Enhancing UI Element Recognition with a Dual-System Approach through Dynamic Code Generation

Anonymous ACL submission

Abstract

Grounding is central to AI agents on smartphones and requires recognizing relevant UI elements on graphical interfaces. However, existing grounding methods typically prioritize either efficiency or accuracy, and struggle to balance both under real-world UI variations. To address this challenge, we adopt a dual-system approach: System 1 efficiently recognizes UI elements using predefined rules, while System 2 provides deeper analytical reasoning when System 1 fails. To bridge the two systems, we propose **GroundCoder**, a multi-agent system that extracts representative UI features (e.g., visual appearance and layout) based on System 2's reasoning and generates executable code for System 1. The generated code transfers System 2's analytical capabilities to System 1 by encoding them as executable rules, enabling fast and efficient recognition of UI elements beyond predefined patterns. To systematically evaluate our approach, we construct Eleva, a dataset of UI elements collected from popular mobile applications, covering diverse devices, display modes, and application modes. Experiments on Eleva show that our method preserves efficiency comparable to rule-based methods while improving recognition accuracy by **34.6%** over existing mainstream methods. We further discuss implications for using generative code in UI recognition to support more robust grounding in dynamic mobile environments.

1 Introduction

Grounding on GUIs is a fundamental capability for GUI agents. The efficiency and accuracy of recognizing the appropriate UI elements based on task requirements are crucial to the performance of AI agents (Cangelosi, 2010; Zheng et al., 2024).

Prior AI agents rely on hard-coded rules, or local models to recognize UI elements. While effective in stable environments, these methods struggle with rich UI variations (Huang et al., 2023). While LLM-based methods improve the generalization ability, they typically require significant computational resources, suffer from efficiency issues (Wang et al., 2023a), and display instability (Huang et al., 2025), making them impractical for real-time scenarios. Thus, a critical challenge remains: how to enable AI agents to achieve both efficiency and accuracy in UI element recognition?

To address this, we emulate human UI recognition strategies that balance efficiency and accuracy via a dual-process mechanism (Sharp et al., 2007). Followed by Kahneman's theory (Kahneman et al., 2002), our approach employs System 1, which leverages prior experience for rapid feature extraction (e.g., expecting search bars at the top), and System 2, which performs deliberate semantic refinement to resolve ambiguities. This hierarchical approach enables rapid engagement while maintaining robust understanding.

To maximize both efficiency and accuracy, we introduce **GroundCoder**, a multi-agent system that acts as a bridge between System 2 and System 1, as shown in Figure 1. GroundCoder leverages System 2's advanced LLM intelligence to dynamically generate and update UI recognition code for System 1's efficient recognition. When new UI variations are encountered, GroundCoder can update the existing code using new cases, continuously enhancing System 1's capabilities and improving the overall system's efficiency and accuracy.

GroundCoder is designed to abstract representative features from limited samples and use them to generate executable code. It leverages multi-agent collaboration and multimodal LLMs to select useful features from numerous low-level features from UI images and layouts, and high-level features generated through lightweight models and tools. Additionally, to ensure the effectiveness of the generated code, GroundCoder relies on multiple agents for code inspection and analysis.

In this work, we demonstrate how to combine

human cognitive strategies, large model reasoning, and generative code methods to create an adaptive system that efficiently recognizes UI elements while flexibly responding to UI variations. Our work contributes valuable insights into AI agents' grounding, providing a potential pathway for the trade-off between efficiency and accuracy.

This paper makes the following contributions:

- We introduce Eleva [1], a highly targeted dataset that contains 10 popular applications and 3,182 classes of UI elements, with a total of 32,807 samples distributed across these classes. These samples span diverse UI variations across multiple devices, apps and display modes, and have been carefully filtered to include challenging cases that existing static rules cannot fully cover.
- We propose a UI recognition method that combines a dual-system approach with dynamic code generation, inspired by human cognitive strategies. By leveraging System 1's fast code execution and System 2's LLM-based deep analysis, we provide a solution that balances efficiency and accuracy. The dynamic generation and updating of recognition code underpin System 1's operation, ensuring continuous optimization of the entire system.
- We implement our dual system powered by GroundCoder. Evaluation on the Eleva dataset demonstrates that our system improves recognition accuracy by over 34.6% compared to mainstream methods, while maintaining efficiency comparable to rule-based methods. Based on these experimental insights, we summarize several points for further optimizing code generation tailored to UI recognition.

2 Task Formulation and Dataset Construction

Building on prior work((Deka et al., 2017a; Chen et al., 2021)), we formalize our task and construct the corresponding dataset as follows:

2.1 Task Formulation

Given the requirements for AI agents grounding on GUIs, our task is to recognize UI elements on a GUI page G that match a given semantic label. Let $\{e_1, e_2, \ldots, e_n\}$ represent the set of UI elements in G, where each e_i has a corresponding semantic label $\sigma(e_i)$. Given an input query s, the goal is to identify the e_i such that: $\sigma(e) \approx s, e \in G$.

[1] https://anonymous.4open.science/r/Eleva-C751

Most existing approaches relying on large multimodal models address this task by mapping semantics to UI elements (Song et al., 2024; Wang et al., 2023a), effectively simulating the inverse mapping $\hat{\sigma}^{-1}$ in real-time, and identifying $\hat{e} = \hat{\sigma}^{-1}(s)$. However, these methods require significant computational resources and are prone to instability issues (Wang et al., 2023b).

To address this, our approach is as follows.

For all known UI elements, we record their corresponding semantic labels, forming a semantic label set $S = \{s_1, s_2, \ldots, s_m\}$. For each semantic label s_i, all UI elements e that satisfy $\sigma(e) = s_i$ are considered to belong to the same class, and they share a common recognition code R_{s_i}. This recognition code is generated using efficient features and rules, supporting fast execution. The recognition condition is satisfied only when $\sigma(e) = s_i$, i.e., $R_{s_i}(e) = \text{true}$ if and only if $\sigma(e) = s_i$.

Thus, for the given task, we first find $\hat{s}$ such that $\hat{s} \approx s$ and $\hat{s} \in S$. Then, we enumerate all UI elements in G and identify the one for which $R_{\hat{s}}(\hat{e}) = \text{true}$, and $\hat{e}$ is the recognized UI element.

2.1.1 Handling UI Element Classification

The concept of semantic similarity among UI elements is critical for this task. While prior studies define UI elements of the same class based on static attributes such as visual appearance or metadata (Hao et al., 2014; Li et al., 2017), we take a more functional approach. In this paper, two UI elements are classified as belonging to the same class if they serve the same role in executing a task operation. Figure 2 illustrates several UI elements labeled as "share." Although their appearances differ, they are semantically the same class, performing the same function in the task.

From a UI grounding perspective, a complete task involves a sequence of operations. UI elements triggered by the same operation in a task are considered to belong to the same class, even across different UI variations. However, if a UI update significantly changes the sequence of operations required to complete the task, the affected elements may no longer be classified in the same class.

