COMPONENT-AWARE MULTI-FRAMEWORK WEBUI CODE GENERATION WITH LIGHTWEIGHT VERIFICATION

Jingwu Chen **Xiaoshuang Chen** **Gaode Chen** **Liang Chen** **Yong Jiang***

ABSTRACT

Front-end code generation from visual inputs has achieved encouraging progress with multimodal large language models. However, most existing methods remain centered on HTML-style outputs and do not adequately address the requirements of modern component-based development. When targeting frameworks such as React, Vue, and Angular, the model must generate not only visually aligned code, but also framework-compliant project structures, reusable components, valid bindings, and cross-file dependencies. This makes executability a fundamental challenge in multi-framework WebUI code generation. In this paper, we propose CompoVerify, a component-aware generation framework for executable WebUI synthesis across multiple front-end frameworks. The key idea is to explicitly model reusable interface patterns and component boundaries before code generation, so that the system can move beyond flat page translation and produce framework-consistent project structures. Building on this representation, we introduce a framework-conditioned generation process that organizes component decomposition, file layout, and implementation decisions according to framework-specific conventions. To further improve reliability, we incorporate a lightweight verification module that detects common generation failures, including structural inconsistencies, missing dependencies, invalid bindings, and framework-specific syntax errors, and enables efficient correction before full compilation. By coupling component-aware structural modeling with low-cost verification, the proposed framework shifts WebUI code generation from visually plausible outputs toward executable and reusable front-end projects.

1 INTRODUCTION

Recent multimodal large language models and vision-language models have substantially improved design-to-code and screenshot-to-code generation, making front-end engineering a compelling testbed for grounded code synthesis Liu et al. (2023; 2024); Hong et al. (2024); Chen et al. (2023); Wang et al. (2024). Early systems mostly targeted simplified domains such as mobile GUI DSLs, HTML reconstruction, or sketch-driven prototyping Beltramelli (2018); Chen et al. (2018); Jain et al. (2019); Xu et al. (2021). More recent benchmarks such as Design2Code, WebCode2M, Web2Code, Interaction2Code, DesignBench, FullFront, and FrontendBench have pushed the field toward more realistic webpages, larger training corpora, and broader engineering workflows Si et al. (2024); Gui et al. (2024); Yun et al. (2024); Xiao et al. (2024; 2025); Sun et al. (2025); Zhu et al. (2025). Yet a central gap remains between visually plausible code and executable multi-file projects.

This gap is especially visible when generation moves from standalone HTML to component-based frameworks. In HTML-centric settings, a model can still obtain competitive visual similarity while relying on flat markup, redundant fragments, or brittle inline styles Laurencon et al. (2024); Si et al. (2024). In contrast, React, Vue, and Angular impose stronger conventions on component decomposition, file organization, template syntax, imports, bindings, and framework-specific semantics Xiao et al. (2025); Sun et al. (2025); Zhu et al. (2025). As a result, visual fidelity alone is no longer a sufficient proxy for practical usability.

*Corresponding author. Email: jiangyong@bytedance.com.

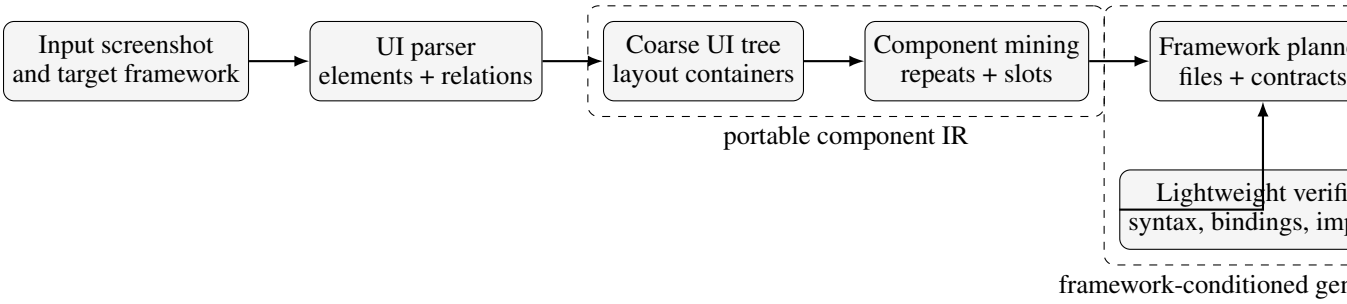

Figure 1: Overview of CompoVerify. The model first derives a coarse structural view of the interface, mines repeated components to form a portable intermediate representation, and then performs framework-conditioned project generation under lightweight verification.

