Render Structure Uncertainty for HTML Repair in MLLM-based UI-to-Code Generation

Haoran Ma[1*] **Jiechao Gao**[2*] **Shisong Tang**[3]
Bing Han[5] **Jifeng Hu**[4 †] **Hechang Chen**[4 †]

[1]Harvard University [2]Stanford University [3]Tsinghua University
[4]Jilin University [5]ByteDance Inc.

Abstract

Existing MLLM-based UI-to-Code generation methods can generate complete HTML/CSS codes through end-to-end or multi-stage pipelines, yet the generated codes often contain structural errors such as incorrect repeated layouts, wrong component boundaries, or missing interaction slots. Subsequent generations may continue along the erroneous structure, causing the error to propagate and amplify. This paper studies post-generation repair: how can we locate and repair unreliable local DOM structures? We observe that this repair process is ambiguous in two ways: a mismatch between the target screenshot and the output may correspond to multiple plausible DOM subtrees, and a single DOM subtree may admit multiple repairs with different rendered behaviors. We propose Render Structure Uncertainty (RSU) for HTML repair, a browser grounded uncertainty layer that can be attached to any base HTML generator. RSU parses the target screenshot into a visual structure graph and converts the browser execution trace of the generated code into a DOM-anchored render structure graph. By matching these graphs, RSU yields structural mismatches and a small set of candidate DOM holes, preserving error attribution uncertainty. For each hole, RSU generates multiple local candidates, renders them in the full page context, and clusters them using render-structure signatures that encode layout relations, repeated patterns, content slots, semantic roles, and invalid states, thereby modeling repair uncertainty. The retained representative candidates form a compact Partial DOM candidate set, over which we perform global reranking to produce the final HTML code. In this way, RSU reframes HTML repair into uncertainty-guided search in render-structure space, avoiding premature commitment to either a single error location or a single local repair. Experiments show that RSU improves the generation quality of base models, producing higher-quality HTML with better layout consistency and fewer local structural errors. Code is available here.

1 Introduction

With multimodal large language models (MLLMs), UI-to-Code systems can now directly map webpage designs to executable HTML/CSS [18, 6, 25], reducing repetitive implementation effort and accelerating development workflows. However, real webpages are structured artifacts whose correctness depends on DOM hierarchy, repeated layouts, component boundaries, and content slots [8, 32]. This makes UI-to-Code generation a demanding task where visual similarity alone is insufficient [27].

*Equal contribution.
†Corresponding author: chenhc@jlu.edu.cn and jifenghu@jlu.edu.cn

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

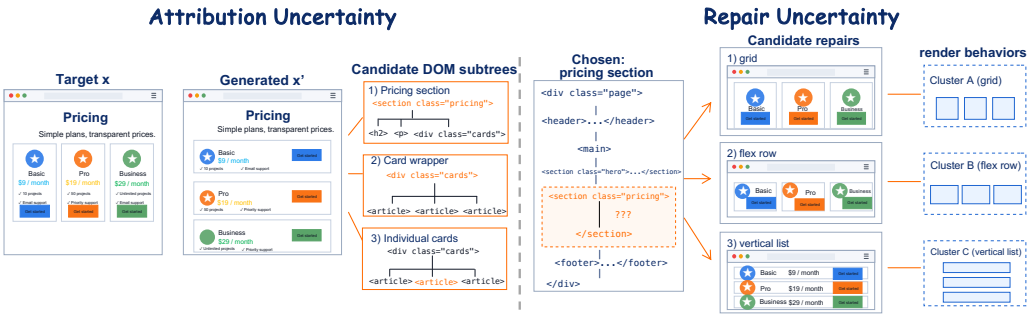


Figure 1: (a) Attribution uncertainty. A structural mismatch between the target page and the generated page, such as a row layout becoming a column layout, does not uniquely determine which DOM subtree is responsible. The error may come from the whole pricing section, an intermediate card wrapper, or individual card nodes. (b) Repair uncertainty. After choosing a candidate hole, multiple local repairs can lead to distinct rendered structures, such as a grid, a flex row, or a vertical list.

Recent MLLM-based UI code generation has advanced along three lines. Dataset and benchmark efforts such as Design2Code [18] and WebCode2M [8] scale evaluation and training with real webpage screenshots, rendered references, and layout-rich code pairs, revealing persistent weaknesses in element recall and layout fidelity. Structure-aware methods then reduce long-horizon drift by decomposing generation: DCGen [20] segments screenshots before reassembly, UICopilot [10] separates coarse HTML hierarchy from fine-grained synthesis, and LaTCoder [9] uses Layout-as-Thought to preserve block-level layout. More explicit layout and agentic systems further expose intermediate structure: LayoutCoder [22] builds element relations and UI layout trees for layout-guided fusion, while ScreenCoder [11] uses modular grounding, planning, and generation agents. Together, these works show a clear shift from black-box one-shot prompting toward structured and layout-aware UI code generation.

Despite these advances, most UI-to-Code research focuses on generation, while the problem of fixing structural errors after generation remains underexplored [27]. In practice, generated pages often contain local structural errors such as repeated layouts in the wrong direction, misplaced component boundaries, or missing interaction slots [20]. Addressing these errors requires two uncertain decisions, as illustrated in Figure 1. In Figure 1(a), a structural mismatch does not uniquely identify the responsible DOM subtree; a row-to-column error may come from the pricing section, an intermediate wrapper, or item-level constraints, which we call *attribution uncertainty*. In Figure 1(b), even after selecting a candidate DOM subtree, multiple valid local implementations may induce different rendered behaviors, such as a grid, a flex row, or a vertical list, which we call *repair uncertainty*. These two uncertainties motivate us to formulate post-generation Partial DOM repair as uncertainty-guided search over both DOM locations and rendered local implementations.

To address these, we propose Render Structure Uncertainty (RSU), a browser-grounded partial DOM repair framework that can be attached to any existing HTML generator. Instead of comparing the target and generated pages at the pixel level, RSU lifts both sides into a shared render-structure space. It parses the target screenshot into a visual structure graph that captures regions, repeated groups, layout relations, and content slots. In parallel, it executes the generated code in a browser and builds a DOM-anchored render structure graph, where each rendered region is linked back to the DOM subtree that produced it. By matching the two graphs, RSU identifies structural mismatches and proposes a small set of candidate DOM holes, without prematurely committing to a single error location. For each hole, RSU then generates multiple local repair candidates, renders them in the full-page context, and groups them by render-structure signatures that capture layout relations, repeated patterns, slots, semantic roles, and invalid states. The retained no-op candidate and cluster representatives form a compact Partial DOM candidate set, over which RSU performs global reranking to produce the final HTML. In this way, RSU turns post-generation repair into uncertainty-guided search over DOM locations and rendered repair behaviors, rather than a deterministic edit on a single predicted structure. Our contributions are summarized as follows:

- We formulate post-generation UI code repair as render-structure uncertainty modeling, identifying attribution uncertainty from structural mismatches to DOM holes and repair uncertainty from DOM holes to rendered candidate behaviors.
- We propose RSU, a browser-grounded Partial DOM repair framework, which aligns target visual structures with DOM-anchored render structures, proposes candidate DOM holes, clusters local repairs by rendered behavior, and reranks a compact candidate set.
- Experiments show that RSU improves HTML repair quality by producing layouts with stronger structural consistency, more complete content and interaction slots, and fewer local structural errors.