2.1.2 Handling Personalized Parameters

In certain cases, UI elements may exhibit minor variations. For instance, a username field displaying "User A" versus "User B" should still be considered a similar UI element, as both serve the same functional role (e.g., selecting a recipient in a mes-

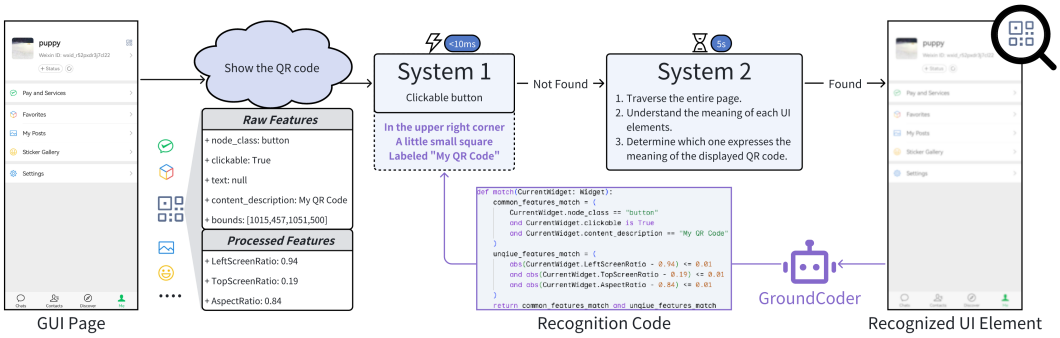


Figure 1: Human-inspired dual-system approach for UI element recognition. The task is to display a QR code, and the system must recognize the corresponding UI element. System 1 focuses on efficiency but cannot identify the element alone, while System 2 performs semantic analysis to locate the correct UI element. GroundCoder extracts efficient features from System 2 to generate executable code that improves System 1's efficiency in future tasks.

saging task). However, the distinguishing text acts as a parameter unique to each instance.

Thus, within our formalization, UI elements of the same class may contain parameterized differences, where the varying text is treated as a specific feature rather than an indicator of a distinct element class. This implies that for UI elements with personalized attributes, the code for recognition requires additional parameters as input for differentiation. Furthermore, these parameters are inherently part of the input semantics.

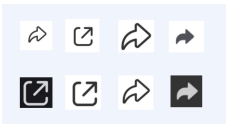


Figure 2: Samples of UI elements with the semantic label "share" across different UI variations.

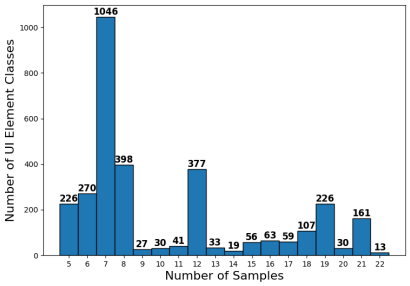


Figure 3: Distribution of the number of samples across different UI element classes.

2.2 Eleva: Data Collection for UI Variations

Given the task formulation, we emphasize the need to recognize UI elements based on semantic labels across different UI variations. However, existing datasets do not meet these requirements. Many available datasets, such as Rico (Deka et al., 2017a) and APM (Zhang et al., 2021), have two main limitations: firstly, their semantic annotations are too coarse to support the level of granularity required for task execution; secondly, they do not focus on UI variations across different versions or states.

To address this gap, we collected Eleva, a dataset that emphasizes UI variations, considering factors including device model, display modes, and app-specific modes. The details are shown in Table 2. These variations reflect the cognitive challenges humans encounter when interacting with UIs in different contexts, providing broader coverage of real-world usage scenarios. Table 3 highlights the main advantages of Eleva compared to existing mainstream datasets, making it more aligned with the task formulation. Due to factors including devices, app settings and page states, the number of samples for each UI element class varies. The specific distribution is shown in Figure 3. The dataset construction involves two main phases: raw data collection and further semantic annotation. The construction details can be found in Appendix A.

3 GroundCoder: Bridging Dual Systems for UI Element Recognition

Inspired by Kahneman's dual-process theory (Kahneman et al., 2002), our approach adopts a dual-system design integrating complementary strengths to achieve both runtime efficiency and adaptability in UI element recognition tasks. Specifically, we define two distinct yet collaborative systems:

- System 1 focuses on efficiency and speed. It performs rapid UI element recognition using

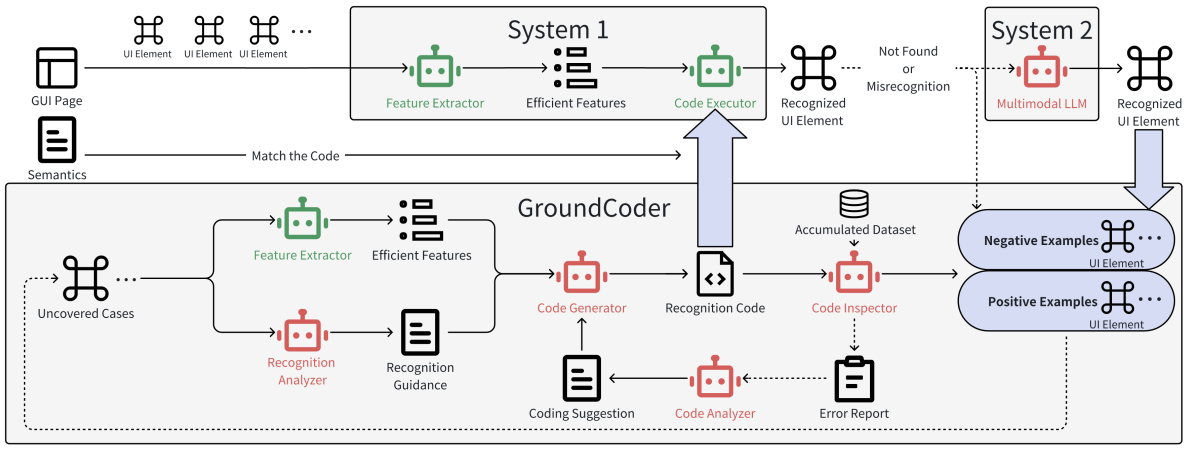


Figure 4: The workflow of our proposed dual system. System 1 uses existing Python code for efficient UI element recognition. If it fails, System 2 applies a multimodal LLM for deeper semantic analysis. The GroundCoder system extracts features from new or incorrect cases, updating the code for future use in System 1. The green agents represent high efficiency, while the red agents, relying on LLMs, are less efficient but more intelligent.

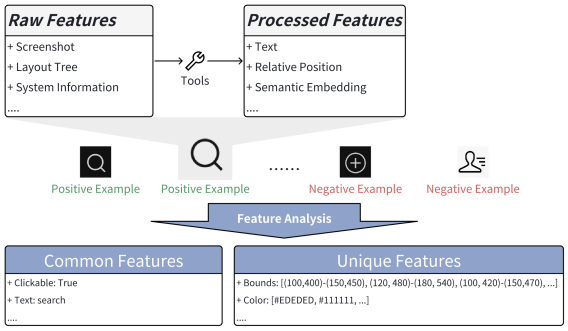


Figure 5: The feature extractor extracts raw features of UI elements and uses extensive tools to generate additional processed features. It also analyzes the features of positive and negative examples, extracting common and unique features from positive examples to assist in the subsequent code generation.

lightweight, dynamically generated code, enabling real-time interaction.