A second limitation of many current approaches is that they implicitly treat a webpage as a single monolithic sequence generation problem. This design ignores a crucial property of modern front-end engineering, namely repeated structure. Real webpages often contain cards, navigation items, repeated lists, metric blocks, and shared templates. Prior work has begun to exploit hierarchy, layout, or self-correction Gui et al. (2025b); Wu et al. (2025); Chen et al. (2025); Jiang et al. (2025); Gui et al. (2025a), but reusable component structure is still weakly modeled in multi-framework settings. Without an explicit notion of components, generation often duplicates similar blocks, misses shared props, and produces inconsistent file-level abstractions.

To address these issues, we propose CompoVerify, a component-aware multi-framework WebUI code generation framework with lightweight verification. CompoVerifyfirst converts the input screenshot into a coarse UI tree enriched with element roles, spatial relations, and normalized style cues. It then mines repeated subtrees to form a portable component intermediate representation that exposes reusable components, repeated slots, and framework-agnostic properties. Based on this intermediate representation, a framework-conditioned planner produces a project blueprint describing file layout, component signatures, imports, and framework-specific constraints. A writer then generates code under this blueprint, while a lightweight verifier performs low-cost checks prior to full compilation to catch common failure modes such as missing imports, invalid template bindings, malformed component boundaries, and cross-file inconsistencies.

This paper makes four contributions. First, we formulate multi-framework WebUI code generation as an executability-oriented project synthesis problem rather than a screenshot-to-markup translation problem. Second, we introduce a component-aware intermediate representation that explicitly models reusable structure before code generation. Third, we design a lightweight verification mechanism that provides actionable feedback while avoiding the cost of repeated full builds. Fourth, we present a complete conference-style paper draft with experimental protocol and reporting templates grounded in the recent UI-to-code literature Si et al. (2024); Xiao et al. (2025); Sun et al. (2025); Zhu et al. (2025). Because this manuscript is authored without user-provided experimental results, quantitative tables are intentionally left as placeholders for later insertion rather than populated with fabricated numbers.

2 Related Work

UI code generation. Early UI-to-code work studied synthetic or constrained settings, including pixel-to-DSL generation, mobile GUI skeleton creation, sketch-based prototyping, and web image to code conversion Beltramelli (2018); Chen et al. (2018); Jain et al. (2019); Xu et al. (2021). The recent wave of multimodal LLMs has dramatically expanded the task to real webpages, multi-file implementations, and richer design inputs Si et al. (2024); Gui et al. (2024); Yun et al. (2024); Laurencon et al. (2024). Closest to our work are hierarchical or layout-guided systems such as UICopilot, LayoutCoder, DesignCoder, ScreenCoder, DeclarUI, and LaTCoder, which decompose generation using structure, layout, or iterative refinement Gui et al. (2025b); Wu et al. (2025); Chen et al. (2025); Jiang et al. (2025); Zhou et al. (2024); Gui et al. (2025a). Our method differs by

centering reusable components as an explicit intermediate representation and by targeting consistent generation across multiple front-end frameworks.

Benchmarks for front-end engineering. Design2Code introduced a realistic screenshot-to-webpage benchmark and automatic visual evaluation Si et al. (2024). WebCode2M and Web2Code focused on larger corpora for training and evaluation Gui et al. (2024); Yun et al. (2024). Interaction2Code extended evaluation to interactive behaviors Xiao et al. (2024). FullFront and FrontendBench argued that practical front-end engineering requires broader workflow coverage and executable end-to-end assessment Sun et al. (2025); Zhu et al. (2025). DesignBench further emphasized multi-framework and multi-task evaluation Xiao et al. (2025). Our experimental design follows this line of work and prioritizes compilation success, framework consistency, and visual fidelity together.

UI understanding and multimodal code models. High-quality UI code generation depends on strong screen understanding. Representative datasets and models include RICO, UIBert, ActionBert, ScreenQA, Pix2Struct, and CogAgent Deka et al. (2017); Bai et al. (2021); He et al. (2021); Hsiao et al. (2022); Lee et al. (2022); Hong et al. (2024). Generalist multimodal models such as LLaVA, LLaVA-NeXT, InternVL, and Qwen2-VL provide competitive perception and OCR capabilities that are useful for UI parsing and grounding Liu et al. (2023; 2024); Chen et al. (2023); Wang et al. (2024). We build on these advances but focus on the downstream project synthesis problem.

