lacks structural information such as spatial relationships among elements. We propose A11y-Compressor, a framework that transforms linearized accessibility trees into compact and structured representations. Our implementation, Compressed-a11y, applies a lightweight and structured transformation pipeline with modal detection, redundancy reduction, and semantic structuring. Experiments on the OSWorld benchmark show that Compressed-a11y reduces input tokens to 22% of the original while improving task success rates by 5.1 percentage points on average.

1 Introduction

AI agents that interact with graphical user interfaces (GUIs) have advanced rapidly with multimodal large language models (MLLMs) (Lin et al., 2025; Hong et al., 2024). These agents perform tasks by interpreting complex on-screen environments, such as booking flights or responding to emails. While cloud-hosted closed-source MLLMs achieve strong performance, their real-world deployment is constrained by privacy risks, latency, and operational costs (Zhang et al., 2025b,a). As a result, locally deployed open-source MLLMs have emerged as a practical alternative (Niu et al., 2024; Wang et al., 2024). However, these models face significant challenges in grounding, i.e., aligning UI elements with executable actions (Wu et al., 2025).

Effective grounding critically depends on how GUI environments are represented as observations (Cheng et al., 2024; Zheng et al., 2024). Existing approaches can be broadly categorized into image-based and text-based representations (Xie et al., 2024; Zhou et al., 2024). Image-based representations provide rich visual information but often struggle with precise element localization (Cheng et al., 2024; Lin et al., 2025). In contrast, text-based representations explicitly encode semantic attributes such as element roles, names, and positions, facilitating more reliable target identification (Zhou et al., 2024; Xie et al., 2024).

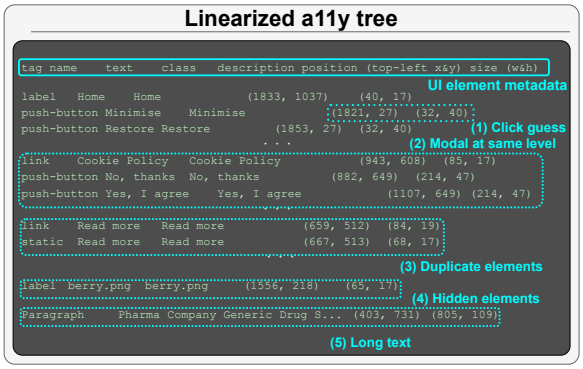

Figure 1: Examples of a linearized a11y tree, illustrating several issues: (1) providing position and size information can lead to unnecessary click-coordinate inference; (2) modal UIs are represented at the same hierarchical level as background elements, obscuring visual layering; (3) the same visual element may appear under multiple tags, creating ambiguity in element selection; (4) visually hidden elements are still included in the UI tree; and (5) elements with the Paragraph role tend to contain long text spans, increasing context length usage.

The accessibility (a11y) tree is a used text-based representation that organizes UI elements hierarchically (Zhou et al., 2024; Xie et al., 2024). Despite its effectiveness, it has two key limitations. First, its hierarchical structure does not align with the visual layout, making it difficult to capture spatial relationships and semantic regions (Kerboua et al., 2025). Second, it contains substantial redundancy due to exhaustive attribute preservation, which increases token consumption and diffuses model attention (Kerboua et al., 2025; Xie et al., 2024). Prior approaches, such as element selection

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop), pages 563–580

and linearization, partially address redundancy but still fail to preserve spatial ordering and intrinsic GUI structures (Kerboua et al., 2025; Deng et al., 2023; Xie et al., 2024). In particular, the linearized a11y tree is widely adopted as an observation representation for agents; however, it exhibits several challenges, as illustrated in Figure 1.

To address these limitations, we propose A11y-Compressor, a framework for constructing compact and structured GUI observations from linearized a11y trees. The framework consists of three stages: modal detection, which reconstructs foreground-background relationships; redundancy reduction, which removes irrelevant or repetitive elements; and semantic structuring, which organizes elements into meaningful groups. This design preserves essential structural information while significantly reducing token overhead, enabling more effective grounding for local MLLMs.

Our main contributions. (1) We propose A11y-Compressor, a structured framework for constructing efficient GUI observation representations from the linearized a11y tree. (2) We demonstrate that observations generated by our framework significantly improve task success rates while reducing input token consumption for local MLLM-based GUI agents compared with existing observation formats on a GUI agent benchmark.

2 Related Work

Observation Representations. A central challenge in GUI agents is how to represent the environment for decision-making. Existing approaches can be broadly categorized into image-based and text-based representations. Image-based methods directly process screenshots using vision-language models, capturing rich visual context but often struggling with precise element localization (Cheng et al., 2024; Lin et al., 2025). In contrast, text-based representations encode structured information such as element roles, attributes, and positions, enabling more accurate grounding (Zhou et al., 2024; Xie et al., 2024). Hybrid approaches combining both modalities have also been explored, but they often incur higher computational costs.

Accessibility Trees and Compression. The accessibility (a11y) tree is a widely adopted text-based representation for GUI agents, as it provides a hierarchical view of UI elements and their attributes (Zhou et al., 2024; Xie et al., 2024). Prior work has proposed various techniques to improve its efficiency, including element filtering, attribute selection, and linearization (Kerboua et al., 2025; Deng et al., 2023; Xie et al., 2024). These methods primarily focus on reducing redundancy and token usage. However, they often overlook the structural mismatch between hierarchical representations and the visual layout, leading to loss of spatial and semantic relationships. In contrast, our work focuses on constructing compact representations that preserve both structural and semantic information, enabling more effective grounding.

3 A11y-Compressor

As illustrated in Figure 2, A11y-Compressor is a structured framework that transforms linearized a11y trees into compact, semantically coherent GUI observations. Given a linearized a11y tree as input, the transformation is implemented as a three-phase pipeline: Modal Detection, Redundancy Reduction, and Semantic Structuring, each addressing a distinct aspect of observation construction. The framework is modular, allowing each phase to be instantiated with different algorithms as long as they satisfy structural objectives. The resulting representation preserves essential GUI structure while significantly reducing token overhead, enabling more efficient and reliable grounding.

3.1 Modal Detection

This phase identifies foreground modal UI elements and separates them from background elements to make interaction constraints explicit. Foreground elements such as modal UI elements (e.g., dialogs or pop-ups) introduce front–back relationships that restrict interaction with background elements; however, in the a11y tree, these elements are listed in parallel, potentially leading agents to select non-interactable background elements.

To address this issue, this phase detects modal UI elements and separates them from background elements, thereby constructing interaction constraints that arise from visual stacking relationships. In practice, modal elements are identified based on accessibility attributes in the linearized a11y tree, such as tags and task-specific keywords (e.g., *cookie*, *accept*, or action-related labels). In addition, for modals triggered by interactions, newly appearing UI elements within the same screen state are identified as modal regions. Elements that satisfy modal characteristics are assigned to the modal set M, while remaining elements are treated as

