Seeing is Fixing: Cross-Modal Reasoning with Multimodal LLMs for Visual Software Issue Repair

Kai Huang[1]*, Jian Zhang[2]†, Xiaofei Xie[3], Chunyang Chen[1]
[1]Technical University of Munich, [2]Nanyang Technological University, [3]Singapore Management University
{kai-kevin.huang, chun-yang.chen}@tum.de, jian_zhang@ntu.edu.sg, xfxie@smu.edu.sg

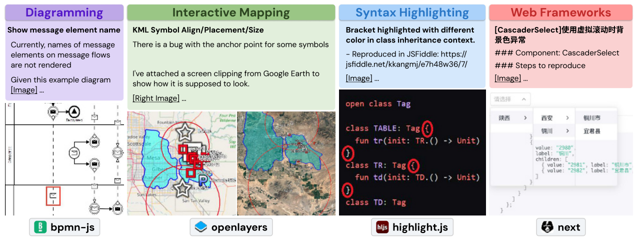
***Abstract*—Large language model (LLM)-based automated program repair (APR) techniques have shown promising results in resolving real-world github issue tasks. Existing APR systems are primarily evaluated in unimodal settings (e.g., SWE-bench), relying solely on textual issue descriptions and source code. However, these autonomous systems struggle to resolve multimodal problem scenarios (e.g., SWE-bench M) due to limitations in interpreting and leveraging visual information. In multimodal scenarios, LLMs need to rely on visual information in the graphical user interface (GUI) to understand bugs and generate fixes. To bridge this gap, we propose GUIRepair, a cross-modal reasoning approach for resolving multimodal issue scenarios by understanding and capturing visual information. Specifically, GUIRepair integrates two key components including Image2Code and Code2Image to enhance fault comprehension and patch validation. Image2Code extracts relevant project documents based on the issue report, then applies this domain knowledge to generate the reproduced code responsible for the visual symptoms, effectively translating GUI images into executable context for better fault comprehension. Code2Image replays the visual issue scenario using the reproduced code and captures GUI renderings of the patched program to assess whether the fix visually resolves the issue, providing feedback for patch validation. We evaluate GUIRepair on SWE-bench M, and the approach demonstrates significant effectiveness. When utilizing GPT-4o as the base model, GUIRepair solves 157 instances, outperforming the best open-source baseline by 26 instances. Furthermore, when using o4-mini as the base model, GUIRepair can achieve even better results and solve 175 instances, outperforming the top commercial system by 22 instances. This emphasizes the success of our new perspective on incorporating cross-modal reasoning by understanding and capturing visual information to resolve multimodal issues.**

***Index Terms*—Large Language Model, Automated Program Repair, Autonomous Programming, Multimodal Issue**

I. Introduction

Automated program repair (APR) techniques [1]–[5] aim to automatically resolve software defects, reducing manual effort and improving software quality [6]. With the advent of the large language model (LLM) era, LLM-driven APR techniques [4] show promising results in the software engineering (SE) community [7], [8]. Researchers have proposed many autonomous systems to repair software bugs [9]–[13]. Recently, the APR community has begun to focus on a new challenge, SWE-bench [14], designed to evaluate LLMs in solving real-world GitHub issue instances [15]–[20]. These issues are typically represented by textual components including natural language

* This work was done during Kai Huang's visit to Singapore Management University.
† Corresponding author: Jian Zhang (jian_zhang@ntu.edu.sg).



Show message element name
Currently, names of message elements on message flows are not rendered
Given this example diagram
[Image] ...

KML Symbol Align/Placement/Size
There is a bug with the anchor point for some symbols
I've attached a screen clipping from Google Earth to show how it is supposed to look.
[Right Image] ...

Bracket highlighted with different color in class inheritance context.
– Reproduced in JSFiddle: https://jsfiddle.net/kkangmj/e7h48w36/7/
[Image] ...

```
open class Tag

class TABLE: Tag {
  fun tr(init: TR.() -> Unit)
}
class TR: Tag {
  fun td(init: TD.() -> Unit)
}
class TD: Tag
```

[CascaderSelect]使用虚拟滚动时背景色异常
Component: CascaderSelect
Steps to reproduce
[Image] ...

Fig. 1: Four visual task instances from SWE-bench M.

and programming language, few of which contain an image (i.e., screenshot). However, many real-world software engineering tasks involve visual assets, and widely-used front-end libraries enable users to build applications with graphical user interfaces (GUIs). As shown in Figure 1, this is especially common in domains such as web development and interactive application design. When bugs occur in such applications, users often report issues in the repository, accompanied by both textual descriptions and screenshots. Developers maintaining these libraries rely not only on the code and issue descriptions but also heavily on visual cues from the screenshots to understand and diagnose the problems. Without visual context, it becomes difficult to identify the root cause of the issue or to verify whether it has been resolved.

Unfortunately, there has been limited research on automated repair for visual software issues, despite their practical significance. Existing approaches [9]–[13], [15]–[21] primarily target unimodal scenarios that involve only textual information, such as code and natural language. These methods are typically evaluated on benchmarks like SWE-bench [14], Defects4J [22], and HumanEval [23], which are not designed to address multimodal tasks involving both visual and textual elements [24]. Recent findings [24] reveal that top-performing systems on unimodal benchmarks struggle significantly when applied to multimodal settings. For example, SWE-agent [16], which performs strongly on SWE-bench, successfully solves only 12% of tasks in the SWE-bench Multimodal benchmark (SWE-bench M). This sharp performance decline highlights the unique challenges of multimodal scenarios and underscores the urgent need to advance existing APR systems to better handle visual problems.

To reveal the underlying causes, we analyze the workflows of existing AI systems and identify two key limitations that hinder their effectiveness on visual issues: **1) Insufficient Understanding of Visual Issues.** Most LLM-driven autonomous systems focus on analyzing textual issue descriptions and source code to identify and repair defects. However, they often overlook the relationship between visual outputs (e.g., screenshots) and the underlying code components. Directly incorporating images from issue reports poses challenges because LLMs typically lack the domain-specific context needed to interpret visual scenarios. They may struggle to identify which code components are responsible for the visual issue or how these components interact to produce the faulty behavior. Consequently, this limited understanding of visual symptoms can hinder accurate fault localization and reduce the effectiveness of generated patches. **2) Lack of Feedback for Repair Behaviors.** Existing autonomous systems typically rely on executing test suites to validate and select valid patches [9]–[11]. However, SWE-bench M omits test cases to prevent potential data leakage, removing a critical feedback signal for patch validation. While some approaches [16], [17] generate test cases as substitutes, they are not well-suited for visual issues. On the one hand, certain visual problems rely on pixel-level comparison in visualization-based test cases [24], which are challenging for language models to generate accurately. On the other hand, although reproducing issue scenarios can provide feedback on patch effectiveness [16], [25], most multimodal task instances in SWE-bench M lack reproduction scripts [24], making it difficult to recreate the necessary visual conditions. As a result, a key challenge is how to reliably reproduce visual scenarios to enable effective patch validation.

In this paper, we propose GUIRepair, a cross-modal transformation solution designed for multimodal scenarios. GUIRepair addresses the challenges of multimodal issue resolution by enabling seamless transformation between textual and visual information, thereby facilitating both the understanding and utilization of visual content. Specifically, GUIRepair comprises two core components: Image2Code and Code2Image. For simplicity, the term *image* is used to refer to a screenshot in this paper. Image2Code enhances bug comprehension by interpreting visual information to support accurate fault localization, while Code2Image generates visual representations from code to provide feedback for patch validation. **1) Image2Code**: Before performing fault localization, GUIRepair first selectively retrieves relevant project documentation based on the multimodal issue report to better understand the visual problem. It then uses these retrieved documents as domain knowledge within the knowledge injection paradigm to generate the code that reproduces the visual issue. This process transforms high-level visual information into its corresponding code behavior, providing essential contextual knowledge to map the visual problem to specific code components. In essence, by reasoning from visual issue scenarios to their textual code representations, GUIRepair enables large language models to develop a more comprehensive understanding of the problem using both visual and textual features. **2) Code2Image**: After generating patches, GUIRepair replays the bug using the previously generated reproduction code and captures the resulting behavior in the GUI. It then evaluates whether the visual presentation of the patched program aligns with the expected outcome described in the issue report, thereby filtering for potentially correct patches. This step provides critical feedback for patch validation by converting the abstract fixing behavior into concrete, observable visual effects. That is, by translating textual repair behaviors into visual outcomes, GUIRepair enables language models to verify patch effectiveness more intuitively through multimodal cues. Finally, these two core components are seamlessly integrated into the overall repair workflow including fault localization, patch generation and patch validation. This bidirectional transformation enables GUIRepair to reason across visual and programmatic modalities, significantly enhancing the accuracy and effectiveness of automated issue repair.

We evaluate GUIRepair on the SWE-bench M benchmark comparing against state-of-the-art autonomous systems. Our results show that GUIRepair is able to achieve the best performance (157 resolved instances) than other open-source approaches when keeping the same base model GPT-4o. In particular, GUIRepair's two key components with Image2Code and Code2Image resolved 21 more instances than the basic repair workflow, marking a 15.44% improvement. Furthermore, when using the advanced reasoning model o4-mini, GUIRepair can achieve amazing results (175 resolved instances), which solves 22 instances more than the best commercial system.

Overall, the main contributions of this paper are as follows:

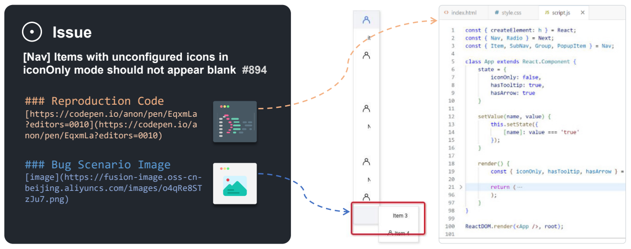
- **Dimension.** To the best of our knowledge, this is the first work to explicitly address issues with visual cues in front-end libraries, which opens a new dimension in automated issue resolution by incorporating textual and visual modalities.
- **Technique.** We propose GUIRepair, a bidirectional cross-modal transformation approach to solve multimodal software development problems. GUIRepair performs image-to-code transformation to diagnose issues by extracting semantic features from GUI screenshots and mapping them to code-level representations. Furthermore, it applies code-to-image transformation to validate repair patches by rendering the updated GUI and assessing its visual correctness.
- **Evaluation.** We conduct extensive experiments on the popular SWE-bench M benchmark, comparing GUIRepair against a range of open-source and commercial autonomous systems. GUIRepair resolves 157 issues, outperforming all existing approaches. Notably, its two key components (Image2Code and Code2Image) contribute a 15.44% improvement in overall effectiveness, highlighting the impact of cross-modal reasoning in addressing complex multimodal program repair tasks. We have released our code and experimental data [26] for public use.

