# ITERUI: LEARNING DOM-GROUNDED LOCAL VISUAL CODE EDITING FOR UI-TO-CODE

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Vision-language models (VLMs) can generate executable webpages from UI screenshots, yet correcting their remaining visual discrepancies remains challenging. Existing full-page refinement treats each correction as another page-generation step, leaving the correspondence between a local visual error and its responsible code implicit and potentially disturbing already-correct regions. We instead formulate UI refinement as *local visual code editing*, where each action pairs a grounded DOM component with an executable code update whose effect is evaluated after full-page rendering. Based on this formulation, we propose **IterUI**, a closed-loop framework that detects discrepancies between a reference screenshot and the current rendering, grounds them to DOM components, and generates targeted patches from localized visual evidence, component code, and ancestor layout context. IterUI masks non-target pixels while preserving the original image canvas, retaining the position and scale information needed for visual correction. Each patch is reintegrated into the complete page and accepted only after render-based verification. Because the framework requires only editable source code, browser renders, and the rendered DOM, it can also refine outputs from existing VLMs without updating their parameters. We further train **IterUI-Coder-8B** on refinement trajectories and optimize its code-generating actions using **Group-Scored Relative Policy Optimization (GSRPO)**, which jointly evaluates groups of rendered candidates to construct comparative visual rewards. Across four UI-to-code benchmarks, using a test-time judge distinct from the training reward model, IterUI-Coder-8B achieves the highest VLM scores among the evaluated open-source models. On UIHTML, IterUI further raises its VLM score from 66.80 to 69.12.

## 1 INTRODUCTION

Recent vision-language models (VLMs) have made substantial progress in generating HTML code from UI design mockups (Beltramelli, 2018; Bhathal and Gupta, 2025; Si et al., 2025; Jiang et al., 2025). Given a reference screenshot, these models can often produce a complete webpage with a plausible layout and component structure (Xu et al., 2025; Zhang et al., 2026). However, the rendered result may still differ from the reference in spacing, typography, alignment, colors, and element proportions (Si et al., 2025; Deng et al., 2026; Zheng et al., 2026). Correcting these residual errors is essential for high-fidelity UI generation, but requires a model to connect what looks wrong in the rendered page with where and how the underlying code should be changed.

The central difficulty is *translating visual discrepancies into effective code edits*. A model may recognize a mismatch in the rendered page while still needing to determine which code controls it and what update would correct it. Existing refinement approaches commonly request an entirely revised page (Yang et al., 2025; Deng et al., 2026), leaving this visual-to-code correspondence implicit within a long implementation. As illustrated in Figure 1, full-page revisions can leave local errors unresolved and introduce unintended changes elsewhere (Xiong et al., 2026). Moreover, the visually mismatched element is not always the correct edit target: fixing a button's alignment may require modifying its parent container's layout rules. Effective refinement therefore calls for an action that connects the visual discrepancy to an explicit code target and update.

Therefore, we formulate visual refinement for UI-to-code as *local visual code editing*. Instead of predicting a complete revised page, each refinement action couples a grounded DOM component with an



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

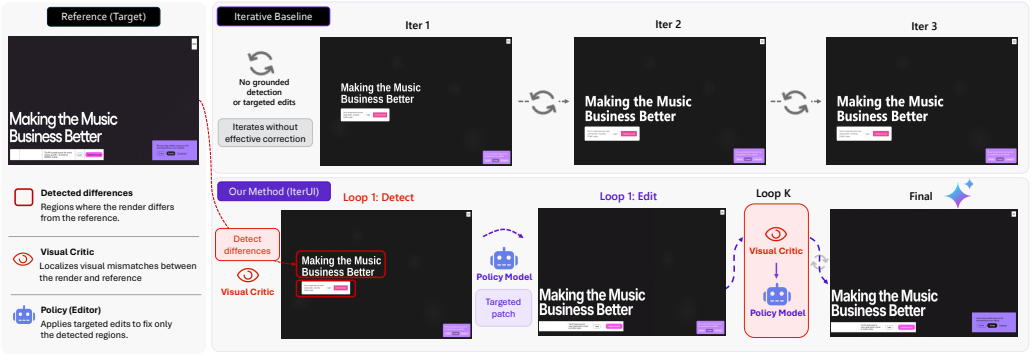



Figure 1: Comparison of global and localized visual refinement. In the illustrated example, full-page revisions leave local discrepancies unresolved. IterUI identifies mismatched regions, grounds them to DOM components, and applies targeted code patches with visual verification.

executable code update, whose effect is evaluated after reintegration into the full page. We instantiate this formulation with **IterUI**, a closed loop of discrepancy detection, DOM grounding, localized component editing, and render-based verification. IterUI identifies mismatched regions between the reference and current render, maps them to explicit DOM targets, and generates component patches using localized visual evidence, component code, and ancestor layout context. Coordinate-preserving masks suppress unrelated visual content without removing position and scale information. Each patch is then rendered within the complete page and retained only when it improves the result. Because IterUI requires only editable source code, browser renders, and the rendered DOM, it can also refine outputs from existing VLMs without updating their parameters.