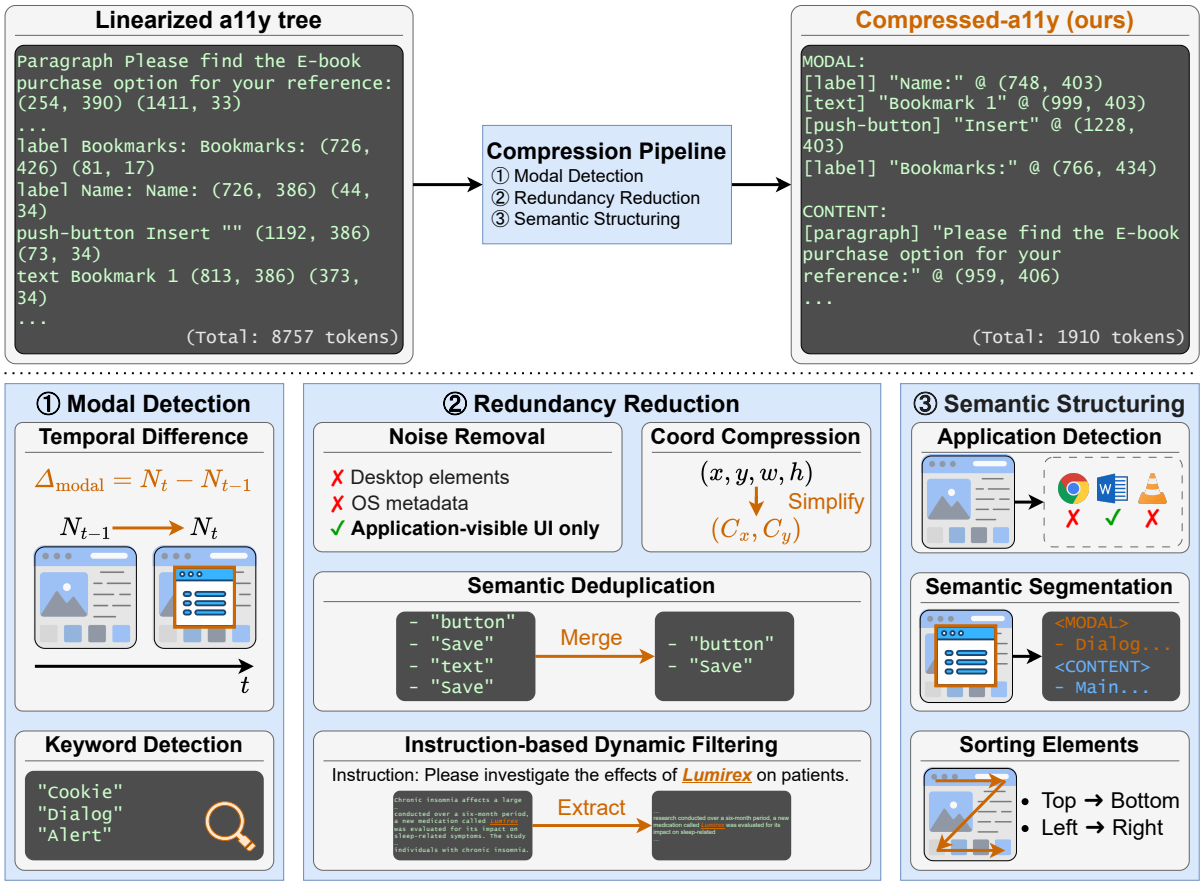

Figure 2: Overview of the A11y-Compresser framework. Given a linearized a11y tree, the pipeline applies modal detection, redundancy reduction, and semantic structuring to generate a compact and structure-preserving observation representation. This representation enables more efficient and effective grounding for GUI agents.

background elements B. Formally,

$$(M, B) = f_{\text{modal}}(x), \tag{1}$$

where x denotes the linearized a11y tree, and (M, B) are passed to the subsequent phase.

3.2 Redundancy Reduction

This phase reduces redundant and irrelevant information in the observation representation to improve the efficiency and reliability of grounding for GUI agents. Linearized a11y trees often contain forms of redundancy, including duplicated UI elements, verbose textual content, and elements irrelevant to the current task (e.g., background or off-window components). These factors increase input length and hinder the agent's ability to identify correct interaction targets. In addition, the spatial representation of UI elements can introduce ambiguity. The linearized a11y tree represents each element using its top-left coordinate and bounding box size; however, the top-left coordinate does not always correspond to an effective interaction point. To simplify spatial reasoning, this phase converts bounding box representations into center coordinates.

To address these issues, this phase applies rule-based preprocessing, including filtering irrelevant elements, merging duplicates, normalizing attributes, and compressing text. Formally, given modal and background elements (M, B) from the previous phase, it produces refined element sets:

$$(M', B') = f_{\text{reduce}}(M, B), \tag{2}$$

where M' and B' denote the modal and background elements after redundancy reduction. The refined element sets M' and B' are then passed to Section 3.3 for semantic structuring.

3.3 Semantic Structuring

This phase organizes UI elements into semantically meaningful regions to better reflect the functional structure of the GUI interface. The linearized a11y tree often does not explicitly represent high-level semantic information about the GUI interface, such

as which application it belongs to or what functional role each UI element plays. As a result, GUI agents must implicitly infer the current screen context and the functional meaning of UI elements (e.g., which button performs which operation).

To address this, this phase augments the observation with explicit semantic structure. This phase first identifies the application associated with the current interface, then partitions UI elements into semantic regions based on spatial layout and application-specific heuristics. These regions correspond to coherent functional areas (e.g., taskbars or navigation panels). Semantic regions are determined using application-specific heuristics derived from spatial layout and grounding patterns. Formally, given the background elements B' obtained from the previous phase, this phase reorganizes them into a set of semantic regions:

$$R = f_{\text{region}}(B'), \tag{3}$$

where $R = r1, r2, \&, rk$ denotes the set of detected semantic regions. The detected modal elements $M2$ are incorporated into the structured representation to construct the final observation used by the GUI agent:

$$O = f_{\text{struct}}(R, M'), \tag{4}$$

where O denotes the semantically structured observation used for grounding and action selection.

4 Experiments

4.1 Experiments Setup

We evaluate the effectiveness of A11y-compressor on the GUI agent benchmark OSWorld (Xie et al., 2024). Our evaluation set consists of 358 tasks from the standard task set, excluding tasks that could not be executed due to environment-dependent errors. The task distribution across application domains is as follows: web browsing (Chrome: 44), office work (LibreOffice Calc: 46, Impress: 47, Writer: 23), email management (Thunderbird: 15), media editing (GIMP: 26, VLC: 17), software development (VS Code: 23), basic OS operations (24), and cross-application tasks involving multiple applications (93). This task set broadly covers practical GUI operation scenarios. In all evaluation experiments, we employ Qwen3-VL-32B (Bai et al., 2025) as the MLLM for inference. Although we focus on Qwen3-VL-32B for controlled evaluation, the proposed representation is model-agnostic and applicable to other MLLMs that accept textual GUI observations.

4.2 Implementation of A11y-compressor

To evaluate the proposed framework, we implement A11y-Compressor using a rule-based approach. Each phase is instantiated with heuristic rules derived from the structural characteristics of GUI interfaces, capturing common GUI patterns while remaining lightweight for efficient preprocessing.

Modal Detection. The modal detection phase identifies foreground UI elements that block interactions with background elements. Modal elements are detected using two complementary strategies: temporal-difference detection and keyword-based detection. Temporal-difference detection compares the linearized a11y tree at step t with that at step $t-1$. If the screen state remains unchanged but new UI elements appear, they are treated as modal candidates. Keyword-based detection identifies UI elements containing representative modal-related keywords (e.g., cookie). By combining these signals, foreground modal elements are separated from background elements. Detailed rules are provided in Appendix A and Appendix B.

Redundancy Reduction. The redundancy reduction phase removes duplicated or irrelevant UI elements and compresses textual content in the observation representation. It also converts bounding box representations into center coordinates to simplify spatial reasoning. For textual compression, keywords are first extracted from the task instruction. If a `paragraph` tag contains a matching keyword, the surrounding context is preserved; otherwise, only a predefined number of leading characters is retained. Detailed implementation rules are provided in Appendix C.

Semantic Structuring. The semantic structuring phase organizes UI elements into semantically coherent regions. UI elements are first sorted from top-left to bottom-right based on their center coordinates, then partitioned into functional regions (e.g., APP_LAUNCHER, CONTENT) using application-specific heuristics derived from spatial layout. Detailed rules are provided in Appendix D.

To support the OSWorld benchmark (Xie et al., 2024), we instantiate each phase using rule-based heuristics designed from 145 screen states across nine application domains (e.g., Chrome, Writer, VS Code). These heuristics rely solely on structural and visual characteristics, without task-specific tuning, and are derived from data independent of the evaluation set to avoid benchmark bias.