Code generation, self-refinement, and feedback. Modern code models such as Code Llama and CodeT5+ provide strong backbones for generation and repair Roziere et al. (2023); Wang et al. (2023). Several works show that iterative self-feedback, self-debugging, and verbal reflection can improve code quality without retraining Madaan et al. (2023); Chen et al. (2024); Shinn et al. (2023); Yao et al. (2023). Our verifier is inspired by this general philosophy, but it is tailored to front-end projects: instead of relying solely on full execution or generic reflection, it applies low-cost framework-specific checks to offer targeted corrective signals before invoking compilation.

3 Problem Formulation

Given an interface screenshot x and a target framework $f \in \{\mathrm{React}, \mathrm{Vue}, \mathrm{Angular}\}$, the goal is to generate an executable project y_f that reconstructs the input interface while respecting framework-specific syntax and engineering conventions. Unlike HTML-only settings, y_f is a structured object containing multiple files, component declarations, imports, templates, bindings, styles, and asset references.

We model generation as a conditional mapping

$$P(y_f \mid x, f) = P(y_f \mid \mathcal{G}(x), \mathcal{F}_f), \tag{1}$$

where $\mathcal{G}(x)$ is a portable structural representation extracted from the screenshot, and $\mathcal{F}_f$ is a framework contract specifying valid project-level constraints for framework f.

We evaluate a generated project along three dimensions. Executability measures whether the project can pass parsing or compilation. Fidelity measures whether the rendered page visually matches the reference screenshot. Engineering coherence measures whether repeated UI structures are abstracted into reusable components rather than duplicated as flat markup. This last criterion is often ignored in prior work even though it strongly affects maintainability and downstream editability.

4 Method

4.1 Overview

Figure 1 summarizes CompoVerify. The framework contains four stages: a UI parser, a component miner, a framework-conditioned planner and writer, and a lightweight verifier. The parser converts the screenshot into a coarse tree of elements and layout containers. The miner detects repeated subtrees and canonicalizes them into reusable components with slot placeholders. The planner maps the component IR to a framework-specific project blueprint. The writer instantiates code file by file under this blueprint. Finally, the verifier performs bounded correction with cheap framework-aware checks.

4.2 COMPONENT-AWARE STRUCTURE INDUCTION

We first infer a coarse UI tree $\mathcal{T} = (V, E)$ from the screenshot. Each node $v \in V$ is associated with an element type $t(v)$, a semantic role $r(v)$, a bounding box $b(v)$, textual content $c(v)$ when available, and a light style descriptor $s(v)$ including coarse font, spacing, color, and alignment attributes. The parser can be instantiated using any strong screenshot understanding backbone, including Pix2Struct-style sequence prediction or a modern multimodal parser Lee et al. (2022); Hong et al. (2024); Wang et al. (2024).

The tree is intentionally coarse. Instead of attempting to recover exact DOM nodes from pixels, we only require enough structural information to identify layout containers, repeated children, and likely reusable blocks. This design reduces sensitivity to small OCR or alignment errors and matches the abstraction level needed for component synthesis.

To improve structural stability, we assign each node a normalized label

$$\ell(v) = \big(t(v), r(v), q(b(v)), q(s(v))\big), \tag{2}$$

where $q(\cdot)$ discretizes geometry and style features into coarse bins. These labels are used downstream for repeat detection.

4.3 PORTABLE COMPONENT IR

A key observation is that repeated interface blocks, such as cards or navigation items, should be treated as components before framework-specific code generation. We therefore mine recurring subtrees from $\mathcal{T}$ using bottom-up canonicalization inspired by subtree hashing and Weisfeiler-Lehman style refinement Shervashidze et al. (2011); Merkle (1989).

For each node v, we compute a canonical signature

$$h(v) = H\Big(\ell(v), h(u_1), \ldots, h(u_m)\Big), \tag{3}$$

where $u_1, \ldots, u_m$ are ordered children and $H(\cdot)$ is a deterministic hash over normalized subtree structure. Candidate components are selected from signatures that recur above a frequency threshold and satisfy structural stability constraints. We then convert repeated literal content into slots or props, such as `title`, `subtitle`, `imageSrc`, or `value`.

The resulting intermediate representation is

$$\mathcal{G} = (\mathcal{T}, \mathcal{C}, \Pi), \tag{4}$$

where $\mathcal{C}$ is a set of reusable component templates and Π is a set of slot instantiations. Importantly, $\mathcal{G}$ is framework agnostic. It captures component boundaries, repetition, and typed placeholders without committing to JSX, Vue templates, or Angular decorators.