2 Related Works

UI-to-Code benchmarks Early UI-to-Code benchmarks framed the task as screenshot-to-DSL or GUI-skeleton translation [3, 4, 23], before shifting toward HTML/CSS through screenshot pre-training [13] and large-scale synthetic supervision [12]. With MLLMs, the focus moved to real-world webpages and structurally richer evaluation: Design2Code [18] introduces a five-dimensional rendered-screenshot protocol on 484 curated pages, Web2Code [32] adds an MLLM-as-judge framework, WebCode2M [8] scales to 2.56M pairs and proposes TreeBLEU for DOM hierarchy recall, WAFFLE [15] contributes structure-aware fine-tuning with hierarchical evaluation, and DesignBench [27] extends evaluation to React/Vue/Angular and to generation, editing, and repair tasks. A complementary line targets specialised settings: interaction [26], multi-page resource handling [19], sketch input [14], agentic end-to-end synthesis [16], and the related chart-to-code domain [21, 30]. Despite this progression, existing benchmarks predominantly assess code from a single forward pass and treat the generated DOM as a static artefact rather than a render trace that can be inspected, localised, and selectively repaired.

UI-to-Code methods Method-side progress on MLLM-based UI-to-Code spans four overlapping paradigms. The first decomposes the screenshot before generation, with DCGen [20] segmenting and reassembling per-region code, UICopilot [10] predicting a coarse skeleton before fine-grained synthesis, and ComUICoder [29] pursuing reusable component-level generation through semantic segmentation. The second exposes layout as an explicit intermediate: LaTCoder [9] reasons over layout-as-thought blocks, LayoutCoder [22] constructs element relations and a layout tree for layout-guided fusion, and ScreenCoder [11] organises grounding, planning, and generation into modular agents. A third line targets training-side improvement, with UICoder [24] self-generating and filtering synthetic UI code for fine-tuning, WAFFLE [15] introducing structure-aware attention and contrastive vision–code alignment, VinciCoder [33] applying coarse-to-fine visual reinforcement learning, ChartCoder [34] extending similar ideas to the chart-to-code domain, and EfficientUICoder [28] compressing input and output tokens to mitigate UI-specific redundancy. A fourth keeps the generator fixed and adds inference-time correction loops, including DesignCoder [5] for hierarchical self-correction, DeclarUI [35] for compiler-driven optimisation guided by page transition graphs, WebGen-Agent [16] for multi-level visual feedback with step-level reinforcement learning, and UI2Code$^{\mathrm{N}}$ [31] for test-time scalable interactive refinement. Together, these works mark a clear shift from black-box one-shot prompting toward structured, layout-aware, and feedback-driven generation.

3 Method

3.1 Problem Definition and Method Overview

Given a target webpage screenshot x, a base UI-to-Code model generates an initial HTML/CSS page y. We execute y in a browser and collect a render trace $\Omega_y = (x', T_y, B_y, C_y, S_y)$, where $x' = R(y)$ is the rendered screenshot, T_y is the DOM tree, B_y denotes element bounding boxes, C_y denotes computed styles, and S_y contains visibility, roles, clickability, and invalid states. Our goal is to produce an improved complete page $\hat{y}$ without retraining the base HTML generator.

RSU lifts screenshot comparison from pixels to render structures. It parses the target screenshot into a visual structure graph G_x, builds a DOM-anchored render structure graph G_y from Ω_y, and matches them to obtain structural mismatches. These mismatches are mapped to a small set of candidate

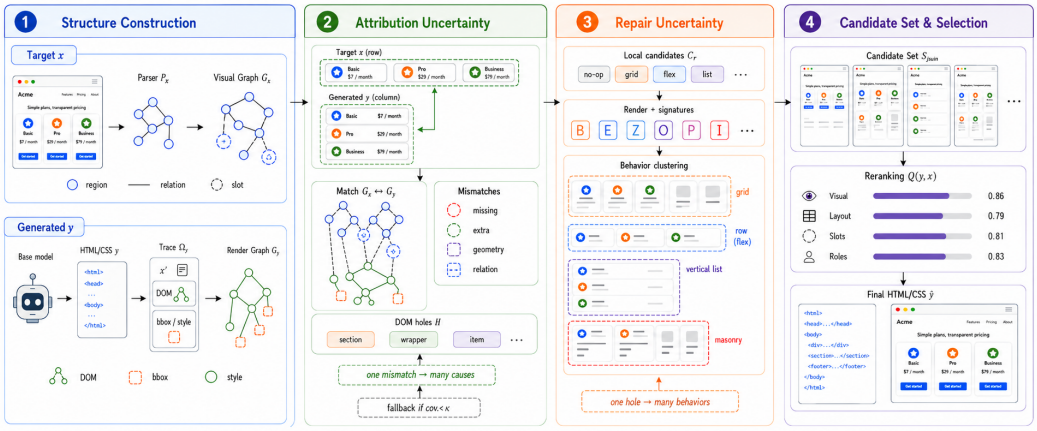


Figure 2: The post-generation repair process of RSU.

DOM holes, where each hole denotes a replaceable DOM subtree. RSU then generates multiple local candidates for each hole, renders them in the full-page context, clusters them by rendered behavior, and composes the retained representatives into a Partial DOM candidate set for global reranking. Formally, RSU implements $(x, y, x') \mapsto \hat{y}$ at test time.

3.2 Structure Graph Construction

We represent a screenshot x as a structure graph $G = (N, E)$, where N is the set of visual nodes and E is the set of relations. Node types include section, container, repeat group, card, text block, button, image, input, and icon group. Relations include inside, same row, same column, above, below, aligned left, aligned center, similar size, and repeat.

The target screenshot structure graph is denoted as $G_x = (N_x, E_x) = P_\theta(x)$, where P_θ is a screenshot structure parser. We render $y^\star$, extract DOM bounding boxes, tags, roles, text, images, and clickable elements, and compress the rendered DOM into a structure graph as pseudo-labels for training P_θ. During compression, invisible nodes and visually redundant wrappers are removed, while layout-controlling nodes, visible leaf nodes, and repeat groups are preserved.

The generated page structure graph is denoted as $G_y = (N_y, E_y)$. Unlike G_x, G_y is constructed from the browser rendering result of the generated code y. We render y, traverse the DOM, and collect each element's bounding box, computed style, tag, role, text length, clickable state, visibility, and parent-child relations. We then filter invisible nodes, merge visually redundant wrappers, and preserve nodes with layout-controlling properties, such as `display:flex`, `display:grid`, `position`, `gap`, `align-items`, and `justify-content`. Each structure node $n \in N_y$ in G_y is associated with a DOM anchor, denoted as $a(n) \in T_y$. For a single button or image, the anchor is usually the element itself. For a repeat group, the anchor is defined as the lowest common layout ancestor of the repeated item anchors, written as $a(n) = \mathrm{LayoutLCA}(a_1, \ldots, a_k)$. This anchor lets RSU map visual structure errors back to editable code subtrees.