II. Motivation

In this section, we analyze the gap of existing autonomous systems in resolving visual software problems, which motivates us to design a cross-modal reasoning approach.

A. Challenge 1: Fault Understanding.

In unimodal scenarios, user-described issue symptoms are typically conveyed through text [14]. However, multimodal issue reports often involve both textual and visual elements [24].

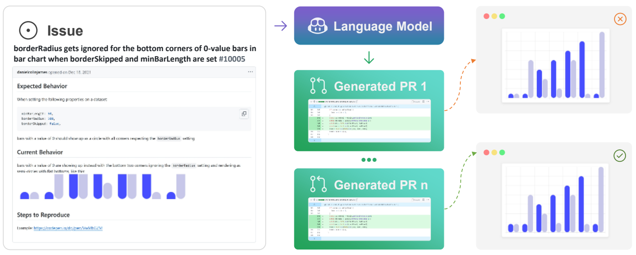

Issue

[Nav] Items with unconfigured icons in iconOnly mode should not appear blank #894

```
### Reproduction Code
[https://codepen.io/anon/pen/EqxmLa?editors=0010](https://codepen.io/anon/pen/EqxmLa?editors=0010)

### Bug Scenario Image
[image](https://fusion-image.oss-cn-beijing.aliyuncs.com/images/o4qRe8STzJu7.png)
```

```
const { createElement: h } = React;
const { Nav, Radio } = Next;
const { Item, SubNav, Group, PopupItem } = Nav;

class App extends React.Component {
    state = {
        iconOnly: false,
        hasTooltip: true,
        hasArrow: true
    }

    setValue(name, value) {
        this.setState({
            [name]: value === 'true'
        });
    }

    render() {
        const { iconOnly, hasTooltip, hasArrow } =

        return (⋯
        );
    }
}

ReactDOM.render(<App />, root);
```

Fig. 2: A motivating example of alibaba-fusion/next-895 [28].


Issue

borderRadius gets ignored for the bottom corners of 0-value bars in bar chart when borderSkipped and minBarLength are set #10005

danielcolinjames opened on Dec 18, 2021

Expected Behavior

When setting the following properties on a dataset:

```
minBarLength: 90,
borderRadius: 100,
borderSkipped: false,
```

bars with a value of 0 should show up as a circle with all corners respecting the borderRadius setting

Current Behavior

bars with a value of 0 are showing up instead with the bottom two corners ignoring the borderRadius setting and rendering as semi-circles with flat bottoms, like this:

Steps to Reproduce

Example: [illegible]

Fig. 3: A motivating example of chartjs/Chart.js-10157 [34].

As shown in Figure 2, these reports usually contain three layers of information: 1) natural language descriptions, 2) visual issue images, and 3) reproduced code that links symptoms to underlying components. Text alone often fails to capture visual issues. A recent study [24] found that 80% of visual symptoms cannot be fully described in text, and images are essential for issue-solving in 83.5% of cases.

Even when text and images are available, models may still struggle to analyze visual symptoms of users' applications and identify root causes in the used front-end library. This is because LLMs may lack human expert knowledge about the specific front-end project [27], making it difficult for them to understand the code behavior behind the visual information. This gap stems from the lack of reproduced code, which clarifies the relationship between visual symptoms and code behavior. Yet, only 17% of SWE-bench M reports include such code [24], leaving most cases without sufficient context. This highlights a key challenge in multimodal bug resolution: inferring code behavior from visual symptoms for effective fault localization.

B. *Challenge 2: Patch Validation.*

AI systems typically generate multiple candidate patches, but only the top-ranked one is selected in order to best meet developers' needs (e.g., SWE-bench uses the Pass@1 metric). Therefore, accurately identifying which patches are effective is a crucial post-processing step. However, most current systems typically cannot perform patch validation on visual scenes directly [29], limiting their ability to provide meaningful feedback on patch correctness.

While some approaches validate patches in unimodal settings using textual test cases [17], generating reliable visual oracles that involves pixel-level accuracy remains difficult, especially for issues requiring visual testing [24], [30], [31]. For example, data visualization and interactive mapping frameworks like *Chart.js* [32] and *OpenLayers* [33] often rely on pixel-level visual tests to verify correctness. As shown in Figure 3, we illustrate a *Chart.js* issue [34]: *borderRadius is ignored for the bottom corners of 0-value bars when* `borderSkipped` *and* `minBarLength` *are set*. Different proposed fixes (PR 1–N) produce subtle but distinct visual effects. Therefore, unlike in textual domains, generating such image test cases is infeasible for LLMs, as they cannot produce accurate visual references. This highlights a central challenge in visual patch validation: capturing and assessing visual effects as the repair feedback.

III. Approach

In this section, we introduce GUIRepair to solve visual software issues by reasoning the correlation between visual information from user-submitted reports and code components from the front-end library. Inspired by the recent success of Agentless in SWE tasks [17], GUIRepair adopts a similar pipeline, which is organized into four main steps: Fault Comprehension, Fault Localization, Patch Generation, and Patch Validation. Figure 4 illustrates the overall workflow. In the fault comprehension phase (❶–❷), GUIRepair first mines project-specific knowledge and generates reproduction code to help LLMs better understand the issue of the library. This reproducible code is essential for replicating the problem scenario in user applications. In the fault localization phase (❸–❹), GUIRepair leverages the knowledge injection paradigm [35], utilizing the domain knowledge acquired in the previous phase to assist LLMs in accurately identifying the code components associated with specific visual elements. During the patch generation phase (❺), GUIRepair prompts the LLM to perform multiple sampling rounds, producing candidate patches for the library based on the previously learned knowledge of the bug scenario. Finally, in the patch validation phase (❻–❼), instead of generating test cases, GUIRepair captures the visual effects of generated patches by rendering GUI of the reproduced code and provides feedback to select the most promising patch.

A. *Fault Comprehension*

As highlighted in Section II-A, existing AI systems often struggle with multimodal software engineering (SWE) tasks due to limited understanding of the interplay between visual and code representations. This lack of multimodal comprehension impairs effective fault localization. To address this challenge, GUIRepair introduces the Image2Code component, which enhances fault comprehension by enabling models to acquire project-specific domain knowledge and reason about the code components behind visual scenarios. As shown in Figure 4, Image2Code comprises two key steps: ❶ Knowledge Mining, and ❷ Reproduction Generation.

1) Knowledge Mining: To facilitate effective reasoning, it is essential for the LLM to familiarize itself with the relevant project context such as usage patterns [36], so as to act more like a human developer. As shown in Figure 5a, project documents (docs) contain important domain knowledge about what components are in the project and how to use them. Hence, GUIRepair collects project-specific knowledge and incorporates

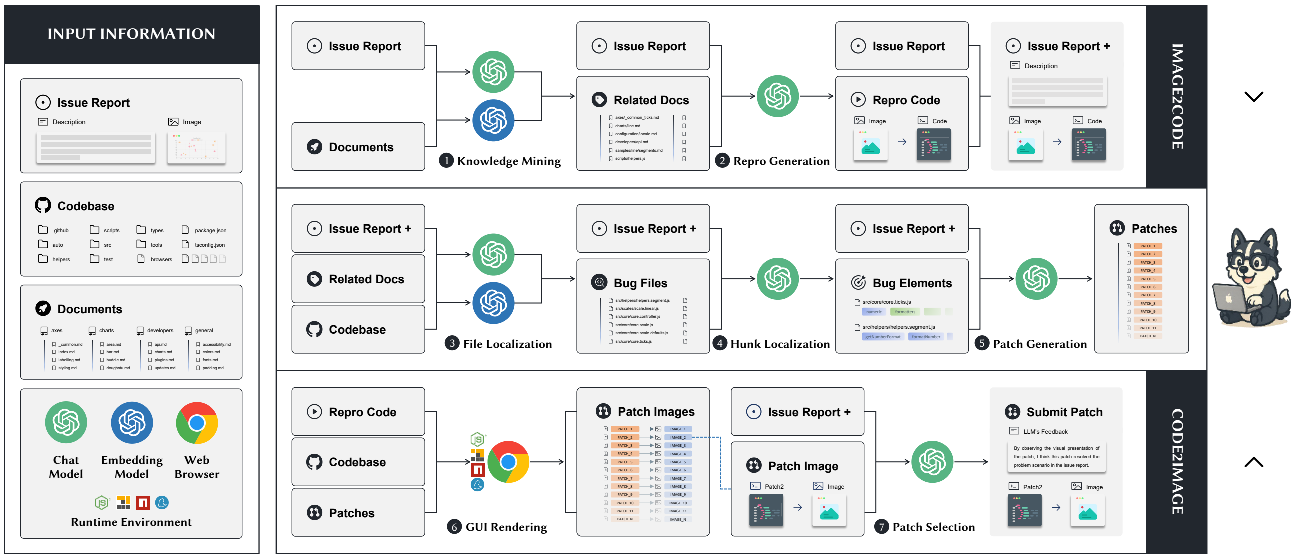
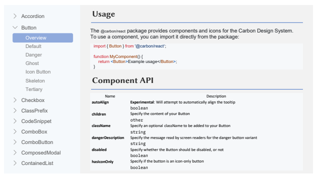
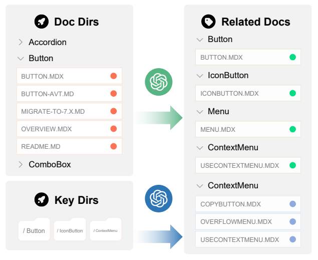


Fig. 4: The overview of GUIRepair. Note: The left shows the system input, and the right illustrates the step-by-step repair workflow.

it using the knowledge injection paradigm [35], enabling the model to better comprehend the problem in subsequent reasoning tasks. As illustrated in Figure 4, GUIRepair takes both the issue report and project documentation as input. By processing the issue report, the model heuristically identifies and selects documents that are most relevant to the described scenario. Specifically, we first prompt the chat model (e.g., GPT-4o [37]) to analyze the issue report and, based on its initial understanding, identify potentially relevant documents from the project's documentation directory. Due to the limited context window of LLMs, it is impractical to load and process all documents at once. Considering how human developers browse documentation, we instead provide only the directory structure, allowing the model to select the Top-N most relevant document filenames. However, selecting documents solely based on filenames may miss important content, as the model cannot access the full text during this step. To address this, and inspired by retrieval-augmented generation (RAG) techniques in Agentless [17], we further use an embedding model (e.g., text-embedding-3 [38]) to retrieve relevant content from the documentation repository. This retrieval is guided by the issue report and the chat model's preliminary understanding. To reduce retrieval costs in large projects with hundreds or thousands of files, we limit the search to key directories identified in the previous chat model phase and return the Top-



(a) Example of doc content.