4.4 FRAMEWORK-CONDITIONED PLANNING AND WRITING

Given $\mathcal{G}$ and a framework contract $\mathcal{F}_f$, the planner emits a project blueprint consisting of a file graph, component signatures, import dependencies, and framework-specific implementation rules. The contract contains hard constraints such as legal file extensions, template rules, metadata declarations, binding syntax, component registration requirements, and style scoping conventions.

The planner solves two problems. First, it maps each portable component to a concrete file or module under the target framework. Second, it decides how slots, repeated instances, and container layouts should be implemented using framework-native idioms. For example, repeated children may be realized using array mapping in React, `v-for` in Vue, or `*ngFor` in Angular. The planner also determines whether local state, props, and scoped styles are required.

The writer then generates code conditioned on the blueprint rather than the screenshot alone. This reduces the search space by separating high-level structure from local realization. In practice, we serialize the blueprint into a structured prompt or a planner output consumed by a code model, similar in spirit to structured generation pipelines used in recent UI code systems Gui et al. (2025b); Wu et al. (2025); Jiang et al. (2025).

Algorithm 1 Framework-conditioned generation with bounded correction

Require: Screenshot x, target framework f, max repairs K
$\mathcal{T} \leftarrow \textsc{ParseUI}(x)$
$\mathcal{G} \leftarrow \textsc{MineComponents}(\mathcal{T})$
$b \leftarrow \textsc{Plan}(\mathcal{G}, \mathcal{F}_f)$
for $k = 1$ to K **do**
    $y^{(k)} \leftarrow \textsc{Write}(b)$
    $r^{(k)} \leftarrow \textsc{Verify}(y^{(k)}, \mathcal{F}_f)$
    if $r^{(k)}$ is Pass **then**
        return $y^{(k)}$
    else
        $b \leftarrow \textsc{RepairPlan}(b, r^{(k)})$
    end if
end for
return $\arg\max_k \mathcal{S}(y^{(k)})$

4.5 Lightweight verification

Full compilation or browser execution is expensive, especially during iterative generation. We therefore design a lightweight verifier that catches a substantial fraction of errors before invoking the build system. Let $y^{(k)}$ denote the project after the k-th generation attempt. The verifier computes

$$\mathcal{S}(y^{(k)}) = \sum_{j=1}^{J} \alpha_j \phi_j\big(y^{(k)}, \mathcal{F}_f\big), \tag{5}$$

where each ϕ_j is a binary or soft check and α_j is its weight. We use five families of checks.

Syntax and balance. We verify bracket and tag balance, malformed JSON or TypeScript metadata, and incomplete structured outputs.

Dependency consistency. We ensure required imports, component registrations, and file references are present and non-duplicated.

Binding validity. We check that template variables, props, loops, and event handlers are declared in the relevant scope.

Component contract consistency. We verify that a component is called with the props it declares and that slot names match the component template.

Framework contract compliance. We validate framework-specific invariants such as Angular metadata-template linkage, Vue template script alignment, or React client component boundaries.

If $\mathcal{S}(y^{(k)})$ is below a threshold, the verifier returns a compact correction plan. The writer is then allowed a bounded number of repair attempts. This yields a low-cost generate-verify-repair loop tailored to front-end code.

4.6 Training objective

Although CompoVerifycan be instantiated in an inference-only manner with a frozen multimodal model and rule-based verifier, we also describe a trainable variant for completeness. Let z denote the serialized project blueprint and y the target project. We optimize

$$\mathcal{L} = \mathcal{L}_{\text{tree}} + \lambda_1 \mathcal{L}_{\text{comp}} + \lambda_2 \mathcal{L}_{\text{plan}} + \lambda_3 \mathcal{L}_{\text{code}} + \lambda_4 \mathcal{L}_{\text{verify}}. \tag{6}$$

Here $\mathcal{L}_{\text{tree}}$ supervises the coarse tree prediction, $\mathcal{L}_{\text{comp}}$ supervises repeat and slot detection, $\mathcal{L}_{\text{plan}}$ supervises the project blueprint, $\mathcal{L}_{\text{code}}$ is token-level code generation loss, and $\mathcal{L}_{\text{verify}}$ trains a small verifier head to predict common failure types. This decomposition makes the method compatible with corpora such as WebCode2M, Web2Code, and synthetic HTML-screenshot pairs Gui et al. (2024); Yun et al. (2024); Laurencon et al. (2024).