4.3 Baseline Methods

We evaluate Compressed-a11y, an observation representation generated by A11y-Compressor, and compare it with three commonly used baselines. **(1) Screenshot.** A raw GUI screenshot directly provided as input to the MLLM. **(2) Linearized a11y tree.** A textual representation obtained by linearizing the hierarchical a11y tree into a one-dimensional sequence. **(3) LineRetriever (Kerboua et al., 2025).** A method that dynamically selects task-relevant lines from the a11y tree based on their contribution to actions. We extend the original web-based method to multiple application domains and use a lightweight, low-latency LLM (Qwen3-4B (Yang et al., 2025)) as the retriever.

For quantitative evaluation, we compare Compressed-a11y with Linearized a11y Tree and LineRetriever in terms of success rate and token efficiency. Screenshot-based observations are evaluated only in terms of success rate due to their distinct token characteristics. For qualitative analysis of modal interaction, we compare Compressed-a11y with Screenshot and Linearized a11y Tree to examine their impact on agent reasoning and actions. Additional qualitative results for LineRetriever are provided in Appendix E.

4.4 Evaluation Metrics

We evaluate agent performance using two quantitative metrics and a qualitative case study. **Success Rate (SR).** SR is defined as the proportion of tasks successfully completed within a maximum of 15 interaction steps. To mitigate the impact of non-deterministic factors in the OSWorld environment, such as system response delays and variability in MLLM outputs, we conduct two trials for each task and consider a task successful if at least one trial succeeds. **Average Input Tokens.** We measure the average number of input tokens provided to the MLLM per application domain to quantify the efficiency of the observation representation. **Case Study.** In addition to quantitative evaluation, we analyze the behavior of different observation representations through a qualitative case study involving a task with a modal dialog.

4.5 Ablation Study

To analyze the contribution of each phase of the A11y-compressor framework, we conduct an ablation study. We compare the full three-phase pipeline with variants that apply each phase individually, including modal detection, redundancy reduction, and semantic structuring.

5 Results

5.1 Quantitative Results

Token Efficiency. Figure 3 compares the average number of input tokens for each observation representation across application domains. Overall, while LineRetriever effectively reduces the average input token count compared to the linearized a11y tree, it still produces large input sizes in domains with inherently high token complexity. In contrast, Compressed-a11y consistently limits the number of input tokens to approximately 3,500 or fewer across all application domains. These results demonstrate that Compressed-a11y effectively suppresses token growth even for large-scale and complex UIs, enabling more efficient processing by local MLLMs.
Task Success Rate. Table 1 reports task success rates across the different observation representations. Compressed-a11y achieves the highest overall average success rate (0.207), outperforming all baseline representations. In particular, it achieves success rates of 0.304 and 0.467 for LibreOffice Writer and Thunderbird, respectively, showing substantial improvements over the baselines in these domains. Although LineRetriever improves performance over the linearized a11y tree in several domains, its average success rate remains slightly lower than the linearized a11y tree baseline.

5.2 Case Study: Modal Dialog Handling

Figure 5 presents a representative case study of a task involving a modal dialog. The agent must correctly handle a privacy consent modal before interacting with the underlying flight booking interface. With screenshot-based observations, the agent recognizes the modal but generates inaccurate click coordinates, leading to interactions at incorrect locations. As a result, it repeatedly fails to interact with the intended UI elements and eventually reaches the maximum step limit. With the linearized a11y tree, the agent fails to recognize that the modal blocks interaction with the background interface and attempts to interact with elements behind it, again resulting in failure.

In contrast, Compressed-a11y successfully identifies the modal and its relevant UI elements. After initially attempting to close the dialog, the agent recognizes that additional actions are required and enables the necessary toggles before confirming the

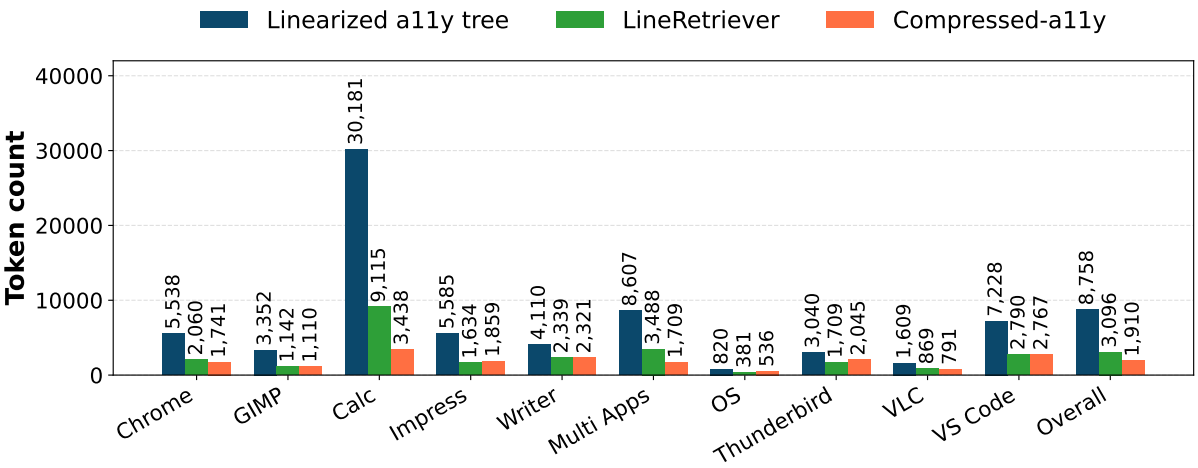

Figure 3: Average input token counts of observation representations across application domains. Compressed-a11y consistently achieves lower input token counts than both the linearized a11y tree and LineRetriever across domains, while maintaining a compact and stable representation even for applications with complex user interfaces.

Method	Chrome	GIMP	Calc	Impress	Writer	Multi Apps	OS	Thunderbird	VLC	VS Code	Overall
Screenshot	0.045	0.115	0.000	0.021	0.043	**0.108**	0.208	0.000	0.118	0.043	0.070
Linearized a11y tree	0.182	0.192	0.000	0.149	0.087	**0.108**	0.333	0.267	**0.294**	0.304	0.156
LineRetriever	0.136	0.192	0.022	**0.191**	0.087	**0.108**	0.333	0.133	0.176	**0.348**	0.151
Compressed-a11y (ours)	**0.250**	**0.231**	**0.043**	**0.191**	**0.304**	**0.108**	**0.375**	**0.467**	**0.294**	**0.348**	**0.207**

Table 1: Task success rates across various application domains. Compressed-a11y consistently achieves the highest or tied-highest success rates across most domains. The highest score for each domain is highlighted in bold.

dialog. This structured representation allows the agent to correctly complete the modal interaction and proceed with the task.

5.3 Ablation Study

Table 2 presents a component-wise analysis of the A11y-Compresser framework. The resulting representation (Compressed-a11y) achieves the highest overall success rate (0.207) and outperforms all ablated variants acrossamong mostall configurations.application domains. The redundancy reduction–only variant achieves performance comparable to the linearized a11y tree baseline (0.156). In particular, it attains a success rate of 0.467 on Thunderbird, matching the full pipeline in this domain. Both modal detection only and semantic structuring only variants achieve lower overall success rate (0.134). While modal detection shows relatively stable performance across domains, semantic structuring exhibits more domain-dependent behavior. For example, semantic structuring achieves the highest success rate in Calc and Multi Apps, but fails to solve any tasks in the Thunderbird domain. These results show that combining all three phases leads to the most consistent performance across application domains.

6 Discussion

6.1 Effectiveness of Compressed-a11y

Compressed-a11y consistently achieves the highest task success rate across most tasks while reducing the number of input tokens to approximately 22% of that required by the baseline linearized a11y tree, corresponding to an average improvement of 5.1 percentage points. Token reduction rates varied across application domains and were closely related to the complexity of the UI in each domain.