3.3 Attribution Uncertainty in Structure Matching

This stage models *attribution uncertainty*: a structural mismatch between the target and generated pages does not uniquely determine which DOM subtree should be modified. We first match the target structure graph G_x and the DOM-anchored render structure graph G_y to obtain structural mismatches

$$\mathcal{M} = \mathrm{Match}(G_x, G_y). \tag{1}$$

The matching is hierarchical: top-level sections are aligned first, followed by text blocks, images, buttons, cards, and repeat groups inside matched sections. The resulting mismatches include missing nodes, extra nodes, geometry mismatches, and relation mismatches. Importantly, these mismatches indicate what is structurally inconsistent, but not a unique DOM cause.

We therefore map each mismatch to a small set of plausible DOM holes. For a relation mismatch involving generated nodes $n_1, \ldots, n_k$, we add

$$v = \mathrm{LayoutLCA}(a(n_1), \ldots, a(n_k)) \tag{2}$$

to the candidate holes, where $a(n) \in T_y$ is the DOM anchor of a generated structure node. To preserve attribution uncertainty, we also include the nearest layout parent and major wrapper child of v when they satisfy the replaceability constraints. For a missing node, we use the generated DOM anchor of its nearest matched parent region. For an extra node, we use the extra node anchor, and additionally the parent container anchor when multiple extra nodes share one container. For a geometry mismatch, we include the matched generated node anchor and its nearest layout ancestor.

Let $\mathcal{V}$ denote all candidate holes. Each $v \in \mathcal{V}$ explains a subset of mismatches $\mathcal{M}(v) \subseteq \mathcal{M}$. We select a compact attribution set $\mathcal{H}$ by greedy set cover, choosing at each step the candidate that covers the most uncovered mismatches. Ties favor the LayoutLCA of relation mismatches, then smaller visible area and lower CSS coupling. We exclude `html`, `body`, and global root wrappers, and require each hole to occupy between 1% and 50% of the page area.

The selected holes are $\mathcal{H} = \{v_1, \ldots, v_B\}$. This set is not a deterministic diagnosis of the unique error source; it is a compact search space over plausible DOM locations. If the selected holes explain too few mismatches, the initial DOM is considered unsuitable for local repair. We define

$$\mathrm{Cover}(\mathcal{H}) = \frac{\left|\bigcup_{v \in \mathcal{H}} \mathcal{M}(v)\right|}{|\mathcal{M}|}. \tag{3}$$

If $\mathrm{Cover}(\mathcal{H}) < \kappa$, the system enters global fallback, where full-page candidates are generated and reranked.

3.4 Repair Uncertainty through Rendered Local Candidate Behaviors

Given $\mathcal{H}$, RSU models the second uncertainty source, *repair uncertainty*. Even for a selected hole v, the local implementation may remain ambiguous: a pricing card group can be implemented as CSS grid, flex row, semantic repeated cards, or a vertical list, each inducing a different rendered behavior.

For each $v \in \mathcal{H}$, we generate $\mathcal{C}_v = \{s_v^0, s_v^1, \ldots, s_v^{K-1}\}$, where s_v^0 is the no-op candidate that preserves the original subtree. By default, $K = 8$, with one no-op and seven generated candidates. Candidate generation is conditioned on the mismatch type: row-versus-column mismatches trigger grid, flex row, semantic repeated cards, and responsive grid hypotheses; missing buttons require inserting the interaction slot into the target parent region; repeat-count mismatches require matching the target count and direction. All candidates rewrite only the selected subtree, preserve key text, images, and interaction slots, and use scoped CSS without changing global styles.

For each $s \in \mathcal{C}_v$, we replace the subtree as $y[v \leftarrow s]$, render the page, and extract the local render-structure signature

$$\phi_v(s) = (\mathcal{B}_v(s), \mathcal{E}_v(s), \mathcal{Z}_v(s), \mathcal{O}_v(s), \mathcal{P}_v(s), \mathcal{I}_v(s)). \tag{4}$$

Here, $\mathcal{B}_v$ stores visible bounding boxes normalized to the hole parent container; $\mathcal{E}_v$ stores layout relations such as inside, same row, same column, aligned, above, and below; $\mathcal{Z}_v$ stores text, button, image, input, and icon slots; $\mathcal{O}_v$ stores roles; $\mathcal{P}_v$ stores repeat count, direction, spacing, and item structure; and $\mathcal{I}_v$ stores invalid states such as overlap, overflow, hidden text, and unclickable regions.

We compare candidates by rendered behavior rather than source code. For $s_i, s_j \in \mathcal{C}_v$, we define

$$d_v(s_i, s_j) = \frac{d_{\mathrm{box}} + d_{\mathrm{rel}} + d_{\mathrm{slot}} + d_{\mathrm{role}} + d_{\mathrm{repeat}} + d_{\mathrm{invalid}}}{6}. \tag{5}$$

Here, d_{box} is $1 - \mathrm{meanIoU}$ after Hungarian matching, d_{rel} is the Jaccard distance between relation sets, d_{slot} is the normalized ℓ_1 distance between slot counts, d_{role} is the Jaccard distance between role sets, d_{repeat} compares repeat count and direction, and d_{invalid} compares invalidity counts.

We cluster $\mathcal{C}_v$ with agglomerative clustering, obtaining $\Pi_v = \{\pi_{v,1}, \ldots, \pi_{v,m_v}\}$. Each cluster corresponds to one rendered local behavior. Its probability is $p_{v,j} = |\pi_{v,j}|/|\mathcal{C}_v|$. The representative $r_{v,j}$ is the candidate with the highest full-page score Q in cluster $\pi_{v,j}$. Let $j^\star = \arg\max_j Q(y[v \leftarrow r_{v,j}], x)$ be the best cluster. The repair uncertainty of v is

$$U_v = \sum_{j=1}^{m_v} p_{v,j}\, d_v(r_{v,j^\star}, r_{v,j}). \tag{6}$$

A large U_v indicates multiple competing rendered behaviors for the same DOM hole, while a small U_v indicates that candidates collapse to a similar local implementation.

3.5 Using Uncertainty to Construct Candidate Set

The two uncertainties define the final search space. Attribution uncertainty determines the hole set $\mathcal{H}$, while repair uncertainty determines which local candidates are retained. RSU thus avoids committing to a single DOM location or a single local repair before full-page evaluation.

For each hole v, we keep $\mathcal{R}_v \subseteq \mathcal{C}_v$, with at most $L_c = 3$ candidates by default. We always keep the no-op candidate s_v^0. Among non-no-op clusters, representatives are ranked by $Q(y[v \leftarrow r_{v,j}], x)$, and the top $L_c - 1$ are retained.