5 EXPERIMENTAL PROTOCOL

5.1 DATASETS

We recommend training on large-scale screenshot-code corpora and evaluating on realistic multi-framework benchmarks. Specifically, WebSight, Web2Code, and WebCode2M can be used for pretraining or instruction tuning, while Design2Code, DesignBench, FullFront, and FrontendBench provide realistic evaluation settings Laurencon et al. (2024); Yun et al. (2024); Gui et al. (2024); Si et al. (2024); Xiao et al. (2025); Sun et al. (2025); Zhu et al. (2025). Interaction2Code can be used as an auxiliary transfer benchmark to probe whether component-aware structure helps on dynamic edits or repairs Xiao et al. (2024).

5.2 BASELINES

The primary comparisons should include strong multimodal baselines and structure-aware UI code systems. We suggest evaluating generic VLMs or code-capable MLLMs such as LLaVA-NeXT, InternVL, Qwen2-VL, and CogAgent, together with recent task-specific systems including UICopilot, LayoutCoder, DesignCoder, ScreenCoder, DeclarUI, and LaTCoder Liu et al. (2024); Chen et al. (2023); Wang et al. (2024); Hong et al. (2024); Gui et al. (2025b); Wu et al. (2025); Chen et al. (2025); Jiang et al. (2025); Zhou et al. (2024); Gui et al. (2025a). For the code-writing backbone, open code models such as Code Llama and CodeT5+ are natural choices Roziere et al. (2023); Wang et al. (2023).

5.3 METRICS

We recommend reporting four groups of metrics. First, **compilation success rate** measures end-to-end executability and should be the primary metric in multi-framework settings Xiao et al. (2025); Zhu et al. (2025). Second, **render fidelity** should be measured by screenshot similarity metrics such as CLIP-based similarity together with pixel-level or layout-aware metrics used in existing benchmarks Si et al. (2024); Chen et al. (2025). Third, **engineering coherence** should quantify component reuse, average duplication ratio, and file-level consistency. Fourth, **verification efficiency** should report the fraction of failures caught before full compilation and the average number of repair iterations.

5.4 IMPLEMENTATION DETAILS

A practical setup is to use a parser derived from Pix2Struct or a strong UI VLM, followed by a planner and writer implemented with a code-capable LLM Lee et al. (2022); Hong et al. (2024); Roziere et al. (2023). The verifier can be implemented as a lightweight static analyzer mixed with learned failure prediction. During inference, we cap repair iterations at $K \in \{2, 3, 4\}$ to keep latency manageable. Since the final goal is project generation rather than pure token likelihood, we recommend selecting the best candidate by the verifier score and only then invoking the full framework build.

6 RESULTS AND ANALYSIS

Important note. This manuscript is intentionally written as a complete paper draft without inventing experimental numbers. All quantitative cells below are placeholders that should be replaced with actual results once experiments are available. The narrative around the expected findings is written to match the method and current literature, but should be revised after real measurements are collected.

We expect CompoVerifyto offer its clearest gains on executability and engineering coherence. Because the framework explicitly reasons over repeated structure and project contracts, it should reduce duplicated blocks, missing imports, and binding mismatches that frequently hurt CSR on component-based frameworks Xiao et al. (2025); Sun et al. (2025). Gains in visual similarity may be smaller but should remain competitive because the component IR preserves layout and repeated local structure.

Cross-framework executability. The central hypothesis is that the benefit of component awareness grows with framework strictness. React often tolerates relatively direct component synthesis, while

Table 1: Main comparison on multi-framework WebUI code generation. Metrics should include compilation success rate (CSR), CLIP similarity, and a component reuse score (CRS). Replace *TBD* with actual numbers.

Method	React			Vue			Angular		
	CSR	CLIP	CRS	CSR	CLIP	CRS	CSR	CLIP	CRS
LLaVA-NeXT	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*
InternVL / Qwen2-VL	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*
UICopilot	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*
LayoutCoder	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*
DesignCoder	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*
ScreenCoder	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*
CompoVerify	***TBD***	***TBD***	***TBD***	***TBD***	***TBD***	***TBD***	***TBD***	***TBD***	***TBD***

Table 2: Ablation study. Remove each module from CompoVerifyand report the drop in executability and reuse.

Variant	CSR	CLIP	CRS
Full CompoVerify	*TBD*	*TBD*	*TBD*
without component mining	*TBD*	*TBD*	*TBD*
without framework planner	*TBD*	*TBD*	*TBD*
without lightweight verifier	*TBD*	*TBD*	*TBD*
flat one-shot generation	*TBD*	*TBD*	*TBD*

Vue and especially Angular impose stricter template and file-level rules. The lightweight verifier is therefore expected to contribute larger relative gains on Vue and Angular, where template-binding and metadata consistency matter more Xiao et al. (2025); Zhu et al. (2025).