- System 2 emphasizes accuracy and flexibility. When System 1 encounters unfamiliar or complex UI variations, System 2 leverages state-of-the-art multimodal LLMs to perform deeper semantic reasoning and accurately identify target UI elements.

To effectively integrate these two systems, we propose **GroundCoder**, which dynamically extracts insights obtained from System 2 and translates them into efficient executable code for System 1. This process not only allows System 1 to maintain its efficiency but also empowers System 1 with semantic recognition capabilities that were traditionally associated with more computationally intensive systems.

As shown in Figure 4, GroundCoder integrates the two systems (System 1 and System 2) through dynamic code generation and iterative updates. At runtime, System 1 initially attempts to recognize the target UI element using previously generated efficient code. If recognition succeeds, the task is completed immediately. Otherwise, System 2 is activated to conduct deeper semantic analysis using advanced multimodal reasoning. The outcome of the interaction with the recognized element determines its classification: successfully executed actions yield positive examples, whereas failed interactions, indicating a misrecognition, are treated as negative examples. These exapmles are then used by GroundCoder's internal agents to update and refine the recognition code. Through this mechanism, GroundCoder continuously enhances System 1's robustness to UI variations.

To support the generation of recognition code, **GroundCoder** incorporates several specialized agents, each tasked with different responsibilities. The motivation behind these agents lies in three core objectives:

- For code generation, we designed *feature extractor* and *code generator*.
- To gain more generalized recognition insights, we designed *recognition analyzer*.
- To enable iterative optimization, we designed *code inspector* and *code analyzer*.

Each agent serves a distinct role in generating and refining recognition code, ensuring that the system adapts efficiently to diverse UI structures.

3.1 Feature Extractor

The *feature extractor* is responsible for converting raw GUI data into a set of rich, high-efficiency features that facilitate UI element recognition.

As shown in Figure 5, the agent includes the following designs:

Multi-Source Raw Features: The feature extractor collects features from multiple data sources, such as screenshots, layout trees accessed by Accessibility APIs, and real-time system information. This multi-source approach ensures that even if one feature fails due to incomplete or inconsistent data, other features can provide compensatory information. For example, Accessibility API might fail to extract certain UI components if they're not labeled properly (Zhang et al., 2021). In such cases, image-based features can help fill in the gap.

Extensible Processed Features: The feature extractor can use various lightweight tools to generate additional features, such as text information processed via OCR. Some models can generate compact embeddings that capture critical visual features, such as color, shape, and positioning. These embeddings refined into high-level features through unsupervised methods like clustering or dimensionality reduction (Li et al., 2021; Wang et al., 2021; Jiang et al., 2022). Additionally, manually designed heuristic rules can be applied to generate further features, such as calculating the aspect ratio of elements based on absolute coordinates or determining their relative size on the screen. These processed features can be easily extended as more lightweight tools or models are developed.

Feature Analysis for Positive and Negative Examples: The feature extractor can process multiple UI elements of the same class simultaneously, which we consider as positive examples. Additionally, we incorporate negative examples, which are other UI elements on the same page that are not the target element. These negative examples are filtered using a set of heuristic rules to exclude meaningless or invisible elements. The feature extractor extracts features from both the positive and negative examples. To assist subsequent agents in selecting the most representative features for code generation, the feature extractor identifies both shared and unique features. Shared features are employed to differentiate positive examples from negative ones, while unique features ensure the code can handle subtle variations within the same class of UI elements.

3.2 Code Generator

Given rich UI element features, the *code generator* generates executable code for UI element recognition by combining high-efficiency features into instructions. The agent bridges the gap between the reasoning phase and the practical implementation required for UI recognition.

The agent includes the following designs:

Chain of Thought: As shown in Figure 9, the code generator employs a chain of thought approach to guide the LLM in generating recognition code. Firstly, the LLM is tasked with analyzing the semantic meaning of each feature to ensure a comprehensive understanding of the elements. Secondly, the LLM prioritizes the use of relative features over absolute ones, ensuring that the generated code is adaptable across diverse UI variations. Thirdly, the LLM is asked to consider and select the necessary shared features that effectively distinguish the positive elements from the negative elements in the same page. The LLM must also account for unique features within the same class of UI elements, ensuring that the generated code can handle variations by accommodating a suitable range of values.

Parameterization: As mentioned in Section 2.1, some UI elements require contextual parameters to be identified correctly. In such cases, the code's parameters should include not only the UI elements to be verified but also specific parameters relevant to the current recognition task, such as the recipient of a message or the name of a selected item. These parameters are extracted from the input semantics. Figure 6 illustrates an example of recognition code for UI elements with parameters.

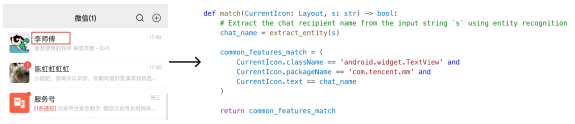


Figure 6: To recognize the "chat recipient" button, the code needs to extract personalized parameters from the input text through entity recognition. The `extract_entity` function should be a predefined function in System 1.

3.3 Recognition Analyzer

The number of samples is much smaller compared to the number of features, making it challenging for the code generator to identify effective, generalizable features from a limited set of samples. To

address this, we designed the *recognition analyzer*, which leverages multimodal LLMs to mimick human understanding of UI elements, guiding the extraction of the most relevant features.

As shown in Figure 9, the recognition analyzer uses the LLM to perform a macro-level analysis of both positive and negative examples, identifying the methods used to distinguish similar from dissimilar elements. It explores the patterns that humans might use to recognize these UI elements, providing valuable insights to assist in generating recognition code.

3.4 Code Inspector

LLMs exhibit a phenomenon of "forgetting," meaning that as the recognition code is continuously updated, it cannot always be guaranteed to work on previously encountered cases. To address this issue, we introduce the *code inspector*, which is responsible for verifying the effectiveness of the code. The code inspector ensures that updates to the code are only finalized when the results for all cases are correct, or when contradictions are handled manually.

The agent includes the following designs:

Batch Testing: Based on the data collected so far, batch testing is performed to validate the recognition code. The testing includes all accumulated UI elements of the target class (positive examples) as well as other UI elements on the same page (negative examples). The batch testing process identifies any "bad cases," which are then fed back into the initial recognition process to update and refine the code. Since the recognition code is efficient, batch testing consumes minimal time and resources.

Error Analysis: The code inspector not only identifies "bad cases" but also provides an analysis of errors, including syntax issues in the code, the rules causing failures, and whether overly loose rules lead to misrecognition. It also checks for contradictions between cases. Based on this, the code inspector generates detailed error reports that help pinpoint root causes, enabling more efficient refinement and faster convergence of the code.

Manual Intervention: If the number of iterations exceeds 5 and the code has not yet converged, manual intervention is requested. This issue may arise when contradictions exist between cases, which can typically be resolved by adding necessary features or by removing outdated cases. Another possibility is that the LLM's ability to refine the code is limited. In such situations, manual intervention involves reviewing and adjusting the recognition code, adding or modifying rules to aid convergence, or providing prompts to guide the LLM in generating more accurate code.

