DesignRepair: Dual-Stream Design Guideline-Aware Frontend Repair with Large Language Models

Mingyue Yuan[†‡], Jieshan Chen[‡¶*], Zhenchang Xing[‡‖], Aaron Quigley[†‡], Yuyu Luo[§], Tianqi Luo[§]
Gelareh Mohammadi[†], Qinghua Lu[†‡], Liming Zhu[†‡]
[†]University of New South Wales, Australia
[‡]CSIRO's Data61, Australia
[§]The Hong Kong University of Science and Technology (Guangzhou), China
[¶] Institute for Advanced Study, Technical University of Munich, Germany
[‖]Australian National University, Australia
Email: [†]{mingyue.yuan, g.mohammadi}@unsw.edu.au, [§]{yuyuluo, tluo553}@hkust-gz.edu.cn
[‡]{jieshan.chen, zhenchang.xing, a.quigley, qinghua.lu, liming.zhu}@data61.csiro.au

***Abstract*—The rise of Large Language Models (LLMs) has streamlined frontend interface creation through tools like Vercel's V0, yet surfaced challenges in design quality (e.g., accessibility, and usability). Current solutions, often limited by their focus, generalisability, or data dependency, fall short in addressing these complexities. Moreover, none of them examine the quality of LLM-generated UI design. In this work, we introduce DesignRepair, a novel dual-stream design guideline-aware system to examine and repair the UI design quality issues from both code aspect and rendered page aspect. We utilised the mature and popular Material Design as our knowledge base to guide this process. Specifically, we first constructed a comprehensive knowledge base encoding Google's Material Design principles into low-level component knowledge base and high-level system design knowledge base. After that, DesignRepair employs a LLM for the extraction of key components and utilizes the Playwright tool for precise page analysis, aligning these with the established knowledge bases. Finally, we integrate Retrieval-Augmented Generation with state-of-the-art LLMs like GPT-4 to holistically refine and repair frontend code through a strategic divide and conquer approach. Our extensive evaluations validated the efficacy and utility of our approach, demonstrating significant enhancements in adherence to design guidelines, accessibility, and user experience metrics.**

***Index Terms*—Frontend Code Repair, Design Guideline, UI Design, Large Language Models**

I. Introduction

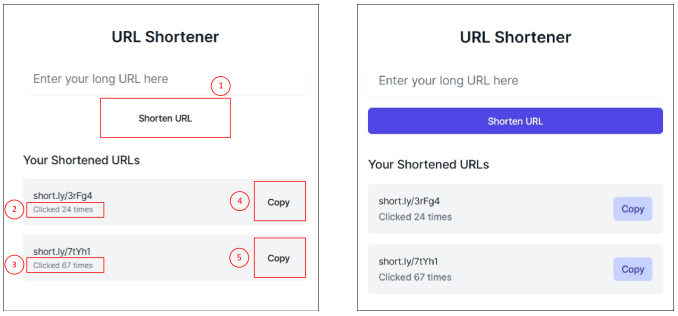
The emergence of advanced large language models (LLMs), such as GPT-4 [1] and Meta's Llama [2], has catalyzed transformative changes across various fields, including legal [3]–[6], finance [7], and software development [8], unlocking tremendous new possibilities. For frontend software development, commercial and open-source tools, such as Vercel's V0 [9], Imagica [10] and OpenV0 [11], have attracted considerable interest using the capabilities of LLMs. These applications adeptly transform user-provided textual or visual prompts into concrete, well-structured high-fidelity user interfaces, along with their associated front-end code, providing substantial support to designers and developers. To augment the quality of the generated UI and its accompanying code, these tools commonly integrate with established UI design libraries (e.g., HTML, CSS, JS frameworks, and libraries), including popular ones like React [12], Shadcn [13] and Tailwind CSS [14]. These libraries serve as foundational knowledge bases, empowering these tools to generate aesthetically pleasing and technically sound designs.

Despite their advanced capabilities, these tools continue to face two significant challenges. Firstly, while they are adept at converting prompts into well-structured UI elements, their compliance with established design principles, such as material design [15], often falls short (see Section IV). This inadequacy is critical, as strict adherence to these principles is fundamental to crafting effective, accessible, and user-friendly interfaces. Such high-quality interfaces play a vital role in setting an application apart from its competitors, boosting user downloads, reducing user complaints, and enhancing user retention [16]–[21]. Secondly, these tools exhibit limitations in the repair and refinement of generated designs. While they allow for additional prompts for UI adjustments, they often rely on users to manually identify and describe needed changes. This process can be cumbersome, requiring extensive labeling and in-depth knowledge [18], [19], [22]. Fig. 1a shows an example. When prompted with "URL shorten Chrome extension," Vercel's V0 successfully generated an appealing user interface. Nonetheless, upon closer examination, it exhibited five distinct design issues related to contrast and color in text and button elements, marked in red.

Many studies have been proposed to assess different facets of UI design development to ensure its quality, encompassing accessibility [23]–[28], tapability [29], animation effects [18], and broader design principles and concerns [16], [19], [30]. Industrial tools like Playwright [23] and Google Lighthouse [24] primarily address accessibility concerns, such as text contrast for readability. Swearing et al. [29] utilized a classifier-based technique to identify discrepancies between the perceived and intended tappability of UI elements. Yang et al. [19] introduced

* Jieshan Chen is the corresponding author.
This work is partially funded by the Dieter Schwarz Foundation and the Technical University of Munich – Institute for Advanced Study, Germany.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00109



(a) Initial UI Displaying Text and Button Color Contrast Issues, Highlighted in Red

(b) Repaired UI using our method, showcasing resolved issues

Fig. 1: "URL Shortener" by Vercel's V0: Initial Design vs. Improved Design.

UIs-Hunter, a rule-based method for detecting deviations from material design principles in UIs. Despite these advancements, they either tend to focus on specific, isolated issues within UI design, or they encounter challenges in generalizing to a broader spectrum of design aspects. Moreover, rule-based methods require extensive manual efforts which limits their usability and generalisability.

To tackle the aforementioned challenges and fill the existing research gaps, we introduce DesignRepair, a novel knowledge-driven LLM-based approach specifically designed for enhancing the quality assurance and remediation of frontend UI and code. DesignRepair adopts Material Design 3 [15] as the standard for evaluating quality and as the foundation of our knowledge base. To guarantee efficient and effective knowledge retrieval, catering to varying levels of repair granularity and diverse design requirements, we meticulously analyzed and restructured Material Design 3 into two organized knowledge bases: the component knowledge base and the system design knowledge base. The component knowledge base focuses on the detailed, fine-grained design principles of individual components, whereas the system design knowledge base provides broader guidance on the overall design perspective, ensuring coherence in elements such as layout and color harmony.

Given the frontend code and its corresponding rendered page, DesignRepair initially extracts essential information, such as components and property groups, followed by retrieving relevant knowledge from our knowledge bases. This approach, considering both the source code and the user-perceived experience, provides a holistic and detailed examination. Finally, DesignRepair adopts a divide and conquer approach to iteratively repair the code. The process progresses from focusing on individual elements to considering the overall design, ensuring a comprehensive and systematic approach to repair frontend code.

We conducted an evaluation to measure the effectiveness and usefulness of DesignRepair. We collected 196 design issues from AI-generated frontend code and 115 from GitHub projects consisting of UIs and their frontend code. Our evaluation shows that DesignRepair effectively identifies and repairs design issues, achieving a recall of 89.3% and precision of 86.6% for AI-generated issues, and a recall of 85.2% with precision of 90.7% for GitHub issues. Additionally, our experiments on each step highlights the effectiveness of the strategies we proposed. Finally, a user study with 26 participants further confirms the high quality of our repairs. All our code and prompts are released at our GitHub repository [1].