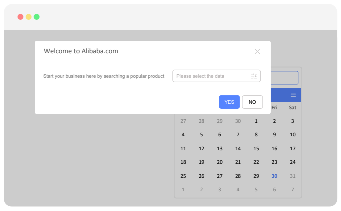


(b) Example of mined docs.

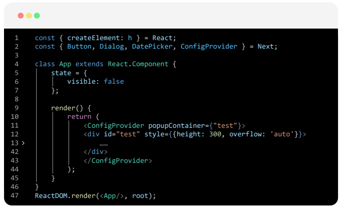
Fig. 5: Example of documents.

N documents with the highest similarity scores. Ultimately, we obtain two sets of relevant documents: one selected by the chat model through directory browsing, and another retrieved via the embedding-based similarity search. In Figure 5b, these two sets are merged to form a final collection of contextually relevant documents, which helps LLMs acquire project knowledge related to the issue scenario.

2) Repro Generation: Similar to website generation [39], [40] and bug reproduction [25], we design the reproduction (repro) generation step to enhance the model's understanding of visual problems by generating the corresponding code that reproduces the issue. To support this, we reorganize the model input by incorporating previously mined relevant documents as domain knowledge. This domain knowledge, which is closely related to the issue scenarios, is injected directly into the prompt as reference material [35]. As shown in Figure 6, we present the repro code behind a visual issue *next-1509* [41]. Figure 6a shows the symptoms of the visual issue, where the calendar UI is incorrectly displayed below the dialog box, preventing users from directly using the calendar. The repro code in Figure 6b reveals which code components were called in the visual problem (line 1-2) and how these code components were organized to trigger the problem scenario (line 4-47). These



(a) Visual issue in *next-1509*.



```
1   const { createElement: h } = React;
2   const { Button, Dialog, DatePicker, ConfigProvider } = Next;
3
4   class App extends React.Component {
5       state = {
6           visible: false
7       };
8
9       render() {
10          return (
11              <ConfigProvider popupContainer={"test"}>
12              <div id="test" style={{height: 300, overflow: 'auto'}}>
13 >                ......
42              </div>
43              </ConfigProvider>
44          );
45      }
46  }
47  ReactDOM.render(<App/>, root);
```

(b) Repro code for *next-1509*.

Fig. 6: Example of the reproduced code and the visual issue.

code details in Figure 6b are crucial for understanding the visual problem and pinpointing the relevant code elements. The goal of this step is to enable the model to learn project-specific knowledge in a manner similar to a human expert, thereby allowing it to reconstruct the code behind the visual issue. In many multimodal scenarios, reproducing an issue requires building a complete front-end project, typically composed of HTML, CSS, and JavaScript (JS) files [24]. Among these, the JS file contains the dynamic logic that reproduces the visual problem, while the HTML and CSS files generally contain static content that does not require frequent changes. Therefore, we instruct the model to generate only the JS code needed to reproduce the multimodal scenario. This output reveals the code components responsible for the visual problem and illustrates how they are structured to trigger the faulty behavior. The generated code (as shown in Figure 6b) is then embedded into the original issue report. The resulting enriched issue report (i.e., Issue Report+), now includes the natural language description, the programming language reproduced code, and the visual representations of the issue. The Issue Report+ can be used in subsequent localization, generation, and validation processes to enhance fault understanding and implement fixes. Indeed, an LLM may not always generate completely accurate reproduction code. Minor discrepancies may exist between the generated output and the original visual report, such as differences in button sizes or text highlight colors. However, this is tolerable, as our primary goal is to expose the underlying code behavior responsible for the visual problem. As long as the reproduced code sufficiently captures the essential behavior of the issue (e.g., API misuse), it serves its purpose in helping the model understand the multimodal scenario. Furthermore, during the patch validation phase, we will re-render the issue scenario in a controlled local environment using the generated reproduction code. This eliminates noise from non-essential elements, allowing the LLM to focus more effectively on the core fault. As a result, minor detail discrepancies in the reproduced code have minimal impact on the repair process.

B. Fault Localization

After obtaining the reproduced code and related documents for the visual problem, the project-specific domain knowledge is used to enhance fault localization. As shown in Figure 4, GUIRepair performs fault localization in two stages: ❸ file localization and ❹ hunk localization.

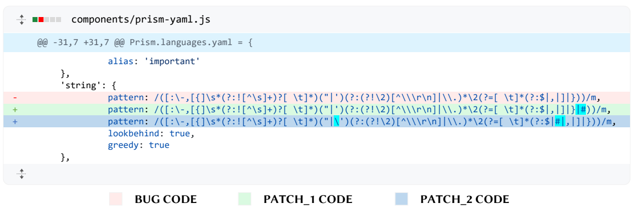
1) File Localization: In the file localization phase, GUIRepair leverages Issue Report+ (with reproduced code) to understand the fault scenario and identify potentially related code files within the codebase. To achieve this, we inject project-specific documents into the chat model, enabling it to analyze the enriched issue report and identify files likely to contain the bug. Since having the LLM read the entire codebase would exceed its context limit, we follow Agentless to extract the *repository structure format* [17]. Based on this structure, the model is prompted to return the paths of suspicious files after reviewing the overall layout of the codebase. In addition, we use the embedding model to retrieve Top-N relevant code files from the codebase based on the Issue Report+, helping to mitigate the chat model's limitation of not being able to view file content directly during the file localization phase. Similar to the document retrieval strategy described in Section III-A1, the embedding model restricts its search to the directories identified by the chat model as likely containing the bug, which reduces the overhead of scanning the entire codebase by focusing only on files within key directories. Finally, we merge the suspicious file paths identified by both the chat model and the embedding model to form the final set of suspicious files.

Note that the final set may include a dozen or more files, which can exceed the model's context limit during the hunk localization. To address this, we set a maximum number N of key bug files. If the file set exceeds this limit, we filter and retain only the Top-N most relevant files for further analysis. In detail, we provide the LLM with the code content of each suspicious file and ask it to select Top-N key bug files. Inspired by the *skeleton format* [17], we input a streamlined format of each file to avoid context limits. The original skeleton format retains class/function headers, variable declarations, and comments. GUIRepair extends this by preserving import statements to capture file-level dependencies and removing variable declarations to streamline the input. Based on this lightweight representation and its understanding of issue scenarios, the chat model identifies the Top-N key bug files.

2) Hunk Localization: In the hunk localization phase, we start with the list of key bug files and aim to identify the specific bug code hunks. We localize at the class or function level, rather than pinpointing exact code edit locations [17]. This higher-level granularity simplifies the process while still capturing meaningful fault contexts. Specifically, we provide the chat model with the full contents of all key bug files, enabling it to identify the classes or functions most likely responsible for the issue based on the enriched issue report. Unlike earlier stages that use a compressed skeleton format, this step retains complete file content, as some bugs may occur outside classes or functions (e.g., in configuration files or global code blocks). Full context is thus required for accurate fault localization. To alleviate context limitations, we compress only the content within classes and functions by defining a maximum hunk size of N lines. If a class or function exceeds this limit, we only keep its header, along with internal variable declarations and comments. The model then returns the names of all suspected bug-related classes or functions. For each identified element, we extract its complete class or function body as the localized hunk. If the bug is located outside of a class or function such as in a configuration block or global scope, the model returns the relevant element name. In these cases, we define a context window (e.g., N lines) and extract a surrounding code segment that includes the bug element, which is then used as the localized hunk.

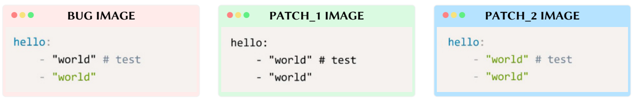
C. Patch Generation

In the ❺ patch generation process, our goal is to generate candidate patches for the bug code snippets localized in the previous phase. Here, we follow the learning-based APR



```
components/prism-yaml.js
@@ -31,7 +31,7 @@ Prism.languages.yaml = {
                alias: 'important'
        },
        'string': {
-               pattern: /([:\-,[{]\s*(?:![^\s]+)?[ \t]*)("|')(?:(?!\2)[^\\\r\n]|\\.)*\2(?=[ \t]*(?:$|,|]|}))/m,
+               pattern: /([:\-,[{]\s*(?:![^\s]+)?[ \t]*)("|')(?:(?!\2)[^\\\r\n]|\\.)*\2(?=[ \t]*(?:$|,|]|}|#))/m,
+               pattern: /([:\-,[{]\s*(?:![^\s]+)?[ \t]*)("|\')(?:(?!\2)[^\\\r\n]|\\.)*\2(?=[ \t]*(?:$|#|,|]|}))/m,
                lookbehind: true,
                greedy: true
        },
```

(a) The bug/patch_1/patch_2 code behaviors in *prism-1602*.



```
hello:
    - "world" # test
    - "world"
```

(b) The visual images effects of bug/patch_1/patch_2 code in *prism-1602*.

Fig. 7: Example of front-end visual image effects corresponding to bug/fix code behaviors (Code2Image).

work [11]–[13], [23], [42]–[63] to input class/function level bug code snippets and then ask the chat model to generate patches directly. For the format of the generated patches, we adopt the *Search/Replace edit* strategy used in Agentless [17], due to its simplicity and effectiveness. A *Search/Replace edit* consists of two key components: (1) a *search* string, representing the original buggy code to be replaced, and (2) a *replace* string, indicating the code snippet that should take its place. To integrate the patches into the project codebase, we search for occurrences of the specified *search* snippet within the source files and replace them with the corresponding *replace* snippet. This step enables seamless backfilling of the generated patches into the codebase. Finally, we convert each patch into the unified git diff format using the *git diff* command. These diff-formatted patches provide a clearer picture of the change behaviors and are used in the subsequent patch validation.

D. Patch Validation

In the LLM-driven APR workflow, LLMs typically sampled multiple times [60]–[63] to generate a diverse set of candidate patches. However, in the SWE task, a robust system is expected to achieve high accuracy using only its top-ranked patch, as measured by the Pass@1 evaluation metric. This requirement more accurately reflects the problem-solving capability of the AI system. Therefore, the goal of the patch validation phase is to identify and select the most promising patches from the generated patch space. To facilitate this process, we introduce the Code2Image component, which assesses patches by capturing the visual representation of repair behavior. The validation process consists of two phases: ❻ GUI rendering, and ❼ patch selection.