3.5 Code Analyzer

The recognition code needs to be continuously updated as new cases arise, whether discovered in new recognition tasks or identified as bad cases by the code inspector. Therefore, we designed the *code analyzer* to interpret the existing code and assist the code generator in understanding the original rule logic when updating the code. It analyzes errors reported by the code inspector and provides insights for more effective code updates.

As shown in Figure 9, the code analyzer uses the LLM to analyze the current code and the error cases, offering suggestions for code updates. These suggestions are incorporated into the prompt provided to the code generator, aiding in the code update process and accelerating convergence.

4 Evaluation

We evaluate the performance of the dual system powered by GroundCoder in UI element recognition, comparing it with the baseline systems. We also provide an in-depth analysis of the effectiveness of the code generated by GroundCoder.

4.1 System Implementation

We have implemented the dual system powered by GroundCoder, along with several baseline systems, including both efficiency-oriented and adaptability-oriented approaches. As defined in Section 2.1, the input to these systems is a GUI page and a given semantic label, and the output is the UI element corresponding to that label.

Our System: The code generated by GroundCoder is implemented in Python. System 2 utilizes the GPT-4o multimodal model to recognize the UI element. For the feature extractor in both System 1 and GroundCoder, the raw features of UI elements include screenshots, system runtime information, and layout trees accessed via the Accessibility API. Processed features include a set of relative features (such as aspect ratio and boundary positions relative to the screen) and structural features derived from the layout tree (e.g., node depth, subtree size, and path to the root node). In the GroundCoder, the code analyzer and recognition analyzer utilize GPT-4o, while other agents rely on DeepSeek-R1.

Baselines: We selected efficiency-oriented and adaptability-oriented baselines, respectively. Details regarding baseline selection and experimental configurations are provided in Appendix B.

4.2 Results

We present the results of our experiments, including a comparison between our system and the baseline systems, as well as an in-depth analysis of our system's performance with increasing cases and across different UI variations.

4.2.1 Comparison with Baselines

We assess the accuracy and average recognition time per UI element for each system on the test set. The results are presented in Table 1.

In terms of accuracy, our system outperforms the best baseline *Pixel* by 34.6% in five-fold cross-validation. Compared to other evaluation results, the performance of these baselines on Eleva is generally lower. This is because, during data collection, we intentionally selected UI elements with higher recognition difficulty and filtered them using predefined rules. Within our system, in cases where System 1 fails to recognize a UI element, System 2 is triggered for further recognition. Among the cases correctly identified by our system, 84.6% were successfully recognized by System 1. In the cases where recognition failed, 98.7% were misidentified, and 1.3% had extra elements identified.

While *Pixel* also performed well, it deeply relies on template matching of images. When dealing with UI variations caused by skin-related changes, the method's accuracy could significantly decrease, potentially dropping to zero. However, such variations have no impact on our system, as skin-related changes do not affect the layout tree, which is central to the recognition code.

In terms of runtime, even with 15.4% cases triggering System 2, the overall average time is comparable to efficiency-oriented baselines, while the accuracy far exceeds them.

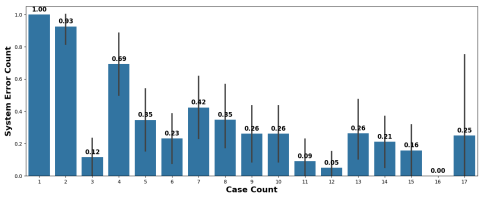


Figure 7: The number of new errors made by System 1 as new cases are input under cold-start conditions.

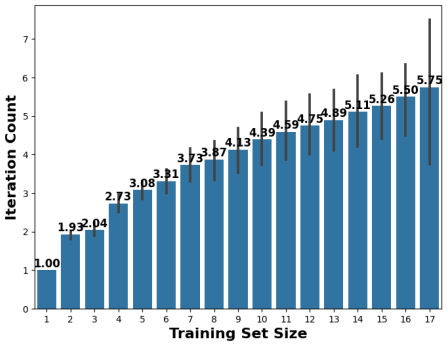


Figure 8: For the progressive input method, each new training case generates additional iterations.

4.2.2 Performance with Increasing Cases

Our results show that as the number of cases increases, the performance of our system improves.

For the progressive input method, as shown in Figure 10, accuracy improves with fluctuations as the training set size increases, and the frequency of System 2 calls decreases with similar fluctuations. This suggests that GroundCoder enhances System 1's capability, and System 1 increasingly takes over, resulting in improved accuracy and efficiency for the entire system. The small fluctuations may be due to the varying difficulty of recognizing different classes of UI elements, as well as the differences in the training set sizes among classes.

As shown in Figure 8, the number of iterations required for code refinement changes with the training set size. The upward trend gradually levels off, indicating that the code's effectiveness improves and new cases no longer necessarily lead to code updates. However, the volatility increases, which we speculate is related to forgetting issues that arise in LLMs after multiple iterations, as discussed in Section C.3.

4.2.3 Performance Across Variations

We evaluate the system's performance by splitting the training and testing sets according to different UI variations, including devices, display modes, and app modes. The overall results are shown in Table 1. Performance varies significantly across apps, as illustrated in Table 4.

Compared to other train-test split methods, the app-mode-based split results in poorer performance for our system across most of the apps. This suggests that the UI element differences between the regular and easy modes within each app are substantial, leading to a reduction in the generalization

Table 1: Comparison of our system and several baselines on the Eleva dataset.

System	Accuracy Five-fold (%)	Accuracy Device (%)	Accuracy Display Mode (%)	Accuracy App Mode (%)	Time (in seconds)
Scale	26.47	17.63	24.72	32.85	**0.0002**
Pixel	59.18	54.79	57.23	47.06	0.628
SeeClick	4.73	5.02	3.68	5.24	0.392
Sim	16.42	18.13	21.05	18.79	0.0052
PureLLM (GPT-4o)	53.97	56.91	51.86	54.42	6.48
PureLLM (DeepSeek-R1)	68.67	69.52	68.31	62.85	92.92
Ours (one-time)	**93.84**	**94.03**	**88.29**	**85.71**	0.54
Ours (progressive)	89.96	81.37	81.98	75.73	1.37

of recognition code generated from one mode to another. The largest discrepancy is observed in WeChat, where the layout differences between the two modes are significant, causing the layout features that are beneficial for convergence in one mode to be ineffective in the other mode. With the device-based split, performance is more balanced across apps, possibly because layout tree structures remain relatively stable across different device versions, thereby enabling stronger generalization.

The results show that the required generalization across different variations influences feature selection. Fortunately, our system supports dynamic code updates. When an unrecognized case is encountered, the system can re-analyze representative features and iteratively update the code until it converges on all encountered cases.

5 Limitations and Future Work

Managing Semantic Relationships of UI Elements: Currently, GroundCoder categorizes UI elements into discrete categories based on semantics. However, in real-world scenarios, the relationships between UI elements are often more complex. A key challenge is how to handle the varying states of UI elements. Many UI elements exhibit different states (e.g., selected vs. unselected, active vs. inactive), and determining whether such states should be classified as the same element or distinct entities is a fundamental question. This issue is inherently tied to the granularity of semantic recognition. Future improvements in GroundCoder will need to refine this granularity of semantic understanding, enabling better handling of UI elements with state-dependent semantics.

