Coding with Eyes: Visual Feedback Unlocks Reliable GUI Code Generating and Debugging

Zhilin Liu, Ye Huang *, Ting Xie,
Ruizhi Zhang, Wen Li, and Lixin Duan

Shenzhen Institute for Advanced Study,
University of Electronic Science and Technology of China

Abstract. Recent advances in Large Language Model (LLM)-based agents have shown remarkable progress in code generation. However, current agent methods mainly rely on text-output-based feedback (*e.g.* command-line outputs) for multi-round debugging and struggle in graphical user interface (GUI) that involve visual information. This is mainly due to two limitations: 1) GUI programs are event-driven, yet existing methods cannot simulate user interactions to trigger GUI element logic. 2) GUI programs possess visual attributes, making it difficult for text-based approaches to assess whether the rendered interface meets user needs. To systematically address these challenges, we first introduce InteractGUI Bench, a novel benchmark comprising 984 commonly used real-world desktop GUI application tasks designed for fine-grained evaluation of both interaction logic and visual structure. Furthermore, we propose VF-Coder, a vision-feedback-based multi-agent system for debugging GUI code. By perceiving visual information and directly interacting with program interfaces, VF-Coder can identify potential logic and layout issues in a human-like manner. On InteractGUI Bench, our VF-Coder approach increases the success rate of Gemini-3-Flash from 21.68% to 28.29% and raises the visual score from 0.4284 to 0.5584, indicating the effectiveness of visual feedback in GUI debugging.

1 Introduction

In recent years, the rapid progress of LLM-based agents has substantially advanced automated code generation. Models now perform competitively on widely used benchmarks such as HumanEval and MBPP [4,5], and have begun to power practical development assistants like Copilot and Cursor. However, effectively generating and debugging GUI code remains a challenging problem.

Existing code agents (*i.e.* Fig. 1.a) primarily rely on textual feedback for self-repair, iteratively optimizing code through the textual information like execution logs or error messages. Some studies further incorporate external knowledge sources [30,37]or structured information (*e.g.* abstract syntax trees or XML Trees) [8, 13, 39] for better visual effects. However, the absence of visual perception regarding the actual rendered interface prevents the code agent from accurately assessing layout and rendering correctness.

* Corresponding author

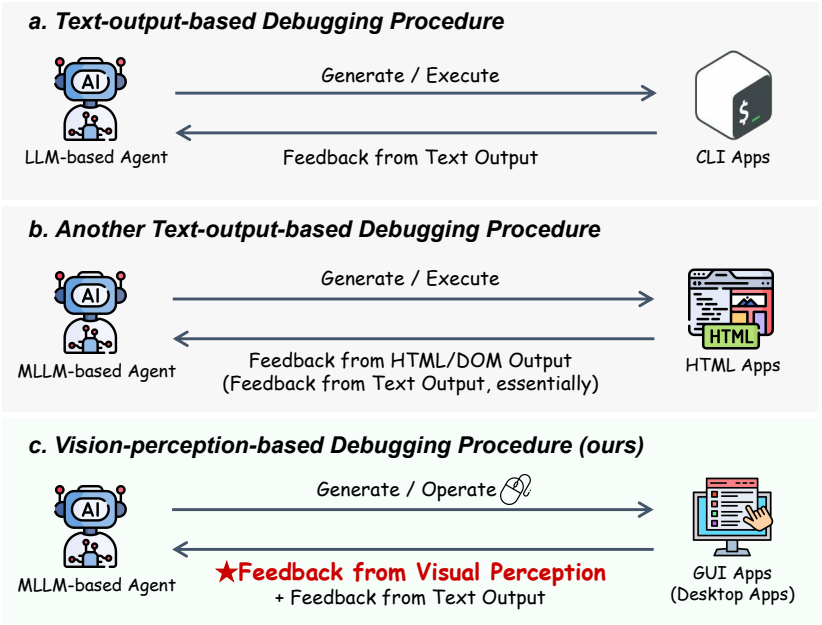

Fig. 1: Comparison between (a, b) traditional text-output-based debugging procedure and (c) our vision-perception-based debugging procedure (VF-Coder), which enables direct perception of rendered GUIs for interactive correction.

Although the advancement of Multimodal Large Language Models (MLLMs) offers visual perception for code agents, making them now able to generate GUI code directly from real UI screenshots or design drafts for front-end development tasks (*i.e.* Fig. 1.b), the debugging stage of these methods still relies on textual feedback such as UI layout trees [31] or DOM structures [36].

This implies that the corrective capabilities of code agents remain confined to the textual level, lacking the ability to directly harness visual information for end-to-end dynamic debugging. Furthermore, existing benchmarks and methods focus mainly on the visual reproduction of static webpages, and difficult to migrate to other GUI forms, such as Desktop GUI apps. Also, existing benchmarks struggle to handle complex multi-page structures and long-range interaction logic in real-world scenarios. Therefore, these limitations reveal two core challenges in generating real-world GUI code:

(1) **Functional logic modeling:** GUI apps follow an event-driven paradigm that emphasizes interactivity. Only user operations (*e.g.* clicks, text inputs, or focus changes) can trigger behavioral or state changes, and varying interaction sequences may lead to diverse interface states and logical outcomes. This makes it difficult for models to capture dynamic dependencies and maintain functional correctness across interactions.

(2) **Visual structural alignment:** Unlike command-line interface (CLI) programs that lack a visual component, the usability of a GUI is tightly coupled with its visual organization. The generated interface must not only be functionally correct but also preserve consistent layout structure, component styling, and visual hierarchy.

To address these challenges, we first introduce **InteractGUI Bench**, a high-quality benchmark tailored for real-world desktop GUI scenarios. We observed that existing datasets and benchmarks focus predominantly on Web scenarios, ignoring the diverse layout challenges and limited debugging information inherent in desktop GUI apps. Thus, we carefully curated up to 984 diverse real-world desktop GUI apps as test cases, covering nearly all commonly used apps available on the Internet. Each case consists of multiple interface screenshots and a textual instruction, reflecting the authentic workflow of human developers building GUIs from design drafts.