1) GUI Rendering: In this phase, we aim to provide visual feedback for patch validation by capturing how the patch alters the third-party application's behavior as rendered in the GUI. To achieve this, we leverage the reproduced code generated by the LLM during the fault comprehension phase to construct a runnable front-end project. This project is then executed in a web browser to visually reflect the effect of the applied patch within the original bug scenario. As shown in Figure 7, different repair behaviors can produce different visual effects on the front end. Although automatically replicating the execution environment would be ideal, it involves installing complex dependencies and adjusting JS package imports, which is an error-prone task for LLMs. As a practical compromise, we follow the procedure used in the SWE-bench M study [24] and perform this process manually. First, we manually set up the local runtime environment for each JS project by following the repository's developer or contributor guidelines and installing the necessary package manager and dependencies. Second, we build front-end projects to replay the bugs using the reproduced code, manually importing the built JS package into the HTML file. We then run the reproduced code in a local browser to capture a screenshot of the bug scenario. In some cases, minor manual modifications are needed to adjust package paths for correct execution. Third, we apply each candidate patch from the patch space to the buggy repository. For each patched version, we rebuild the project to obtain a new JS package and run it in the browser. The resulting visual output is captured to represent the behavior of each patch. Finally, we obtain one bug scenario image and multiple patch scenario images.

2) Patch Selection: In the patch selection phase, we evaluate the effectiveness of each patch using the previously captured visual information. Specifically, we first perform pixel-level comparisons between each patch scenario image and the bug scenario image. Patches that result in no visual change are considered ineffective and are filtered out. The remaining patch scenario images are then sequentially input to the LLM, which assesses whether the visual output reflects a successful fix based on its understanding of the issue report. Once the model identifies a patch as effective, the validation process terminates. The validated patch is then retained as the resolution for the issue. To illustrate this process, consider the *prism-1602* issue [64], where YAML strings fail to highlight when a comment appears on the same line (i.e., "world"). The buggy code (Figure 7a) results in incorrect rendering in the front-end application (Figure 7b). Our tool initially generates two candidate fixes: Patch_1 and Patch_2. Patch_1 disables syntax highlighting entirely, while Patch_2 correctly restores it. By observing these visual differences and associating with the previous understanding of issue symptoms, the LLM selects Patch_2 as the effective fix. This example demonstrates how visual effects guide the model in choosing appropriate patches.

IV. Experiment Setup

A. Research Question

- **RQ1: How well does the overall effectiveness of GUIRepair compare to state-of-the-art SWE autonomous systems?** This question evaluates GUIRepair's overall capabilities in solving multimodal issues. (Overall Effectiveness)
- **RQ2: How much does the design choice of GUIRepair contribute to the overall repair capability?** This question explores how Image2Code and Code2Image components affect the overall repair performance. (Ablation Study)
- **RQ3: Can GUIRepair generalize to additional multimodal task instances as well as other base models?** This question

assesses GUIRepair's generalization ability on additional repositories and other models. (Generalizability Study)

B. Benchmark

We evaluate GUIRepair in SWE-bench M [24], which is a benchmark for evaluating AI systems to fix bugs in visual, user-facing JavaScript software. It contains 619 task instances from 17 popular JavaScript libraries used for web interface design, diagramming, data visualization, syntax highlighting, and interactive mapping, each featuring images crucial to problem-solving. SWE-bench M is divided into two parts: Test split (SWE-bench M test) contains 517 instances. Development split (SWE-bench M dev) contains 102 instances. We use the standard test benchmark SWE-bench M test to evaluate the overall effectiveness, and the additional development set SWE-bench M dev for conducting the generalizability study.

C. Baseline

We compare GUIRepair against recent 13 SWE systems. These baseline tools represent the state-of-the-art performance on SWE-bench M. We directly select the top representative techniques on the official SWE-bench M leaderboard [65] as of May 2025 (around the paper submission time) for comparison. We include open-source as well as commercial or closed-source baselines, and the recent Computer Use Agent [66]. We additionally provide a GPT-4o [37] based implementation for Agentless Lite [29] to serve as a complementary baseline, it is currently the best open-source system on SWE-bench M.

D. Implementation

We follow recent SWE studies [17], [24] to chose gpt-4o-2024-08-06 [37] as the chat model and text-embedding-3-small [38] as the embedding model for the implementation of GUIRepair. Table I shows the specific parameter settings. More parameter setting details can be found on [26].

TABLE I: Parameter setting.

Phase	Temperature	Samples
❶	0.0	1
❷	0.0	1
❸	1.0	2
❹	1.0	2
❺	0.0/1.0	1/39
❼	0.0	1

V. Evaluation

A. RQ1: Overall Effectiveness

In the main experiment, we will investigate the overall effectiveness of GUIRepair on the SWE-bench M test. Table II presents the evaluation result of existing AI systems. The effectiveness of GUIRepair in solving multimodal instances exceeds that of all SWE systems currently on the SWE-bench M leaderboard. Specifically, GUIRepair solves 4 more task instances than the best closed-source commercial system, Globant Code Fixer Agent [67], and 17 more task instances than Zencoder [68]. This shows that GUIRepair's simple workflow is comparable to complex agent systems in terms of repair capability. Besides, GUIRepair also solves 26 more task instances than the best open-source system Agentless Lite [29]. Note that since closed-source systems do not release the base

TABLE II: Results on SWE-bench M test. Note: 🔒 refers closed-source commercial systems. Ⓐ refers claude-3-5-sonnet-20240620. ⊛ refers gpt-4o-2024-08-06.

SWE System	Base Model	Resolved	%Resolved	$Avg. Cost
Globant Code Fixer Agent [67] 🔒	NA	153	29.59%	-
Zencoder [68] 🔒	NA	140	27.08%	-
Agentless Lite [29]	Ⓐ Claude3.5	131	25.34%	$0.25
Agentless Lite [29]	⊛ GPT-4o	127	24.56%	$0.18
Computer-Use Agents [66]	⊛ GPT-4o	104	20.12%	-
SWE-agent Multimodal [24]	⊛ GPT-4o	63	12.19%	$2.94
SWE-agent [16]	Ⓐ Claude3.5	63	12.19%	$1.52
SWE-agent JavaScript [24]	Ⓐ Claude3.5	62	11.99%	$3.11
SWE-agent [16]	⊛ GPT-4o	62	11.99%	$2.07
SWE-agent Multimodal [24]	Ⓐ Claude3.5	59	11.41%	$3.11
SWE-agent JavaScript [24]	⊛ GPT-4o	48	9.28%	$0.99
Agentless [17]	Ⓐ Claude3.5	32	6.19%	$0.42
RAG [24]	⊛ GPT-4o	31	6.00%	$0.17
RAG [24]	Ⓐ Claude3.5	26	5.03%	$0.15
Agentless [17]	⊛ GPT-4o	16	3.09%	$0.38
GUIRepair	⊛ GPT-4o	157	30.37%	$0.29

TABLE III: Resolved multimodal issues by repository.

Repo	Num	GUIRepair ⊛	Globant [67] NA	Zencoder [68] NA	AgentlessL [29] Ⓐ	AgentlessL [29] ⊛	SWE-agentM [24] ⊛
next	39	9 (23.08%)	10 (25.64%)	6 (15.38%)	6 (15.38%)	3 (7.69%)	0 (0.00%)
bpmn-js	54	35 (64.81%)	34 (62.96%)	28 (51.85%)	27 (50.00%)	28 (51.85%)	15 (27.78%)
carbon	134	12 (8.96%)	12 (8.96%)	15 (11.19%)	9 (6.72%)	9 (6.72%)	2 (1.49%)
eslint	11	6 (54.55%)	2 (18.18%)	2 (18.18%)	2 (18.18%)	1 (9.09%)	0 (0.00%)
lighthouse	54	3 (5.56%)	7 (12.96%)	7 (12.96%)	4 (7.41%)	3 (5.56%)	3 (5.56%)
grommet	21	1 (4.76%)	1 (4.76%)	4 (19.05%)	2 (9.52%)	1 (4.76%)	0 (0.00%)
highlight.js	39	4 (10.26%)	4 (10.26%)	4 (10.26%)	3 (7.69%)	1 (2.56%)	1 (2.56%)
openlayers	79	76 (96.20%)	76 (96.20%)	64 (81.01%)	73 (92.41%)	75 (94.94%)	41 (51.90%)
prettier	13	2 (15.38%)	2 (15.38%)	1 (7.69%)	0 (0.00%)	1 (7.69%)	1 (7.69%)
prism	38	7 (18.42%)	4 (10.53%)	7 (18.42%)	5 (13.16%)	5 (13.16%)	0 (0.00%)
quarto-cli	24	2 (8.33%)	1 (4.17%)	2 (8.33%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
scratch-gui	11	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
Resolved		157	153	140	131	127	63
%Resolved		30.78%	30.00%	27.45%	25.69%	24.90%	12.35%

model they use, it is unfair to make a comparison directly with systems implemented with different models. Thus, if keeping the consistent base model, the GPT-4o based implementation of GUIRepair solves 30 more task instances than the best baseline Agentless Lite. Recall that GUIRepair adds design choices for multimodal scenarios to the agentless workflow. And the fact that our approach achieves better results than the general agentless system [17], [29], which potentially suggests that our novel design further improves the ability of existing agentless systems to tackle visual problems. Meanwhile, compared to agent systems [16], [67], [68], the agentless-based GUIRepair inherits the cost advantage of agentless approaches [17], [29], which requires only $0.29 on average to solve a multimodal instance. Overall, GUIRepair is a success in terms of overall effectiveness. Not only does it remain relatively inexpensive in repair cost, but it rivals and surpasses state-of-the-art closed-source and open-source systems in terms of repair capability.

Further, we will also explore the difference in repair capability between GUIRepair and existing systems in solving multimodal problems. Here, we present the unique issues resolved by GUIRepair compared to existing Top-5 SWE systems in Figure 8, and analyze the differences in the ability of these systems to resolve different repositories in Table III. As shown in Fig-

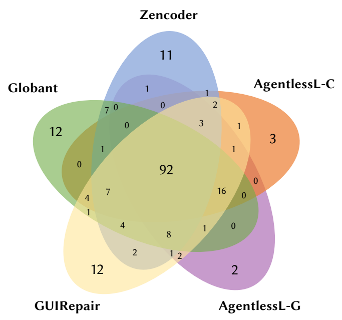


Fig. 8: Venn diagram for multimodal issue fixes. Note: AgentlessL-C/G refers Claude3.5/GPT4o-based Agentless Lite.

ure 8, different SWE systems maintain unique fixes, suggesting that the different approaches differ in the characterization of the resolved issues. In particular, although GUIRepair is very close to the state-of-the-art closed-source commercial system Globant [67] (Globant Code Fixer Agent) in terms of repair capability (157 vs. 153), they both possess unique issue resloved (12 vs. 12), which suggests that GUIRepair solves task instances that are difficult to solve by the closed-source system, while the closed-source system at the same time compensates for GUIRepair's shortcomings in terms of repair capability. Moreover, the overall distribution shows that GUIRepair complements rather than duplicates the capabilities of the other systems. In addition, as shown in Table III, different SWE systems have their strengths in solving the issues of different repositories. Specifically, we use the green background to highlight the best results from each repository. GUIRepair, Globant, and Zencoder achieved the best results in 7, 5, and 6 repositories. Overall, GUIRepair is orthogonal to existing systems in terms of repair capability, and it is competitive and complementary compared to top commercial systems.