Handling Page Classification: In this paper, identifying similar UI elements assumes that the page on which the element resides is correctly identified, as outlined in Section 2.2 where pages are assumed to correspond one-to-one. In other words, even if two UI elements appear visually similar, they should not be considered as belonging to the same class if they are located on different pages. This is because they are likely to trigger different tasks based on their page context. In the future, we plan to extend GroundCoder to support page recognition as well, classifying pages based on their semantic content. This would enable a verification step to ensure the correct page correspondence, further enhancing the system's robustness and adaptability.

Expanding the Capabilities of Generated Code: The current generated code primarily consists of logical expressions based on feature extraction, but it lacks the sophistication needed to handle more complex recognition rules. To address this limitation, we plan to develop an expert knowledge base that will integrate with the *code generator*. This knowledge base will support the generation of more advanced logic and rules, improving the code's adaptability and generalization capabilities.

6 Conclusion

We presented a dual-system GUI grounding approach that balances efficiency and accuracy via generative code. Our multi-agent system, GroundCoder, bridges cognitive-inspired System 1 (rule-based recognition) and System 2 (semantic analysis) by dynamically generating recognition rules. To evaluate this, we introduced Eleva, a large-scale dataset with 32,807 samples across 3,182 semantic classes. Results show that our system achieves a 34.6% accuracy improvement over mainstream techniques while maintaining high efficiency and robustness across UI variations. Future work will focus on enhancing LLM capabilities to further optimize code generation and system scalability.

References

Angelo Cangelosi. 2010. Grounding language in action and perception: From cognitive agents to humanoid robots. *Physics of Life Reviews*, 7(2):139–151.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

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. *Preprint*, arXiv:2401.10935.

Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017a. 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, page 845–854, New York, NY, USA. Association for Computing Machinery.

Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017b. 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*, pages 845–854, Québec City QC Canada. ACM.

Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, page 204–217, New York, NY, USA. Association for Computing Machinery.

Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Trans. Inf. Syst.*, 43(2).

Tian Huang, Chun Yu, Weinan Shi, Bowen Wang, David Yang, Yihao Zhu, Zhaoheng Li, and Yuanchun Shi. 2023. Interaction proxy manager: Semantic model generation and run-time support for reconstructing ubiquitous user interfaces of mobile services. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 7(3).

Peiyuan Jiang, Daji Ergu, Fangyao Liu, Ying Cai, and Bo Ma. 2022. A review of yolo algorithm developments. *Procedia Computer Science*, 199:1066–1073. The 8th International Conference on Information Technology and Quantitative Management (ITQM 2020 & 2021): Developing Global Digital Economy after COVID-19.

Daniel Kahneman, Shane Frederick, and 1 others. 2002. Representativeness revisited: Attribute substitution in intuitive judgment. *Heuristics and biases: The psychology of intuitive judgment*, 49(49-81):74.

Gang Li and Yang Li. 2023. Spotlight: Mobile ui understanding using vision-language models with a focus. *Preprint*, arXiv:2209.14927.

Tao Li, Gang Li, Jingjie Zheng, Purple Wang, and Yang Li. 2022. Mug: Interactive multimodal grounding on user interfaces. *Preprint*, arXiv:2209.15099.

Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. Sugilite: Creating multimodal smartphone automation by demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, page 6038–6049, New York, NY, USA. Association for Computing Machinery.

Toby Jia-Jun Li, Lindsay Popowski, Tom Mitchell, and Brad A Myers. 2021. Screen2vec: Semantic embedding of gui screens and gui components. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA. Association for Computing Machinery.

Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M. Mitchell, and Brad A. Myers. 2019. Pumice: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, page 577–589, New York, NY, USA. Association for Computing Machinery.

Thomas F. Liu, Mark Craft, Jason Situ, Ersin Yumer, Radomir Mech, and Ranjitha Kumar. 2018. Learning design semantics for mobile apps. In *The 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, pages 569–579, New York, NY, USA. ACM.

Helen Sharp, Yvonne Rogers, and Jenny Preece. 2007. *Interaction Design: Beyond Human Computer Interaction*. John Wiley & Sons, Inc., Hoboken, NJ, USA.

Yunpeng Song, Yiheng Bian, Yongtao Tang, Guiyu Ma, and Zhongmin Cai. 2024. Visiontasker: Mobile task automation using vision based ui understanding and llm task planning. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, UIST '24, New York, NY, USA. Association for Computing Machinery.

Bryan Wang, Gang Li, and Yang Li. 2023a. Enabling conversational interaction with mobile ui using large language models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA. Association for Computing Machinery.

Bryan Wang, Gang Li, and Yang Li. 2023b. Enabling Conversational Interaction with Mobile UI using Large Language Models. *arXiv preprint*. ArXiv:2209.08655 [cs].

Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. 2021. Screen2Words: Automatic Mobile UI Summarization with Multimodal Learning. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 498–510, Virtual Event USA. ACM.

Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, Aaron Everitt, and Jeffrey P Bigham. 2021. Screen recognition: Creating accessibility metadata for mobile applications from pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA. Association for Computing Machinery.

Xiaoyi Zhang, Anne Spencer Ross, Anat Caspi, James Fogarty, and Jacob O. Wobbrock. 2017. Interaction proxies for runtime repair and enhancement of mobile application accessibility. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, page 6024–6037, New York, NY, USA. Association for Computing Machinery.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. 2024. Gpt-4v(ision) is a generalist web agent, if grounded. *Preprint*, arXiv:2401.01614.

A DataSet Construction

A.1 Raw Data Collection

Although our dataset is not primarily intended to capture task workflows, we still collect data following predefined workflows. This approach ensures that corresponding pages from multiple samples of the same workflow can be directly aligned, facilitating accurate identification and classification of UI elements across variations.

The detailed procedure is as follows:

1. The experts define multiple workflows for each application, ensuring comprehensive coverage of commonly encountered UI pages while minimizing overlap between similar pages across different workflows.

2. We utilize crowdsourcing for data collection. We hired 8 annotators, each of whom used their own devices to exhaustively annotate all required applications and their variations (including display modes and app-specific modes), completing the data collection for all workflows.

Notably, we only consider UI variations that affect visual appearance and layout. Significant differences—such as changes in the sequence of pages traversed during the workflow—are beyond the scope of our dataset, as these cases prevent meaningful alignment of UI elements across different UI variations.

A.2 Dataset Annotation

The raw data includes a rich set of UI pages exhibiting many UI element variations. From this dataset, we need to select a subset of meaningful UI elements for targeted annotation, including both classified and semantic labeling.

UI Element Selection and Classified Annotation: Assume that for a given workflow, we have collected n samples from different UI variations. The i^{th} sample consists of a sequence of m pages, $\{P_1^i, P_2^i, ..., P_m^i\}$. The UI element triggering the transition from page P_j^i to $P_{j+1}^i (1 \leq j < m)$ is labeled as E_j^i. Naturally, the elements $\{E_j^1, E_j^2, ..., E_j^n\}$ belong to the same UI element class.