Our contributions are as follows:

- We systematically analyzed Material Design guidelines and converted them into structured knowledge bases, i.e., low-level component knowledge base and high-level system design knowledge base, enabling efficient and effective utilization of knowledge in design quality assurance processes.
- We introduced DesignRepair, a novel dual-stream knowledge-driven, LLM-based method, uniquely designed to detect and repair design quality issues in frontend code. This approach simultaneously considers both the source code and the user-perceived rendered page, ensuring outstanding performance in design quality enhancement.
- We conducted extensive experiments and a user study with 26 participants, which demonstrate the effectiveness and usefulness of our approach.

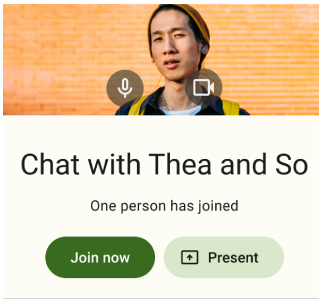
II. Background

A. Material Design Guidelines

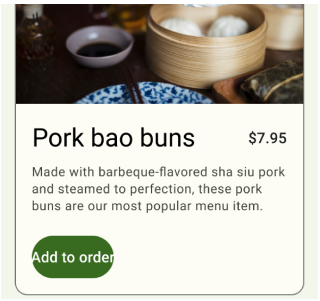
Material Design 3 [15] is a broad design system built and supported by Google designers and developers, and can be implemented across Android, iOS, and web platforms. Its latest version, Material Design 3, provides guidelines and examples to support important practices of user interface, ensuring personal, adaptive, and expressive experiences. It covers topics ranging from fine-grained design guidelines such as the application of individual components and color usage for specific meanings, to the high-level composition of components and common design patterns.

The guidelines consist of **three main sections**: foundations, styles, and components. `Foundations` inform "the basis of any great user interface, from accessibility standards to essential patterns for layout and interaction." For example, it offers advice on simplifying the hierarchy of UI elements, making them more straightforward to perceive and understand. `Styles` are "the visual aspects of a UI that give it a distinct look and feel." For instance, they may include recommendations on color palettes to maintain a consistent appearance across the app's user interfaces, and guidelines for choosing colors for primary, secondary, and tertiary accent groups to enhance the user experience. `Components` are "interactive building blocks for creating a user interface. They can be organized into categories based on their purpose: Action, containment, communication, navigation, selection, and text input." These guidelines present best practices for selecting and utilizing components to effectively communicate the intended design purpose. These three sections are interconnected, collectively providing a comprehensive framework for designing intuitive and visually cohesive user interfaces.

[1] https://github.com/UGAIForge/DesignRepair



(a) ***Recommended***: Use visually prominent filled buttons for the most important actions



(b) ***Do/Don't***: A button container's width shouldn't be narrower than its label text

Fig. 2: Examples of *recommended* and *don't* design guidelines in Material Design 3. Both guidelines are related to Button component in the `Components` section, and regard to the *Usage* aspect. We categorised the *recommended* guideline as a soft constraint, and *do/don't* guidelines as hard constraints.

Within these three sections, they further classify the guidelines according to **various aspects of UI/UX design principles**, such as anatomy, behavior, responsive layout, usage, placement, and other categories. For instance, the *Usage* aspect focuses on how and when to utilize a specific component, color, or layout. Moreover, they utilize clear text and illustrative examples to explain **what to do (*do*), what to avoid (*don't*), and what is *recommended***. Adhering to the *do* and *don't* guidelines is crucial for delivering an optimal user experience; violating them can lead to user confusion or dissatisfaction. On the other hand, following the *recommended* guidelines, while not mandatory, can enhance user engagement and satisfaction by refining the overall usability and aesthetic appeal of the interface. For instance, Fig. 2a in the `Components` section includes a *recommended* guideline for the Button component, stating "use visually prominent filled buttons for the most important action," illustrated with a "Join now" button example. On the other hand, Fig. 2b offers a *don't* guideline, "Don't: A button container's width shouldn't be narrower than its label text. " shown with an example where a button's label exceeds its container width, leading to a poor user experience. These two examples both focus on the *Usage* aspect of the Button component.

III. Approach

Fig. 3 illustrates the overview of our approach, DesignRepair, which consists of three phases, namely, an offline knowledge base construction (Section III-A), online page extraction ((Section III-B) and knowledge-driven repair phases (Section III-C).

For the offline knowledge base construction phase (Fig. 3-A), we built a knowledge base (KB) consisting of two parts, i.e., a low-level Component Knowledge Base (KB-Comp) and a high-level System Design Knowledge Base (KB-System). This knowledge base functions as a domain expert, offering guidance for addressing potential UI issues. Given the frontend code and rendered page, we employ a parallel dual-pipe method to extract the used components and

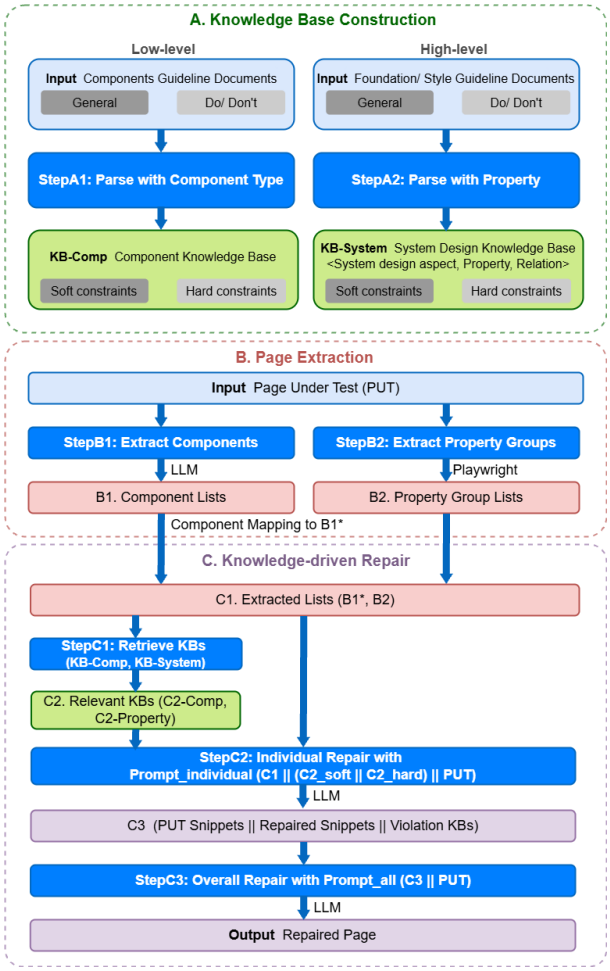
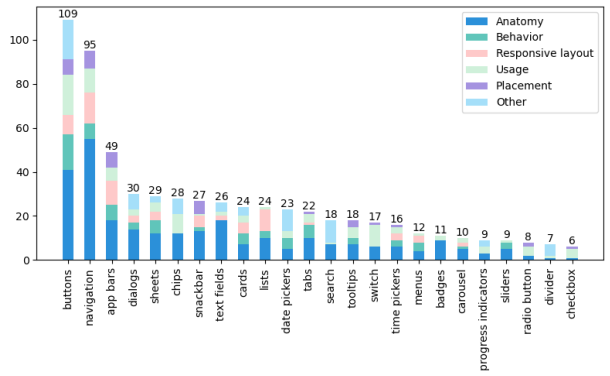


Fig. 3: Framework Overview of Our Approach

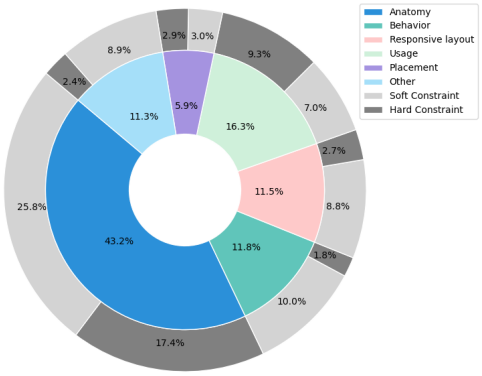
their corresponding property groups (as shown in Fig. 3-B). Finally, we implement a knowledge-driven, LLM-based repair method (Fig. 3-C). This approach allows us to carefully analyze and repair issues concurrently. By employing a divide and conquer strategy, we tackle each component/property group separately before integrating the repairs. This ensures a cohesive, optimized final output, achieved through a thorough and scrutinized repair process.