Furthermore, unlike most existing benchmarks that solely evaluate static interfaces, InteractGUI Bench establishes a fully executable sandbox environment and introduces Interactive Evaluation Script (IES), which can simulate real interactions and interface transitions on actually running GUI apps. Backed by the Assistive Technology Service Provider Interface (AT-SPI) and a specially trained visual evaluation model, we provide a comprehensive assessment ranging from atomic attributes (components, colors, interactions), precise layout, all the way to overall GUI structure. We systematically evaluated mainstream MLLMs on InteractGUI Bench. Taking PySide6 as an example, even state-of-the-art models achieve only a 26.99% task success rate, showing the difficulty and complexity of our benchmark.

In addition, we propose **VF-Coder**, a multi-agent framework that integrates visual feedback into the entire lifecycle of GUI code generation and debugging. Inspired by GUI Agent methods [20, 29, 40], VF-Coder allows the model to directly observe and manipulate rendered interfaces to detect and repair potential logical or display errors in real time. On InteractGUI Bench, using Gemini-3-Flash as the base model, VF-Coder achieves a 6.61% improvement in task success rate (from 21.68% to 28.29%), significantly outperforms existing text-based methods, which achieve only about a 3% improvement. This result demonstrates the effectiveness of visual feedback in enhancing the quality of GUI apps.

Our main contributions are as follows:

1. We systematically identify the two key challenges in GUI code generation: functional logic modeling and visual structural alignment. Both capture the dual difficulties of maintaining interactivity and visual consistency.
2. We introduce InteractGUI Bench, a high-quality benchmark comprising 984 real-world desktop GUI apps equipped with an end-to-end automated evaluation framework. Leveraging AT-SPI and the visual evaluation model, it enables comprehensive measurement and accurate judgment.
3. We propose VF-Coder, the first visual feedback-driven framework that introduces GUI Agent to solve the code generation task. By simulating the real debugging process of human developers, VF-Coder establishes a closed-loop

mechanism of "visual perception - dynamic interaction - code refactoring", achieving the automated repair of functional logic errors and visual rendering flaws, significantly enhancing the robustness and visual consistency of generated apps.

2 Related Work

2.1 Benchmarks for Code Generation

Early benchmarks mainly evaluated models' capabilities at the line or function level of code generation [4, 5, 10, 12, 18]. With the rapid advancement of LLMs, many research works have expanded toward real-world app development tasks.

For example, SWE-Bench [14] and CodeAgentBench [37] require models to perform functionality repair and feature development directly within large codebases. EvoCodeBench [16] and DevEval [17] further evaluate whether models can generate target code that meets specified requirements under full project context and constraints. Web-Bench [33] extends to web development scenarios, emphasizing the agent's capacity for long-term context understanding across multi-step development processes. However, these benchmarks remain constrained to text-output-based feedback, thereby ignoring the inherent interactivity and visual characteristics of GUI environments.

Another line of research attempts to evaluate visual aspects using the image-to-code paradigm. For instance, WebSight [15] synthesizes two million HTML screenshot pairs, providing a large-scale foundation for studying visual code generation. However, its code similarity metrics may cause models to overfit superficial patterns rather than true visual semantics.

To better capture front-end design fidelity, Design2Code [22] measures performance via visual similarity between generated pages and reference images. Nonetheless, such methods focus solely on static appearance while ignoring end-to-end interaction.

To move beyond static evaluation, Interaction2Code [32] uses webpage screenshots before and after user interactions to simulate state transitions, encouraging models to reason about component functionality. Yet, its dual-image differencing mechanism can only represent simple single-step interactions, falling short of capturing the multi-stage state dependencies and complex logic of real GUI applications. Overall, existing image-to-code benchmarks primarily focus on visual consistency in web front-end generation, but they lack systematic evaluation of multi-stage interaction logic and desktop-level GUI programming.

2.2 Code Agents

Recent LLM-based code generation agents primarily rely on text-output-based feedback for iterative refinement [3, 19, 23, 35, 38], leveraging execution outputs [38] and other external knowledges [1] to perform self-correction and substantially improve generation accuracy. *e.g.*, CodeNav [8] automatically retrieves

relevant code snippets from real repositories and debugs them based on execution results. ROCODE [13] continuously monitors compilation logs during generation and backtracks once errors are detected. QualityFlow [11] adopts a multi-agent workflow that generates unit tests to more comprehensively verify code functionality. However, these methods are not tailored for GUI-oriented scenarios and thus fail to model the visual and interactive characteristics inherent to GUI.

Several studies have begun to explore the use of visual information. However, they typically incorporate it only once at the input stage, while the debugging process remains dominated by textual feedback. For instance, LayoutCoder [31] constructs structured UI layout trees to enhance LLMs' understanding of interface hierarchy and spatial arrangement, whereas DesignRepair [36] leverages rendered DOM trees to identify and repair visual defects.

Although these methods improve the visual consistency between generated code and reference screenshots, the lack of runtime visual perception and interaction prevents them from dynamically reasoning about rendered interfaces. As a result, they are limited to generating static UI and struggle to handle GUI programs involving multi-page navigation and complex event-driven logic.

Our work distinguishes itself by incorporating visual feedback–driven vulnerability awareness and interactive debugging. By performing real interactive testing through screenshots, our method enables the agent to efficiently diagnose and fix program errors.

3 Proposed InteractGUI Bench

This section introduces the proposed InteractGUI Bench, a high-quality benchmark that consists of 984 GUI code generation tasks from 984 real-world desktop GUI apps. Each task comprises multiple GUI screenshots of a real-world Desktop application and corresponding natural-language instructions. We first describe the data acquisition and task construction pipeline, followed by a detailed explanation of the automated verification and evaluation process.

3.1 Data Collection

Raw Data Acquisition: To ensure the authenticity and diversity of the benchmark, we collected data from Flathub, a centralized app store for Linux. Flathub encompasses thousands of desktop applications, each with high-resolution, real-world GUI screenshots. Using a Python-based crawler, we retrieved the top 1200 GUI apps from the "Popular" list, capturing their descriptions and corresponding interface screenshots as the foundational data source.