Ablation analysis. Ablations should show that component mining mainly improves CRS and indirectly improves CSR by reducing duplicated or inconsistent structures. Removing the framework planner should hurt all frameworks but especially Angular, where file-level conventions and declarations matter. Removing lightweight verification should primarily reduce CSR while having smaller impact on CLIP, supporting the claim that executability errors are often orthogonal to visual fidelity.

Why lightweight verification matters. Recent front-end benchmarks show that full end-to-end generation is highly brittle, and that even strong MLLMs fail on code organization, repairs, and interaction details Sun et al. (2025); Zhu et al. (2025); Xiao et al. (2024). Our verifier is deliberately simpler than compiler-in-the-loop optimization used in some declarative UI systems Zhou et al. (2024). The point is not to replace compilation, but to filter obvious failures earlier and to provide structured feedback in a bounded loop. This design keeps inference practical while still moving beyond one-shot generation.

Qualitative analysis. Qualitative case studies should compare generated projects before and after component mining. In repeated-card or dashboard layouts, the flat baseline typically emits many near-duplicate blocks with slight inconsistencies in spacing or naming. By contrast, CompoVerifyshould recover a shared card component with stable props and cleaner file structure. Similar benefits should appear on navigation bars, pricing tables, metrics panels, and repeated list items.

Generalization to edit and repair settings. Because CompoVerifyoutputs an explicit project blueprint and reusable component hierarchy, it should also support edit and repair tasks more naturally than monolithic generators. Benchmarks such as DesignBench and FrontendBench include realistic edit or debugging scenarios in which component boundaries matter Xiao et al. (2025); Zhu et al. (2025). A natural extension is to edit the IR or blueprint first and then regenerate only the affected files, reducing unnecessary churn.

Error taxonomy. Table 4 organizes the failure modes that should be reported in qualitative analysis. We expect the dominant errors of generic multimodal generators to shift from perception and layout

Table 3: Verifier analysis. Report the share of failures caught before full build and the average repair budget.

Failure family	Catch rate	Avg. repair steps
Malformed syntax / tags	*TBD*	*TBD*
Missing imports / files	*TBD*	*TBD*
Invalid bindings / scope	*TBD*	*TBD*
Component contract mismatch	*TBD*	*TBD*
Framework-specific violations	*TBD*	*TBD*

Table 4: Suggested error taxonomy for qualitative analysis. The last column can be filled with empirical prevalence once experiments are completed.

Category	Typical symptom	Root cause	Most affected frameworks	Prevalence
Perception error	Missing icon, wrong text, or inaccurate local alignment	Weak OCR, tiny visual element omission, or poor grounding of dense regions	All	*TBD*
Layout error	Incorrect grouping, stacking, or relative spacing	Coarse layout parsing fails to recover container boundaries or repeated blocks	All	*TBD*
Componentization error	Similar blocks duplicated instead of abstracted into shared components	Missing repeat detection or unstable slot extraction	React, Vue, Angular	*TBD*
Binding error	Undefined template variables, loop items, or event handlers	Local generation ignores scope or prop declarations	Vue, Angular	*TBD*
Dependency error	Missing imports, missing registration, or broken file references	Multi-file coordination failure during project synthesis	React, Angular	*TBD*
Contract violation	Invalid framework-specific syntax or metadata mismatch	Writer violates target framework conventions despite visually plausible code	Vue, Angular	*TBD*

mistakes on simpler HTML tasks toward componentization, binding, and dependency errors on framework-based tasks. This hypothesis is consistent with the framework-specific findings reported by recent multi-framework benchmarks Xiao et al. (2025); Sun et al. (2025). An important benefit of our component IR is that it localizes many failures at the component boundary level, which makes post-hoc diagnosis more actionable than monolithic token-level debugging.

Cost and practicality. A common concern for iterative generation is inference overhead. Our design attempts to control this cost by making the expensive step, namely full compilation, a last resort rather than the default feedback source. In reporting final experiments, we recommend measuring the average number of verifier-triggered repairs, wall-clock latency, and build invocations per solved sample. This protocol helps clarify whether gains in executability come from better structure alone or from excessive trial-and-error. Since front-end project failure often renders the entire output unusable, a modest increase in low-cost verification can still be justified when it prevents repeated manual debugging downstream.