A. Knowledge Base Construction

As introduced in Section II-A, we utilize Material Design 3 as our foundational knowledge base. By systematically analyzing the guidelines for efficient knowledge retrieval, we restructured them into two parts: a low-level Component Knowledge Base (KB-Comp) and a high-level System Design Knowledge Base (KB-System). KB-Comp provides detailed guidance for atomic-level elements, while KB-System addresses the broader aspects of aesthetic and functional harmony in interface design. For example, a guideline from KB-Comp might ensure a button is sized appropriately for accessibility purpose, while a guideline from KB-System would focus on how this button fits into the overall page layout and flow. Within these knowledge bases, we further classified the information into soft constraints (i.e., *recommended* guidelines) and hard constraints



(a) Distribution by Component Type



(b) Distribution by Component Design Aspects with Soft/Hard Constraint.

Fig. 4: Distribution of Component Guidelines

(i.e., *do/don't* guidelines), catering to different levels of design concerns. For instance, a soft constraint would suggest font styles that align with the brand's aesthetic, whereas a hard constraint may specify the minimum text size for readability.

1) ***Component knowledge base****:* We created the component knowledge base (KB-Comp) at the component level, drawing from `Components` section in the Material Design (see Section II-A). Each component is linked with a range of guidelines. In total, we obtained 24 component types associating with Material Design guidelines. Fig. 4a illustrates the number of guidelines for each component type and the distribution of various UI/UX design aspects (like usage, responsive layout) that these guidelines address. The distribution of guidelines across these component types demonstrates a notable imbalance, showing a long-tail distribution pattern. For instance, buttons and navigation components are the most abundant, with 109 and 95 entities respectively, while checkboxes are the least represented, with only 6 entities. To facilitate understanding, Fig. 5 (a) shows two hard constraints and two soft constraints regarding the button component. In this example, the first hard constraint aligns with the guideline shown in Fig. 2b, and the first soft constraint correlates with the guideline in Fig. 2a.

We further conducted an in-depth analysis of our component knowledge base in terms of design aspects, identifying the distribution of soft and hard constraints within each design aspect. In total, our KB-Comp includes 228 hard constraints (113 *do*s, 115 *don'ts*), and 399 soft constraints (*recommended* guidelines). Fig. 4b shows the distribution of different design aspects and the percentages of hard and soft constraints. The anatomy aspect comprises 43.2% of the guidelines, whereas placement represents 5.9%. Notably, the number of soft constraints frequently matches or surpasses the number of hard constraints. This suggests that while soft constraints might be less mandatory, they offer a wealth of valuable information to convey better user experience.

In conclusion, based on our detailed knowledge base, we propose knowledge-driven LLM-based method, which can effectively manage the long-tail distribution of design guidelines, ensuring an equitable application of these principles across all component types.

2) ***System Design knowledge base****:* To mitigate the challenge of hallucinations in LLMs and enhance their analysis capabilities, our approach involves a precise association of high-level design guidelines with actionable properties. This is based on the understanding that properties are the direct targets for addressing high-level design issues.

To this end, we reformulated high-level design guidelines, i.e., `Foundations` and `Styles` sections (see Section II-A), into a structured system design knowledge base (KB-System). In total, We identified 12 system design aspects (e.g., typography), summarized 7 properties (e.g., clickable and spacing) and 15 mapping relations between them. For example, design aspects '*role*' and '*utilities*' in `Styles` - Color section, as well as the design aspect accessibility '*contrast*' in `Foundations` - Color section are both connected to the 'color' property. These relations are crucial for marking guidelines, aiding LLMs in understanding the intricate relationships between design elements and their practical applications in a web development context. Based on this, we reconstructed the design guidelines into triples of <*system design aspect, property, relation*>. Fig. 5 (b) shows two examples of the system design knowledge base.

Foundations have 7 key aspects including Flow, Layout, Implementation, Structure, Label, Text, and Color. For instance, the 'Flow' aspect relates to how users navigate through an interface, while 'Color' pertains to both aesthetic appeal and accessibility. In total, we identified 118 knowledge entities for Foundations.

Styles consist of 5 key aspects: Typography, Elevation, Icons, Color, and Shape. These elements craft the UI's visual identity, adaptable through Material themes. In our work, we structured 80 knowledge entities related to Styles, with Icons specifically linked to icon buttons and Shapes applicable across all component types.

Property Groups include 7 categories: Group, Clickable, Spacing, Platform, Label, Text, and Color. These are actionable property levels that we map to high-level guidelines for effective detection and repair.

(1) Group: This property is associated with the layout and accessibility elements within the UI, focusing on aspects such

(a) Component Knowledge Base Example

```
"component_type": "buttons",
"description": "A component helping people initiate actions...",
"constraint": {
    "hard": {
              "Usage#None#A button container's width shouldn't be narrower than
               its label text.",
             "Usage#None#Don't clutter your UI with too many buttons. Consider
              presenting lower-priority actions in overflow menus..."
            ...},
    "soft": {
             "Usage#None#Use visually prominent filled buttons for the most
             important actions...",
              "Anatomy#Container#"Button containers hold the label text..."
            ...},
```

(b) System Design Knowledge Base Example

```
{
    "material_design_section": "styles",
    "system_design_aspect": "typography",
    "property": "text",
    "constraint": "hard",
    "relation": "accessibility size",
    "content": "Don't mix the sizes of your symbol and text"
},{
    "material_design_section": "foundations",
    "system_design_aspect": "layout",
    "property": "spacing",
    "constraint": "soft",
    "relation": "layout",
    "content": "Padding refers to the space between UI elements.
                Padding is measured in increments of 4dp..."
}
```

Fig. 5: Examples of Component and System Design Knowledge entities

as landmark recognition and prioritization in navigation. It is linked to the 'Structure', 'Flow', and 'Layout' of Foundations.

(2) Clickable: This property predominantly relates to interactive elements like buttons, focusing on accessibility aspects such as focus order and target size. It is also connected to the 'Elevation' style, emphasizing floating elements.

(3) Spacing: Integral to layout design, this property focuses on the spatial arrangement and coherence of UI elements. It is linked to the 'Layout' aspect in Foundations.

(4) Platform: This property addresses the layout implementation across different platforms, ensuring accessibility and consistency in various environments. It is associated with 'Implement' and 'Layout' in Foundations.

(5) Label: Expanded to encompass alternative text and captions for images, videos, icons, and buttons, this property plays a pivotal role in enhancing accessibility. It is linked to 'Label' for content accessibility in Foundations, as well as 'Structure' for accessibility labeling.

(6) Text: This property group covers aspects of typography and content accessibility size. It is connected to the 'Typography' style and the 'Text' and 'Structure' aspects of Foundations, focusing on size and accessibility in headings.

(7) Color: Color is intricately linked to both the aesthetic and functional roles in UI design, including accessibility considerations like contrast. This property group is connected to the 'Color' aspect in both Styles and Foundations.

B. Page Extraction

In this phase, given the code and the rendered page as inputs, DesignRepair examine the usage of component in the code, and system design aspects in the rendered output of the page.

This dual analysis is important as it bridges the gap between technical code and the user perceived experience. It allows for detailed exploration of component usage within the code, alongside addressing broader design concerns evident in the rendered output at the property group level.

*1) **Components Extraction**:* The first aspect of our strategy is to identify a list of component types within the Page Under Test (PUT) using GPT-4. This initial step is important for the efficient retrieval of relevant component knowledge from KB-Comp in subsequent repair stages.