Manual Cleaning and Annotation: Due to the broad range of sources, the raw data contained significant noise. We built a dedicated annotation system to address two key challenges through manual review:

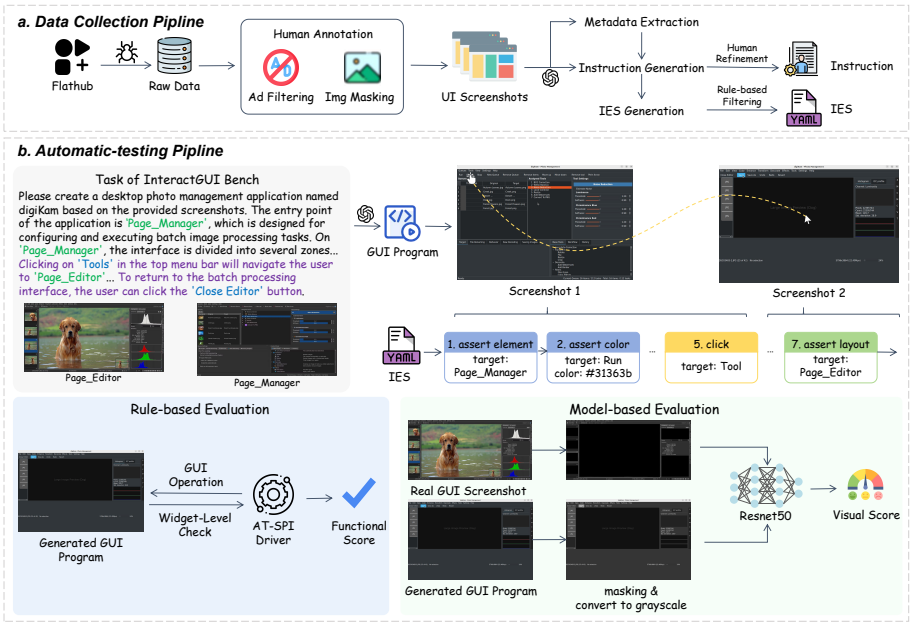

Fig. 2: The data collection and automatic-testing pipeline. (a) shows the process of collecting real-world desktop UI screenshots from Flathub, constructing task instructions, and building the IES. (b) illustrates the workflow for testing generated GUI apps using a series of steps defined by the IES. Zoom in to see better.

- Filtering Non-GUI Content: Some screenshots included promotional posters or commercial advertisements, which could cause model inference to deviate from the target. These were manually excluded to ensure data purity.
- Resource Sensitivity Handling: MLLMs cannot autonomously generate logos or images. To prevent such uncontrollable factors from biasing the visual evaluation, we manually performed masking on all icons and images within the screenshots, allowing the assessment to focus on visual structure and interaction logic.

Following this refinement, we distilled the initial 1200 candidate apps into 984 high-quality instances. Each instance contains one or more GUI screenshots of the same desktop app.

Metadata Extraction and Instruction Generation: To transform static screenshots into executable tasks, we developed a multi-stage synthesis pipeline using Gemini-3-Flash [6]. We first prompt the model to analyze collected screenshots and descriptions, extracting structured metadata that includes component name and navigation logic between each interfaces. Based on this metadata, we then use model to generate natural language instructions that mimic human intent. The instructions emphasize cross-page interaction, effectively bridging

multiple independent screenshots into a unified GUI development task . All generated instructions are then manually revised to ensure that their transition logic aligns with human intuition and that all mentioned elements exist on the corresponding pages of the input screenshots.

3.2 Evaluation

Construction of Interactive Evaluation Script: To support the evaluation of multi-interface layouts and interaction logic of generated GUI apps, we design the Interactive Evaluation Script (IES) for each task. IES is a YAML-formatted declarative interaction protocol that describes interface states, user operations, and expected outcomes in a structured form. We define the following three types of assertion operations and three types of interaction operations. By combining these operations in IES, we enable dynamic verification and interaction for the generated GUI apps.

The assertion operations are used to verify visual and semantic consistency:

- **Assert Element:** Specifies the expected components that should appear on the current page, which is used to verify the structural completeness of the interface and detect missing or misplaced elements.
- **Assert Color:** Specifies the color and style attributes of key elements, used to assess visual consistency and conformity with design specifications.
- **Assert Layout:** Evaluates the visual structural similarity between a specified page and its corresponding reference UI screenshot.

The interaction operations simulate concrete user interactions:

- **Click:** Declares a click action on a specific component to simulate clicking behavior and evaluate its interactivity.
- **Input Text:** Declares a text-input operation within the target input field.
- **Dropdown Selection:** Declares a selection from a dropdown menu.

Regarding the construction pipeline, we first utilize Gemini-3-Flash to generate a draft IES based on the task instruction. Subsequently, a rule-based script performs consistency validation: each step defined in the IES is precisely matched against the Metadata and instruction, while strictly constraining interaction operations to occur only on components with navigation attributes. This rigorous process ensures the reliability of the IES, enabling closed-loop verification of both visual rendering and interaction logic for the generated GUI apps.

Rule-based Evaluation: We leverage the Assistive Technology Service Provider Interface (AT-SPI) to achieve component-level access and manipulation. AT-SPI is a Linux-supported system interface that provides access to the internal structure of GUI apps through the accessibility tree. This allows us to not only retrieve metadata such as component positions, hierarchies, states, and visibility to support the **Assert Element** operation but also to execute **Click**, **Input Text** and **Dropdown Selection** operations.

For **Assert Color**, since AT-SPI does not directly expose color or style attributes, we further utilize the component coordinates it provides to capture the corresponding screen regions at runtime. We then perform a pixel-wise color comparison by calculating the Euclidean distance d in the RGB color space between the rendered pixels and the target color; a threshold of $d < 80$ is applied to determine if the assertion passes.