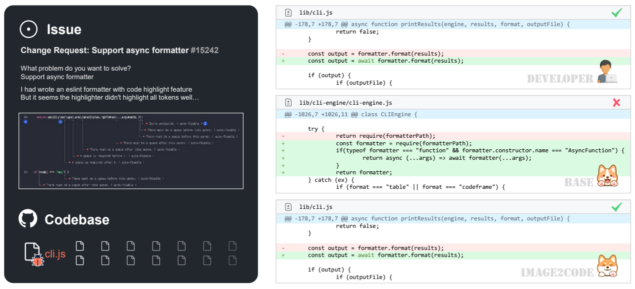
B. RQ2: Ablation Study

In the ablation experiment, we aim to evaluate the contributions of GUIRepair's key components (Image2Code and Code2Image) in enhancing the repair capability. Here, we designed four variants to reveal the effects of design choices:

- **GUIRepair$_{base}$**: To provide a basic baseline to present the repair capabilities of the basic agentless approach in multimodal scenarios, we remove two key components (Section III-A and III-D) of the GUIRepair approach and keep only the core fault localization and patch generation modules (Section III-B and III-C). This variant, called GUIRepair$_{base}$, receives multimodal issue reports and codebases directly as input and generates only one valid patch to be submitted.
- **GUIRepair$_{I2C}$**: To present the improvement of the Image2Code component for the basic agentless workflow to solve multimodal scenarios, we add the Image2Code to the basic workflow. This variant, called GUIRepair$_{I2C}$, not only receives visual issue reports and codebases, but also analyzes the code behavior behind the visual information by mining project-specific knowledge to understand visual scenarios.
- **GUIRepair$_{C2I}$**: To indicate the enhancement of the Code2Image component for the basic agentless workflow to solve multimodal scenarios, we add the Code2Image to the basic workflow. This variant, called GUIRepair$_{C2I}$, also receives issue reports and codebases as input, in particular, it helps to select potentially correct patches from the patch space by capturing the visual effects of the repair behavior.
- **GUIRepair$_{full}$**: To explore the contribution of the two key components to overall effectiveness, we refer to the complete method implementation as GUIRepair$_{full}$. In fact, the two key components of GUIRepair, along with the basic agentless approach, form the complete workflow, and the process is an interdependent one, so that only the simultaneous use of the two key components can maximize the repair benefits.

TABLE IV: Results of GUIRepair variants on SWE-bench M.

Variants	Components		Result	
	Image2Code	Code2Image	Resolved	Avg. $Cost
GUIRepair$_{base}$	-	-	136	$0.08
GUIRepair$_{I2C}$	✓	-	146 (+10)	$0.10
GUIRepair$_{C2I}$	-	✓	148 (+12)	$0.27
GUIRepair$_{full}$	✓	✓	157 (+21)	$0.29



Change Request: Support async formatter #15242

What problem do you want to solve?
Support async formatter

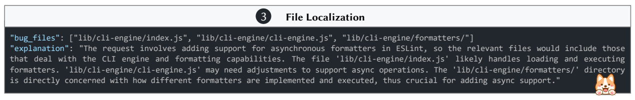
I had wrote an eslint formatter with code highlight feature
But it seems the highlighter didn't highlight all tokens well…

```
lib/cli.js
@@ -178,7 +178,7 @@ async function printResults(engine, results, format, outputFile) {
                return false;
        }

-       const output = formatter.format(results);
+       const output = await formatter.format(results);

        if (output) {
                if (outputFile) {
```

```
lib/cli-engine/cli-engine.js
@@ -1026,7 +1026,11 @@ class CLIEngine {

        try {
-               return require(formatterPath);
+               const formatter = require(formatterPath);
+               if(typeof formatter === "function" && formatter.constructor.name === "AsyncFunction") {
+                       return async (...args) => await formatter(...args);
+               }
+               return formatter;
        } catch (ex) {
                if (format === "table" || format === "codeframe") {
```

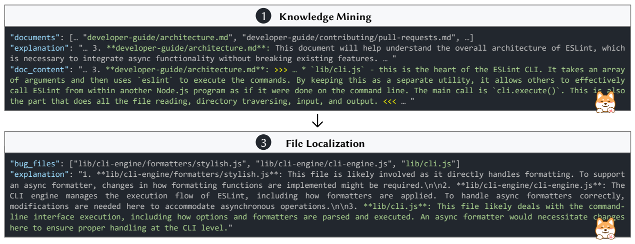
```
lib/cli.js
@@ -178,7 +178,7 @@ async function printResults(engine, results, format, outputFile) {
                return false;
        }

-       const output = formatter.format(results);
+       const output = await formatter.format(results);

        if (output) {
                if (outputFile) {
```

(a) The developer/GUIRepair$_{base}$/GUIRepair$_{I2C}$ patch of eslint-15243.



```
"bug_files": ["lib/cli-engine/index.js", "lib/cli-engine/cli-engine.js", "lib/cli-engine/formatters/"]
"explanation": "The request involves adding support for asynchronous formatters in ESLint, so the relevant files would include those that deal with the CLI engine and formatting capabilities. The file 'lib/cli-engine/index.js' likely handles loading and executing formatters. 'lib/cli-engine/cli-engine.js' may need adjustments to support async operations. The 'lib/cli-engine/formatters/' directory is directly concerned with how different formatters are implemented and executed, thus crucial for adding async support."
```

(b) The reasoning trace of GUIRepair$_{base}$ to solve eslint-15243.



```
"documents": [… "developer-guide/architecture.md", "developer-guide/contributing/pull-requests.md", …]
"explanation": "… 3. **developer-guide/architecture.md**: This document will help understand the overall architecture of ESLint, which is necessary to integrate async functionality without breaking existing features. … "
"doc_content": "… 3. **developer-guide/architecture.md**: >>> … * `lib/cli.js` - this is the heart of the ESLint CLI. It takes an array of arguments and then uses `eslint` to execute the commands. By keeping this as a separate utility, it allows others to effectively call ESLint from within another Node.js program as if it were done on the command line. The main call is `cli.execute()`. This is also the part that does all the file reading, directory traversing, input, and output. <<< … "
```

```
"bug_files": ["lib/cli-engine/formatters/stylish.js", "lib/cli-engine/cli-engine.js", "lib/cli.js"]
"explanation": "1. **lib/cli-engine/formatters/stylish.js**: This file is likely involved as it directly handles formatting. To support an async formatter, changes in how formatting functions are implemented might be required.\n\n2. **lib/cli-engine/cli-engine.js**: The CLI engine manages the execution flow of ESLint, including how formatters are applied. To handle async formatters correctly, modifications are needed here to accommodate asynchronous operations.\n\n3. **lib/cli.js**: This file likely deals with the command-line interface execution, including how options and formatters are parsed and executed. An async formatter would necessitate changes here to ensure proper handling at the CLI level."
```

(c) The reasoning trace of GUIRepair$_{I2C}$ to solve eslint-15243.

Fig. 9: The case study of eslint-15243.

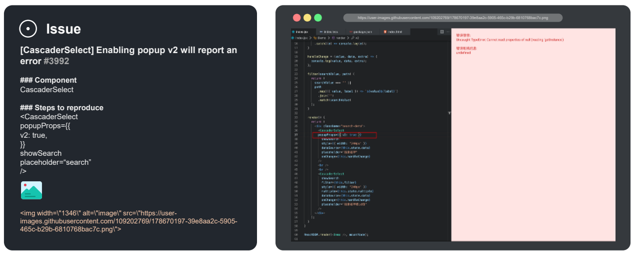
As shown in Table IV, we demonstrate the performance of different variants on SWE-bench M test. Next, we will analyze the impact of design choices on the repair capability.

1) Basic Capability of the Agentless Workflow: As shown in Table IV, GUIRepair$_{base}$ provides a reference baseline to indicate the performance achieved by the basic agentless workflow. Specifically, GUIRepair$_{base}$ solves 136 instances, which is close to the performance of Agentless Lite (Table II). This is because both GUIRepair$_{base}$ and Agentless Lite follow the basic agentless workflow, and thus they are comparable in terms of repair capability. In particular, we note that GUIRepair$_{base}$ maintains a lower cost compared to existing agentless systems, Agentless Lite [29] and Agentless [17], [24]. This is due to the fact that our implementation further simplifies the fault localization process. Compared to Agentless Lite, which locates faults by retrieving the codebase, GUIRepair$_{base}$ only retrieves the key file directories, thus reducing the file retrieval cost. Compared to Agentless, which uses three levels (file/element/edit-level) of fault location granularity to locate specific edit locations, GUIRepair$_{base}$ uses only two levels (file-level and hunk-level) of fault localization to locate hunk-level code snippets, thus further simplifying the complexity of the localization process. Benefiting from our simplified design, our implementation of the basic agentless approach has a lower cost advantage while maintaining good repair effectiveness.

2) *Improved Capability of the Image2Code Component:* As shown in Table IV, GUIRepair$_{I2C}$ presents improvements of the Image2Code component to the basic repair workflow. Specifically, GUIRepair$_{I2C}$ solves 10 more instances than GUIRepair$_{base}$, achieving a 7.35% improvement. Let's recall that one of the purposes of the Image2Code component is to enhance fault localization by mining the project-specific knowledge to understand the visual information. To further clarify the contribution of the Image2Code, here we provide a case study to analyze how the Image2Code component works. As shown in Figure 9, we demonstrate a instance *eslint-15243*, which was successfully resolved by GUIRepair$_{I2C}$, but the GUIRepair$_{base}$ fix failed. In Figure 9a, GUIRepair$_{I2C}$ accurately located the bug file *cli.js* and generated a patch consistent with the developer patch. Yet GUIRepair$_{base}$ did not locate the correct bug file, which caused the fix to fail. Further, we try to analyze the reasoning trace behind these two variants to reveal why GUIRepair$_{I2C}$ was able to locate the bug file. As shown in Figure 9b, GUIRepair$_{base}$ reads the issue report directly to locate suspicious files, however, since the model does not have project-specific knowledge, it struggles to accurately locate the relevant code files behind the visual issue. In contrast, in Figure 9c, GUIRepair$_{I2C}$ first learns the project details in the knowledge mining phase, where it reads the key documents and learns the role of the *cli.js*. Then, benefiting from the fact that the model has learned that project knowledge, when implementing fault localization, the model considers *cli.js* as a suspicious file. In this way, GUIRepair$_{base}$ misses the opportunity to localize to the bug file due to its lack of project knowledge, while GUIRepair$_{I2C}$ compensates for this by using the Image2Code, which improves the overall effectiveness.