The Partial DOM candidate set is

$$\mathcal{S}_{\mathrm{pdom}}(x, y) = \left\{ \mathrm{Compose}\big(y, \{v \leftarrow r_v\}_{v \in \mathcal{H}}\big) \mid r_v \in \mathcal{R}_v \right\}. \tag{7}$$

Here, $\mathrm{Compose}$ simultaneously replaces each selected hole v with a retained candidate r_v, producing a complete HTML/CSS page. The final output is

$$\hat{y} = \arg \max_{\tilde{y} \in \mathcal{S}_{\mathrm{pdom}}(x,y)} Q(\tilde{y}, x). \tag{8}$$

The scoring function Q is structure-oriented. If a candidate page exhibits disappearing main content, large hidden text regions, overly small interactive bounding boxes, large overflow, or severe overlap, we set $Q(\tilde{y}, x) = 0$. Otherwise,

$$Q(\tilde{y}, x) = \frac{V(\tilde{y}, x) + L(\tilde{y}, x) + S(\tilde{y}, x) + R_o(\tilde{y}, x)}{4}. \tag{9}$$

Here, V is visual similarity, L is layout relation F1, S is slot coverage F1, and R_o is role consistency F1. We use equal weights to avoid hand-crafted weighting.

The candidate set operationalizes both uncertainties. Since $\mathcal{H}$ may include multiple plausible DOM holes, RSU avoids premature attribution. Since each $\mathcal{R}_v$ may include multiple rendered-behavior representatives, RSU avoids premature local implementation. Global reranking resolves these alternatives in the full-page context.

The constructed candidate set has three deterministic properties. First, if the render-structure clustering radius is small and Q is stable with respect to the render-structure distance, one representative per cluster incurs only bounded loss in the best achievable score. Second, reranking over $\mathcal{S}_{\mathrm{pdom}}$ weakly dominates independently choosing a local top-1 candidate for each hole and composing them. Third, since each $\mathcal{R}_v$ contains the no-op candidate, the original page $y \in \mathcal{S}_{\mathrm{pdom}}$, and thus $Q(\hat{y}, x) \geq Q(y, x)$. These properties hold only with respect to the internal score Q, and do not guarantee human evaluation or risk calibration.

4 Experiment

4.1 Experimental Setup

Datasets. We evaluate on two benchmarks. **Design2Code** [18] contains 484 real-webpage screenshot–HTML pairs spanning diverse domains and complexity levels, and serves as the established standard for UI-to-Code evaluation. **ScreenBench** [11] extends coverage with 1,000 contemporary websites featuring complex nested layouts built with CSS Grid and Flexbox, providing a more challenging out-of-distribution test bed. For training the structure parser P_θ, we use a separate web-crawl corpus whose training split has no overlap with either test set, following the data-isolation protocol of [18].

Evaluation metrics. We adopt the Design2Code evaluation protocol [18], which compares rendered screenshots against reference screenshots along five dimensions: **Block** (coverage of matched visual element blocks), **Text** (character-level similarity across matched block pairs), **Position** (normalised centre-coordinate proximity between matched elements), **Color** (perceptual text-colour difference between matched blocks), and **CLIP** (high-level visual semantic similarity of rendered screenshots).

Table 1: Comparison of five UI-to-Code pipeline strategies across four MLLM backbones on Design2Code and ScreenBench. All strategies share the same backbone MLLM for every sub-call, ensuring a fair comparison of pipeline design.

		Design2Code					**ScreenBench**				
Backbone	Strategy	Block	Text	Position	Color	CLIP	Block	Text	Position	Color	CLIP
GPT-4o	Direct	0.838	0.955	0.893	0.871	0.908	0.736	0.826	0.715	0.692	0.766
	DCGen	0.846	0.959	0.904	0.876	0.912	0.743	0.833	0.726	0.698	0.771
	UICopilot	0.852	0.962	0.911	0.879	0.915	0.750	0.840	0.736	0.702	0.776
	ScreenCoder	0.857	**0.971**	0.916	0.881	0.918	0.759	0.838	0.746	**0.712**	0.781
	RSU (Ours)	**0.876**	0.966	**0.942**	**0.891**	**0.924**	**0.781**	**0.851**	**0.774**	0.706	**0.789**
Claude-3.5-Sonnet	Direct	0.827	0.948	0.879	0.862	0.901	0.724	0.817	0.706	0.679	0.754
	DCGen	0.835	0.952	0.889	0.866	0.904	0.731	0.822	0.718	**0.691**	0.759
	UICopilot	0.843	0.955	0.899	0.867	0.909	0.739	0.826	0.729	0.684	0.764
	ScreenCoder	0.850	**0.964**	0.911	0.869	0.910	**0.751**	0.831	0.739	0.685	0.768
	RSU (Ours)	**0.866**	0.960	**0.928**	**0.879**	**0.918**	0.744	**0.852**	**0.763**	0.683	**0.779**
Gemini-2.5-Pro	Direct	0.833	0.961	0.891	0.870	0.899	0.732	0.816	0.741	0.685	0.771
	DCGen	0.840	0.964	0.901	0.874	0.903	0.739	0.821	0.751	0.691	0.775
	UICopilot	0.848	0.966	0.910	0.877	0.906	0.747	0.825	0.762	0.694	0.779
	ScreenCoder	0.856	0.969	0.921	0.879	0.909	0.755	**0.834**	0.753	0.696	0.783
	RSU (Ours)	**0.871**	**0.974**	**0.941**	**0.905**	**0.916**	**0.774**	0.828	**0.773**	**0.731**	**0.791**
Qwen2.5-VL	Direct	0.814	0.943	0.806	0.822	0.884	0.714	0.819	0.603	0.622	0.752
	DCGen	0.821	0.946	0.816	0.825	0.887	0.720	0.823	0.614	0.626	0.756
	UICopilot	0.828	0.949	0.827	0.824	0.890	0.727	0.826	0.625	0.628	0.760
	ScreenCoder	0.836	**0.955**	0.843	0.827	0.892	0.735	**0.835**	0.637	0.631	0.762
	RSU (Ours)	**0.852**	0.950	**0.862**	**0.836**	**0.897**	**0.751**	0.830	**0.655**	**0.638**	**0.769**

All screenshots are rendered in a fixed Chromium environment via Playwright (viewport 1440×900, device scale factor 1, animations disabled, fixed random seed), and all base models, RSU, and baselines share the same environment to ensure fair comparison.

Baselines. To isolate the contribution of the RSU *pipeline design* from the choice of underlying model, we hold the backbone fixed and compare five generation strategies applied to the same MLLM. **(1) Direct** prompts the MLLM once with the screenshot to produce a complete HTML page, serving as the single-pass baseline. **(2) DCGen** [20] follows a divide-and-conquer strategy that decomposes the screenshot into sub-regions, generates code for each independently, and assembles the results. **(3) UICopilot** [10] adopts a two-stage hierarchical approach: a structure model first predicts a coarse DOM skeleton, and a code-generation agent then fills each node and applies a global refinement pass. **(4) ScreenCoder** [11] employs a modular multi-agent design in which a grounding agent identifies semantic regions, a planning algorithm constructs a hierarchical layout tree, and a generation agent produces code component by component. We instantiate all strategies on four backbone MLLMs—GPT-4o [17], Claude-3.5-Sonnet [1], Gemini-2.5-Pro [7], and Qwen2.5-VL [2]—and evaluate on both Design2Code and ScreenBench.