Model-based Evaluation: For **Assert Layout**, while rule-based evaluation enables automated interaction and basic verification, it remains limited in identifying global rendering deviations such as disproportionate scaling, overlapping elements, or misalignments. To address this, we trained a specialized visual evaluation model utilizing a dual-stream architecture based on ResNet-50 [9]. This model processes pairs of reference screenshots and generated screenshots as input to learn high-dimensional visual representations in the feature space. It ultimately outputs a similarity score $s \in [0, 1]$, thereby quantifying the perceptual discrepancy between the rendered interface and the authentic target.

Due to the scarcity of large-scale annotated data for GUI layout alignment, we designed an automated synthesis pipeline to generate training samples via procedural perturbations of real interfaces. Specifically, we utilized Gemini-3-Flash to reconstruct 2048 GUI interfaces from our benchmark into high-fidelity HTML/CSS code representations. To further enhance the model's generalization to real-world distributions, we additionally integrated 484 real-webpage instances from Design2Code [22].

For each original instance, ten variants were constructed to cover a diverse range of layout scenarios, ranging from minor fluctuations to complete structural failure. First, three variants underwent random proportional scaling within a $\pm 30\%$ range; as scaling preserves the layout structure, their similarity labels were set to 1.0. This subset encourages the model to focus on overall layout correctness rather than pixel-level differences. Subsequently, by randomly injecting varying degrees of element deletion, positional shifts, layout collapse and style tampering at the HTML/ CSS code level, we constructed five additional variants with increasing layout damage. During this process, we introduced a weight-based penalty mechanism that linearly deducts the initial score (1.0) based on the extent of structural damage, to assign precise labels to these variants. The last two varants were randomly selected from other unrelated instances, with labels set to 0. This construction strategy ensures that the model can learn a full spectrum of visual representations.

In total, we curated a dataset of 25k pairs, split 8:1:1 for training/validation/ testing. We evaluate the visual evaluation model on the held-out test set and further invite human annotators to conduct a blind preference study on 200 randomly sampled tasks from InteractGUI Bench. It achieves a low MAE of 0.09 while reaching an 81.45% agreement rate with human judgments. These results demonstrate that the model not only detects fine-grained layout discrepancies but also aligns well with human perception, providing a robust visual assessment for GUI generation tasks.

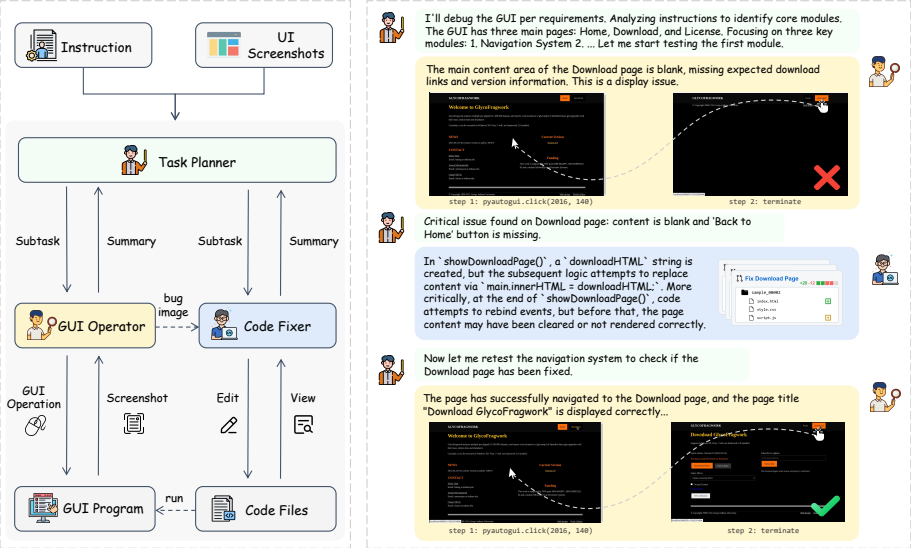

Fig. 3: Overall architecture (left) and interaction details(right) of VF-Coder. Zoom in to see better. The right presents a real case of VF-Coder: The GUI Operator detects a rendering error on the download page through visual interaction. Under the coordination of the Task Planner, the Code Fixer successfully identifies and repairs the issue.

4 VF-Coder

Fig. 3 shows the overall framework of our proposed VF-Coder, which consists of three agents, including Task Planner, Code Fixer, and GUI Operator, that collaboratively analyze and repair logical and visual errors of GUI apps, effectively enhancing code quality.

4.1 Task Planner

The Task Planner serves as the central controller of VF-Coder, responsible for task decomposition, agent coordination, and overall workflow management. It consists of a MLLM equipped with a session memory module. Upon receiving task instruction and UI screenshots, the Task Planner divides the task into a sequence of subtasks and assigns them to the appropriate agents (*i.e.* GUI Operator for runtime interaction and validation or Code Fixer for code modification).

After each subtask, Task Planner collects the agents' feedback, updates its internal state, and determines the next action accordingly. Once all detected issues are resolved, the Task Planner terminates the debugging process and produces the final GUI app.

4.2 GUI Operator

GUI Agents (or Computer-Use Agent, CUA) [24,25,34] have emerged as a prominent research frontier in recent years. By transforming the user instruction into step-by-step executable operations on the screen, GUI Agents are revolutionizing the paradigm of human-computer interaction [20]. Inspired by this paradigm, we design GUI Operator, a vision-capable GUI agent that autonomously navigates and interacts with app interfaces to identify runtime execution anomalies.

The GUI Operator receives the task instructions, screenshots, and specific inspection task from the Task Planner. Upon invocation, the GUI Operator first initializes and launches the target app within an isolated sandbox environment. By perceiving real-time screenshots and terminal logs, the operator predicts and executes the corresponding interaction operations. The environment then automatically updates and returns the interface state, facilitating subsequent interactions within an iterative feedback loop. Through this process, the GUI Operator can proactively detect logical and visual errors during runtime. Once an issue is detected, it immediately terminates the current session and sends feedback to the Task Planner for subsequent repair and revalidation steps.

4.3 Code Fixer

The Code Fixer is responsible for fixing the GUI code based on the task instruction, UI screenshots and the bug description provided by the Task Planner. After being invoked, it automatically parses the source files, localizes the erroneous code segments, and synthesizes corresponding patches. After completing the repair, the Code Fixer summarizes the modifications to the Task Planner, which subsequently determines whether the app requires further validation or additional debugging iterations.