3) *Enhanced Capability of the Code2Image Component:* As shown in Table IV, GUIRepair$_{C2I}$ presents enhancements of the Code2Image component to the basic agentless approach. Specifically, GUIRepair$_{C2I}$ solves 12 more multimodal task instances than GUIRepair$_{base}$, achieving a 8.82% improvement. Such results indicate that the Code2Image component is critical for improving repair capabilities. It should be noted that although GUIRepair$_{C2I}$ achieves better results, it is almost three times more expensive to repair than GUIRepair$_{base}$. The reason for the increase in repair cost is mainly due to the multiple sampling performed by the patch generation in order to fully explore the patch space and the large amount of image information read when receiving visual feedback in patch selection phase. However, such a repair cost is still acceptable and it is still cheaper than many agent systems. Overall, capturing visual effects of repair behaviors to provide fix feedback is an effective validation scheme for visual issues.

4) *Full Capability of the Complete Pipeline:* We have presented the unique contributions of two key components. However, using any single component alone cannot fully unlock



Issue

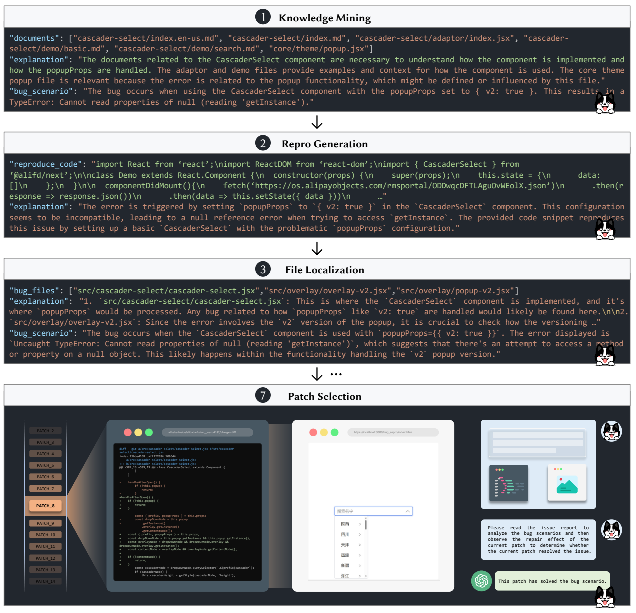
[CascaderSelect] Enabling popup v2 will report an error #3992

Component
CascaderSelect

Steps to reproduce
<CascaderSelect
popupProps={{
v2: true,
}}
showSearch
placeholder="search"
/>

<img width=\"1346\" alt=\"image\" src=\"https://user-images.githubusercontent.com/109202769/178670197-39e8aa2c-5905-465c-b29b-6810768bac7c.png\">

(a) The issue report of next-4182. Note: left is the textual description, right is the issue image.



```
"documents": ["cascader-select/index.en-us.md", "cascader-select/index.md", "cascader-select/adaptor/index.jsx", "cascader-select/demo/basic.md", "cascader-select/demo/search.md", "core/theme/popup.jsx"]
"explanation": "The documents related to the CascaderSelect component are necessary to understand how the component is implemented and how the popupProps are handled. The adaptor and demo files provide examples and context for how the component is used. The core theme popup file is relevant because the error is related to the popup functionality, which might be defined or influenced by this file."
"bug_scenario": "The bug occurs when using the CascaderSelect component with the popupProps set to { v2: true }. This results in a TypeError: Cannot read properties of null (reading 'getInstance')."
```

```
"reproduce_code": "import React from ‘react’;\nimport ReactDOM from ‘react-dom’;\nimport { CascaderSelect } from ‘@alifd/next’;\n\nclass Demo extends React.Component {\n  constructor(props) {\n    super(props);\n    this.state = {\n      data: []\n    };\n  }\n\n  componentDidMount(){\n    fetch(‘https://os.alipayobjects.com/rmsportal/ODDwqcDFTLAguOvWEolX.json’)\n      .then(response => response.json())\n      .then(data => this.setState({ data }))\n      …"
"explanation": "The error is triggered by setting `popupProps` to `{ v2: true }` in the `CascaderSelect` component. This configuration seems to be incompatible, leading to a null reference error when trying to access `getInstance`. The provided code snippet reproduces this issue by setting up a basic `CascaderSelect` with the problematic `popupProps` configuration."
```

```
"bug_files": ["src/cascader-select/cascader-select.jsx","src/overlay/overlay-v2.jsx","src/overlay/popup-v2.jsx"]
"explanation": "1. `src/cascader-select/cascader-select.jsx`: This is where the `CascaderSelect` component is implemented, and it's where `popupProps` would be processed. Any bug related to how `popupProps` like `v2: true` are handled would likely be found here.\n\n2. `src/overlay/overlay-v2.jsx`: Since the error involves the `v2` version of the popup, it is crucial to check how the versioning …"
"bug_scenario": "The bug occurs when the `CascaderSelect` component is used with `popupProps={{ v2: true }}`. The error displayed is `Uncaught TypeError: Cannot read properties of null (reading 'getInstance')`, which suggests that there's an attempt to access a method or property on a null object. This likely happens within the functionality handling the `v2` popup version."
```

Please read the issue report to analyze the bug scenarios and then observe the repair effect of the current patch to determine whether the current patch resolved the issue.

This patch has solved the bug scenario.

(b) The reasoning trace of GUIRepair$_{full}$ to solve next-4182.

Fig. 10: The case study of next-4182.

the complete performance of GUIRepair. On the one hand, Image2Code-only GUIRepair$_{I2C}$ only helps to understand visual scenarios by mining project-specific knowledge for improving fault localization, ignoring a crucial role (i.e., guided patch validation) of reasoning the code behavior behind the image information. On the other hand, Code2Image-only GUIRepair$_{C2I}$ only works on instances with reproduced code, which means it cannot solve most of the visual problems (only around 17% of instances contain reproduced link). As shown in Table IV, the full implementation, GUIRepair$_{full}$, solves 157 instances, which is better than any other variant. Such results suggest that implementing the full workflow simultaneously is the only way to maximize the fixing benefits. To clarify more clearly how the two components work together, we show an instance *next-4182* that can only be resolved by GUIRepair$_{full}$. As shown in Figure 10a, the issue report does not provide the complete repro code, which results in GUIRepair$_{C2I}$ unable to implement patch validation. Furthermore, even though GUIRepair$_{I2C}$ is able to generate the repro code, it is not equipped with the Code2Image module, resulting in the inability to fully utilize the repro generation. Therefore, these variants cannot solve this case. Here, we show the reasoning trace of

TABLE V: Results of GUIRepair on additional repositories.

Note: all systems listed here use GPT-4o as the base model.

Repo	Num	GUIRepair$_{full}$	GUIRepair$_{base}$	SWE-agentM	AgentlessJS	RAG
wp-calypso	37	0	0	0	0	1
Chart.js	24	6	5	5	0	3
react-pdf	11	0	0	0	0	0
marked	14	1	1	1	0	3
p5.js	16	7	4	4	1	4
Total Resolved		**14**	10	10	1	11

TABLE VI: Results of GUIRepair with different base models.

Repo	Num	GPT-4o		GPT-4.1		o4-mini	
		🐱$_{base}$	🐱$_{full}$	🐱$_{base}$	🐱$_{full}$	🐱$_{base}$	🐱$_{full}$
next	39	6	9	5	8	6	10
bpmn-js	54	35	35	32	33	37	37
carbon	134	7	12	15	16	14	19
eslint	11	2	6	1	5	5	6
lighthouse	54	3	3	3	4	6	6
grommet	21	1	1	2	2	2	4
highlight.js	39	4	4	3	3	3	3
openlayers	79	76	76	76	76	77	77
prettier	13	0	2	1	2	4	4
prism	38	1	7	7	9	4	6
quarto-cli	24	1	2	3	3	2	3
scratch-gui	11	0	0	0	0	0	0
Total Resolved		136	**157**	148	**161**	160	**175**
Avg. $Cost		\$0.08	\$0.29	\$0.07	\$0.25	\$0.08	\$0.36

GUIRepair$_{full}$ in Figure 10b. Using Image2Code, the model successfully generates repro code (❷ Repro Generation) for the bug scenario after learning project-specific knowledge (❶ Knowledge Mining) and helps the model locate the correct bug file *cascader-select.jsx* (❸ File Localization). Then, the Code2Image component uses the previously generated repro code to capture the visual effects presented by the fixing behavior. When the iteration validates to Patch_8 (❼ Patch Selection), the model observes that the bug component renders properly on the web page and considers the patch resolved this issue. In this case, the reproduced code is a key factor in helping to solve the problem by tightly linking the Image2Code and Code2Image components to better understand the bug and validate the patch. Since most visual instances lack reproduction links, the full repair potential can only be unlocked by using both key components together. Overall, GUIRepair requires not only the Image2Code to understand the visual problem and generate the reproduced code, but also the Code2Image to use the reproduced code to capture the visual effect of the fixing behavior, which is an inextricably intertwined process.

C. RQ3: Generalizability Study

In the generalizability experiment, we focus on the performance of GUIRepair on additional projects and base models. We follow the recent study [24] testing the performance of GUIRepair on SWE-bench M dev to investigate whether our approach can be generalized to additional projects. We choose GPT-4.1 [69] (flagship GPT model for complex tasks) and o4-mini [70] (faster, more affordable reasoning model) to analyze whether our method can work effectively on other base models. As shown in Table V, GUIRepair achieves the best results on SWE-bench M dev. In particular, GUIRepair$_{full}$ solves 4 more instances than GUIRepair$_{base}$, showing that our approach is equally effective in other projects. In addition, we show the performance of GUIRepair on different models in Table VI. Specifically, GUIRepair$_{full}$ resolves 21, 13, and 15 more instances than GUIRepair$_{base}$ when using GPT-4o, GPT-4.1, and o4-mini, respectively. Notably, with the reasoning model o4-mini, GUIRepair outperforms the best commercial system, Globant, by 22 instances (175 vs. 153). Such results show that our approach is not only effective across different models, but also significantly benefits from enhanced reasoning capabilities [71], surpassing SOTA commercial solutions. Overall, these findings highlight the generalizability of GUIRepair across diverse projects and model backbones.

TABLE VII: Median values for metrics about each task instance.

Note: GUIRepair listed here use GPT-4o as the base model.