7 Discussion

Why components are the right abstraction. Componentization is not just a stylistic preference. It is the dominant engineering abstraction across modern front-end frameworks, and it is especially important when repeated structures appear in the target UI. If a model only optimizes screenshot similarity, it can often ignore this abstraction and still look visually correct. However, such outputs are hard to maintain, brittle under edits, and more likely to break when transplanted into real codebases. This is why we argue that reusable structure should be modeled before framework realization.

Limitations. Our method still relies on heuristic choices in structure normalization, component mining thresholds, and verifier rule design. Although these heuristics are cheaper than full compiler-

in-the-loop optimization, they may need adaptation when extending beyond the current frameworks. Another limitation is that the coarse UI tree may miss subtle semantics such as animation behavior, responsive breakpoints, or latent interactions. These gaps partly explain why interactive webpage generation remains difficult even for strong MLLMs Xiao et al. (2024); Sun et al. (2025). Finally, our current draft does not yet include real quantitative results, so the claimed advantages should be interpreted as a well-motivated research hypothesis until experiments are completed.

Future directions. There are at least three promising extensions. First, one can integrate richer grounding signals from DOM, design files, or accessibility trees when available. Second, the component IR could support more explicit responsive layout and state transitions, bringing the framework closer to interactive generation. Third, the verifier could be upgraded with learned failure classifiers or agentic correction policies, drawing inspiration from self-debugging and reflection methods in code generation Chen et al. (2024); Shinn et al. (2023); Madaan et al. (2023); Yao et al. (2023).

8 Conclusion

We presented CompoVerify, a component-aware multi-framework WebUI code generation framework with lightweight verification. The method reframes front-end generation as executable project synthesis and introduces a portable component intermediate representation to expose reusable structure before framework-specific writing. By combining structure induction, framework-conditioned planning, and low-cost verification, CompoVerifytargets a more realistic notion of quality than screenshot similarity alone. Although the current document is a writing-ready research draft with placeholder quantitative tables, it defines a concrete technical path for studying executable multi-framework front-end code generation under reusable component abstractions.

References

Chongyang Bai, Xiaoxue Zang, Ying Xu, Srinivas Sunkara, Abhinav Rastogi, Jindong Chen, and Blaise Aguera y Arcas. Uibert: Learning generic multimodal representations for ui understanding. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, 2021.

Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 2018.

Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. From ui design image to gui skeleton: A neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*, 2018.

Xinyun Chen, Maxwell Lin, Nathanael Scharli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, 2024.

Yunnong Chen, Shixian Ding, YingYing Zhang, Wenkai Chen, Jinzhou Du, Lingyun Sun, and Liuqing Chen. Designcoder: Hierarchy-aware and self-correcting ui code generation with large language models. *arXiv preprint arXiv:2506.13663*, 2025.

Zhe Chen, Weiyun Wang, Yue Cao, Zhaoyang Liu, et al. Internvl: Scaling up vision foundation models and aligning for generic visual-linguistic tasks. *arXiv preprint arXiv:2312.14238*, 2023.

Biplab Deka, Zifeng Huang, Amy X. Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, 2017.

Yi Gui, Yao Wan, Zhen Li, Zhongyi Zhang, Dongping Chen, Hongyu Zhang, et al. Webcode2m: A real-world dataset for code generation from web page screenshots. *arXiv preprint arXiv:2404.06369*, 2024.

Yi Gui, Zhen Li, Zhongyi Zhang, Guohao Wang, Tianpeng Lv, Gaoyang Jiang, Yi Liu, Dongping Chen, Yao Wan, Hongyu Zhang, et al. Latcoder: Converting webpage design to code with layout-as-thought. *arXiv preprint arXiv:2508.03560*, 2025a.

Yi Gui, Yao Wan, Zhen Li, Zhongyi Zhang, Dongping Chen, Hongyu Zhang, Yi Su, Bohua Chen, Xing Zhou, Wenbin Jiang, et al. Uicopilot: Automating ui synthesis via hierarchical code generation from webpage designs. In *Proceedings of the ACM Web Conference*, 2025b.

Zecheng He, Srinivas Sunkara, Xiaoxue Zang, Ying Xu, Lijuan Liu, Nevan Wichers, Gabriel Schubiner, Ruby Lee, and Jindong Chen. Actionbert: Leveraging user actions for semantic understanding of user interfaces. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxuan Zhang, Juanzi Li, et al. Cogagent: A visual language model for gui agents. *arXiv preprint arXiv:2312.08914*, 2024.