Given a PUT code In_{PUT}, we propose a systematic component extraction prompt $P_{\text{comp\_extra}}$ to analyse a PUT, represented by the equation:

$$B1 = P_{\text{comp\_extra}}\left(In_{\text{PUT}}\right)$$

This process results in a list of component type names from the PUT, defined by: $B1 = \{\text{Component}_i \mid i = 0, 1, \ldots, k\}$

*2) **Property Extraction**:* The second aspect of our extraction strategy leverages the Playwright tool [23], a robust testing tool developed by Microsoft, to dissect the rendered frontend page into various properties. This approach is critical for understanding the complex relationships between different page elements and their various properties, which are integral to addressing broader design issues.

Properties are analyzed to ensure consistency in design elements such as spacing and color schemes, as well as overall layout coherence. This stage focuses on elements that are more readily apparent in the rendered output rather than just within the code itself. The use of the Playwright tool enables us to pre-render the page and extract key DOM information such as XPath, color, framed within seven specified categories: Group, Clickable, Spacing, Platform, Label, Text, and Color.

We process these properties to produce *<outerHTML, property>* pairs, yielding code snippets and property values. This meticulous organization of data is crucial in pinpointing problem areas within the code, thereby facilitating an in-depth and thorough analysis by the LLM. It enhances the efficacy of the subsequent repair phase. The output from this stage is represented as $B2$, a list of 7 extracted property groups.

C. Knowledge-Driven Repair

In the knowledge-driven repair phase, we leverage the reasoning capabilities of LLMs coupled with a broader spectrum of knowledge bases from Section III-A to perform detailed repairs at both the component and property levels of frontend code. This phase marks a critical transition from the initial analysis to the actual refinement and enhancement of the code based on the insights gained.

The process is divided into three key steps. For each component or property group, we first retrieve the related guidelines from our knowledge bases (Fig. 3 Step C1). With related knowledge/guidelines and associated component/property group, we adopt LLMs to conduct targeted repair (Fig. 3 Step C2). This method ensures that the specific needs of various components and properties are addressed with focused attention. Finally, we assemble, reassess and synthesize the individual repairs into a cohesive, optimized, and final repaired output (Fig. 3 Step C3). By adopting the divide-and-conquer strategy, we ensure detailed, thorough, and holistic repair.

*1) **Relevant Knowledge Retrieval**:* In this step, we first preprocess the retrieved components to map them with our component knowledge base, and then retrieve corresponding knowledge from KB-Comp and KB-System.

(a) Component Mapping and Knowledge Retrieval: To map extracted web page components to the component names in our KB-Comp for alignment, we use LLMs for reasoning with the prompt $P_{\text{map\_kb}}$. This approach allows us to handle the diversity in terminology used across different front-end libraries such as Shadcn [13], Tailwind CSS [14] without the need of defining any exhaustive rule-based mapping function. For instance, a 'navigation bar' component might be referred to as 'Navbars' in Tailwind CSS, while in Shadcn UI it could be labeled as 'Navigation Menu'. Such variability in naming conventions can be effectively addressed by LLMs without the need to define an exhaustive mapping function. The mapping process is represented by:

$$B1^* = P_{\text{map\_kb}}\left(B1_i \,|\, i = 0, 1, \ldots, k\right)$$

Where $B1^*$ is the resultant list of components mapped to our KB-Comp equivalents. The corresponding knowledge items for $B1^*$ are then retrieved from KB-Comp, denoted as $C2_{\text{comp}}$.

(b) Property Knowledge Retrieval: For each property group in $B2$, we directly retrieve related knowledge items from our KB-System, denoted as $C2_{\text{property}}$.

(c) Aggregation: Finally, we aggregate the extracted information and the retrieved knowledge as the following:

$$C1 = \{B1^* \cup B2_i \,|\, i = 0, 1, \ldots, j\}$$

$$C2 = \{C2_{\text{comp}} \cup C2_{\text{property}_i} \,|\, i = 0, 1, \ldots, h\}$$

In $C1$, we have a list of mapped components and property types extracted from the PUT. Furthermore, $C2$ contains the corresponding knowledge entities from our knowledge bases. Examples of these entities are illustrated in Fig 5, serving as preparation for the repair stages.

*2) **Individual Repair**:* In the individual repair phase, we implement an LLM-driven, detail-oriented process to inspect and repair frontend code in relation to specific components and property groups. This phase also employs a distinct approach for handling soft and hard constraints to guide the analysis and repair process (denoted as $C3$). The output of this phase is formatted as a JSON object, comprising sections for poor design code snippets, associated guideline references, and repair suggestions.

Formally, given a PUT code In_{PUT} alongside the extracted lists $C1$ and their corresponding knowledge lists $C2$, we apply an iterative repair prompt $P_{\text{individual}}$:

$$C3 = P_{\text{individual}}\left(In_{\text{PUT}} \,||\, C1 \,||\, (C2_{\text{soft}} \,||\, C2_{\text{hard}})\right)$$

In the LLM prompting stage, we differentiate the guiding narrative for soft and hard constraints. For hard constraints (denoted as $C2_{\text{hard}}$), we instruct the LLM as follows: *"Here are the guidelines you must follow, we name it 'hard constraints'. Remember this is mandatory. Once you find a bad design not following the guideline, you must fix it."* This directive emphasizes the obligatory nature of these constraints in the repair process.

Conversely, for soft constraints (denoted as $C2_{\text{soft}}$), we prompt the LLM with: *"Here are the general guidelines you can use, we name it 'soft constraints'. Remember this is not mandatory, regarded as optional."* This approach suggests a more flexible consideration of these guidelines, allowing for discretionary application in the repair process.

The resulting $C3$ list comprises the code in question, the associated knowledge, and repaired code snippets for each component and property:

$$C3 = \left\{ \begin{array}{l} \text{"bad\_design\_code"}_i, \\ \text{"detailed\_reference\_guidelines"}_i, \;\; | \, i = 0, 1, \ldots, m \\ \text{"suggestion\_fix\_code"}_i \end{array} \right\}$$

This structured approach enables the LLM to deliver nuanced repair suggestions, tailored to the specific context and requirements of each detected issue, aligning closely with the established design principles and usability guidelines.

*3) **Overall Repair**:* In this final step, we integrate the repair suggestions obtained from both the component-level and property-group analyses. This integration ensures a holistic repair of the frontend code.

Formally, given the original PUT code In_{PUT} and the set of partially repaired code snippets $C3$, our objective is to use these inputs to produce the final, fully repaired page. To achieve this, we employ prompt with P_{all}, which is designed to direct the overall repair process:

$$\text{Repaired\_page} = P_{\text{all}}\left(In_{\text{PUT}} \,||\, C3\right)$$

The P_{all} prompt effectively merges the various repair suggestions, taking into account any potential overlaps or conflicts between the proposed fixes. The goal is to ensure that the final output, represented as Repaired_page, is not only free from identified design flaws but also aligns with the established design guidelines and usability standards. Due to the page limit, all prompts can be obtained at our GitHub repository.

IV. Experiments

In this section, we evaluate the effectiveness and usefulness of our proposed DesignRepair by investing three research questions (RQs):

- RQ1: How effective is our approach in detecting violations according to Design Guidelines compared to the baseline?
- RQ2: How effective are individual strategies and the overall repair approach?
- RQ3: What is the perceived quality of the fixes generated by our approach from user perspectives?

A. Subjects

We conducted two experiments to evaluate the performance of our frontend code repair approach on both generated code from Vercel's V0 and real projects from GitHub.

1) Vercel's V0 Projects: Our research focuses on examining code and the rendered pages generated by LLM tools, specifically targeting adherence to design guidelines and enabling code-level adjustments. Our work aims to assist developers with limited knowledge of UI design guidelines who utilize LLM-generated UI mock-ups as starting points. Vercel's V0 [9] generated projects were chosen for this purpose, as it is a leading LLM tool that attracted 100K users within three weeks of its release [31]. Additionally, Vercel's V0 is widely used to assist in frontend development, with 259K results on GitHub as of December 2024.