RSU configuration. We use the following default hyperparameters: $B = 3$ holes per page, $K = 8$ local candidates per hole (including one no-op candidate that retains the original subtree), $L_c = 3$ cluster representatives retained per hole, agglomerative clustering distance threshold 0.3, and global-fallback coverage threshold $\kappa = 0.6$ (triggered when the selected holes cover fewer than 60% of detected mismatches). All candidate renders are independent and can be executed in parallel; given sufficient parallelism, wall-clock latency is proportional to a single render.

4.2 Main Results

Table 1 presents a comprehensive comparison of five pipeline strategies across four backbone MLLMs on both benchmarks. RSU achieves the best average performance on the majority of metrics, and this advantage holds across every backbone and benchmark combination, highlighting the generalizability and robustness of the approach. Among the baselines, structural fidelity improves progressively as the pipeline becomes more structured: DCGen yields modest gains through divide-and-conquer decomposition; UICopilot achieves a larger boost by scaffolding generation with an explicit DOM skeleton; ScreenCoder attains the strongest baseline performance by decomposing the task into specialised grounding, planning, and generation agents. Despite this progression, all three baselines share a fundamental limitation: they commit to a structural plan before generation begins and have

Table 2: Ablation of hole-proposal components on Design2Code (GPT-4o base).

Variant	Block	Text	Position	Color	CLIP
RSU (full)	**0.876**	**0.966**	**0.942**	**0.891**	**0.924**
w/o G_x (pixel-residual proposal)	0.858	0.963	0.913	0.879	0.916
w/o G_x (token-entropy proposal)	0.860	0.964	0.916	0.880	0.917
w/o G_x (DOM-complexity proposal)	0.856	0.961	0.910	0.878	0.915
w/o DOM anchor (structure match only)	0.849	0.958	0.906	0.883	0.916
LayoutLCA only (no parent/wrapper)	0.866	0.963	0.929	0.888	0.921

Table 3: Ablation of repair-uncertainty components on Design2Code (GPT-4o base).

Variant	Block	Text	Position	Color	CLIP
RSU (full)	**0.876**	**0.966**	**0.942**	**0.891**	**0.924**
no mismatch conditioning	0.850	0.959	0.913	0.885	0.918
no clustering (top-L_c by score)	0.861	0.961	0.923	0.887	0.920
source edit-distance clustering	0.866	0.963	0.928	0.889	0.922
token-probability ranking	0.857	0.960	0.920	0.886	0.919
local top-1 per hole (no Partial DOM)	0.867	0.963	0.930	0.889	0.921
no no-op candidate	0.871	0.964	0.936	0.887	0.922

no mechanism to detect or correct layout errors afterward. RSU addresses this limitation directly by operating as a post-hoc repair layer: it identifies structure-graph mismatches after rendering, localises them to DOM-anchored holes, and selects repairs from a diverse candidate set, producing consistent gains over every pipeline that relies solely on upfront planning.

4.3 Ablation Study

We ablate each RSU component individually on Design2Code (GPT-4o base), keeping all other hyperparameters at their defaults (B=3, K=8, L_c=3).

Structure graph and hole proposal. Table 2 isolates each component of the hole-selection pipeline. Replacing G_x with any of the three heuristic alternatives (pixel-residual, token-entropy, or DOM-complexity) produces a consistent drop across Position and Block, and all three heuristics underperform by a similar margin—indicating that the bottleneck is the absence of structure-graph matching, not the choice among heuristics. Removing DOM anchors while retaining structure matching causes the sharpest overall degradation, since repairs without DOM anchors operate on subtrees that do not correspond to the identified mismatches. Restricting hole candidates to the $\mathrm{LayoutLCA}$ alone, without the nearest layout parent and primary wrapper child, incurs a smaller but reliable drop, confirming that the attribution-uncertainty preservation step is necessary for correctly localising mismatches in complex nested layouts.

Repair uncertainty modeling. Table 3 ablates the Partial DOM candidate set and render-structure clustering. Removing mismatch conditioning yields the largest drop among all repair variants, confirming that generating candidates conditioned on the specific mismatch type is essential for producing diverse and targeted repairs. Disabling render-structure clustering (replacing cluster representatives with a flat score-ranked list) leads to a notable drop in Position and Block, consistent with the need for structurally diverse representatives that cover distinct rendering behaviours. Replacing render-structure distance with source-code edit distance incurs a smaller but consistent drop, since code-level differences do not reliably reflect rendering differences. Replacing global Partial DOM selection with per-hole greedy top-1 selection degrades all metrics, demonstrating that joint reranking captures cross-hole layout dependencies that hole-independent selection misses. Finally, removing the no-op candidate has the smallest effect on average scores yet measurably increases the fraction of pages where RSU regresses the base, validating its role in the non-degradation guarantee.

Hyperparameter sensitivity. Figure 3 sweeps $B \in \{1, 2, 3, 4\}$, $K \in \{4, 8, 12\}$, and $L_c \in \{1, 2, 3, 4\}$, plotting each configuration's Position score against total browser render count alongside a render-budget-matched full-page best-of-K baseline as a reference. RSU configurations consistently and uniformly dominate the baseline across the entire render budget range, with the per-budget best RSU score lying clearly above the corresponding baseline at every render count. Marginal gains

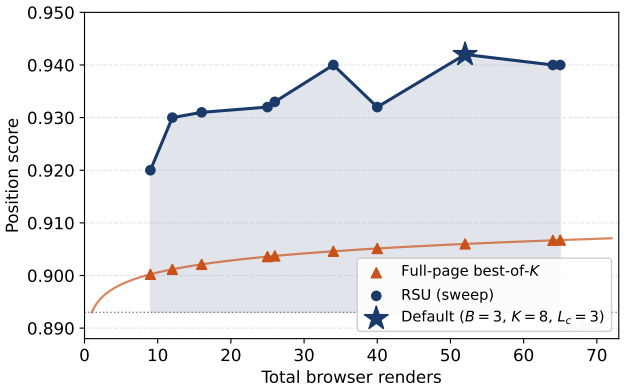


Figure 3: Hyperparameter sensitivity on Design2Code (GPT-4o base). Position score, i.e., the metric most sensitive to structural layout errors, is plotted against total browser render count for all combinations of B, K, and L_c. RSU configurations (blue) uniformly dominate the render-budget-matched full-page best-of-K baseline (orange). The default configuration (B=3, K=8, L_c=3) is marked with a star; gains diminish near and beyond this point.

Table 4: Budget-matched comparison on Design2Code (GPT-4o base). All methods are matched to RSU's budget of 52 browser renders ($1 + BK + L_c^B = 52$ with B=3, K=8, L_c=3).