The linearized a11y tree contain large amounts of redundant accessibility information, particularly in application domains with complex UIs such as Calc and VS Code. By aggregating and compressing such redundant information, Compressed-a11y achieves more pronounced token reductions in these domains. Meanwhile, in application domains with smaller reduction rates, such as OS and VLC, the number of UI elements was inherently limited and the redundancy of accessibility information was relatively low; consequently, the impact of compression was correspondingly smaller. Notably, despite variations in compression rate, Compressed-a11y maintains the highest task success rate across most application domains.

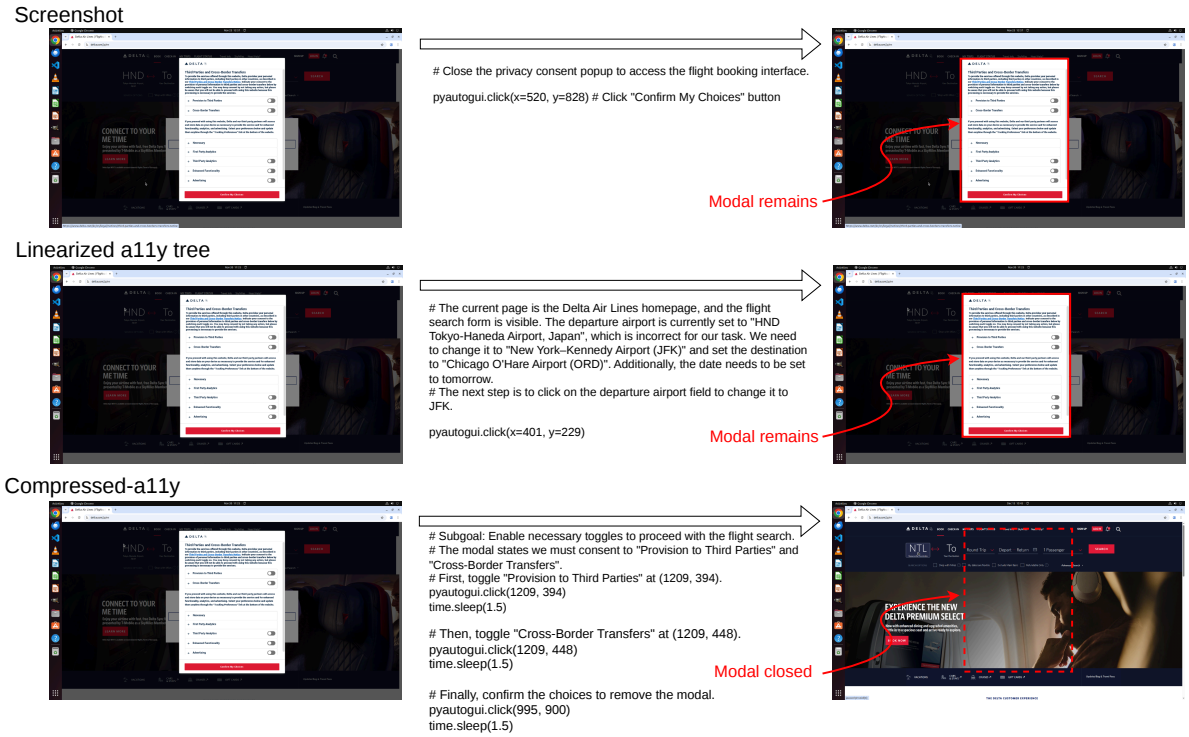

Figure 4: Example of modal dialog handling with different observation representations. Screenshot-based observations suffer from inaccurate coordinate grounding leading to incorrect interactions. The linearized a11y tree fails to capture modal-induced interaction constraints and attempts to interact with background elements. In contrast, Compressed-a11y explicitly models the modal structure, enabling correct interaction sequencing and successfully successful task completion.

These results indicate that Compressed-a11y effectively extracts task-relevant information while preserving essential GUI structure, enabling more efficient screen understanding and interaction by MLLMs. Furthermore, the effectiveness of Compressed-a11y varies across application domains. This variation likely arises because the compression pipeline incorporates application-specific designs, such as UI region segmentation based on structural characteristics of each interface. When these designs align well with the UI characteristics of a given application, Compressed-a11y yields larger performance gains; otherwise, the improvements are more limited. This observation suggests that considering application-specific UI layout characteristics is important for effectively organizing and highlighting task-relevant elements in compressed observation representations.

6.2 Modal Interaction Analysis

The qualitative case study highlights the importance of explicitly representing modal dialogs in observation representations. In screenshot-based observations, the agent often fails due to imprecise coordinate grounding, while the linearized a11y tree does not explicitly distinguish modal elements from background UI components. As a result, the agent may attempt to interact with elements that are temporarily inaccessible, leading to failed interactions. These observations indicate that observation representations should explicitly encode modal dialogs and provide clear grounding information for UI elements within the modal interface. Without such modal-aware representations, agents may incorrectly attempt to interact with background elements that are temporarily inaccessible.

6.3 Comparisons to Other Approaches

Through comparative experiments, we observed that even with a strong Local-MLLM, relying solely on screenshots is insufficient for solving general GUI tasks. This finding underscores the necessity of structured observation representations.

The linearized a11y tree, which serves as the foundation of Compressed-a11y, achieved substantially higher task success rates than screenshots.

Method	Chrome	GIMP	Calc	Impress	Writer	Multi Apps	OS	Thunderbird	VLC	VS Code	Overall
Full pipeline (Compressed-a11y)	0.250	0.231	0.043	0.191	0.304	0.108	0.375	0.467	0.294	0.348	0.207
Modal detection only	0.159	0.077	0.022	0.191	0.087	0.108	0.250	0.200	0.118	0.261	0.134
Redundancy reduction only	0.182	0.154	0.043	0.191	0.130	0.108	0.167	0.467	0.176	0.261	0.156
Semantic structuring only	0.159	0.115	0.109	0.170	0.000	0.118	0.333	0.000	0.059	0.217	0.134

Table 2: Ablation study of the proposed A11y-Compresser framework across application domains. We compare the full three-phase pipeline (Compressed-a11y) with variants that apply each phase individually, including modal detection, redundancy reduction, and semantic structuring. Each column reports the task success rate for the corresponding application domain, highlighting the best-performing configuration and enabling direct comparison of the contribution of each component.

However, as discussed earlier, for tasks involving complex UIs, the resulting observation representations tended to become redundant, potentially constraining the performance of the MLLM.

LineRetriever achieved a compact representation by using an LLM to extract only important lines. Nevertheless, it failed to preserve the global structural context required for GUI manipulation, resulting in inconsistent improvements in task success rates. Because many tasks involve exploration, retrieval-based representations are prone to missing critical UI elements and their relationships in the early interaction stages. Furthermore, dependence on the inference results of the retriever LLM may have introduced variability in the extracted content, leading to unstable task performance.

By compressing observations while preserving task-relevant information, Compressed-a11y outperforms existing methods in both token efficiency and task success rate. These results indicate that compressing the linearized a11y tree into a form that better captures overall UI structure improves GUI agent performance.

6.4 Ablation Study

The ablation analysis further clarifies the role of each phase in the A11y-Compressor framework. The full pipeline achieves the highest overall success rate, indicating that performance gains arise from the combination of multiple processing phases rather than any single component alone.

Among the single-phase variants, redundancy reduction only achieves the highest overall success rate. A possible explanation is that converting UI coordinates to center-based positions reduces ambiguity in click actions. When coordinates are represented by the top-left corner of a bounding box, clicking at that position may not always successfully interact with the intended UI element, depending on its visual shape. In such cases, task success may depend on whether the model correctly infers a valid click position within the bounding box, introducing variability in task performance. By converting coordinates to center-based positions, this ambiguity is reduced, leading to more reliable and consistent interactions.