Vercel's V0 platform enables users to experiment with and generate UI designs based on their provided prompts. By default, users' attempts are shared publicly, allowing for community visibility. Additionally, Vercel's V0 manually reviews prototypes created with their tool, showcasing those that are notably good and representative.

We randomly chose 64 projects from Vercel's V0 featured projects and user submissions to ensure a diverse selection. Each project was fully examined based on the following rules:

(1) Projects with specific and clear page definitions were included, while those with vague prompts like "give me an icon" without additional explanation were excluded. (2) Projects included different use cases such as landing pages, product pages, and login pages. (3) Projects that were well-designed and had undergone at least three iterations of human modification and prompting were included.

Consequently, we selected 20 high-quality human-AI collaborative front-end code projects generated by Vercel's V0.

To conduct recognition of guideline violations, two of the authors first discussed and manually reviewed the guidelines and examples on the Material Design guidelines to reach a common understanding. They then independently reviewed the selected projects, manually identifying components, property groups, associated design guidelines, and design violations. In cases of disagreement, a third author was consulted to review the relevant guidelines and provide their judgment, which was then referred to for resolution.

Through careful discussion and consensus on the annotations, we pinpointed 669 components/property groups across these projects, corresponding to 3,765 design guidelines, of which 196 were violated. Note that on average, each project is associated with 188.25 design guidelines. This statistic underscores the complexity and the extensive range of design principles that need to be considered in modern UI development.

2) Github Projects: To further evaluate the effectiveness of our method in real-world projects, we searched for "top frontend projects" and "frontend example projects" on GitHub and Google, identifying the top 6 popular projects. These projects represent various real-world website uses, ranging from commercial product frontend websites to blogs, with star counts ranging from 1.1k to 5.6k. Their popularity and representativeness have proven their usefulness and quality.

We randomly selected 66 files out of 577 files from these projects, ensuring diverse functionalities. These projects utilize a variety of representative packages, including React [12], Next.js [32], Material-UI [33], and TypeScript [34], which can examine the capabilities of our DesignRepair in detecting and repairing design issues across these different technologies. Following the same protocol in Section IV-A1, we identified 1,793 components/property groups across these projects, of which 115 guidelines were violated.

3) Subject Analysis: For Vercel's V0 projects, there were 33.45 components/property group per file and 9.8 guideline violations per project. Of these, 149 (76%) were component-related and 47 (24%) were property-related. Additionally, 27 (14%) were soft constraint violations, while 169 (86%) were hard constraint violations. For GitHub projects, there were 27 components/property group per file and 1.74 guideline violations per project. Among these, 60 (52%) were component-related and 55 (48%) were property-related. Soft constraint violations accounted for 15 (13%), and hard constraint violations accounted for 100 (87%).

Based on our analysis, the generated frontend code from Vercel's V0 was found to have complexity comparable to real GitHub projects (33.45 vs. 27 components/property group per file). This suggests the potential of LLM-generated code to match the structural richness required for real frontend development. However, GitHub projects have significantly fewer violations per project compared to Vercel's V0 projects (1.74 vs. 9.8), highlighting a substantial gap in quality. This indicates that while LLMs can generate complex code, their outputs are prone to a higher rate of errors and require enhanced automated quality assurance mechanisms to improve code quality. In addition, Vercel's V0 projects have a higher percentage of component-related issues (over 52%) compared to property-level issues. Real-world projects, although generally better designed, still have minor issues such as inconsistent padding or missing alt-text.

In conclusion, we found that human-generated issues differ significantly from AI-generated ones, making it valuable to analyze them separately for better insights.

B. RQ1: Effectiveness in detecting violations

*1) **Experimental Setup**:* We used UIS-Hunter [19] as our baseline, as it also focuses on detecting design violations against the material design. It primarily focuses on do/don't guidelines (i.e., hard constraints), and only supports 23 types of UI components and examines 126 visual design guidelines. In comparison, our tool collects 399 guidelines for 24 UI component types, including both do/don't and general guidelines.

To make a fair comparison, we selected corresponding instances involving hard constraint violations: 127 issues from Vercel's V0 and 48 issues from GitHub, to perform the experiment. We adopted the precision and recall metrics. A detection result is considered as a true positive if the elements are actually faulty.

During the experiment, we identified limitations in UIS-Hunter due to its rule-based approach, which struggles with component mapping. For instance, elements like buttons

within a navigation bar should be classified under multiple types, but UIS-Hunter restricts them to a single type. Moreover, UIS-Hunter was specifically designed for Android apps and lacks the capability to handle the complexity of web components commonly used in frameworks such as React and Tailwind CSS. To address these issues and adapt UIS-Hunter for the experimental data (as some component names in the experimental data differ from those supported in UIS-Hunter), we focused on evaluating its performance in detecting guideline violations. Two authors manually mapped component names to their corresponding types, which took about one hour per project. In contrast, our method enables automated component analysis and mapping, demonstrating significantly greater flexibility than the rule-based approach.

*2) **Results**:* As shown in Table I, for recall, UIS-Hunter achieves 32.3% for Vercel's V0 projects and 22.9% for GitHub projects, while our method achieves 92.1% and 83.3%, respectively.

UIS-Hunter's lower true positive rates stem from its reliance on fixed rules and thresholds, which result in many issues being overlooked. We identified three primary failure reasons for UIS-Hunter. First, it is unable to evaluate active state components. It cannot assess the correctness of components in active states and can only identify issues in static screenshots through image processing. Second, UIS-Hunter fails to detect responsive design issues such as line breaks not displaying correctly in larger windows. Addressing these issues would require multiple detections, a capability that this method lacks. Moreover, it cannot capture the semantics of components, resulting in an inability to detect subtle component issues. For instance, UIS-Hunter cannot distinguish and evaluate the semantic differences between buttons for "follow" and "message", which require different active colors based on their semantics.

In contrast, our algorithm overcomes these limitations, ensuring high recalls and consistency detection among components, such as ensuring uniform font colors across multiple buttons. It also accurately identifies minor visual violations, such as padding differences between the left and right, which purely visual-based methods may miss.

For precision, as shown in Table I, UIS-Hunter achieves 55.4% for Vercel's V0 projects and 47.8% for GitHub projects, while our method achieves 87.3% and 90.9%, respectively. UIS-Hunter's lower precision is largely due to false positives caused by OCR errors, such as spaces in long text strings. Additionally, its rigid rule-based approach-such as requiring colored dividers in lists or limiting the number of icons-further contributes to the higher false positive rate.

Overall, our comparative analysis with the baseline demonstrates that our method effectively addresses component guideline mapping issues caused by different language packages, accurately detects guideline violations in LLM-generated code, and is extendable to real-world projects.

C. RQ2: Effectiveness in repairing violations

*1) **Experimental Setup**:* To evaluate the effectiveness of DesignRepair in correcting guideline violations in frontend code, we conducted an assessment of each step module. This evaluation includes analyzing Component Extraction, Component Mapping, Violation Detection, various aspects of Violation Repair, and the overall performance of the tool.

In the evaluation, we compared outputs with manually annotated ground truth for each step. For "Component Extraction", an extraction is deemed correct if the component name can be found in the file. For "Component Mapping", a mapping is considered as correct if the mapped type matches an entry in the Material Design component list. Successful detection and repair are defined as correctly identifying and fixing faulty elements based on the guidelines.

To evaluate the effectiveness of component/property and soft/hard constraint strategies separately, we tested each strategy in isolation and evaluate them on their respective guideline subsets. For example, for component-only violation repair strategy, we disabled the property repair module and calculated results only for component-specific guidelines.