Method	Render calls	Block	Text	Position	Color	CLIP
Base (Direct)	1	0.838	0.955	0.893	0.871	0.908
Full-page best-of-K	52	0.851	0.959	0.906	0.876	0.913
Full-page self-repair	52	0.849	0.960	0.903	0.875	0.912
Random hole repair	52	0.843	0.956	0.896	0.872	0.909
Pixel-residual hole repair	52	0.858	0.963	0.913	0.879	0.916
Token-entropy hole repair	52	0.860	0.964	0.916	0.880	0.917
DOM-complexity hole repair	52	0.856	0.961	0.910	0.878	0.915
RSU (Ours)	52	**0.876**	**0.966**	**0.942**	**0.891**	**0.924**

diminish as the budget grows beyond the default settings, yet the qualitative ordering between RSU and the baseline remains stable throughout, confirming that the default configuration already provides a strong cost–performance trade-off.

4.4 Budget-Matched Comparison

To verify that RSU's gains are not attributable to a larger compute budget, Table 4 compares RSU against six alternatives matched on the same total number of browser renders, using GPT-4o on Design2Code. RSU outperforms all budget-matched alternatives by a clear margin, confirming that the gains come from structural localisation rather than extra sampling. Full-page best-of-K and self-repair yield only modest improvements over the base, as they repeat complete-page generation without targeting the structural sources of error. Random hole repair provides negligible benefit, demonstrating that location specificity is essential. Structurally-motivated heuristics (pixel-residual, token-entropy, DOM-complexity) yield small but consistent gains, and the remaining gap between these heuristics and RSU isolates the contribution of structure-graph-based localisation.

5 Conclusion

We presented RSU, a browser-grounded post-generation repair framework that lifts UI-to-Code repair into render-structure space and explicitly models two sources of ambiguity: attribution uncertainty over which DOM subtree is responsible for a structural mismatch, and repair uncertainty over which local implementation best resolves it. RSU proposes candidate DOM holes via structure-graph matching, clusters local repair candidates by rendered behavior, and performs global reranking over a Partial DOM candidate set. Experiments across four backbone MLLMs on two benchmarks confirm

consistent improvements over structured pipelines, with budget-matched comparisons and ablations validating that the gains come from structural localisation rather than additional compute.

Acknowledgement

This work is supported in part by the National Natural Science Foundation of China (No. 62476110, No. U2341229), and the Key R&D Project of Jilin Province (No. 20240304200SF).

References

[1] Anthropic. Claude 3.5 sonnet. `https://www.anthropic.com/news/claude-3-5-sonnet`, 2024.

[2] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. Qwen2.5-vl technical report, 2025. URL `https://arxiv.org/abs/2502.13923`.

[3] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI symposium on engineering interactive computing systems*, pages 1–6, 2018.

[4] 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*, pages 665–676, 2018.

[5] 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.

[6] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023.

[7] Google. Gemini API. `https://ai.google.dev/gemini-api`, 2024.

[8] Yi Gui, Zhen Li, Yao Wan, Yemin Shi, Hongyu Zhang, Bohua Chen, Yi Su, Dongping Chen, Siyuan Wu, Xing Zhou, et al. Webcode2m: A real-world dataset for code generation from webpage designs. In *Proceedings of the ACM on Web Conference 2025*, pages 1834–1845, 2025.

[9] 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. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 721–732, 2025.

[10] 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 on Web Conference 2025*, pages 1846–1855, 2025.

[11] 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.

[12] Hugo Laurençon, Léo Tronchon, and Victor Sanh. Unlocking the conversion of web screenshots into html code with the websight dataset. *arXiv preprint arXiv:2403.09029*, 2024.

[13] Kenton Lee, Mandar Joshi, Iulia Raluca Turc, Hexiang Hu, Fangyu Liu, Julian Martin Eisenschlos, Urvashi Khandelwal, Peter Shaw, Ming-Wei Chang, and Kristina Toutanova. Pix2struct: Screenshot parsing as pretraining for visual language understanding. In *International Conference on Machine Learning*, pages 18893–18912. PMLR, 2023.

[14] Ryan Li, Yanzhe Zhang, and Diyi Yang. Sketch2code: Evaluating vision-language models for interactive web design prototyping. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3921–3955, 2025.

[15] Shanchao Liang, Nan Jiang, Shangshu Qian, and Lin Tan. Waffle: Fine-tuning multi-modal model for automated front-end development. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 24786–24802, 2025.

[16] Zimu Lu, Houxing Ren, Yunqiao Yang, Ke Wang, Zhuofan Zong, Junting Pan, Mingjie Zhan, and Hongsheng Li. Webgen-agent: Enhancing interactive website generation with multi-level feedback and step-level reinforcement learning. *arXiv preprint arXiv:2509.22644*, 2025.

[17] OpenAI. Hello GPT-4o. `https://openai.com/index/hello-gpt-4o/`, 2024.

[18] Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2code: Benchmarking multimodal code generation for automated front-end engineering. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3956–3974, 2025.

[19] Yuxuan Wan, Yi Dong, Jingyu Xiao, Yintong Huo, Wenxuan Wang, and Michael R Lyu. Mrweb: An exploration of generating multi-page resource-aware web code from ui designs. *arXiv preprint arXiv:2412.15310*, 2024.

[20] Yuxuan Wan, Chaozheng Wang, Yi Dong, Wenxuan Wang, Shuqing Li, Yintong Huo, and Michael Lyu. Divide-and-conquer: Generating ui code from screenshots. *Proceedings of the ACM on Software Engineering*, 2(FSE):2099–2122, 2025.

[21] Chengyue Wu, Zhixuan Liang, Yixiao Ge, Qiushan Guo, Zeyu Lu, Jiahao Wang, Ying Shan, and Ping Luo. Plot2code: A comprehensive benchmark for evaluating multi-modal large language models in code generation from scientific plots. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3006–3028, 2025.

[22] Fan Wu, Cuiyun Gao, Shuqing Li, Xin-Cheng Wen, and Qing Liao. Mllm-based ui2code automation guided by ui layout information. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1123–1145, 2025.

[23] Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P Bigham. Screen parsing: Towards reverse engineering of ui models from screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 470–483, 2021.

[24] Jason Wu, Eldon Schoop, Alan Leung, Titus Barik, Jeffrey P Bigham, and Jeffrey Nichols. Uicoder: Finetuning large language models to generate user interface code through automated feedback. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 7511–7525, 2024.

[25] Jie Wu, Haoran Ma, Shisong Tang, Yulin Xu, Xiaoyu Kang, and Jiechao Gao. MulF-Coder: Framework-conditioned multi-agent for MLLM-based multi-framework front-end code generation. In *Forty-third International Conference on Machine Learning*, 2026. URL `https://openreview.net/forum?id=xUW3aH4ie3`.

[26] Jingyu Xiao, Yuxuan Wan, Yintong Huo, Zixin Wang, Xinyi Xu, Wenxuan Wang, Zhiyao Xu, Yuhang Wang, and Michael R Lyu. Interaction2code: Benchmarking mllm-based interactive webpage code generation from interactive prototyping. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 241–253. IEEE, 2025.

[27] 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.