4.4 Memory and Communication Management

Each agent maintains an independent session memory that records its local context within the current debugging session. After completing a subtask, both GUI Operator and Code Fixer send feedback to the *Task Planner*, which integrates it into its global context for subsequent decision-making. To ensure task focus and prevent cross-interference between subtasks, temporary histories of GUI Operator and Code Fixer are cleared after each execution, while the Task Planner retains a persistent high-level memory of the overall workflow.

4.5 Discuss with Kimi agent

We note that the Kimi agent recently introduced a feature for debugging GUI applications using screenshots. Consequently, a direct comparison or in-depth analysis is not feasible. To still provide context, we conduct a manual evaluation of its performance on our proposed InteractGUI Bench. The results of this evaluation are presented and discussed in the Experiment section.

5 Experiments

5.1 Experimental Setup

Models and Code Agent Frameworks: In this section, we first conduct a systematic evaluation of five mainstream MLLMs on InteractGUI Bench to establish performance baselines, including GPT-5.2 [21], Claude-4.5-Sonnet [2], Gemini-3.1-Pro [7], Gemini-3-Flash [6] and Kimi-2.5 [28].

Then, we use Gemini-3-Flash as the backbone model, comparing our VF-Coder with two representative frameworks: 1) Gemini CLI [27], an open-source AI agent developed by Google that performs autonomous planning, shell command execution, and web access; 2) Cursor CLI [26], the command-line version of Cursor that provides full access to Cursor's features, including codebase indexing and automated code refinement via linter integration.

Implementation Details: All models and code agents are prompted to generate GUI code based on PySide6 (see Supplementary for detailed prompts). We keep the default configurations for all Baseline models and compared frameworks. For VF-Coder, to balance inference cost and performance, we set the maximum number of planning iterations for the Task Planner to 10. The GUI Operator is allowed to execute at most 5 steps for each subtask while retaining the most recent 4 interaction histories as context. The framework automatically terminates if the step limit is reached.

Evaluation Metrics Following SWE-bench [14], we adopt **% Resolved** (pass@1), **$ Avg. Cost** and **Avg. Visual Score** to evaluate task completion, efficiency, and visual fidelity: % Resolved represents the percentage of tasks that pass all operations defined in the IES; $ Avg. Cost denotes the average economic cost per task. Under the same model settings, this directly reflects token consumption and computational overhead; Avg. Visual Score measures the average visual similarity between rendered and ground-truth interfaces via our pre-trained visual evaluation model.

To further identify performance bottlenecks, we additionally report four fine-grained metrics: **FS** (Fail-to-Start), **AE**, **AC**, and **CK**. FS is the ratio of tasks that fail to launch due to compilation or runtime errors; AE is the success rate of **Assert Element** operations, reflecting the presence of essential UI components; AC is the success rate of **Assert Color** operations, reflecting the accuracy of visual style; CK is the success rate of **Click** operations, reflecting the logical integrity of interactive functionalities.

5.2 Results on proposed InteractGUI Bench

Baseline Models Performance: Based on the experimental results in Table 1 , we make the following observations: 1) As shown in the table, GPT-5.2 achieves the highest task success rate at 26.99%, closely followed by Gemini-3.1-Pro.

Table 1: Performance comparison of baseline LLMs under Desktop (Pyside6) settings on proposed InteractGUI Bench. All metrics are reported in percentages except Avg. Cost and Avg. Visual Score. Best results are **bolded**.

Model	% Resolved ↑	$ Avg. Cost ↓	Avg. Visual Score ↑	FS ↓	AE ↑	AC ↑	CK ↑
GPT-5.2	**26.99**	0.1746	**0.4839**	**14.53**	**58.28**	**67.53**	**72.52**
Claude-4.5-Sonnet	23.18	0.1563	0.3378	15.57	53.94	62.75	66.50
Gemini-3.1-Pro	23.57	0.2076	0.4816	21.66	51.47	57.36	61.03
Gemini-3-Flash	21.68	0.0147	0.4284	25.20	49.05	58.35	58.62
Kimi-2.5	11.31	**0.0202**	0.3867	27.08	34.04	45.86	45.77

Table 2: Performance comparison of related code agent methods and VF-Coder under the Desktop (Pyside6) setting.

Method	% Resolved ↑	$ Avg. Cost ↓	Avg. Visual Score ↑	FS ↓	AE ↑	AC ↑	CK ↑
Gemini-3-Flash	21.68	0.0147	0.4284	25.20	49.05	58.35	58.62
+Gemini CLI	24.81	**0.1309**	0.5190	7.14	50.69	66.91	61.66
+Cursor CLI	25.46	0.1726	0.5028	**1.59**	53.45	66.95	**73.64**
+VF-Coder	**28.29**	0.2064	**0.5584**	4.47	**58.41**	**75.93**	71.33

This indicates that even the top-performing models are still far from saturating the InteractGUI Bench, highlighting that our benchmark remains significantly challenging for current MLLMs. 2) In terms of visual similarity, GPT-5.2 also achieves the best performance with a score of 0.4839. There exists a loose correlation between visual scores and task success rates, as fully functional GUI applications are relatively less prone to severe visual discrepancies.

VF-Coder vs. Text-Only Agents: As shown in Tab. 2, with the integration of our VF-Coder framework, Gemini-3-Flash achieves a % Resolved rate of 28.29%, outperforming the strongest baseline, GPT-5.2 (26.99%), with comparable cost ($0.2064 vs. $0.1746). Despite introducing additional overhead, VF-Coder achieves the best cost-performance efficiency among all compared agents, measured by $\frac{\Delta\%\text{Resolved}}{\Delta\$\text{Avg. Cost}}$ (34.48 vs. Cursor CLI's 23.9 and Gemini CLI's 26.9). Beyond % Resolved, the substantial improvements in Avg. Visual Score (0.4284 → 0.5584), AE (49.05 → 58.41), AC (58.35 → 75.93), and CK (58.62 → 71.33) further show clear gains at both the visual and element levels, making the additional cost a reasonable trade-off. These results indicate the effectiveness of VF-Coder in fixing visual problems.