Since transition-triggering elements alone are limited, the annotators manually select 5 representative UI element classes on each page $P_j^1 (1 \leq j \leq m)$ in the first sample. For each selected class, the annotators label corresponding UI elements of the same class on the other sample pages $\{P_j^2, P_j^3, ..., P_j^n\}$. Consequently, each page is annotated with at least 5 UI element classes, each represented by multiple samples corresponding to different UI variations. To define "representative" UI element classes, the annotation platform automatically applies simple rules to assist the annotators in selecting classes that are meaningful and not easily identifiable through basic rules alone (such as by tree path or coordinates). This ensures that the selected UI element variations are more representative.

Specifically, although the experts strive to minimize duplicate pages in each workflow during data collection, it is unavoidable that some pages are revisited. Therefore, for pages that are traversed multiple times and have similar UI elements annotated, the class of those elements should be merged.
Semantic Annotation: After classifying UI elements, we assign a semantic textual label to each class. The annotators are provided with visualizations, including both screenshots and layout information. GPT-4o generates an initial semantic description based on visual input, which the annotators subsequently verify and refine as needed.

Table 2: Overview of the Eleva dataset. The mobile applications were selected based on popularity, page content richness, and diversity of UI features.

Application	Device	Display Mode	App Mode	UI Element Classes	UI Element Samples
TikTok	HUAWEI MatePad Pro-MRX-W39 HUAWEI MatePad Pro WGRR-WO9 HUAWEI MatePad Pro-GOT-W29 REDMI Turbo 14 HONOR 90GT REDMI K70 Ultra OPPO Reno9 Pro iQOO Neo5	Light Dark	Regular Easy	314	5380
Ctrip				596	3834
WeChat				420	5776
Weibo				343	2528
Tecent Meeting				191	2250
Pinduoduo				224	2590
12306				278	2641
Lark				413	2820
Taobao				226	3772
RedNote				177	1216

Table 3: Comparison of Eleva with existing UI datasets with semantic labeling.

Dataset	**UI Variations**	**Granularity of Classes**	**Number of Classes**	**Number of Samples per Class**
Rico (Deka et al., 2017b; Liu et al., 2018)	Single device	UX concepts of buttons	197	1~2720
APM (Zhang et al., 2021)	4 resolutions	Common UI types	12	1808~41,285
Eleva	8 devices, 2 display modes, 2 app modes	Task-related semantics	3182	5~22

B Experiment Details

B.1 Baseline Selection

Efficiency-oriented Baselines: These baselines represent mainstream methods adopted for fast UI element recognition. They rely on basic methods:

- *Scale*: It locates UI elements on the current page by matching their coordinates (x, y) relative to the screen size, based on values in the train-set.
- *Pixel*: It converts the image to grayscale, scales the images in the training set across various scales, and uses the OpenCV function `matchTemplate` with the `TM_CCOEFF_NORMED` algorithm for template matching.
- *SeeClick*: It uses a 9.6B model to recognize UI elements on the screenshot (Cheng et al., 2024).
- *Sim*: It uses a similarity function based on the integration of layout trees and screenshots. For each UI element on the current page, it computes a weighted similarity score by comparing it to elements in the training set, based on all accessible features. An empirical threshold is applied for classification, where any element with a higher similarity score is considered a match. This method was initially utilized in prior works on accessibility enhancement (Zhang et al., 2017), and has since been commonly applied in systems with real-time efficiency requirements (Li et al., 2019; Huang et al., 2023).

Adaptability-oriented Baselines: Adaptability-oriented methods usually leverage LLMs for deeper semantic understanding of UI elements, enabling better adaptation to UI variations. They align with the majority of current LLM-based AI agents (Li et al., 2022; Li and Li, 2023). We selected the GPT-4o multimodal model as a representative, which uses semantic inputs to identify corresponding UI elements from images. The image is uniformly divided into 144 sections (a 16×9 grid), and GPT-4o outputs the identifier for the target region, which is then used to match the corresponding UI element.

B.2 Experiment Setup

We conduct the experiments using our proposed Eleva dataset. For each UI element class, we split the samples into training and testing sets. We use two types of data splits: random 5-fold cross-validation (80% for training, rounded down, and 20% for testing, rounded up) and a variant-based split. In the variant-based split, the data is divided based on device model (each two devices used as a test set once, with the remaining devices used for training), display mode (alternating between light and dark modes for testing and training), and app-specific modes (alternating between regular and easy modes for testing and training). The final evaluation is based on the average performance across all rounds of the split.

For *Scale*, *Pixel*, *Sim* and our proposed dual system, the training set must be inputted. For our system, there are two methods of inputting the training set: a one-time input of the entire training set or a progressive input of individual training cases.

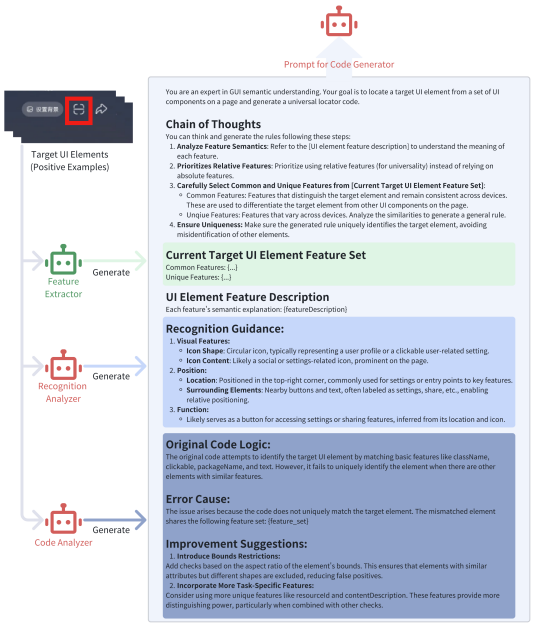
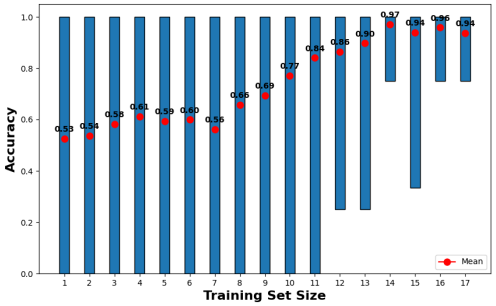
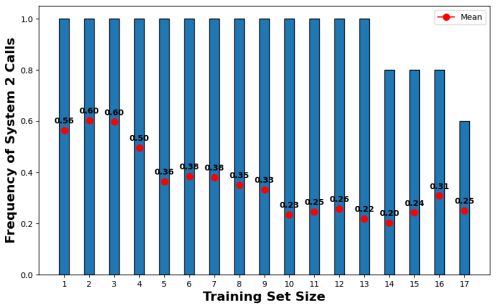


Figure 9: Prompt design for the code generator, including a description of the code generation task, guidance provided by the recognition analyzer, and analysis of the original code (if available), along with error analysis and improvement suggestions from the code analyzer.



(a) Minimum, maximum, and average accuracy of the code generated by GroundCoder as training set size changes.