Repo	Num	Codebase Lines	Codebase Files	Gold Patch Lines	Gold Patch Files	Gold Patch Funcs	GUIRepair Resolved
next	39	200K	1731	57	5	5	9 (23.08%)
bpmn-js	54	99K	593	15	1	2	35 (64.81%)
carbon	134	**822K**	4250	45	3	4	**12 (8.96%)**
eslint	11	398K	1567	23	1	2	6 (54.55%)
lighthouse	54	**1076K**	834	29	2	3	**3 (5.56%)**
grommet	21	**612K**	1186	8	1	1	**1 (4.76%)**
highlight.js	39	105K	1476	16	2	2	4 (10.26%)
openlayers	79	544K	1761	32	2	3	76 (96.20%)
prettier	13	348K	6855	48	2	5	2 (15.38%)
prism	38	184K	2721	12	2	2	7 (18.42%)
quarto-cli	24	**1399K**	3639	12	2	2	**2 (8.33%)**
scratch-gui	11	100K	514	**562**	**12**	**9**	**0 (0.00%)**

VI. Discussion

A. Failure Cases and Current Limitations

1) Large-scale codebases. GUIRepair struggles with repositories that contain highly complex and large-scale codebases. As shown in Table VII, in *carbon*, *lighthouse*, *grommet*, *quarto-cli*, the number of codebase lines exceeds 600K. In these cases, the model has difficulty navigating the search space and identifying the correct fix locations. GUIRepair achieves a resolution rate below 10% on these projects, suggesting that the current pipeline does not scale well to such complex scenarios.
2) Structurally complex patches. Even in smaller codebases, GUIRepair may fail when the required repair behavior is structurally complex. As shown in Table VII, *scratch-gui* has a relatively small codebase (514 Codebase Files), but its gold patches span multiple files, functions, and over 500 lines of edits. None of the current tools, including GUIRepair, successfully resolve any instances in this project. This suggests that generating multi-location or cross-file edits remains a significant challenge for current LLM-based repair systems.
3) Issues involving dynamic interactions. Some visual issues (e.g., *bpmn-js-1083/1092/1143/1192/1236/1256*, *wp-calypso-21409/21492/23991*, *scratch-gui-8492/5039/3342*, etc.) require users to perform a series of dynamic interactions to trigger the underlying bug. While GUIRepair can leverage automated tools such as Playwright for simple actions, reproducing these deeper workflows remains beyond its capability. This level of interaction, which involves multi-step reasoning, conditional branching, and UI state tracking. Designing agents that can infer and execute such interaction flows represents an open challenge for visual issue repair.

B. Lessons Learned and Future Directions

1) Multimodal bugs require multimodal reasoning. Visual artifacts such as rendered UIs often provide decisive contextual signals that complement textual descriptions. Integrating such cues opens up fundamentally new opportunities in APR.
2) LLMs are more effective when scaffolded as agents. GUIRepair's Image2Code and Code2Image workflows highlight that structured reasoning stages, rather than end-to-end prompting, better exploit LLMs' multimodal reasoning abilities.
3) Reproduction itself is diagnostic. Beyond validation, the act of reproducing a bug helps localize faults and infer causality, providing implicit grounding for repair.
4) Visual validation is a powerful complement to testing. Traditional unit tests often overlook layout or interaction issues, incorporating screenshot-based validation offers a perceptual check that aligns more closely with human developer judgment.
5) Benchmarks must evolve. While SWE-bench M is a valuable first step, it is limited to visual issues in front-end projects. Broader multimodal debugging benchmarks, including user interactions, logs, and platform-specific artifacts, will be crucial for advancing this field.

VII. Threats to Validity

Data Leakage. To mitigate the threat of data leakage, we keep the same model as baselines. In the main experiment (Table II), GUIRepair based on GPT-4o still performs better than baselines using the same model. This suggests that even under the same threat of data leakage, our approach still outperforms other schemes. Moreover, our ablation study (Table IV) and generalizability study (Table VI) show that the improvement does not only stem from the powerful base model, but is further enhanced by the design choices of GUIRepair.
Repo Configuration. We manually install dependencies and deploy runtime environments for each project, which is an inefficient process. Fortunately, SWE-bench M just released docker images. Besides, we see the potential for autonomous systems to replace humans in deploying and running code repository [72], and we plan to use these automated techniques in the future to replace manual/hard-encoding practices.

VIII. Related Work

LLM for APR. Researchers have proposed many LLM-based APR tools to solve software engineering problems. Early researchers fixed real-world software defects by fine-tuning [23], [53], [56], [60]–[62], [73]–[75] or prompting [9]–[12], [21], [55], [57]–[59], [63], [76]–[79] LLMs and showed promising results. Recently, researchers [15]–[20], [80]–[82] have begun to focus on github issue resolution [14], [24], and proposed many agent/agentless-based autonomous systems to solve complex repo-level issue scenarios. For example, SWE-agent [16] solves SWE problems by designing Agent Computer Interface to interact with the environment, SpecRover [18] guides the LLM fixing process by specifications inference, and Agentless [20] employs a simplistic pipeline of localization, repair, and validation to solve software development problems, etc. However, most existing SWE systems are not designed for multimodal scenarios, which makes them difficult to solve visual problems [24]. GUIRepair is designed specifically for understanding, reproducing, and validating visual issues. The modules are orthogonal to the foundational repair steps used in Agentless. In addition, although some studies also focus on visual-related issues, they are often designed only for specific bug types and are difficult to generalize to generic visual problem domains. For example, DesignRepair [83] only focuses on UI design-related quality issues, Iris [84] is designed for color-related accessibility issues. As shown in Figure 1, multimodal task instances contain various issue scenarios, leading to the need for more generalized solutions to be applied in the vision software domain. GUIRepair enhances the ability to solve multimodal issues by leveraging cross-modal reasoning to understand and capture visual information.
LLM for GUI. GUI-related software issues directly affect the user experience. A key area of research is GUI testing, especially researchers have used LLM-driven approaches to implement GUI testing [85]. For example, researchers explored applications by enabling LLMs to understand multimodal scenarios and simulate user actions to test and reproduce GUI-related issues [27], [86], [87]. Especially, recent work has investigated reproducing bugs from bug reports [88] or videos [89]. And this key reproduction information has successfully helped to implement soap opera tests generation [90]. Inspired by GUI testing work, GUIRepair implements fault comprehension and patch validation by generating reproduced code and replaying visual scenarios to migrate the experience of GUI testing to APR.

IX. Conclusion

In this work, we propose GUIRepair, a cross-modal reasoning agentless approach that designs Image2Code and Code2Image components for understanding and capturing visual information to enhance the capabilities of autonomous systems for multimodal SWE problems. Our evaluation on the SWE-bench M demonstrates that GUIRepair can achieve the highest performance compared to other open/closed-source systems.

Acknowledgment

This work was partially supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (CRPO-GC1-NUS-001), the CyberSG R&D Cyber Research Programme Office, the Federal Ministry of Education and Research (BMBF), and the fellowship within the IFI programme of the German Academic Exchange Service (DAAD). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, Cyber Security Agency of Singapore and CyberSG R&D Programme Office. We also gratefully acknowledge Singapore Management University, and OpenAI for providing API credits through the OpenAI Researcher Access Program. In addition, Kai Huang would like to thank Ludwig Felder for generously providing access to his workstation for running the experiments.

References

[1] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM (CACM)*, vol. 62, no. 12, pp. 56–65, 2019.

[2] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang, "Evolving paradigms in automated program repair: Taxonomy, challenges, and opportunities," *ACM Computing Surveys (CSUR)*, vol. 57, no. 2, pp. 1–43, 2024.

[3] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 33, no. 2, pp. 1–69, 2023.

[4] B. Yang, Z. Cai, F. Liu, B. Le, L. Zhang, T. F. Bissyandé, Y. Liu, and H. Tian, "A survey of llm-based automated program repair: Taxonomies, design paradigms, and applications," *arXiv preprint arXiv:2506.23749*, 2025.

[5] J. Renzullo, P. Reiter, W. Weimer, and S. Forrest, "Automated program repair: Emerging trends pose and expose problems for benchmarks," *ACM Computing Surveys (CSUR)*, vol. 57, no. 8, pp. 1–18, 2025.

[6] H. Eladawy, C. Le Goues, and Y. Brun, "Automated program repair, what is it good for? not absolutely nothing!" in *46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.

[7] Q. Zhang, C. Fang, Y. Xie, Y. Zhang, Y. Yang, W. Sun, S. Yu, and Z. Chen, "A survey on large language models for software engineering," *arXiv preprint arXiv:2312.15223*, 2023.

[8] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou, "Large language model-based agents for software engineering: A survey," *arXiv preprint arXiv:2409.02977*, 2024.

[9] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," in *47th International Conference on Software Engineering (ICSE)*, 2025, pp. 694–694.

[10] C. Lee, C. S. Xia, L. Yang, J.-t. Huang, Z. Zhu, L. Zhang, and M. R. Lyu, "A unified debugging approach via llm-based multi-agent synergy," *arXiv preprint arXiv:2404.17153*, 2024.

[11] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *33rd International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 819–831.

[12] X. Yin, C. Ni, S. Wang, Z. Li, L. Zeng, and X. Yang, "Thinkrepair: Self-directed automated program repair," in *33rd International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 1274–1286.

[13] J. Kong, M. Cheng, X. Xie, S. Liu, X. Du, and Q. Guo, "Contrastrepair: Enhancing conversation-based automated program repair via contrastive test case pairs," *arXiv preprint arXiv:2403.01971*, 2024.

[14] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can language models resolve real-world github issues?" in *12th International Conference on Learning Representations (ICLR)*, 2024.

[15] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *33rd International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 1592–1604.

[16] J. Yang, C. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," *Advances in Neural Information Processing Systems (NIPS)*, vol. 37, pp. 50 528–50 652, 2024.

[17] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Demystifying llm-based software engineering agents," *Proc. ACM Softw. Eng. (FSE)*, 2025.

[18] H. Ruan, Y. Zhang, and A. Roychoudhury, "Specrover: Code intent extraction via llms," in *47th International Conference on Software Engineering (ICSE)*, 2025, pp. 617–617.

[19] Y. Ma, R. Cao, Y. Cao, Y. Zhang, J. Chen, Y. Liu, Y. Liu, B. Li, F. Huang, and Y. Li, "Lingma swe-gpt: An open development-process-centric language model for automated software improvement," in *34th International Symposium on Software Testing and Analysis (ISSTA)*, 2025.

[20] Y. Ma, Q. Yang, R. Cao, B. Li, F. Huang, and Y. Li, "Alibaba lingmaagent: Improving automated issue resolution via comprehensive repository exploration," *arXiv preprint arXiv:2406.01422*, 2024.

[21] H. Ye, A. Z. Yang, C. Hu, Y. Wang, T. Zhang, and C. Le Goues, "Adverintent-agent: Adversarial reasoning for repair based on inferred program intent," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 1398–1420, 2025.

[22] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *2014 International Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 437–440.

[23] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1430–1442.