To specialize a model for this formulation, we construct refinement trajectories that supervise initial generation, discrepancy detection, component editing, and visual verification, yielding **IterUI-Coder-8B**. We further optimize its code-generating actions with **Group-Scored Relative Policy Optimization (GSRPO)**. GSRPO jointly evaluates groups of integrated and rendered candidates against the same reference and converts their scores into group-relative learning signals, providing comparative visual rewards with fewer evaluator calls than independent scoring.

We evaluate IterUI on four UI-to-code benchmarks against one-shot generation and global iterative revision. Experiments with multiple editing models show that IterUI achieves higher VLM scores than global refinement in evaluated settings. With **IterUI-Coder-8B**, it obtains the highest VLM scores among the evaluated open-source models across all four benchmarks. On UIHTML, applying IterUI improves the model's VLM score from 66.80 to 69.12, compared with 43.58 for Qwen3-VL-32B-Instruct. Further analyses examine discrepancy detection, DOM grounding, region masking, and visual reward design to understand how these components contribute to localized refinement.

Our main contributions are summarized as follows:

- We formulate UI refinement as DOM-grounded local visual code editing, changing the refinement action from full-page regeneration to an explicit component update evaluated through rendering. This formulation makes the visual-to-code correspondence explicit by coupling each localized discrepancy with a concrete edit target and executable update.

- We propose IterUI, which realizes this formulation through localized discrepancy detection, DOM grounding, context-aware component editing, and full-page visual verification, while remaining applicable to existing VLMs.

- We train IterUI-Coder-8B from refinement trajectories and introduce group-scored rendered rewards for relative policy optimization, achieving the strongest VLM scores among the evaluated open-source models across four benchmarks.

# 2 Related Work

## 2.1 Vision-Language Models

Recent advancements in Vision-Language Models (VLMs) have fundamentally transformed multimodal reasoning and zero-shot task execution. Early architectures seamlessly bridged vision encoders with large language models to enable visual instruction tuning Liu et al. (2023). As the demand for analyzing dense and complex images grew, subsequent research pivoted toward fine-grained visual perception. Models introduced explicit spatial grounding mechanisms Peng et al. (2023) and dynamic-resolution processing Wang et al. (2024; 2025), empowering them to accurately localize and interpret dense elements within rich contexts like UI screenshots. Building upon these foundational techniques, frontier proprietary models, such as the Gemini series Comanici et al. (2025), have demonstrated remarkable generalist capabilities, including zero-shot code generation from visual inputs. However, while these general-purpose VLMs possess strong perceptual priors, mapping subtle, continuous visual nuances into highly structured, hierarchical code representations remains an open challenge. Our work leverages the spatial grounding capabilities of modern VLMs but re-purposes them within a specialized critic-policy architecture to explicitly bridge this modality gap.

## 2.2 UI-to-Code Generation Models

UI-to-code has evolved from heuristic reverse engineering and classical computer vision pipelines Beltramelli (2018); Nguyen and Csallner (2015) to deep learning and large-scale datasets such as WebSight Bhathal and Gupta (2025) and Design2Code Si et al. (2025), which frame the task as direct one-shot screenshot-to-HTML generation. To address the limited visual fidelity of one-shot generation, recent work explores iterative and structured paradigms. UI2Code$^N$ Yang et al. (2025) combines multi-turn interactive refinement with reinforcement learning. VisRefiner (Deng et al., 2026) learns difference-aligned code refinement and self-refinement from rendered visual differences; IterUI complements it by making the edit target explicit at inference, grounding each discrepancy to a rendered DOM component and applying a component-scoped patch. At submission time, VisRefiner provides neither a public implementation nor model checkpoint, preventing re-evaluation under our unified protocol. ScreenCoder Jiang et al. (2025) and other layout-aware systems Wan et al. (2025); Gui et al. (2025) improve structural fidelity by decomposing screenshots and generating code modularly, substantially improving macro-structural accuracy. However, existing interactive pipelines still operate mainly at the document level through global rewrites conditioned on textual feedback. IterUI instead uses a strictly localized paradigm, restricting generation to bounding-box-guided component snippets to reduce cascading layout reflows while optimizing pixel-level visual alignment.