[28] Jingyu Xiao, Zhongyi Zhang, Yuxuan Wan, Yintong Huo, Yang Liu, and Michael R Lyu. Efficientuicoder: Efficient mllm-based ui code generation via input and output token compression. *arXiv preprint arXiv:2509.12159*, 2025.

[29] Jingyu Xiao, Jiantong Qin, Shuoqi Li, Man Ho Lam, Yuxuan Wan, Jen-tse Huang, Yintong Huo, and Michael R Lyu. Comuicoder: Component-based reusable ui code generation for complex websites via semantic segmentation and element-wise feedback. *arXiv preprint arXiv:2602.19276*, 2026.

[30] Cheng Yang, Chufan Shi, Yaxin Liu, Bo Shui, Junjie Wang, Mohan Jing, Linran Xu, Xinyu Zhu, Siheng Li, Yuxiang Zhang, et al. Chartmimic: Evaluating lmm's cross-modal reasoning capability via chart-to-code generation. *arXiv preprint arXiv:2406.09961*, 2024.

[31] Zhen Yang, Wenyi Hong, Mingde Xu, Xinyue Fan, Weihan Wang, Jiele Cheng, Xiaotao Gu, and Jie Tang. Ui2codeˆ n: A visual language model for test-time scalable interactive ui-to-code generation. *arXiv preprint arXiv:2511.08195*, 2025.

[32] Sukmin Yun, Haokun Lin, Rusiru Thushara, Mohammad Q 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. *Advances in neural information processing systems*, 37:112134–112157, 2024.

[33] Xuanle Zhao, Deyang Jiang, Zhixiong Zeng, Lei Chen, Haibo Qiu, Jing Huang, Yufeng Zhong, Liming Zheng, Yilin Cao, and Lin Ma. Vincicoder: Unifying multimodal code generation via coarse-to-fine visual reinforcement learning. *arXiv preprint arXiv:2511.00391*, 2025.

[34] Xuanle Zhao, Xianzhen Luo, Qi Shi, Chi Chen, Shuo Wang, Zhiyuan Liu, and Maosong Sun. Chartcoder: Advancing multimodal large language model for chart-to-code generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7333–7348, 2025.

[35] Ting Zhou, Yanjie Zhao, Xinyi Hou, Xiaoyu Sun, Kai Chen, and Haoyu Wang. Declarui: Bridging design and development with automated declarative ui code generation. *Proceedings of the ACM on Software Engineering*, 2(FSE):219–241, 2025.

A RSU Gain Across Backbones

Figure 4 visualises the absolute Position improvement of RSU over the Direct baseline for each backbone on both benchmarks. The gains are consistent across all four backbones on Design2Code, and remain positive on ScreenBench, confirming that RSU's benefit is not specific to any single model family or test distribution.

B RSU Gain Heatmap Across All Metrics

Figure 5 extends the per-backbone analysis to all five evaluation metrics on both benchmarks. Position consistently shows the largest absolute gain across all backbones and both datasets, reflecting RSU's targeted focus on structural layout correction. Block gains are also substantial, while Text gains are more modest—consistent with RSU operating on layout-level DOM holes rather than text content. Color and CLIP gains are smaller but positive in most settings, indicating that layout improvements propagate to perceptual similarity.

C Budget-Matched Comparison

Figure 6 presents the full five-metric breakdown for all budget-matched methods under equal browser render budget. RSU achieves the highest score on every metric, with the most pronounced advantage

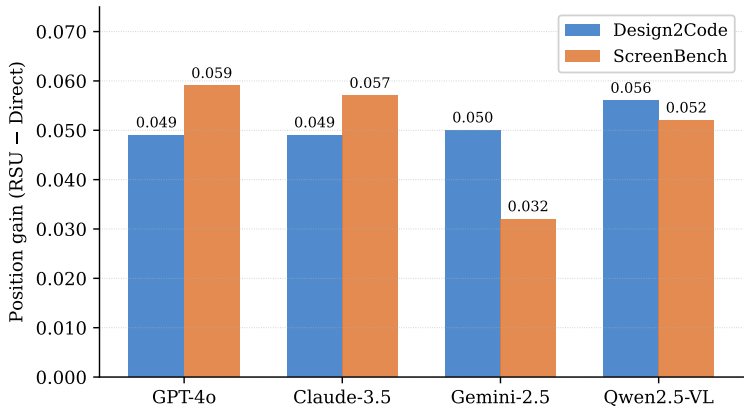


Figure 4: Absolute Position gain (RSU − Direct) per backbone on Design2Code and ScreenBench. Gains are consistent across all four backbones and both benchmarks.

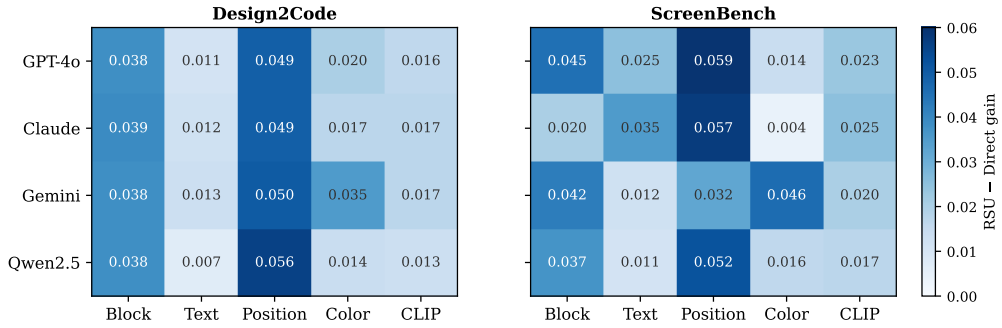


Figure 5: RSU − Direct gain on all five metrics across four backbones on Design2Code (left) and ScreenBench (right). Position and Block show the largest gains; Text gains are smallest, consistent with RSU targeting layout rather than content.

on Position, which is the metric most sensitive to structural layout errors. Among the heuristic baselines, token-entropy and pixel-residual hole repair offer comparable gains but still fall short of RSU on Position and Block, confirming that structure-graph-based localisation provides a qualitatively different signal than pixel- or token-level saliency.

D Repair Uncertainty Score Analysis

Figure 7 plots the repair uncertainty score U_v against the per-hole Position improvement ΔPosition across all selected holes on the Design2Code test set (GPT-4o base). Holes with higher U_v tend to yield larger Position gains, confirming that the uncertainty score correlates with repair difficulty and the potential benefit of uncertainty-guided search. Holes with low U_v, where candidates collapse to a single rendered behavior, contribute negligible gain; in these cases the no-op candidate is typically selected, consistent with the non-degradation guarantee.