[24] J. Yang, C. E. Jimenez, A. L. Zhang, K. Lieret, J. Yang, X. Wu, O. Press, N. Muennighoff, G. Synnaeve, K. R. Narasimhan *et al.*, "Swe-bench multimodal: Do ai systems generalize to visual software domains?" in *13th International Conference on Learning Representations (ICLR)*, 2025.

[25] R. Cheng, M. Tufano, J. Cito, J. Cambronero, P. Rondon, R. Wei, A. Sun, and S. Chandra, "Agentic bug reproduction for effective automated program repair at google," *arXiv preprint arXiv:2502.01821*, 2025.

[26] GUIRepair, "Guirepair," 2025. [Online]. Available: https://sites.google.com/view/guirepair

[27] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.

[28] AlibabaFusion/next-895, "label should show in icononly mode," 2025. [Online]. Available: https://github.com/alibaba-fusion/next/issues/894

[29] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless lite: Rag-based swe-bench software engineering scaffold," 2025. [Online]. Available: https://github.com/sorendunn/Agentless-Lite

[30] pixelmatch, "The smallest, simplest and fastest javascript pixel-level image comparison library," 2025. [Online]. Available: https://github.com/mapbox/pixelmatch

[31] Puppeteer, "Javascript api for chrome and firefox," 2025. [Online]. Available: https://github.com/puppeteer/puppeteer

[32] Chart.js, "Simple yet flexible javascript charting for designers and developers," 2025. [Online]. Available: https://github.com/chartjs/Chart.js

[33] OpenLayers, "A high-performance, feature-packed library for all your mapping needs," 2025. [Online]. Available: https://github.com/openlayers/openlayers

[34] chartjs/Chart.js 1509, "borderradius gets ignored for the bottom corners of 0-value bars in bar chart when borderskipped and minbarlength are set," 2025. [Online]. Available: https://github.com/chartjs/Chart.js/issues/10005

[35] Z. Song, B. Yan, Y. Liu, M. Fang, M. Li, R. Yan, and X. Chen, "Injecting domain-specific knowledge into large language models: a comprehensive survey," *arXiv preprint arXiv:2502.10708*, 2025.

[36] S. B. Hossain, R. Taylor, and M. Dwyer, "Doc2oracll: Investigating the impact of documentation on llm-based test oracle generation," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 1870–1891, 2025.

[37] OpenAI, "Gpt-4o," 2025. [Online]. Available: https://platform.openai.com/docs/models/gpt-4o

[38] ——, "text-embedding-3-small," 2025. [Online]. Available: https://platform.openai.com/docs/models/text-embedding-3-small

[39] Z. Lu, Y. Yang, H. Ren, H. Hou, H. Xiao, K. Wang, W. Shi, A. Zhou, M. Zhan, and H. Li, "Webgen-bench: Evaluating llms on generating interactive and functional websites from scratch," *arXiv preprint arXiv:2505.03733*, 2025.

[40] Y. Wan, C. Wang, Y. Dong, W. Wang, S. Li, Y. Huo, and M. Lyu, "Divide-and-conquer: Generating ui code from screenshots," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 2099–2122, 2025.

[41] AlibabaFusion/next-1509, "The popupcontainer configuration on configprovider does not take effect on dialog," 2025. [Online]. Available: https://github.com/alibaba-fusion/next/issues/1508

[42] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 9, pp. 1943–1959, 2019.

[43] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *44th International Conference on Software Engineering (ICSE)*, 2022, pp. 1506–1518.

[44] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," in *37th International Conference on Automated Software Engineering (ASE)*, 2022, pp. 1–13.

[45] H. Ye and M. Monperrus, "Iter: Iterative neural repair for multi-location patches," in *46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.

[46] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 602–614.

[47] ——, "Dear: A novel deep learning-based approach for automated program repair," in *44th International Conference on Software Engineering (ICSE)*, 2022, pp. 511–523.

[48] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *29th International Symposium on Software Testing and Analysis (ISSTA)*, 2020, pp. 101–114.

[49] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1161–1173.

[50] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, and X. Zhang, "Knod: Domain knowledge distilled tree decoder for automated program repair," in *45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1251–1263.

[51] Q. Zhu, Z. Sun, Y. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2021, pp. 341–353.

[52] Q. Zhu, Z. Sun, W. Zhang, Y. Xiong, and L. Zhang, "Tare: Type-aware neural program repair," in *45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1443–1455.

[53] K. Huang, J. Zhang, X. Meng, and Y. Liu, "Template-guided program repair in the era of large language models," in *47th International Conference on Software Engineering (ICSE)*, 2025, pp. 367–379.

[54] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2023, pp. 172–184.

[55] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2022, pp. 959–971.

[56] C. S. Xia, Y. Ding, and L. Zhang, "The plastic surgery hypothesis in the era of large language models," in *38th International Conference on Automated Software Engineering (ASE)*, 2023, pp. 522–534.

[57] J. Xiang, X. Xu, F. Kong, M. Wu, Z. Zhang, H. Zhang, and Y. Zhang, "How far can we go with practical function-level program repair?" *arXiv preprint arXiv:2404.12833*, 2024.

[58] J. Xu, Y. Fu, S. H. Tan, and P. He, "Aligning the objective of llm-based program repair," in *47th International Conference on Software Engineering (ICSE)*, 2025, pp. 2548–2560.

[59] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1469–1481.

[60] K. Huang, X. Meng, J. Zhang, Y. Liu, W. Wang, S. Li, and Y. Zhang, "An empirical study on fine-tuning large language models of code for automated program repair," in *38th International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1162–1174.

[61] K. Huang, J. Zhang, X. Bao, X. Wang, and Y. Liu, "Comprehensive fine-tuning large language models of code for automated program repair," *IEEE Transactions on Software Engineering (TSE)*, pp. 1–25, 2025.

[62] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah, "How effective are neural networks for fixing security vulnerabilities," in *32nd International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 1282–1294.

[63] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *45th International Conference on Software Engineering (ICSE)*, 2023, pp. 1482–1494.

[64] Prism/prism-1602, "Yaml strings fail with trailing comments," 2025. [Online]. Available: https://github.com/PrismJS/prism/issues/1601

[65] SWE-bench, "Swe-bench multimodal leaderboard," 2025. [Online]. Available: https://www.swebench.com/index.html#multimodal

[66] P. Aggarwal and S. Welleck, "Programming with pixels: Computer-use meets software engineering," *arXiv preprint arXiv:2502.18525*, 2025.

[67] Globant, "Globant code fixer agent," 2025. [Online]. Available: https://ai.globant.com/

[68] Zencoder, "The ai coding agent," 2025. [Online]. Available: https://zencoder.ai/

[69] OpenAI, "Gpt-4.1," 2025. [Online]. Available: https://platform.openai.com/docs/models/gpt-4.1

[70] ——, "o4-mini," 2025. [Online]. Available: https://platform.openai.com/docs/models/o4-mini

[71] Y. Wei, O. Duchenne, J. Copet, Q. Carbonneaux, L. Zhang, D. Fried, G. Synnaeve, R. Singh, and S. I. Wang, "Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution," *arXiv preprint arXiv:2502.18449*, 2025.

[72] I. Bouzenia and M. Pradel, "You name it, i run it: An llm agent to execute tests of arbitrary projects," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 1054–1076, 2025.

[73] W. Wang, Y. Wang, S. Joty, and S. C. Hoi, "Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair," in *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2023, pp. 146–158.

[74] B. Yang, H. Tian, J. Ren, H. Zhang, J. Klein, T. Bissyande, C. Le Goues, and S. Jin, "Morepair: Teaching llms to repair code via multi-objective fine-tuning," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2025.

[75] A. Silva, S. Fang, and M. Monperrus, "Repairllama: Efficient representations and fine-tuned adapters for program repair," *arXiv preprint arXiv:2312.15698*, 2023.

[76] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "Gamma: Revisiting template-based automated program repair via mask prediction," in *38th International Conference on Automated Software Engineering (ASE)*, 2023, pp. 535–547.

[77] H. Zheng, I. Shumailov, T. Fan, A. Hall, and M. Payer, "Fixing 7,400 bugs for 1$: Cheap crash-site program repair," *arXiv preprint arXiv:2505.13103*, 2025.

[78] J. Zhao, D. Yang, L. Zhang, X. Lian, Z. Yang, and F. Liu, "Enhancing automated program repair with solution design," in *39th International Conference on Automated Software Engineering (ASE)*, 2024, pp. 1706–1718.

[79] Y. Xiao, W. Wang, D. Liu, J. Zhou, S. Cheng, and Y. Xiong, "Predicatefix: Repairing static analysis alerts with bridging predicates," *arXiv preprint arXiv:2503.12205*, 2025.

[80] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh *et al.*, "Openhands: An open platform for ai software developers as generalist agents," in *13th International Conference on Learning Representations*, 2024.

[81] A. Antoniades, A. Örwall, K. Zhang, Y. Xie, A. Goyal, and W. Wang, "Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement," *arXiv preprint arXiv:2410.20285*, 2024.

[82] Y. Liu, P. Gao, X. Wang, J. Liu, Y. Shi, Z. Zhang, and C. Peng, "Marscode agent: Ai-native automated bug fixing," *arXiv preprint arXiv:2409.00899*, 2024.

[83] M. Yuan, J. Chen, Z. Xing, A. Quigley, Y. Luo, T. Luo, G. Mohammadi, Q. Lu, and L. Zhu, "Designrepair: Dual-stream design guideline-aware frontend repair with large language models," in *47th International Conference on Software Engineering (ICSE)*, 2025, pp. 2483–2494.

[84] Y. Zhang, S. Chen, L. Fan, C. Chen, and X. Li, "Automated and context-aware repair of color-related accessibility issues for android apps," in *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2023, pp. 1255–1267.

[85] S. Yu, C. Fang, Z. Tuo, Q. Zhang, C. Chen, Z. Chen, and Z. Su, "Vision-based mobile app gui testing: A survey," *arXiv preprint arXiv:2310.13518*, 2023.

[86] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions," in *46th International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.

[87] Y. Su, Z. Xing, C. Wang, C. Chen, S. Xu, Q. Lu, and L. Zhu, "Automated soap opera testing directed by llms and scenario knowledge: Feasibility, challenges, and road ahead," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 757–778, 2025.

[88] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *27th International Symposium on Software Testing and Analysis (ISSTA)*, 2018, pp. 141–152.

[89] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk, "Translating video recordings of mobile app usages into replayable scenarios," in *42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 309–321.

[90] Y. Su, Z. Han, Z. Xing, X. Xia, X. Xu, L. Zhu, and Q. Lu, "Constructing a system knowledge graph of user tasks and failures from bug reports to support soap opera testing," in *37th International Conference on Automated Software Engineering (ASE)*, 2022, pp. 1–13.