In contrast, while Gemini CLI and Cursor CLI decrease the 'FS' to 7.14% and 1.59% respectively, their impact on visual integrity is limited. This suggests that although text-based CLI feedback helps resolve execution-level errors, it lacks the visual information required to maintain layout consistency. This demonstrates that traditional text-based agent frameworks have inherent limitations in GUI tasks, whereas visual feedback provides substantial informational gain for GUI debugging and repair.

Table 3: Performance comparison of Kimi Agent and VF-Coder under the Desktop (PySide6) setting. Some metrics are not reported due to the absence of a public API.

Method	% Resolved ↑	Avg. Visual Score ↑	FS ↓	AE ↑	AC ↑	CK ↑
Kimi-K2.5	16.25	0.4865	10.00	43.67	44.73	42.28
+Kimi Agent	18.75	**0.5648**	**1.25**	**54.28**	61.04	52.31
+VF-Coder	**25.00**	0.5530	3.75	52.50	**66.04**	**61.66**

Comparison with Kimi agent: As discussed in Sec. 4.5, we include a comparison with the Kimi agent to provide additional context. Due to the absence of a public API for the Kimi agent, direct programmatic evaluation was not feasible. Instead, we conducted a manual evaluation by uploading the screenshots and prompts from a subset of 80 applications from our InteractGUI Bench to its web interface and recording the responses.

The results of this manual comparison are presented in Tab. 3. Our proposed VF-Coder achieves a much higher '% Resolved' score (25.00 vs. 18.75) on this subset, showing the strong effectiveness of our approach.

5.3 Ablation Study

To further analyze the role of visual feedback, we decompose the GUI code generation task into the debugging and fixing stages. All ablation experiments are conducted on the InteractGUI Bench using Gemini-3-Flash as the base model. In the debugging stage, we investigate the impact of visual interaction by enabling or disabling the GUI Operator within VF-Coder (denoted as w/ or w/o GUI Operator). When the GUI Operator is disabled (w/o GUI Operator), the system relies exclusively on textual interactions, such as inspecting source code or executing shell commands, to perform debugging tasks. In the fixing stage, we examine the effect of visual feedback by controlling whether the Code Fixer receives the GUI screenshot captured at the moment the bug is detected (w/ or w/o bug screenshot). Without the screenshot (*w/o bug screenshot*), the Code Fixer lacks direct visual access to the interface state at the time of the error. Instead, it must rely solely on the textual bug descriptions provided by the GUI Operator to locate and repair the faulty code.

As shown in Tab. 4, the contribution of visual feedback varies significantly across stages. In the debugging stage, incorporating visual information (w/o bug screenshot) increases the task resolution rate (%Resolved) from 24.79% under text-only interactions (w/o GUI Operator) to 27.64%, an improvement of 2.85%, while the average visual quality score (Avg. Visual Score) rises from 0.5193 to 0.5390. This indicates that visual feedback plays a crucial role in error detection and interface state understanding, and can effectively improve the visual quality of generated interfaces. In the fixing stage, further introducing visual feedback (Full) yields only marginal gains in task resolution rate, with a mere 0.65% increase (27.64%→28.29%). However, there remains improvement in visual fidelity metrics (Avg. Visual Score increases from 0.5390 to 0.5584), suggesting that bug

Table 4: Ablation study on the effects of different agents and visual feedback.

	Debugging	Fixing	% Resolved ↑	$ Avg. Cost ↓	Avg. Visual Score ↑	FS ↓	AE ↑	AC ↑	CK ↑
w/o GUI Operator	textual	textual	24.79	**0.1351**	0.5193	3.25	50.69	61.91	59.63
w/o bug screenshot	visual	textual	27.64	0.1963	0.5390	5.28	53.45	66.95	**73.64**
Full	visual	visual	**28.29**	0.2064	**0.5584**	**4.47**	**58.41**	**75.93**	71.33

screenshots are helpful in conveying visual errors but have limited utility for other types of errors such as functional logic.

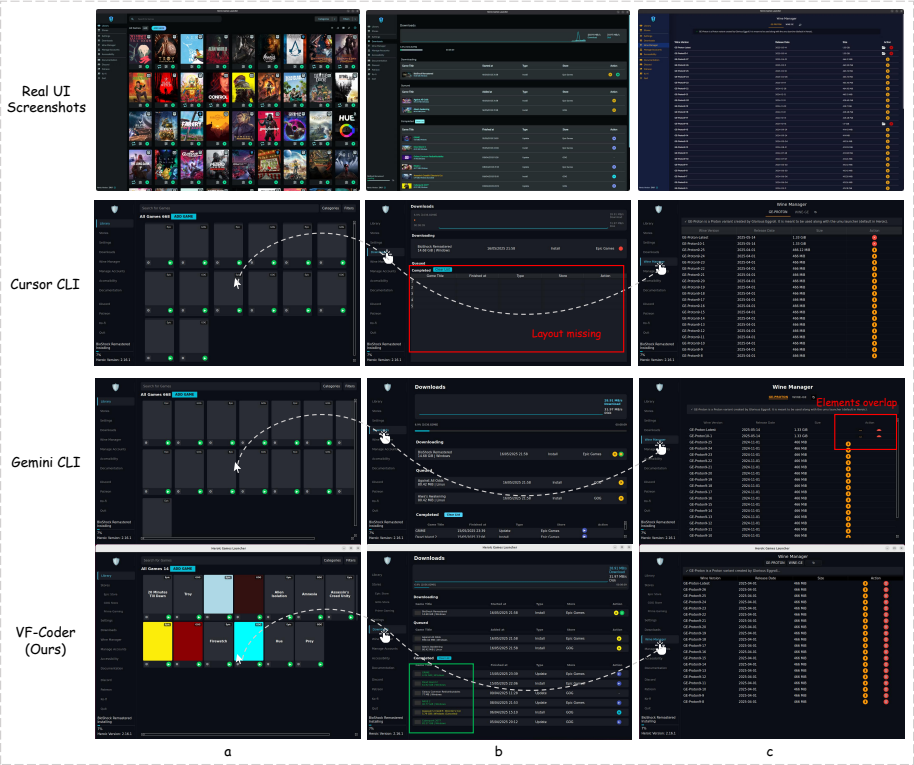

Fig. 4: Visual comparison of Cursor CLI, Gemini CLI and VF-Coder in a real case.