In the Calc domain, semantic structuring only achieves the highest success rate. This may be because explicitly labeling interface regions helps the model focus on task-relevant UI components, reducing distraction from numerous irrelevant elements such as spreadsheet cells.

In contrast, although modal detection only successfully handles modal dialogs, the agent often fails in subsequent steps of the interaction. This limitation explains why modal detection alone does not substantially improve overall task success.

7 Conclusion

In this paper, we propose A11y-Compressor, a framework for constructing compressed observation representations for GUI agents. The framework transforms linearized a11y trees into a structured and compact representation, referred to as Compressed-a11y, by incorporating region segmentation and structural organization. Experimental results on the OSWorld benchmark demonstrate that the proposed framework significantly reduces the number of input tokens to approximately 22% of those required by the linearized a11y tree while improving task success rates across many application domains, achieving an average gain of 5.1 percentage points. In particular, for applications with complex user interfaces, A11y-Compressor effectively suppresses redundant accessibility information while preserving the structural context necessary for successful task execution. Future work will explore the applicability of the proposed representation to closed-MLLMs with larger capacity and stronger reasoning capabilities, as improvements with local MLLMs suggest similar gains.

8 Limitations

Although A11y-Compressor is designed as a general framework for generating compressed GUI observation representations, several limitations remain in the current implementation. The framework operates on the linearized a11y tree as its primary input representation. Consequently, A11y-Compressor cannot directly utilize visual information that is not represented in the accessibility tree, such as icon shapes, colors, or other purely visual cues. This limitation may reduce the effectiveness of the framework in tasks where such visual information plays a critical role in identifying UI elements. In addition, our evaluation is conducted on representative desktop applications in the OSWorld benchmark. The applicability of the framework to other environments, such as mobile interfaces or different application ecosystems, has not yet been investigated. Furthermore, the Compressed-a11y representation used in our empirical evaluation is implemented using rule-based procedures. As a result, several design choices, including threshold values, rely on heuristics, which may limit the robustness and generalizability of the current implementation across diverse interface settings. Future work could extend A11y-Compressor by incorporating more flexible compression strategies, potentially including learning-based approaches, and by evaluating its effectiveness across a wider range of interface environments.

References

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, Wenbin Ge, Zhifang Guo, Qidong Huang, Jie Huang, Fei Huang, Binyuan Hui, Shutong Jiang, Zhaohai Li, Mingsheng Li, and 45 others. 2025. Qwen3-VL Technical Report. https://doi.org/10.48550/arXiv.2511.21631.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. 2024. SeeClick: Harnessing GUI Grounding for Advanced Visual GUI Agents. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Boshi Stevens, Samuel andus, Huan Sun, and Yu Su. 2023. Mind2Web: Towards a Generalist Agent for the Web. In *Advances in Neural Information Processing Systems (NeurIPS)*.

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. 2024. CogAgent: A Visual Language Model for GUI Agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Imene Kerboua, Sahar Omidi Shayegan, Megh Thakkar, Xing Han Lù, Massimo Caccia, Véronique Eglin, Alexandre Aussem, Jérémy Espinas, and Alexandre Lacoste. 2025. LineRetriever: Planning-Aware Observation Reduction for Web Agents. https://doi.org/10.48550/arXiv.2507.00210.

Kevin Qinghong Lin, Linjie Li, Difei Gao, Zhengyuan Yang, Shiwei Wu, Zechen Bai, Stan Weixian Lei, Lijuan Wang, and Mike Zheng Shou. 2025. ShowUI: One Vision-Language-Action Model for GUI Visual Agent. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Runliang Niu, Jindong Li, Shiqi Wang, Yali Fu, Xiyu Hu, Xueyuan Leng, He Kong, Yi Chang, and Qi Wang. 2024. ScreenAgent: A Vision Language Model-driven Computer Control Agent. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence (IJCAI)*.

Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-Agent-v2: Mobile Device Operation Assistant with Effective Navigation via Multi-Agent Collaboration. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Qianhui Wu, Kanzhi Cheng, Rui Yang, Chaoyun Zhang, Jianwei Yang, Huiqiang Jiang, Jian Mu, Baolin Peng, Bo Qiao, Reuben Tan, Si Qin, Lars Liden, Qingwei Lin, Huan Zhang, Tong Zhang, Jianbing Zhang, Dongmei Zhang, and Jianfeng Gao. 2025. GUI-Actor: Coordinate-Free Visual Grounding for GUI Agents. In *Advances in Neural Information Processing Systems (NeurIPS)*.

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. 2024. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. In *Advances in Neural Information Processing Systems (NeurIPS)*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 Technical Report. https://doi.org/10.48550/arXiv.2505.09388.

Chaoyun Zhang, Liqun Li, Shilin He, Xu Zhang, Bo Qiao, Si Qin, Minghua Ma, Yu Kang, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and

Qi Zhang. 2025a. UFO: A UI-Focused Agent for Windows OS Interaction. In *Proceedings of the Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*.

Chi Zhang, Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Han. 2025b. AppAgent: Multimodal Agents as Smartphone Users. In *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems (CHI)*.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. 2024. GPT-4V(ision) Is a Generalist Web Agent, If Grounded. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xian Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

A Overview of Temporal Modal Detection

This section provides a detailed description of our implementation of the modal detection phase in A11y-Compressor. In our implementation, we employ two complementary strategies: temporal-difference-based detection and keyword-based detection.

The temporal-difference-based method serves as the primary approach when consecutive observations correspond to the same screen. In contrast, the keyword-based method is applied in situations where temporal correspondence is unavailable, such as the initial observation or during screen transitions.

We first describe the temporal-difference-based detection method. We adopt a three-stage pipeline consisting of: (1) temporal UI correspondence and same-screen identification, (2) modal candidate extraction based on temporal differences, and (3) rule-based modal validity scoring and decision.

Given two consecutive observations, we first determine whether they correspond to the same screen. If so, newly appeared UI elements are treated as modal candidates and evaluated using a rule-based scoring scheme.

A.1 Temporal UI Correspondence

Let $U^t = \{u_i^t \mid i = 1, 2, \ldots, N_u^t\}$ denote the set of UI elements observed at time step t, where N_u^t is the number of elements. For each UI element u_i^t, we denote its position vector and semantic content (e.g., tag, name, text, class, and description) as p_i^t and c_i^t, respectively.

To establish correspondence across consecutive observations, we define a semantic matching operator that pairs elements with identical semantic content:

$$\Gamma^t(U_A, U_B) = \{(u_j^{t-1}, u_k^t) \mid u_j^{t-1} \in U_A,\ u_k^t \in U_B,\ c_j^{t-1} = c_k^t\}. \quad (5)$$

A.2 Region-Aware Matching

UI elements exhibit different temporal behaviors depending on the region type. We therefore partition UI elements into static and dynamic regions. Let U_{STA}^t and U_{DYN}^t denote UI elements in static and dynamic regions, respectively. Details of region detection are described in Section 3.3.

Static Regions. UI elements in static regions are expected to remain spatially stable across consecutive observations. We define the static matching indicator:

$$\mu_{\mathrm{STA}}^t(u_j^{t-1}, u_k^t) = \begin{cases} 1, & \|p_j^{t-1} - p_k^t\| \le \epsilon_{\mathrm{STA}}, \\ 0, & \text{otherwise.} \end{cases} \quad (6)$$

Here, ϵ_{STA} accounts for positional noise.

Dynamic Regions. In dynamic regions, UI elements may shift due to scrolling or viewport changes. To compensate for this, we estimate a global displacement vector.