(b) Minimum, maximum, and average frequency of System 2 calls during testing as training set size changes.

Figure 10: As the training data increases with the progressive input method, accuracy gradually improves, and the frequency of System 2 calls decreases.

Table 4: Accuracy of each app under different train-test splits, with the training set input all at once. For each app, the bolded values represent the best performance, the underlined values represent the worst performance, and the dashed lines indicate the absence of corresponding variations due to app limitations.

App	Five-fold	Device	Display Mode	App Mode
TikTok	97.03	81.47	**99.31**	95.42
Ctrip	90.28	**94.97**	-	-
WeChat	**83.96**	79.23	82.98	51.38
Weibo	94.51	**97.18**	73.64	-
Tencent Meeting	97.47	97.89	**98.12**	-
Pinduoduo	96.24	**98.53**	98.47	95.12
12306	**95.98**	87.52	-	95.73
Lark	92.54	**93.07**	80.85	-
Taobao	97.32	98.29	**98.61**	89.94
RedNote	88.63	**91.87**	78.49	-

Since UI elements of the same class share the same semantic label in the dataset, as defined in Section 2.1, the success rate of semantic matching during testing can be considered 100%, as reference samples can be directly found in the training set. In contrast, *SeeClick* and the adaptability-oriented baseline do not require a training set. Instead, they directly use the LLM (or self-trained models) for UI element recognition on the test set. UI element recognition is considered successful if the center of the recognized UI element falls within the bounds of the ground-truth UI element.

Additionally, we evaluate the impact of including or excluding key agents, namely the *code inspector*, and *code analyzer*, within the GroundCoder system.

C Insights from the Experiment Results

In this section, we present key insights derived from a thorough analysis of the experimental results. These insights not only validate the effectiveness of GroundCoder but also offer directions for future optimization and improvements.

C.1 Code Analysis

We performed an in-depth analysis of the code generated by GroundCoder to assess the logic and relevance of the selected features. Through this analysis, we identified both strengths and areas for improvement in the system's feature selection process.

Within the generated codes of different UI element classes, the frequency of various features is shown in Figure 11. Certain features were overly

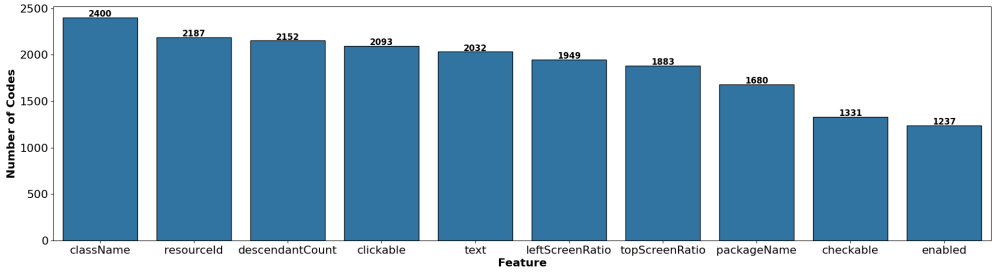


Figure 11: The top 10 most frequently used features in the code.

Table 5: Typical UI element classes and their corresponding recognition code. The red box highlights the UI elements to be recognized, while the green box indicates the UI elements that assist in recognition.

Feature Type	UI Element	Recognition Code

Valid Ranges		`def match(CurrentIcon: 'Layout') -> bool:`
		`  common_features_match = (`
		`  CurrentIcon.ClassName == 'android.view.View' and`
		`  CurrentIcon.clickable == True and`
		`  CurrentIcon.descendantCount == 0 and`
		`  CurrentIcon.level == 9 and`
		`  CurrentIcon.packageName == 'com.ss.android.ugc.aweme'`
		`  )`
		`  unique_features_match = (`
		`  CurrentIcon.resourceId in {`
		`  'com.ss.android.ugc.aweme:id/zvw',`
		`  'com.ss.android.ugc.aweme:id/z7a',`
		`  'com.ss.android.ugc.aweme:id/zk9',`
		`  'com.ss.android.ugc.aweme:id/zf2'`
		`  } and`
		`  # 图标的水平位置必须在屏幕宽度的80%到95%之间`
		`  0.8 <= CurrentIcon.leftScreenRatio <= 0.95 and`
		`  # 图标的垂直位置，根据截图调整到更靠下的范围`
		`  0.4 <= CurrentIcon.topScreenRatio <= 0.7`
		`  )`
		`  return common_features_match and unique_features_match`

Hierarchical Features		`def match(CurrentIcon: Layout) -> bool:`
		`  common_features_match = (`
		`  CurrentIcon.packageName == 'com.taobao.taobao' and`
		`  CurrentIcon.className == 'android.widget.ImageView' and`
		`  CurrentIcon.descendantCount == 0 and`
		`  # 识别该图标的兄弟节点`
		`  CurrentIcon.getSibling() is not None and`
		`  CurrentIcon.getSibling().className == 'android.widget.TextView' and`
		`  CurrentIcon.getSibling().text == '购物车'`
		`  )`
		`  unqiue_features_match = (`
		`  0.60 <= CurrentIcon.leftScreenRatio <= 0.75 and`
		`  0.90 <= CurrentIcon.topScreenRatio <= 0.95`
		`  )`
		`  return common_features_match and unqiue_features_match`

Spatial Relationships		`def match(CurrentIcon: Layout) -> bool:`
		`  # 优先识别导航栏的位置`
		`  nav_bar_focus = (`
		`  0.85 <= CurrentIcon.topScreenRatio <= 0.98`
		`  )`
		`  common_features_match = (`
		`  CurrentIcon.packageName == 'com.tencent.wemeet.app' and`
		`  CurrentIcon.className == 'android.widget.TextView' and`
		`  CurrentIcon.descendantCount == 0 and`
		`  CurrentIcon.text.find('我的') != -1`
		`  )`
		`  return nav_bar_focus and common_features_match`

Fuzzy Text Matching		`def match(CurrentIcon: Layout) -> bool:`
		`  import re`
		`  common_features_match = (`
		`  CurrentIcon.packageName == 'com.MobileTicket' and`
		`  CurrentIcon.descendantCount == 0 and`
		`  )`
		`  unqiue_features_match = (`
		`  CurrentIcon.className in {`
		`  'android.view.View',`
		`  'android.widget.TextView'`
		`  } and`
		`  # 使用正则表达式匹配文本`
		`  bool(re.match(r'.*乘车积分将于\d+日后到期\s*', CurrentIcon.text)) and`
		`  0.02 <= CurrentIcon.leftScreenRatio <= 0.03 and`
		`  0.14 <= CurrentIcon.topScreenRatio <= 0.16 and`
		`  CurrentIcon.level in {11, 12}`
		`  )`
		`  return common_features_match and unqiue_features_match`

emphasized, leading to unnecessarily complex or inaccurate rules, while some crucial features were overlooked. While the code produced accurate recognition results, further optimization could improve its efficiency and accuracy.