# 3 Methodology

## 3.1 Preliminary Study: Making Visual-to-Code Correspondence Explicit

Before presenting our formulation, we first examine whether explicitly connecting a visual discrepancy to its editable code scope improves repair. We use 171 single-edit examples collected from real editing trajectories, each pairing a clean reference with a singly modified implementation. Four configurations progressively expose the correspondence: **A** provides full screenshots and HTML; **B** additionally specifies *where to look* through the oracle region; **C** specifies *where to edit* by adding the corresponding code fragment; and **D** requests *a local update* to that fragment. All four configurations retain the full screenshots and full HTML. We report full-page and local DINO score gains, target-region repair, and off-target damage; Appendix C.1 provides the complete metric definitions.

**Explicit code grounding connects visual errors to edit targets.** Table 1 shows that visual localization alone is insufficient. Adding the oracle region improves Qwen3.8-27B's repair rate from 94.2% to 97.1% but degrades GLM-5.3-Flash from 95.9% to 94.7%. In contrast, providing the code fragment improves both editors, raising repair rates to 98.8% and 97.7% while reducing off-target damage. Thus, localization becomes actionable when tied to an explicit editable code scope.

**Local patching provides a compact grounded repair action.** With the code target explicit, C–D replaces full-page output with a component patch. Repair rates rise modestly to 99.4% and

Table 1: Preliminary study with progressively more explicit visual-to-code grounding.

| Input | Output | Page score Δ ↑ | Local score Δ ↑ | Target Repair ↑ | Off-target Damage ↓ |
|---|---|---|---|---|---|
| *Qwen3.8-27B* | | | | | |
| A: Full images + HTML | Full HTML | +4.38 | +19.94 | 94.2% | 23.4% |
| B: A + Oracle Region | Full HTML | +4.81 | +21.29 | 97.1% | 18.7% |
| C: B + Code Fragment | Full HTML | +4.85 | +21.58 | 98.8% | 17.5% |
| D: Same as C | Code patch | **+4.87** | **+21.66** | **99.4%** | **17.0%** |
| *GLM-5.3-Flash* | | | | | |
| A: Full images + HTML | Full HTML | +4.42 | +19.77 | 95.9% | 22.8% |
| B: A + Oracle Region | Full HTML | +3.55 | +19.13 | 94.7% | 34.5% |
| C: B + Code Fragment | Full HTML | +4.71 | **+21.07** | 97.7% | **20.5%** |
| D: Same as C | Code patch | **+4.76** | +21.02 | **98.2%** | **20.5%** |

98.2%, while off-target damage decreases slightly for Qwen3.8-27B and remains unchanged for GLM-5.3-Flash. These smaller gains suggest that most benefit comes from explicit visual-to-code correspondence, while local patching compactly represents the grounded action. This motivates coupling each localized objective with an explicit DOM target and executable component update.

### 3.2 Problem Formulation: Local Visual Code Editing

We formulate visual refinement for UI-to-code as *local visual code editing*. Rather than predicting a complete revised page, each refinement action specifies an explicit DOM target and an executable update to its component code; the action is evaluated through the rendered change produced after full-page integration. This formulation makes the correspondence between a localized visual error and its editable code scope explicit.

Given a reference screenshot $I_{\text{ref}}$, a current implementation $H_t$, and its rendering $I_t = \mathcal{R}(H_t)$, we define a local editing action as:

$$a_t = (v_t, \hat{c}_t), \tag{1}$$

where $v_t$ is the DOM component grounded from a localized visual discrepancy and $\hat{c}_t$ is a proposed update to the component code. Applying the action produces a candidate implementation

$$\widetilde{H}_{t+1} = \mathcal{U}(H_t, v_t, \hat{c}_t). \tag{2}$$

where $\mathcal{U}$ replaces the code associated with $v_t$ while leaving the remainder of the source unchanged. Source-code locality does not guarantee visual locality: an edit can still trigger layout reflow elsewhere. The candidate must therefore be rendered and verified before it becomes $H_{t+1}$. Each refinement step consequently aligns three scopes: the image region containing the discrepancy, the DOM component that may be edited, and the full-page rendering in which the edit is evaluated.

### 3.3 IterUI framework

Given an initial implementation $H_0 = \mathcal{G}(I_{\text{ref}})$ from a screenshot-to-code generator $\mathcal{G}$, IterUI realizes local visual code editing through discrepancy detection, DOM grounding, localized component editing, and render-based verification. Figure 2 summarizes the refinement loop and training pipeline.