For each matched pair, we define:

$$\Delta p_{j,k}^t = p_k^t - p_j^{t-1}. \quad (7)$$

The global translation is then estimated as:

$$\Delta p_{\mathrm{GLB}}^t = \mathrm{median}\big(\{\Delta p_{j,k}^t\}\big). \quad (8)$$

Using this, the dynamic matching indicator is defined as:

$$\mu_{\mathrm{DYN}}^t(u_j^{t-1}, u_k^t) = \begin{cases} 1, & \|p_j^{t-1} + \Delta p_{\mathrm{GLB}}^t - p_k^t\| \le \epsilon_{\mathrm{DYN}}, \\ 0, & \text{otherwise.} \end{cases} \quad (9)$$

A.3 Same-Screen Identification

We determine whether two consecutive observations correspond to the same screen using a matching ratio over dynamic-region elements:

$$R^t = \frac{1}{|U_{\mathrm{DYN}}^{t-1}|} \sum_{(u_j^{t-1}, u_k^t) \in \Gamma^t(U_{\mathrm{DYN}}^{t-1}, U_{\mathrm{DYN}}^t)} \mu_{\mathrm{DYN}}^t(u_j^{t-1}, u_k^t). \quad (10)$$

If R^t exceeds a predefined threshold, the two observations are regarded as belonging to the same screen.

Rationale for the Denominator. We use U_{DYN}^{t-1} as the denominator to verify whether previously visible UI elements persist. Using U_{DYN}^t would cause large modal overlays to artificially reduce the matching ratio.

Threshold Parameters.

- **Position tolerances ($\epsilon_{\mathrm{STA}}, \epsilon_{\mathrm{DYN}}$):** 25 pixels.
- **Matching ratio threshold:** 0.3.

Exception Handling.

- **Large modal handling:** If the number of matched elements exceeds 10, the screen is regarded as identical regardless of R^t.
- **Sparse screen handling:** If $|U^t| < 15$, we bypass same-screen judgment and proceed directly to modal detection.

A.4 Temporal Difference-based Candidate Extraction

If two observations are determined to belong to the same screen, modal candidates are extracted as newly appeared UI elements.

$$M^t = \Big\{ u_i^t \in U^t \;\Big|\; \forall u_j^{t-1} \in U_{\text{DYN}}^{t-1},\ \mu_{j,i}^{\text{DYN},t} = 0 \\ \wedge\, \forall u_j^{t-1} \in U_{\text{STA}}^{t-1},\ \mu_{j,i}^{\text{STA},t} = 0 \Big\}. \quad (11)$$

These elements represent UI components that newly emerge at time step t.

A.5 Modal Validity Scoring

Let M^t denote a modal candidate set and $m \in M^t$ an individual UI element. We define the total modal score as:

$$M_{\text{total}} = \sum_{m \in M^t} \Big(M_{\text{tag}}(m) + M_{\text{name}}(m) \Big) + M_{\text{count}}(M^t). \quad (12)$$

A.5.1 Tag-based Score

$$M_{\text{tag}}(m) = \begin{cases} +2.0 & \text{if tag}(m) \in \mathcal{R}_{\text{interactive}}, \\ -0.5 & \text{if tag}(m) \in \mathcal{R}_{\text{decorative}}, \\ 0.0 & \text{otherwise.} \end{cases} \quad (13)$$

$\mathcal{R}_{\text{interactive}}$ = {dialog, alertdialog, menu, listbox, tree} $\mathcal{R}_{\text{decorative}}$ = {image, label, heading, paragraph, generic}

A.5.2 Name-based Score

$$M_{\text{name}}(m) = \begin{cases} w_{\text{decide}}, & \text{if is\_interactive}(m) = 1 \\ & \quad \wedge\, \text{name}(m) \cap \mathcal{K}_{\text{decide}} \neq \emptyset \\ w_{\text{func}}, & \text{if is\_interactive}(m) = 1 \\ & \quad \wedge\, \text{name}(m) \cap \mathcal{K}_{\text{func}} \neq \emptyset \\ 0.0, & \text{otherwise.} \end{cases} \quad (14)$$

$\mathcal{K}_{\text{decide}}$ = {OK, Cancel, Save, Yes, No, Login, Agree, Delete} $\mathcal{K}_{\text{func}}$ = {Sort, Filter, Settings, Search, Find}

A.5.3 Cardinality-based Correction

$$M_{\text{count}}(M^t) = \begin{cases} -3.0 & \text{if } |M^t| < 3 \\ & \quad \wedge\, \forall m \in M^t,\ M_{\text{tag}}(m) \leq 0 \\ +1.0 & \text{if } |M^t| \geq 6 \\ 0.0 & \text{otherwise.} \end{cases} \quad (15)$$

A.6 Final Decision Rule

A modal candidate is accepted as a valid modal if:

$$\text{is\_modal}(M^t) = \begin{cases} 1 & \text{if } M_{\text{total}} \geq T_{\text{modal}}, \\ 0 & \text{otherwise.} \end{cases} \quad (16)$$

In all experiments, we set $T_{\text{modal}} = 1.0$.

B Keyword-Based Modal Detection

We next describe the keyword-based detection method. When temporal correspondence is unavailable, we apply this method as a complementary strategy to identify modal elements. Typical examples include cookie consent banners, login dialogs, and informational pop-ups.

This method follows a three-stage pipeline: (1) anchor extraction based on keywords, (2) spatial clustering of anchor elements, and (3) region-level scoring and decision.

B.1 Search Region and Anchor Extraction

Since modals appearing in initial states tend to emerge in characteristic regions depending on the application domain, we first define a heuristic search region. Let $O = \{o_i \mid i = 1, 2, \ldots, N_o\}$ denote the set of UI elements within this region. For each element o_i, we denote its screen position vector as p_i.

We extract anchor elements based on predefined keyword sets. An element is regarded as an anchor if its textual content contains any keyword in K. The resulting anchor set is denoted as $A \subset O$.

Keyword Sets. We define two types of keywords:

- **Content Keywords (K_{content}):** `cookie`, `cookies`, `gdpr`, `privacy`, `consent`
- **Action Keywords (K_{action}):** `accept`, `agree`, `allow`, `reject`, `save`, `confirm`, `close`, ×, `ok`, `policy`, `manage`, `setting`

B.2 Spatial Clustering of Anchor Elements

To construct modal candidate regions, we group anchor elements based on spatial proximity.

Two anchor elements $a_j, a_k \in A$ are considered connected if:

$$\|p_j - p_k\| < \delta, \quad (17)$$

where the distance threshold δ is defined as:

$$\delta = 0.08 \cdot \min(W, H), \quad (18)$$

with W and H denoting the screen width and height. Connected components formed under this criterion are treated as modal candidate regions.

B.3 Edge-Based Immediate Detection

To efficiently detect cookie banners and similar notifications, we introduce a geometric shortcut for edge-aligned regions.

A candidate region is immediately classified as a modal if it satisfies:

- **Vertical Position:**

$$y > 0.75H \quad \vee \quad y < 0.15H \tag{19}$$

- **Aspect Ratio:**

$$\frac{w}{h} > 2.5 \tag{20}$$

These conditions capture horizontally elongated regions located at the top or bottom of the screen, which are typical for consent banners.

B.4 Region-level Scoring

For candidate regions not detected by the edge-based rule, we evaluate modal validity using a composite score:

- **Anchor Count Score:** Based on the number of anchor elements (capped at 20).
- **Centrality Score:** Based on distance from the screen center (maximum 30 points).
- **Structural Score:** Based on the presence of interactive UI components such as buttons, input fields, toggles, or close icons.

The total score is denoted as S_{total}.

B.5 Decision Rule

A candidate region is classified as a modal if:

$$S_{\text{total}} \geq 65.0 \quad \wedge \quad S_{\text{total}} \geq 0.8 \times S_{\max}, \tag{21}$$

where $S_{\max}$ is the maximum score among candidates in the current frame.

B.6 Rejection Criteria

Even if a candidate satisfies the scoring conditions, it is rejected if:

- The number of anchor elements is too small.
- The region area is too small.
- The region corresponds to a navigation bar or search form.
- The region lacks a clear closing or cancel mechanism.
- The region covers most of the screen, indicating a full page transition.