For example, the system tended to overrely on "obvious" features, such as `className`, `packageName`, `text`, `resourceId`, and interaction attributes in the layout tree. While these features are useful, they are often common across many UI elements and can lead to misrecognition in more complex or irregular layouts. In contrast, dynamic features, such as relative positioning or spatial relationships between elements, were often neglected. For instance, in layouts with multiple dynamic popups, the relative positioning and nesting of UI elements became crucial. We also found that some processed features, although potentially significant differentiators in certain UI layouts, were underutilized by the GroundCoder.

By analyzing both successful and failed cases, we identified key features that significantly impacted the success of UI element recognition. Correspondingly, Table 5 presents several typical cases, highlighting the following key feature types.

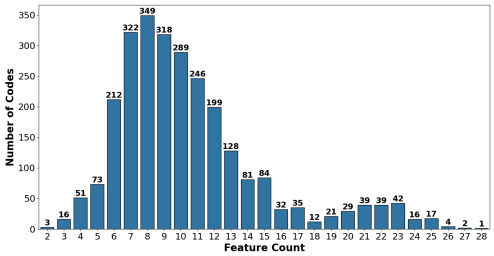
- **Valid ranges for relative and absolute coordinates:** The position of UI elements is a crucial feature, but requires some tolerance for errors. By defining valid coordinate ranges, the system can ensure elements are located within expected spatial areas, reducing misidentification caused by screen resolution differences or dynamic layout changes.

- **Hierarchical structure-related features:** These features capture the parent-child relationships in the UI layout, such as the number of child elements, parent element features, and layout tree depth. Additionally, some sibling nodes are important, such as in recognizing icons, where the adjacent text label can be identified first, often found within the sibling subtree in the layout tree.

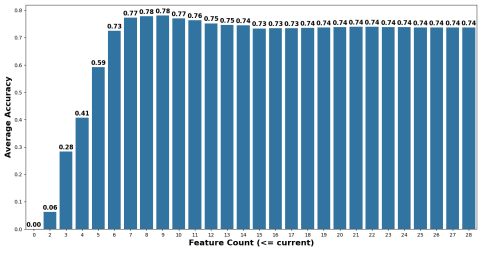
- **Spatial layout relationships:** Accurate recognition heavily relies on the spatial and layout relationships between UI elements. An effective strategy involves a hierarchical approach: first identifying a larger container like a navigation bar or a dialog, and subsequently locating the target element, such as a 'close' button, within that container using simple features.

- **Fuzzy matching of text-based features:** Text-based features, such as the label of a button, often require fuzzy matching due to the dynamic nature of UI content. For instance, the label of a "Chats" button may appear in the `text`, `contentDescription`, or `resourceId` attributes, and may not always match exactly. Variants of the label, like "Chats" and "Chats(1)", require the system to recognize substrings, prefixes, or suffixes, and in some cases, regular expressions are needed.

As a result, we believe the code should be further refined to address biases and omissions in the generated code, improving the logical consistency of the code. Additionally, enhancing the understanding of feature meanings and building a professional knowledge base could guide the LLM to generate more generalizable code.



(a) Distribution of total features used in the code.



(b) Relationship between the upper limit of features in the code and its accuracy. Note that the accuracy here refers to the performance achieved by System 1 using the generated code, without considering System 2's assistance in case of failure.

Figure 12: Feature utilization in the code. Note: This refers to the code automatically generated by GroundCoder, with the training set input all at once, and the code having converged in the training set, without considering any manual corrections.

C.2 Redundancy in Code

During the experiments, we noticed that the generated code sometimes included redundant features that did not positively impact recognition perfor-

Table 6: Comparison of average iterations and human intervention rates between one-time and progressive input methods.

System	Mean Iterations	Human Intervention (%)
Ours (one-time)	1.3 (sd=0.7)	17.6
Ours (progressive)	2.7 (sd=2.0)	11.7

mance. These redundant features not only compromised code readability but also increased the risk of overfitting. For example, the `enabled` feature, which appears frequently in Figure 11, is not a representative feature and can be removed.

The number of features used by the recognition code is shown in Figure 12a. On average, each recognition code used about 10.5 (sd = 4.4) features, of which 2.6 (sd = 1.1) were found to be non-essential according to expert evaluation.

We further compared the code's performance across different feature counts. As shown in Figure 12b, the code with 9 or fewer features achieved the highest average accuracy. Beyond this point, as the number of features increased, the accuracy no longer improved, indicating a decline in generalization ability.

We suggest introducing a code simplification agent in future optimizations to reduce redundant features, improving code readability and the system's generalization across UI variations.

C.3 Handling Overload and Forgetting

We found that when GroundCoder generates and updates the recognition code, using multiple input samples at once significantly outperforms the progressive input of individual training cases. Table 6 compares the iterations required for the one-time input and progressive input methods. As expected, the one-time input method requires fewer iterations. However, the number of iterations for the progressive input method was much lower than we had expected. Upon further analysis, we discovered that 27.2% of the classes could generate effective code for the entire training set using only a single training case. This suggests that GroundCoder can extract representative features and generate generalizable code from limited samples.

For the one-time input method, 60.9% of UI element classes converge within one iteration, 73.5% within two iterations, and in 89.6% of the classes, the one-time input method outperforms the progressive method. Furthermore, the accuracy of the one-time method is higher than that of the progressive method. We attribute this improvement to the fact that multiple samples allow the LLM to learn parallel patterns from various UI layouts.

Both input methods have their drawbacks:

- One-time input faces the issue of LLM overload: We observed that inputting too many samples at once can negatively impact GroundCoder's performance. For example, in UI element classes with more than 12 samples, 85.3% of the classes failed to converge within 5 iterations using the one-time input method.

- Progressive input, while reducing overload, introduces issues of forgetting and internal contradictions: Although one-time input can also encounter these issues, progressive input, which requires more iterations by nature, is more likely to trigger LLM forgetting of earlier inputs. We found that 18.6% of UI elements experienced code loss or conflicts. In these cases, the *code inspector* was triggered 1153 times, helping GroundCoder "remember" the bad cases through further iterations. However, the forgetting issue could not be fully resolved. Specifically, 88.3% of UI elements converged fully within 5 iterations, while the remaining elements required human intervention.

Based on these analyses, we recommend a balanced strategy of using an appropriate number of samples for input, which can both improve accuracy and reduce model training time. Additionally, while the *code inspector* has been proved to effectively mitigate forgetting and contradictions, to fully address these issues, we suggest incorporating a dynamic memory mechanism or more intelligent prompt strategies in future improvements to maintain long-term consistency in generative code.

C.4 Manual Intervention during Runtime

During the system's operation, manual intervention may be required in two scenarios:

- When System 2 lacks sufficient intelligence to identify UI elements, human intervention helps identify the correct elements.

- When GroundCoder fails to converge after multiple iterations, human intervention assists in finalizing the code convergence.

In extreme or uncommon UI designs, expert intervention can quickly identify the shortcomings of the generated code and provide supplementary rules based on real-world scenarios. For example, in new UI versions, experts can annotate new UI element types or important interaction changes, helping the system better recognize these changes in future training.

During the experiments, the proportion of human intervention is shown in Table 6. While the goal is to improve the system's interpretability and reduce intervention costs, efforts should also focus on enhancing the system's intelligence to minimize the need for human intervention.