E Limitations and Impacts

Limitations RSU relies on the quality of the target-side visual structure parser and the generated-side DOM-anchored render graph, so errors in region detection, repeated-group parsing, layout-direction prediction, or DOM anchoring can lead to incorrect candidate holes. Its attribution module is a high-recall proposal mechanism rather than a causal localization guarantee, since real HTML/CSS layouts may depend on global styles, media queries, inherited variables, external assets, and parent-container constraints. The final result is also bounded by the quality of the local candidate generator

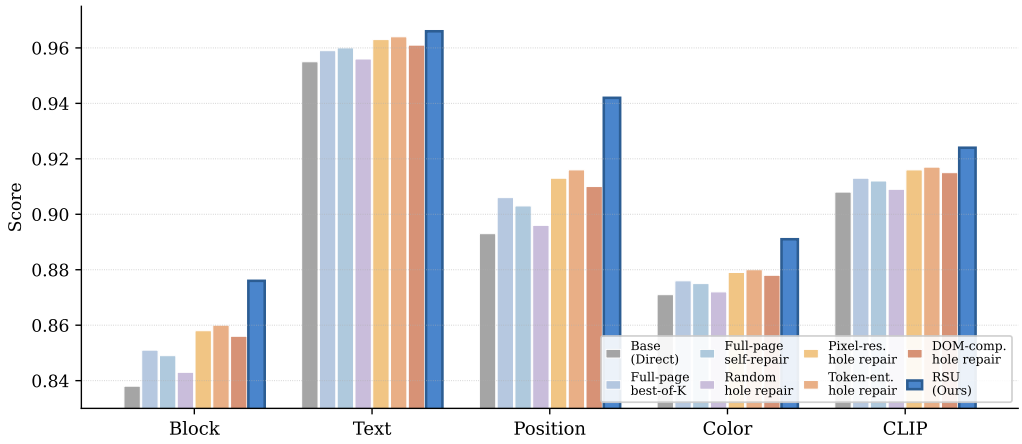


Figure 6: Full metric breakdown for budget-matched methods on Design2Code (GPT-4o base). All methods use 52 browser renders. RSU (dark blue) leads on all five metrics, with the largest margin on Position.

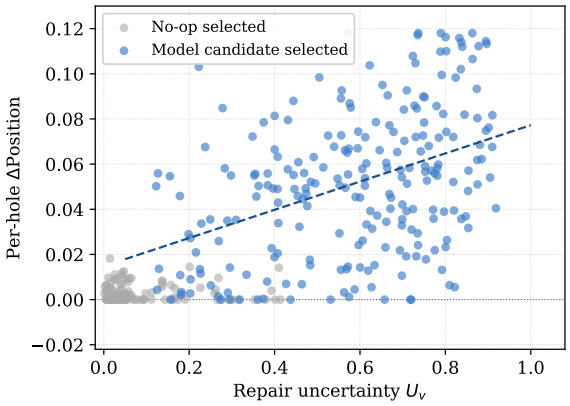


Figure 7: Repair uncertainty U_v vs. per-hole ΔPosition on Design2Code (GPT-4o base). Each point is one selected DOM hole. Gray points indicate holes where the no-op candidate was selected (low U_v, near-zero gain); blue points indicate holes where a model candidate was selected, with gain increasing with U_v.

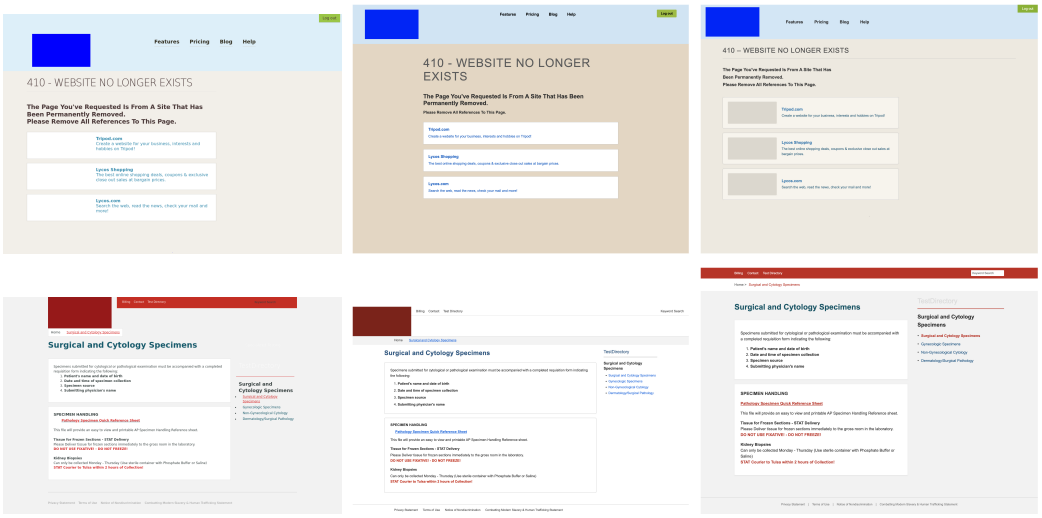
and the page-level scoring function (Q); if no candidate contains a valid repair, or if (Q) is misaligned with human preference, rendering quality may not improve. In addition, the current method does not include a calibration stage, so the Partial DOM candidate set should be viewed as a compact search space rather than a risk-controlled prediction set. Finally, RSU introduces extra browser-rendering and candidate-generation cost, and the current formulation mainly targets static single-viewport pages, leaving multi-state interactions, responsive breakpoints, and strongly global CSS failures as important extensions.

Broader Impacts RSU can improve the reliability and debuggability of UI-to-Code systems by exposing structural mismatches, candidate DOM holes, and alternative rendered implementations, making generated HTML/CSS easier for developers to inspect and refine. This can reduce repetitive frontend work and help build more useful design-to-code assistants, especially when used as a human-in-the-loop repair layer rather than a fully autonomous deployment tool. At the same time, stronger screenshot-to-code repair may facilitate misuse such as cloning proprietary webpages, producing deceptive interfaces, or generating phishing pages with higher visual fidelity. Because RSU executes generated code in a browser, practical deployments should use sandboxing, restrict network access, and audit external resources. Moreover, visual and layout similarity alone may overlook accessibility

issues, so deployed systems should incorporate checks for semantic HTML, keyboard navigation, ARIA validity, label associations, and color contrast before generated interfaces are released.

F Case Study

Figure 8 presents two case studies on Design2Code pages. For each case we show three images: the target screenshot, the **Direct** generation (GPT-4o), and the **RSU-repaired** output.



Target | Direct (GPT-4o) | RSU-Repaired

Figure 8: Qualitative case studies on Design2Code. *Left*: target webpage screenshot. *Middle*: Direct GPT-4o generation. *Right*: RSU output after repair.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction reflect the paper's contributions and scope.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification:Limitations are discussed in Appendix E.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: The paper does not include formal theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Implementation details are given in Section 4.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We have provided codes in abstract section.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: See Section 4.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: The main results are statistical values.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.

- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: See Section 4.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conforms to the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Broader impacts are discussed in Appendix E.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: No high-risk assets are released.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All used public benchmarks and models are properly cited in the paper.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: New assets introduced in the paper are well documented and the documentation is provided alongside the assets.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve human-subject research.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: MLLM backbones (GPT-4o, Claude-3.5-Sonnet, Gemini-2.5-Pro, Qwen2.5-VL) are core components: they serve both as the base UI-to-Code generators and as the local candidate proposers within RSU. Their usage is described in Sections 4.1.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.