TABLE I: Comparison of Violation Detection Performance

	Vercel's V0 Projects		**GitHub Projects**	
	Rec.	**Prec.**	**Rec.**	**Prec.**
UIS-Hunter	32.3	55.4	22.9	47.8
DesignRepair	92.1	87.3	83.3	90.9

TABLE II: Performance Metrics for Each Key Strategy in Vercel's V0 and GitHub Projects

	Vercel's V0 Projects		**GitHub Projects**	
	Rec.	**Prec.**	**Rec.**	**Prec.**
Component Extraction	96.7	93.6	96.5	91.8
Component Mapping	94.1	90.5	94.9	90.2
Violation Detection	92.9	90.1	87.8	93.5
Violation Repair (Comp.)	89.9	87.0	81.7	87.5
Violation Repair (Property)	87.2	85.4	89.0	94.2
Violation Repair (Soft)	70.4	82.6	67.7	77.0
Violation Repair (Hard)	92.3	87.2	88.0	92.6
Violation Repair (All)	**89.3**	**86.6**	**85.2**	**90.7**

*2) **Results**:* **Component Extraction Results:** Upon examining DesignRepair's performance in Component Extraction, we identified 88 component types across 20 Vercel's V0 projects and 64 component types for GitHub projects. As shown in II, DesignRepair achieved 96.7% recall and 93.6% precision on Vercel's V0 projects , and 96.5% recall and 91.8% precision on GitHub projects. Precisions are slightly lower than recalls because the LLM used an aggressive strategy to extract more components when uncertain, minimizing omissions. We found that the missed components in our detection had little impact on mapping performance because they were usually the same type as the correctly identified components, such as "buttons", differing only in their specific names, like "likebutton" or "sharebutton".

Component Mapping Results: We identified 88 component types in 51 Vercel's V0 projects and 39 component types in GitHub projects. As shown in II, our model achieved 94.1% recall and 90.5% precision on Vercel's V0 projects, and 94.9% recall and 90.2% precision on GitHub projects.

Complex components formed from simpler ones, such as navigation bars, cards and lists, achieved nearly 100% precision and recall. The LLM effectively analyzed and identified these groups, which is challenging for rule-based methods. However, simple components defined primarily by CSS, like badges and dividers, were less effectively extracted by our method.

In general, the results demonstrate that the LLM-based analysis method can handle complex component extraction tasks and can deal with various, sometimes anecdotal implementation behaviors, such as using hyperlinks instead of proper button tags to represent a button. Rule-based approaches struggle to correctly identify such variations.

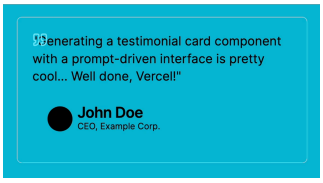
Violation Detection Results: We evaluated guideline violations on 196 Vercel's V0 and 115 GitHub's. As shown in Table II, DesignRepair achieved 92.9% recall and 90.1% precision on Vercel's V0 projects, and 87.8% recall and 93.5% precision on GitHub projects. Our method effectively detects both component and property issues. For components, it identifies violations in anatomy, behavior, layout, usage, and placement. For properties, it detects issues related to labels, text, color, and spacing, focusing on accessibility problems like alt-text and labeling.

In our analysis, our method demonstrates higher recall of candidate fixes on Vercel's V0 compared to GitHub projects. However, recall decreases on complex GitHub pages as DesignRepair's LLM avoids significant adjustments to prevent introducing new issues.

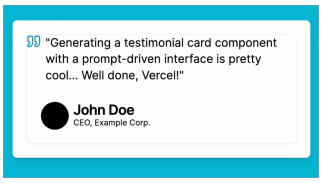
Component/Property Only Repair Results: For component-only repair, we evaluated 149 Vercel's V0 violations instances and 60 GitHub's. As shown in Table II, our method achieved 89.9% recall and 87% precision on Vercel's V0, and 81.7% recall and 87.5% precision on GitHub projects. For property-only repair, we evaluated 47 Vercel's V0 violations instances and 55 GitHub's. Our model achieved 87.2% recall and 85.4% precision on Vercel's V0, and 89% recall and 94.2% precision on GitHub projects.

Our DesignRepair demonstrated better performance in component repairs for Vercel's V0 projects, while property repairs proved more effective for GitHub projects. Upon analysis, we discovered that GitHub projects often contained minor accessibility issues, such as missing alt-text, improper labeling, or errors caused by handwriting, likely due to less rigorous developer writing and review processes. These types of issues are generally easier for LLMs to predict and repair. In contrast, LLM-generated frontend projects present a challenge for DesignRepair to predict values accurately, even with reference sizes from rendered pages. LLMs tend to predict larger values to ensure accessibility, which can compromise page consistency.

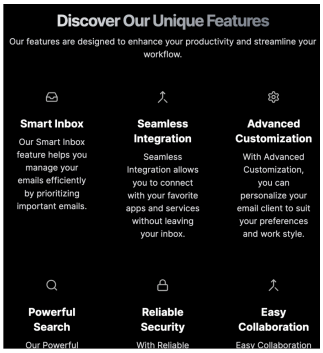
Soft/Hard Constraint Only Repair Results: For soft constraint-only repair, we evaluated 27 Vercel's V0 violations instances and 15 GitHub's. Our model achieved 70.4% recall and 82.6% precision on Vercel's V0 projects, and 67.7% recall and 77% precision on GitHub projects. For hard constraint-only repair, we evaluated 169 Vercel's V0 projects and 100 GitHub projects. The results show 92.3% recall and 87.2% precision for Vercel's V0, and 88% recall and 90.7% precision for GitHub projects.



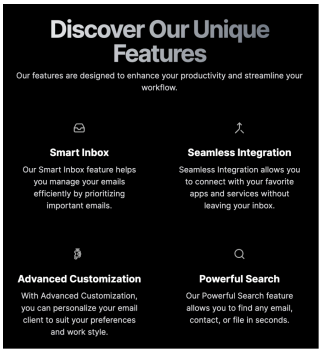
(a) Example 1 before-repair: demonstrates issues with element overlapping and color contrast.



(b) Example 1 after-repair: issues resolved, for enhanced accessibility and aesthetics.



(c) Example 2 before-repair: showcases responsive layout challenges in overall design.



(d) Example 2 after-repair: issues resolved, addressing responsive design issues effectively.

Fig. 6: Examples Illustrating DesignRepair's Repair Effectiveness: Comparative Analysis Before and After Repairs

The performance of soft constraint repair is generally lower than that of hard constraint repair. On GitHub project, results for hard constraint repairs have lower recall but improved precision. These are due to the conservative approach of LLMs to avoid introducing new issues in complex pages, resulting in fewer true positives and false negatives. On Vercel's V0 projects, soft constraint repairs perform better, showcasing the LLM's ability to enhance design diversity and aesthetics when the tested pages have fewer hard constraint issues.

Overall Repair Results: For overall repair, the proposed method achieved 89.3% recall and 86.6% precision on Vercel's V0 projects, and 85.2% recall and 90.7% precision on GitHub projects. Our approach effectively adheres to design guidelines without extensive training. For instance, as illustrated in Fig. 6a and Fig. 6b, DesignRepair adeptly solves element overlapping issue and enhances background color contrast, making the design accessibility-friendly and visually appealing. Furthermore, as showcased in Fig. 6c and Fig. 6d, our tool effectively addresses responsive layout issues, ensuring optimal appearance across various window sizes. Another striking example, presented in Fig. 1b, demonstrates our software's proficiency in unifying component colors and improving aspects like accessibility, text legibility, and con-

trast.

In conclusion, DesignRepair demonstrates a remarkable capacity for frontend code repair, significantly enhancing user experience and interface quality. Its holistic approach, blending various strategies and constraints, proves crucial in navigating and rectifying a wide range of design issues, thus establishing it as an effective tool in contemporary frontend development.