C Redundancy Reduction Process

This section provides a detailed description of our implementation of the redundancy reduction phase in A11y-Compressor. We apply rule-based preprocessing steps, including UI element merging, attribute compression, and dynamic selection, to all UI elements in order to enhance the information density of the observation representation while improving MLLM inference efficiency. Table 3 summarizes these preprocessing steps. Further details on visual and semantic deduplication, as well as instruction-aware filtering, are provided in C.1 and C.2.

C.1 UI Element Deduplication

To reduce the inference load on the MLLMs, we implement a preprocessing step that merges spatially and semantically overlapping UI elements. This process eliminates redundant information while preserving interactive components. The specific logic is implemented as follows.

Deduplication Criteria. Two UI elements are considered "duplicate candidates" if they satisfy both of the following conditions:

- **Spatial Proximity:** We calculate the center coordinates (c_x, c_y) of the bounding box for each element and measure the Euclidean distance between them. If this distance is within a predefined threshold (default: 20.0 pixels), the elements are judged to be in close proximity. *Exception:* If the `name` attributes match exactly, we relax the condition to allow a vertical deviation (y-axis) of up to 30 pixels, tolerating larger horizontal deviations.
- **Semantic Similarity:** We compare the `name` attributes after normalization (lowercase conversion and whitespace removal). Elements are considered semantically similar if there is an exact match or if one string is a substring of the other. *Prevention of Over-merging:* To prevent incorrect merging, if the lengths of the labels differ significantly (e.g., one is more than twice the length of the other), we regard them as distinct meanings and skip the merger.

Tag Priority Strategy. When two elements are identified as duplicate candidates, we prioritize retaining the more interactive element. We define a priority score based on the element's `tag` (lower values indicate higher priority), as shown in Table 4.

Processing Step	Motivation	Processing Overview
Rule-based Noise Removal	To eliminate non-essential hidden elements and OS metadata that bloat the MLLM input context without contributing to task completion.	Filters out background desktop elements and system-level metadata using heuristic rules.
Visual & Semantic Deduplication	To resolve redundancy in the linearized a11y tree where a single UI element is represented by multiple tags, reducing inference load.	Merges spatially overlapping or identical elements, prioritizing interactive tags (e.g., buttons) over static containers.
Attribute Selection & Coordinate Compression	To simplify decision-making by removing excessive geometric details that induce unnecessary reasoning.	Converts bounding boxes (x, y, w, h) into center coordinates (c_x, c_y) and retains only essential attributes (tag, name).
String Normalization	To prevent matching failures caused by inconsistent formatting, whitespace, or unnecessary line breaks.	Normalizes whitespace and removes redundant newlines in attribute values to ensure consistency.
Instruction-based Dynamic Filtering	To avoid distracting the agent with long, irrelevant text paragraphs unrelated to the current task.	Dynamically truncates paragraph text based on keywords extracted from user instructions, retaining only relevant segments.

Table 3: Overview of preprocessing steps for observation space reduction.

Priority	Score	Tags	Rationale
Highest	0	`entry`, `combo-box`, `check-box`, `radio-button`, `toggle-button`, `input`	Elements where users directly input or modify values; crucial for operation.
High	10	`push-button`, `link`, `menu-item`, `button`	Important interactive elements that trigger actions or navigation.
Medium	20	`heading`	Indicates content structure; prioritized over simple static text.
Low	30	`static`, `image`, `group`, others	Elements for information display only; removed when overlapping with interactive elements.

Table 4: Tag Priority for UI Element Preservation. Lower scores indicate higher priority, while higher scores correspond to elements that are more likely to be removed.

Execution Logic. For each pair of duplicate candidates, we execute the merge according to the following rules:

- **Priority Comparison:** Based on the scores in Table 4, the element with the higher priority (lower score) is retained, and the lower priority element is removed.

  Example: If a `push-button` "OK" (Score 10) overlaps with a `static` "OK" (Score 30), the `push-button` is retained.

- **Special Exception:** For a pair consisting of a `link` and a `static` element, the `link` is forcibly prioritized.

- **Tie-breaking:** If both elements have the same priority score (e.g., both are `static`), the element with the longer label string is retained to preserve more information.

C.2 Paragraph Compression

Long text passages with low relevance to the user's task instruction act as noise that increases the inference load on the agent. To address this, we introduce a method to dynamically summarize and filter long content, such as UI elements containing the `Paragraph` tag, based on the instruction content.

Preprocessing and Keyword Extraction. We apply the following normalization steps to both the user instruction and the target text:

1. **Normalization:** Convert all strings to lowercase.

2. **Tokenization:** Replace non-alphanumeric characters with whitespace and split the string into a list of words.

3. **Filtering:** Remove words found in the general stop-word list ($\mathcal{S}_{\text{stop}}$) and extract only words with a length of 2 or more characters to construct the keyword set L.

The stop-word list $\mathcal{S}_{\text{stop}}$ used in this process is defined in Table 5. It includes general function words, task-specific conversational fillers, and common generic UI terms.

Category	Words
General Function Words	the, a, an, in, on, at, to, for, of, with, by, from, is, are, am, be, this, that, it
Task Expressions	please, can, could, would, you, i, my, me, need, want, try, make, let
UI Operations & Generic Nouns	click, tap, press, hit, select, choose, open, go, browse, navigate, find, search, check, uncheck, button, link, tab, menu, window, page, website, site, input, enter, type, fill, text, box, field

Table 5: List of Stop Words for Keyword Extraction.

Context Extraction Based on Keywords. We search the target text (e.g., `Paragraph` content) for any words contained in the keyword set L.

- **Match Found:** When a keyword is found, we identify the index of its first occurrence. We then extract a window of a fixed number of characters (default: 50 characters) before and after the keyword, formatting the result as "`... [extracted text] ...`". This preserves the context relevant to the instruction while reducing the overall length.

- **No Match:** If no words from the instruction are found in the text, we retain only the first $N_{\max}$ characters (default: 100 characters) and truncate the rest (e.g., "`First 100 chars...`").

This dynamic filtering enables the agent to avoid overlooking critical information relevant to the instruction while preventing context overflow and increased inference costs caused by lengthy texts.

D Details of Semantic Structuring and Region Segmentation

This section provides our implementation of the semantic structuring phase of A11y-Compressor, including region segmentation and domain-specific optimizations.

D.1 Element Reordering.

UI elements are reordered based on their center coordinates (c_x, c_y), as the order in a linearized a11y tree does not necessarily reflect the visual layout. Elements are sorted primarily from top to bottom (Y-axis) and secondarily from left to right (X-axis), producing a sequence aligned with the visual reading order.

D.2 Region Segmentation.

We classify UI elements into predefined semantic regions using coordinate information, element tags, and application-specific heuristics. Each region represents a high-level functional unit (e.g., `CONTENT`, `MODAL`) and is associated with structural and interaction-related properties.

D.3 Intra-region Structuring.

Within each region, UI elements are structured using inter-element spatial distances and heading tags. A special token `[BLOCK]` is inserted when the distance between adjacent elements exceeds a threshold Θ, enabling the encoding of two-dimensional layout information into a linear sequence.

D.4 Domain-specific Optimization.

For application domains with high information density, we apply additional token-efficiency optimizations. For example, in spreadsheet applications (e.g., LibreOffice Calc), we retain only value-containing cells, header cells, and instruction-relevant cells. These elements are grouped using row and column indices to reconstruct a structured representation that preserves the tabular layout.

D.4.1 Google Chrome

For web browsers, we strictly separate the page content from the browser's native UI.

Region Definitions:

- `BROWSER_TABS`: The tab area at the top ($Y <$ 150px) containing specific anchors like "new tab" or "close".

- `ADDRESS_BAR`: The area containing the URL bar and navigation buttons (Back, Reload, etc.) ($Y <$ 110px).

- `BOOKMARK_BAR`: The bookmark area located directly below the address bar (110px $< Y <$ 150px).