Yu-Chen Hsiao et al. Screenqa: Large-scale question-answer pairs over mobile app screenshots for screen understanding. *arXiv preprint arXiv:2209.08199*, 2022.

Vanita Jain, Piyush Agrawal, Subham Banga, Rishabh Kapoor, and Shashwat Gulyani. Sketch2code: Transformation of sketches to ui in real-time using deep neural network. *arXiv preprint arXiv:1910.08930*, 2019.

Yilei Jiang, Yaozhi Zheng, Yuxuan Wan, Jiaming Han, Qunzhong Wang, Michael R. Lyu, and Xiangyu Yue. Screencoder: Advancing visual-to-code generation for front-end automation via modular multimodal agents. *arXiv preprint arXiv:2507.22827*, 2025.

Hugo Laurencon, Léonard Tronchon, Matthieu Cord, and Victor Sanh. Unlocking the conversion of web screenshots into html code with the websight dataset. *arXiv preprint arXiv:2403.09029*, 2024.

Kenton Lee, Mandar Joshi, Iulia Turc, Hexiang Hu, Fangyu Liu, Julian Eisenschlos, Urvashi Khandelwal, Peter Shaw, Ming-Wei Chang, and Kristina Toutanova. Pix2struct: Screenshot parsing as pretraining for visual language understanding. *arXiv preprint arXiv:2210.03347*, 2022.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *arXiv preprint arXiv:2304.08485*, 2023.

Haotian Liu, Chunyuan Li, Yuheng Li, Bo Li, Yuanhan Zhang, Sheng Shen, and Yong Jae Lee. Llava-next: Improved reasoning, ocr, and world knowledge. *arXiv preprint arXiv:2401.13601*, 2024.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

Ralph C. Merkle. A certified digital signature. In *Advances in Cryptology—CRYPTO '89 Proceedings*, 1989.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. In *Journal of Machine Learning Research*, volume 12, pages 2539–2561, 2011.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik R. Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2code: Benchmarking multimodal code generation for automated front-end engineering. *arXiv preprint arXiv:2403.03163*, 2024.

Haoyu Sun, Huichen Will Wang, Jiawei Gu, Linjie Li, and Yu Cheng. Fullfront: Benchmarking mllms across the full front-end engineering workflow. *arXiv preprint arXiv:2505.17399*, 2025.

Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, et al. Qwen2-vl: Enhancing vision-language model's perception of the world at any resolution. *arXiv preprint arXiv:2409.12191*, 2024.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

Fan Wu, Cuiyun Gao, Shuqing Li, Xin-Cheng Wen, and Qing Liao. Mllm-based ui2code automation guided by ui layout information. *arXiv preprint arXiv:2506.10376*, 2025.

Jingyu Xiao, Yuxuan Wan, Yintong Huo, Zhiyao Xu, and Michael R. Lyu. Interaction2code: How far are we from automatic interactive webpage generation? *arXiv preprint arXiv:2411.03292*, 2024.

Jingyu Xiao, Ming Wang, Man Ho Lam, Yuxuan Wan, Junliang Liu, Yintong Huo, and Michael R. Lyu. Designbench: A comprehensive benchmark for mllm-based front-end code generation. *arXiv preprint arXiv:2506.06251*, 2025.

Yong Xu, Lili Bo, Xiaobing Sun, Bin Li, Jing Jiang, and Wei Zhou. image2emmet: Automatic code generation from web user interface image. *Journal of Software: Evolution and Process*, 33(8): e2369, 2021.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2023.

Sukmin Yun, Haokun Lin, Rusiru Thushara, Mohammad Qazim Bhat, Yongxin Wang, Zutao Jiang, Mingkai Deng, Jinhong Wang, Tianhua Tao, Junbo Li, et al. Web2code: A large-scale webpage-to-code dataset and evaluation framework for multimodal llms. *arXiv preprint arXiv:2406.20098*, 2024.

Ting Zhou, Yanjie Zhao, Xinyi Hou, Xiaoyu Sun, Kai Chen, and Haoyu Wang. Bridging design and development with automated declarative ui code generation. *arXiv preprint arXiv:2409.11667*, 2024.

Hongda Zhu, Yiwen Zhang, Bing Zhao, Jingzhe Ding, Siyao Liu, Tong Liu, Dandan Wang, Yanan Liu, and Zhaojian Li. Frontendbench: A benchmark for evaluating llms on front-end development via automatic evaluation. *arXiv preprint arXiv:2506.13832*, 2025.