5.4 Case Study

To concretely illustrate the advantages and working mechanism of VF-Coder, we visualize the evaluation trajectories of Cursor CLI, Gemini CLI, and VF-Coder on a representative task in Figure 5.4. Due to the page limit, more comparison

cases are presented in the appendix. Our key observations are as follows: 1) VF-Coder achieves the closest visual fidelity to the Real UI Screenshots. Cursor CLI exhibits significant layout deviations in interface b, while Gemini CLI, suffers from element overlapping in interface c. These are difficult-identified problems for text-output-based methods. With the support of visual perception, VF-Coder not only avoids these obvious visual errors , but also attempts to align with fine-grained style details, such as the precise color matching of fonts in interface b (highlighted by green boxes in the figure), which shows the advantage of Visual Feedback. 2) Despite VF-Coder achieving the best performance, all methods still exhibit visible gaps compared to the Real UI Screenshots. This limitation likely stems from the inherent capability boundaries of base models, reflecting the common challenges faced by current approaches in generating real-world desktop GUI applications.

6 Conclusion

Existing code agents primarily rely on textual feedback to ensure the quality of generated code, which presents significant limitations in GUI code generation scenarios. To systematically evaluate this challenge, we propose InteractGUI Bench, the first benchmark that assesses GUI code generation through interactive execution and visual verification. By simulating realistic user behaviors and jointly examining functional logic and visual structure, this benchmark provides a comprehensive measurement of model capability in GUI contexts. Building upon it, we further introduce VF-Coder, a vision-feedback-based multi-agent framework that integrates interface perception and interactive exploration directly into the debugging loop. Extensive experiments show that VF-Coder significantly improves the reliability of GUI code over text-only agents, highlighting the crucial role of visual feedback in understanding interface states and repairing logical or rendering defects. We hope this study lays a foundation for future research on GUI code generation and visually grounded software intelligence.

Acknowledgement

This research was supported in part by the Young Scientists Fund of the Shenzhen Natural Science Foundation under grant QNXMC20250701093812017, in part by the Science, Technology, and Innovation Project of Shenzhen Longhua District under grant 20260309G23410662, in part by the Future City Research Institute Project, and in part by the Domestic Joint Project Research Program of Inner Mongolia under grant 2048712223181766656.

References

1. Acharya, M., Zhang, Y., Leach, K., Huang, Y.: Optimizing code runtime performance through context-aware retrieval-augmented generation. CoRR **abs/2501.16692** (2025). https://doi.org/10.48550/ARXIV.2501.16692, https://doi.org/10.48550/arXiv.2501.16692
2. Anthropic: Claude sonnet 4.5. https://www.anthropic.com/claude/sonnet (2025)
3. Antoniades, A., Örwall, A., Zhang, K., Xie, Y., Goyal, A., Wang, W.Y.: SWE-search: Enhancing software agents with monte carlo tree search and iterative refinement. In: The Thirteenth International Conference on Learning Representations (2025), https://openreview.net/forum?id=G7sIFXugTX
4. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al.: Program synthesis with large language models. arXiv preprint arXiv:2108.07732 (2021)
5. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.D.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
6. Google, D.: Gemini 3 flash: Best for frontier intelligence at speed. https://deepmind.google/models/gemini/flash/ (2025)
7. Google, D.: Gemini 3.1 pro: Best for complex tasks and bringing creative concepts to life. https://deepmind.google/models/gemini/pro/ (2026)
8. Gupta, T., Weihs, L., Kembhavi, A.: Codenav: Beyond tool-use to using real-world codebases with llm agents. arXiv preprint arXiv:2406.12276 (2024)
9. He, K., Zhang, X., Ren, S., Sun, J.: Identity mappings in deep residual networks. In: European conference on computer vision. pp. 630–645. Springer (2016)
10. Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., et al.: Measuring coding challenge competence with apps. arXiv preprint arXiv:2105.09938 (2021)
11. Hu, Y., Zhou, Q., Chen, Q., Li, X., Liu, L., Zhang, D., Kachroo, A., Oz, T., Tripp, O.: Qualityflow: An agentic workflow for program synthesis controlled by llm quality checks. arXiv preprint arXiv:2501.17167 (2025)
12. Jain, N., Han, K., Gu, A., Li, W.D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., Stoica, I.: Livecodebench: Holistic and contamination free evaluation of large language models for code. In: The Thirteenth International Conference on Learning Representations (2025), https://openreview.net/forum?id=chfJJYC3iL
13. Jiang, X., Dong, Y., Tao, Y., Liu, H., Jin, Z., Li, G.: Rocode: Integrating backtracking mechanism and program analysis in large language models for code generation. In: ICSE. pp. 334–346 (2025), https://doi.org/10.1109/ICSE55347.2025.00133
14. Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., Narasimhan, K.R.: SWE-bench: Can language models resolve real-world github issues? In: The Twelfth International Conference on Learning Representations (2024), https://openreview.net/forum?id=VTF8yNQM66
15. Laurençon, H., Tronchon, L., Sanh, V.: Unlocking the conversion of web screenshots into html code with the websight dataset. arXiv preprint arXiv:2403.09029 (2024)
16. Li, J., Li, G., Zhang, X., Zhao, Y., Dong, Y., Jin, Z., Li, B., Huang, F., Li, Y.: Evocodebench: An evolving code generation benchmark with domain-specific evaluations. In: NeurIPS (2024), http://papers.nips.cc/paper_files/paper/2024/hash/6a059625a6027aca18302803743abaa2-Abstract-Datasets_and_Benchmarks_Track.html