- `PAGE_CONTENT`: The main content area of the rendered web page.

D.4.2 VS Code

To handle the complex pane structure characteristic of IDEs, we perform detailed region segmentation based on precise coordinate thresholds.

Region Definitions:

- APP_LAUNCHER: The OS launcher area on the far left ($X \leq 5\%$).
- MENUBAR: The top menu bar ($Y \leq 12\%$).
- ACTIVITY_BAR: The icon bar on the left side ($2\% \leq X \leq 8\%$).
- SIDE_BAR: The side panel containing the file explorer, etc. ($X \leq 30\%$).
- TAB_BAR: The editor tab area ($7\% \leq Y \leq 16\%$).
- BREADCRUMB: The breadcrumb list directly below the tabs ($10\% \leq Y \leq 18\%$).
- STATUSBAR: The status bar at the bottom ($Y \geq 96\%$).
- CONTENT: The main editor text area (regions other than the above).

D.4.3 Thunderbird

As a mail client, Thunderbird requires advanced processing that dynamically switches segmentation logic depending on the active View (e.g., the 3-pane "Mail View" vs. the "Settings View").

Basic Region Definitions:

- SPACES_BAR: The function switching bar on the far left ($X < 115$px).
- FOLDER_TREE: The mail folder tree structure ($115\text{px} \leq X < 400\text{px}$).
- MESSAGE_LIST: The list of emails (center of screen, $X < 55\%$).
- PREVIEW: The email body preview pane (right side, $X \geq 55\%$).
- TOOLBAR: The search bar and operation buttons at the top.

View-Specific Logic:

- **Mail View:** The output is structured as a 3-part split (Folder List, Mail List, Preview). The boundary of the mail list (SPLIT_MSG_LIST_X) is dynamically estimated from the element layout.
- **Settings View:** When a settings screen is detected, it is split into a "Settings Category (Sidebar)" on the left and "Settings Items (Main)" on the right. Furthermore, we apply a process to simplify "off-screen items" based on the scroll position.

D.4.4 GIMP

This configuration handles the multi-window and docking interface typical of image editing software.

Region Definitions:

- TOOLBOX: The toolbox area on the left ($X < 22\%$).
- DOCKS: The dock area (layers, brushes, etc.) on the right ($X > 78\%$).
- CANVAS: The central image editing area.
- MENUBAR: The top menu ($Y < 10\%$).
- STATUSBAR: The bottom information bar ($Y > 95\%$).

D.4.5 LibreOffice Suite

While sharing a common framework, specific regions are defined for Calc, Impress, and Writer respectively, and when region definitions overlap with the common regions, the application-specific definitions take precedence.

Common Semantic Regions:

- MENUBAR: The top menu strip containing "File", "Edit", etc. ($Y < 10\% \sim 20\%$).
- TOOLBAR: The area immediately below the menubar containing buttons and combo boxes ($Y < 25\%$).
- STATUSBAR: The information bar at the bottom of the window ($Y > 90\% \sim 96\%$).

LibreOffice Calc:

- FORMULA_BAR: Area below the menu containing the formula bar ($9\% < Y < 23\%$).
- SHEET: The spreadsheet cell area.
- SHEET_TABS: Sheet switching tabs at the bottom ($93\% < Y < 96\%$).

LibreOffice Impress:

- SLIDE_LIST: Slide overview on the left ($X < 20\%$).

- PROPERTIES: Property panel on the right ($X > 80\%$).
- CONTENT: The central slide editing view.

LibreOffice Writer:

- CONTENT: The central document editing area.
- PROPERTIES: The properties sidebar (if active).

D.4.6 VLC Media Player

Due to its simple UI structure, we apply a basic vertical split.

Region Definitions:

- MENUBAR: Top menu ($Y < 10\%$).
- TOP_BAR: Toolbar area ($Y < 20\%$).
- CONTENT: Video playback screen or playlist.
- STATUSBAR: Playback controls and seek bar at the bottom ($Y > 92\%$).

D.4.7 OS (Ubuntu)

Since this handles the entire desktop environment, segmentation is performed without reliance on specific window frames.

Region Definitions:

- TOP_BAR: System bar at the very top ($Y < 5\%$).
- APP_LAUNCHER: Dock/Launcher on the far left ($X < 6\%$).
- DESKTOP_ICONS: Desktop icon arrangement recognized as a grid.
- OS_POPUP: Right-click menus or dialogs on the desktop.
- **Window Detection:** We dynamically identify window regions on the screen by pairing labels with input forms and analyzing the placement of "Close/Minimize" buttons, treating each as an independent context.

D.5 Determination of the Threshold Θ

To adaptively handle varying information densities across different applications, we dynamically determine the block segmentation threshold Θ for each screen. First, we estimate a base vertical gap, denoted as G_{base}, which represents the standard line height or margin of the interface. Based on this estimate, we then select an optimal multiplier to derive the final threshold Θ used for block segmentation.

D.5.1 Estimating the Base Gap

We begin by computing the vertical distances between all adjacent UI elements in the content area. Let dy denote the vertical distance between two neighboring elements. To avoid the influence of large structural gaps, such as margins separating distinct sections, we focus only on the lower 70% of the resulting distance values. The base vertical gap, denoted as G_{base}, is then defined as the median of this subset. Finally, G_{base} is clamped to a minimum value to prevent excessive sensitivity to small spacing variations:

$$G_{base} = \max(\text{median}(dy_{0..70\%}), 40 \text{ pixels}).$$

D.5.2 Adaptive Threshold Selection

The segmentation threshold is defined as $\Theta = G_{base} \times N$. We select the multiplier N adaptively from a candidate set $\{3.0, 4.0, 8.0\}$ to balance structural grouping against fragmentation.

The algorithm iteratively tests these multipliers starting from the strictest value (3.0, which yields the lowest threshold). Candidate Θ is rejected if it causes **over-segmentation**, defined by the following criteria:

- The total number of blocks exceeds 50.
- The total number of blocks exceeds 10, and more than 50% of them contain only a single UI element (indicating that the threshold is too low and fragments otherwise coherent lines).

The final Θ is set to the first multiplier N that satisfies these stability conditions, or to the largest multiplier (8.0) if all stricter options fail. This ensures that Θ remains robust against varying layout densities, grouping related elements while correctly separating distinct logical blocks.

E Case Study: LineRetriever

With LineRetriever (Kerboua et al., 2025), the subset of elements extracted by the retriever model caused the agent to be unaware of the modal dialog, leading it to attempt interactions with background elements. However, because the active modal blocked background operations, these attempts resulted in no visual changes on the screen. Consequently, the model switched its strategy from mouse interactions to keyboard operations in an attempt to focus on the target field. This inadvertently shifted the focus to the browser's address bar,

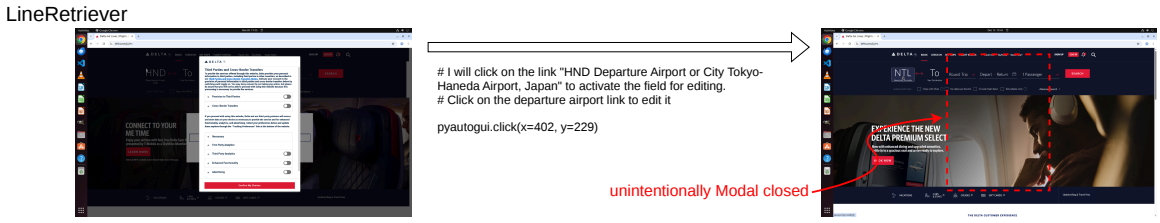

Figure 5: Example of modal dialog handling with LineRetriever. In this instance, the model failed to perceive the active modal due to the limited extraction range of the retriever model. When background mouse interactions failed, the model's fallback to keyboard navigation inadvertently reloaded the page, which closed the modal unintentionally.

triggering a page reload. As a result, the modal was closed unintentionally.