D. RQ3: Perceived usefulness

1) ***Procedure****:* A user study was conducted to evaluate the usefulness and perceived quality of the proposed approach.

We recruited a total of 26 participants (12 female,14 male), aged between 25 and 35. The group comprised 6 researchers, 10 students, and 10 software developers from diverse countries including Australia 8, China 15, and the United States 3. The educational backgrounds of the participants were varied: 5 had Bachelor's degrees, 15 had Master's degrees, and 6 had Ph.D.s. Their affiliations were diverse, with 6 from science agencies, 10 from major internet companies, and 10 from universities.

To align with our research background, the study used both the original and repaired versions of 20 different projects from Vercel V0 (as discussed in Section IV-A). We created two surveys, each containing 10 different pairs of interactive rendered UI page videos, with each participant randomly assigned to one survey. Thus, 13 evaluations per survey and video. Each video pair featured a before and after repair view, and we randomized their order to obscure which was produced by our tool. The videos, each 10 seconds long, displayed the rendered UI page of the code, including interactions with elements and page resizing to demonstrate UI changes and UX. Participants could repeatedly view the videos without time constraints while responding to questions, ensuring detailed observation.

For each pair of videos, participants were asked to review and answer the question: "Please rate each page's visual attractiveness using a 5-point Likert scale." After participants completed their ratings, we conducted interviews to reveal which pages were repaired and discuss their scoring rationale. This approach uncovered patterns, such as why some pages were rated higher before repairs.

2) ***Results****:* The user study's results, depicted in Fig. 7, illustrate the satisfaction ratings for the 20 subjects' UIs before-after repaired. The Likert scores, ranging from 1 to 5, are distributed along the x-axis with before repair scores in blue and after repair scores in red. Notably, there is a significant shift in participant responses towards the higher scores (4 and 5) after repair, indicating improved user satisfaction. The most frequent before-repair score was 3, shifting to 4 after-repair, clearly signifying enhanced satisfaction with the repaired interfaces.

For a more rigorous analysis, we employed the Wilcoxon Signed-Rank test. The test yielded a p-value of approximately $p \approx 6.99 \times 10^{-9}$, far below the standard alpha threshold of 0.05. This statistically significant result indicates a meaningful difference in user satisfaction ratings before and after repairs. As seen in Fig. 8a, the mean satisfaction score rose from 3.04 to 3.92 when applying repair using our work, with a notable shift in score distribution towards higher ratings. This upward trend, coupled with a reduction in standard deviation (from 1.19 to 0.99), suggests a more uniformly positive user experience following the repairs.

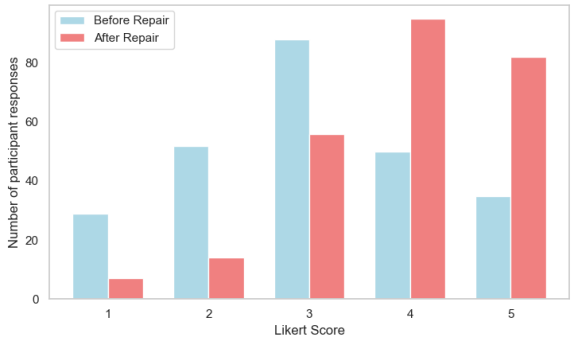
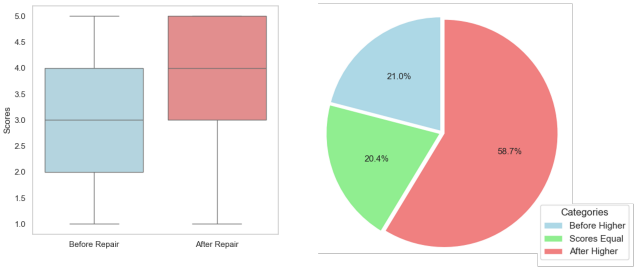


Fig. 7: Frequency of Likert Scores Before and After Repair



(a) Comparison of Scores Before and After Repair

(b) Score Comparisons: Before vs After Repair

Fig. 8: Comparative Analysis of User Satisfaction

Knowing the order of the videos, we calculated how often the before UI page scored lower, equal to, or higher than the after-repair UI page. As shown in Fig. 8b, a notable 58.7% of users regarded the pages as visually superior before-repair. In the interview, participants provided feedback indicating an enhanced user experience, evidenced by greater ease of use, more efficient information retrieval, and a noticeable increase in design attractiveness.

Interestingly, 20.4% of participants rated the before and after repair versions equally. Feedback from these participants's interview indicated that in cases where the frontend code was complex and the errors minor (involving aspects like content and accessibility features), the distinctions in repair quality were not immediately apparent. Conversely, 20.1% of participants expressed a preference for the original design. After being informed about the guideline checks and repairs for readability and interactivity, some participants still preferred the original versions. They felt that certain soft constraint repairs were subjective. Moreover, some design issues, like colorful themes, did not affect usability and were more engaging. Overall, our approach demonstrates its effectiveness through both statistical and descriptive analyses.

E. Threats to Validity

Due to the limitations of manually recognizing design issues in the code, the distribution of identified design issues may

differ from the broader context. For example, annotators found it more challenging to identify soft constraints. Therefore, our statistics may not fully reflect the proportions in AI-generated and GitHub project issues. There may be some bias. Testing on a larger dataset in the future could give additional insights.

V. Discussion and Future Work

Our work presents an extensible, knowledge-based code repair method that effectively utilizes design guidelines without requiring manual rule building and development. This approach's strength lies in its flexibility to support and integrate diverse guidelines. By leveraging LLMs and an extendable knowledge-driven framework, our method can adapt seamlessly to evolving guidelines. Furthermore, future improvements could focus on mechanisms to effortlessly incorporate guideline updates and further narrow the scope of LLM inspection. We have identified a promising direction by combining related specifications with relevant code, properties, and values to achieve more effective performance. We found that shortening the reasoning process can enhance code repair capabilities. Combining LLMs with traditional page analysis tools can create a more efficient workflow by leveraging the unique strengths of each tools. These advancements ensure alignment with industry standards, enabling organizations to tailor the approach to their specific needs.

Additionally, our work provides a first inspection of LLM-generated frontend code issues and methods to improve them, which is valuable for enhancing generated code, especially frontend code. In the future, we can incorporate LLM issue detection and repair during the code generation process, enabling early problem identification and more reliable code creation.

VI. Related work

A. Accessibility Testing Tools

Widely-used accessibility testing tools like Playwright [23], Google Lighthouse [24], and axe-core [25] are primarily designed for addressing accessibility issues like text contrast, icon tapability, image accessibility, display issues, and intention-practice mismatches. These tools lack various aspects of repair capabilities and often miss many issues. A user study [35] found they detected only about 50% of the issues they claimed, highlighting their limitations.

B. Research for UI Issues

To address the limitations of accessibility testing tools, researchers have developed tools targeting specific UI issues. OwlEye [16], IFIX [36], CBRepair [37], and MFIX [36] focus on Android GUI layout issues, web application internationalization presentation problems, and mobile web presentation issues, respectively. Iris [38] and Seenomaly [18] are specifically designed for color-related accessibility issues and detecting violations of design guidelines in animations. While these tools have contributed to advancing QA in their respective domains, they lack general, adaptive, and effective methods to address diverse frontend design issues.

UIS-Hunter [19] represents a vital effort in design smell detection by encompassing 23 types of UI components against 126 visual guidelines. This extensive scope marks a notable advancement in identifying a wide array of design issues. Despite its broad coverage, UIS-Hunter struggles with complex design principles, focusing on component-level guidelines while missing higher-level aspects like layout issues. Furthermore, it can only detect issues and lacks repair ability.