17. Li, J., Li, G., Zhao, Y., Li, Y., Liu, H., Zhu, H., Wang, L., Liu, K., Fang, Z., Wang, L., Ding, J., Zhang, X., Zhu, Y., Dong, Y., Jin, Z., Li, B., Huang, F., Li, Y., Gu, B., Yang, M.: Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. In: ACL (Findings). pp. 3603–3614 (2024), https://doi.org/10.18653/v1/2024.findings-acl.214
18. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al.: Competition-level code generation with alphacode. Science **378**(6624), 1092–1097 (2022)
19. Liu, Y., Liu, Y., Yuan, F., Cao, C., Sun, Y., Peng, K., Chen, W., Li, J., Ma, Z.: Opera: A reinforcement learning–enhanced orchestrated planner-executor architecture for reasoning-oriented multi-hop retrieval. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 40, pp. 32258–32266 (2026)
20. Nguyen, D., Chen, J., Wang, Y., Wu, G., Park, N., Hu, Z., Lyu, H., Wu, J., Aponte, R., Xia, Y., et al.: Gui agents: A survey. arXiv preprint arXiv:2412.13501 (2024)
21. OpenAI: Introducing gpt-5.2. https://openai.com/zh-Hans-CN/index/introducing-gpt-5-2/ (2025)
22. Si, C., Zhang, Y., Li, R., Yang, Z., Liu, R., Yang, D.: Design2code: Benchmarking multimodal code generation for automated front-end engineering. In: NAACL (Long Papers). pp. 3956–3974 (2025), https://doi.org/10.18653/v1/2025.naacl-long.199
23. Sohrabizadeh, A., Song, J., Liu, M., Roy, R., Lee, C., Raiman, J., Catanzaro, B.: Nemotron-CORTEXA: Enhancing LLM agents for software engineering tasks via improved localization and solution diversity. In: Forty-second International Conference on Machine Learning (2025)
24. Song, L., Dai, Y., Prabhu, V., Zhang, J., Shi, T., Li, L., Li, J., Savarese, S., Chen, Z., Zhao, J., et al.: Coact-1: Computer-using agents with coding as actions. arXiv preprint arXiv:2508.03923 (2025)
25. Sun, Y., Zhao, S., Yu, T., Wen, H., Va, S., Xu, M., Li, Y., Zhang, C.: Gui-xplore: Empowering generalizable gui agents with one exploration. In: Proceedings of the Computer Vision and Pattern Recognition Conference. pp. 19477–19486 (2025)
26. Team, C.: Cursor document. https://cursor.com/cn/docs (2025)
27. Team, G.C.: Build, debug & deploy with ai. https://geminicli.com/ (2025)
28. Team, K., Bai, T., Bai, Y., Bao, Y., Cai, S., Cao, Y., Charles, Y., Che, H., Chen, C., Chen, G., et al.: Kimi k2. 5: Visual agentic intelligence. arXiv preprint arXiv:2602.02276 (2026)
29. Wang, H., Zou, H., Song, H., Feng, J., Fang, J., Lu, J., Liu, L., Luo, Q., Liang, S., Huang, S., et al.: Ui-tars-2 technical report: Advancing gui agent with multi-turn reinforcement learning. arXiv preprint arXiv:2509.02544 (2025)
30. Wang, R., Han, X., Ji, L., Wang, S., Baldwin, T., Li, H.: Toolgen: Unified tool retrieval and calling via generation. In: The Thirteenth International Conference on Learning Representations (2025), https://openreview.net/forum?id=XLMAMmowdY
31. Wu, F., Gao, C., Li, S., Wen, X.C., Liao, Q.: Mllm-based ui2code automation guided by ui layout information. Proceedings of the ACM on Software Engineering **2**(ISSTA), 1123–1145 (2025)
32. Xiao, J., Wan, Y., Huo, Y., Wang, Z., Xu, X., Wang, W., Xu, Z., Wang, Y., Lyu, M.R.: Interaction2code: Benchmarking mllm-based interactive webpage code generation from interactive prototyping. arXiv preprint arXiv:2411.03292 (2024)
33. Xu, K., Mao, Y., Guan, X., Feng, Z.: Web-bench: A llm code benchmark based on web standards and frameworks. arXiv preprint arXiv:2505.07473 (2025)

34. Yang, Y., Li, D., Dai, Y., Yang, Y., Luo, Z., Zhao, Z., Hu, Z., Huang, J., Saha, A., Chen, Z., et al.: Gta1: Gui test-time scaling agent. arXiv preprint arXiv:2507.05791 (2025)
35. Ye, H., Yang, A.Z., Hu, C., Wang, Y., Zhang, T., Le Goues, C.: Adverintent-agent: Adversarial reasoning for repair based on inferred program intent. Proc. ACM Softw. Eng. **2**(ISSTA) (Jun 2025). https://doi.org/10.1145/3728939, https://doi.org/10.1145/3728939
36. Yuan, M., Chen, J., Xing, Z., Quigley, A., Luo, Y., Luo, T., Mohammadi, G., Lu, Q., Zhu, L.: Designrepair: Dual-stream design guideline-aware frontend repair with large language models. In: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). p. 2483–2494. IEEE (Apr 2025). https://doi.org/10.1109/icse55347.2025.00109, http://dx.doi.org/10.1109/ICSE55347.2025.00109
37. Zhang, K., Li, J., Li, G., Shi, X., Jin, Z.: Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In: ACL (1). pp. 13643–13658 (2024), https://doi.org/10.18653/v1/2024.acl-long.737
38. Zhang, K., Li, Z., Li, J., Li, G., Jin, Z.: Self-edit: Fault-aware code editor for code generation. arXiv preprint arXiv:2305.04087 (2023)
39. Zhang, Y., Zhao, X., Wang, Z.Z., Yang, C., Wei, J., Wu, T.: cast: Enhancing code retrieval-augmented generation with structural chunking via abstract syntax tree. arXiv preprint arXiv:2506.15655 (2025)
40. Zhou, H., Zhang, X., Tong, P., Zhang, J., Chen, L., Kong, Q., Cai, C., Liu, C., Wang, Y., Zhou, J., et al.: Mai-ui technical report: Real-world centric foundation gui agents. arXiv preprint arXiv:2512.22047 (2025)