C. LLM-based Code Generation and Repair

The emergence of automated program repair research, coupled with the advancements in LLMs, opens up new possibilities for an open approach for program repair. Pre-trained LLMs like CodenBERT [39], CodeX [40], AlphaCode [41], and Code Llama [2] have demonstrated significant improvements in code generation and repair. Research on performance enhancement in program repair has also progressed, with models like CodeT5 [42], CodeGen [43], and InCoder [44] showing competitive or superior performance compared to deep learning-based techniques. These models leverage knowledge-driven methods to enhance repair capabilities without additional model training.

However, while those LLMs have shown remarkable progress in code generation and repair, their focus has primarily been on broader problems and languages, such as algorithms, logic problems, Python, and C. Advanced LLM methods like MetaGPT [45] and KPC [46] have improved Python code project, and Java exception handling issue quality within the LLM QA framework, but frontend code repair using LLMs remains an underexplored area.

In summary, while accessibility testing tools, research tools for specific UI issues, they fail to provide a a broader spectrum and adaptable solution for frontend code repair. The remarkable progress in LLM-based code generation and repair presents an opportunity to explore potential ways to address the extensive design issues of frontend for QA. Our work focuses on this critical unexplored domain, building upon the advancements in LLMs and program repair techniques.

VII. Conclusion

In this paper, we present DesignRepair, a dual-stream, design guideline-aware system developed to provide a guidelines-adaptive method for detecting and repairing frontend code. To evaluate its effectiveness, we analyzed 20 projects created using Vercel's V0 and 66 files from 6 GitHub projects. Our manual examination revealed insights into the tool's adherence to design guidelines. We conducted evaluations and a user study to demonstrate its effectiveness.

References

[1] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang, "Sparks of artificial general intelligence: Early experiments with gpt-4," 2023.

[2] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024.

[3] N. Guha, J. Nyarko, D. E. Ho, C. Ré, A. Chilton, A. Narayana, A. Chohlas-Wood, A. Peters, B. Waldon, D. N. Rockmore, D. Zambrano, D. Talisman, E. Hoque, F. Surani, F. Fagan, G. Sarfaty, G. M. Dickinson, H. Porat, J. Hegland, J. Wu, J. Nudell, J. Niklaus, J. Nay, J. H. Choi, K. Tobia, M. Hagan, M. Ma, M. Livermore, N. Rasumov-Rahe, N. Holzenberger, N. Kolt, P. Henderson, S. Rehaag, S. Goel, S. Gao, S. Williams, S. Gandhi, T. Zur, V. Iyer, and Z. Li, "Legalbench: A collaboratively built benchmark for measuring legal reasoning in large language models," 2023.

[4] Y. Koreeda and C. D. Manning, "Contractnli: A dataset for document-level natural language inference for contracts," *arXiv preprint arXiv:2110.01799*, 2021.

[5] D. Hendrycks, C. Burns, A. Chen, and S. Ball, "Cuad: An expert-annotated nlp dataset for legal contract review," *arXiv preprint arXiv:2103.06268*, 2021.

[6] S. H. Wang, A. Scardigli, L. Tang, W. Chen, D. Levkin, A. Chen, S. Ball, T. Woodside, O. Zhang, and D. Hendrycks, "Maud: An expert-annotated legal nlp dataset for merger agreement understanding," *arXiv preprint arXiv:2301.00876*, 2023.

[7] D. Cheng, S. Huang, and F. Wei, "Adapting large language models via reading comprehension," in *The Twelfth International Conference on Learning Representations*, 2024.

[8] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," 2024.

[9] "V0 development platform," https://v0.dev/, 2024.

[10] "A new way to think and create with computers," https://www.imagica.ai/, 2024.

[11] "openv0: Ai generated ui components," https://github.com/raidendotai/openv0, 2024.

[12] "React: A javascript library for building user interfaces," https://react.dev/, 2024.

[13] "shadcn/ui: Beautifully designed open-source ui components," https://ui.shadcn.com/, 2024.

[14] "Tailwind css: Rapidly build modern websites without ever leaving your html," https://tailwindcss.com/, 2024.

[15] "Material design 3," https://m3.material.io/, 2024.

[16] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Owl eyes: Spotting ui display issues via visual understanding," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 398–409.

[17] J. Chen, A. Swearngin, J. Wu, T. Barik, J. Nichols, and X. Zhang, "Towards complete icon labeling in mobile applications," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–14.

[18] D. Zhao, Z. Xing, C. Chen, X. Xu, L. Zhu, G. Li, and J. Wang, "Seenomaly: Vision-based linting of gui animation effects against design-don't guidelines," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1286–1297.

[19] B. Yang, Z. Xing, X. Xia, C. Chen, D. Ye, and S. Li, "Don't do that! hunting down visual design smells in complex uis against design guidelines," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 761–772.

[20] J. Chen, C. Chen, Z. Xing, X. Xia, L. Zhu, J. Grundy, and J. Wang, "Wireframe-based ui design search through image autoencoder," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 3, pp. 1–31, 2020.

[21] A. K. Vajjala, S. H. Mansur, J. Jose, and K. Moran, "Motorease: Automated detection of motor impairment accessibility issues in mobile app uis," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 993–993.

[22] K. Moran, B. Li, C. Bernal-Cárdenas, D. Jelf, and D. Poshyvanyk, "Automated reporting of gui design violations for mobile apps," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. ACM, May 2018.

[23] "Playwright: Framework for web testing and automation," https://github.com/microsoft/playwright, 2024.

[24] "Lighthouse: Automated auditing, performance metrics, and best practices for the web," https://github.com/GoogleChrome/lighthouse, 2024.

[25] "axe-core: Accessibility engine for automated web ui testing," https://github.com/dequelabs/axe-core, 2024.

[26] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Owl eyes: spotting ui display issues via visual understanding," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. ACM, Dec. 2020.

[27] J. Mahmud, N. De Silva, S. A. Khan, S. H. Mostafavi, S. H. Mansur, O. Chaparro, A. Marcus, and K. Moran, "On using gui interaction data to improve text retrieval-based bug localization," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[28] N. Salehnamadi, A. Alshayban, J.-W. Lin, I. Ahmed, S. Branham, and S. Malek, "Latte: Use-case and assistive-service driven automated accessibility testing framework for android," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–11.

[29] A. Swearngin and Y. Li, "Modeling mobile interface tappability using crowdsourcing and deep learning," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019, pp. 1–11.

[30] J. Chen, J. Sun, S. Feng, Z. Xing, Q. Lu, X. Xu, and C. Chen, "Unveiling the tricks: Automated detection of dark patterns in mobile applications," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 2023, pp. 1–20.

[31] "Announcing v0: Generative ui," https://vercel.com/blog/announcing-v0-generative-ui/, 2023.

[32] "Next.js by vercel - the react framework for the web," https://nextjs.org/, 2024.

[33] "Ready-to-use foundational react components," https://github.com/mui/material-ui, 2024.

[34] "Typescript is javascript with syntax for types." https://www.typescriptlang.org/, 2024.

[35] R. Ismailova and Y. Inal, "Comparison of online accessibility evaluation tools: an analysis of tool effectiveness," *IEEE Access*, vol. 10, pp. 58 233–58 239, 2022.

[36] S. Mahajan, A. Alameer, P. McMinn, and W. G. Halfond, "Automated repair of internationalization presentation failures in web pages using style similarity clustering and search-based techniques," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 215–226.

[37] A. Alameer, P. T. Chiou, and W. G. Halfond, "Efficiently repairing internationalization presentation failures by solving layout constraints," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 172–182.

[38] 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 *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1255–1267.

[39] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[40] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[41] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[42] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[43] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "A conversational paradigm for program synthesis," *arXiv preprint arXiv:2203.13474*, vol. 30, 2022.

[44] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[45] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "Metagpt: Meta programming for a multi-agent collaborative framework," 2023.

[46] X. Ren, X. Ye, D. Zhao, Z. Xing, and X. Yang, "From misuse to mastery: Enhancing code generation with knowledge-driven ai chaining," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 976–987.