**Discrepancy Detection.** Given the current render $I_t = \mathcal{R}(H_t)$, a visual critic $M_{\text{critic}}$ compares it with the reference $I_{\text{ref}}$ and returns up to $K$ localized discrepancies:

$$\mathcal{D}_t = M_{\text{critic}}(I_{\text{ref}}, I_t) = \left\{ \left( b_{\text{ref}}^{(k)}, b_t^{(k)}, f_t^{(k)} \right) \right\}_{k=1}^{K}, \tag{3}$$

where $b_{\text{ref}}^{(k)}$ and $b_t^{(k)}$ denote the corresponding regions in the reference and current renders, and $f_t^{(k)}$ describes their visual discrepancy. Separate boxes allow for component displacement or resizing between the two renders.

**DOM Grounding.** Let $\mathcal{V}_t$ be the set of visible nodes in the rendered DOM tree and $B_t(v)$ the browser-computed bounding box of node $v$. IterUI grounds a render-side discrepancy to the node

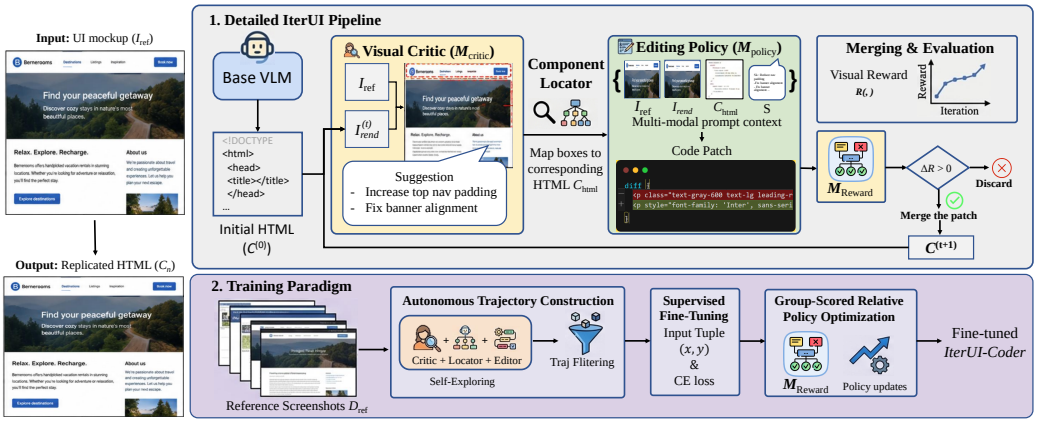



Figure 2: The IterUI paradigm. The pipeline achieves high-fidelity UI alignment via vision-driven, component-level code patching, while the model is trained through an SFT and GSRPO pipeline.

with the highest intersection over union:

$$v_t^{(k)} = \arg\max_{v \in \mathcal{V}_t} \mathrm{IoU}\left(b_t^{(k)}, B_t(v)\right). \tag{4}$$

The selected node defines the source-code boundary that may be replaced.

**Localized Component Editing.** Cropping a target region changes its coordinate system and can obscure the displacement or scale error to be corrected, whereas complete screenshots introduce unrelated visual content. IterUI instead retains the original canvas and masks non-target pixels:

$$\Phi(I, b)_{x,y} = \begin{cases} I_{x,y}, & (x, y) \in b, \\ \kappa, & \text{otherwise}, \end{cases} \tag{5}$$

where $\kappa$ is a uniform masking color. Unlike cropping, $\Phi(I, b)$ preserves the target region's absolute position, scale, and surrounding whitespace. We apply the operation separately to the reference and current renders so that displacement and scale mismatches remain visible.

Because masking removes the appearance of surrounding elements, the editor also receives structural context. Let $c_t^{(k)}$ denote the code associated with $v_t^{(k)}$, and let $X_t^{(k)}$ contain the geometry and layout-relevant computed styles of its ancestor chain, including flex/grid configuration, positioning, alignment, and gaps. The localized editing state and component update are

$$\begin{aligned} x_t^{(k)} &= \left(\Phi(I_{\mathrm{ref}}, b_{\mathrm{ref}}^{(k)}), \Phi(I_t, b_t^{(k)}), c_t^{(k)}, X_t^{(k)}, f_t^{(k)}\right), \\ \hat{c}_t^{(k)} &\sim \mathcal{M}_{\mathrm{editor}}(\cdot \mid x_t^{(k)}). \end{aligned} \tag{6}$$

The masked images retain spatial evidence, while the component and ancestor context expose the layout constraints needed to generate an executable update.

**Render-Based Verification.** A local code update may induce non-local layout changes, so IterUI verifies each candidate after full-page reintegration:

$$\widetilde{H}_{t+1}^{(k)} = \mathcal{U}\left(H_t, v_t^{(k)}, \hat{c}_t^{(k)}\right), \qquad \widetilde{I}_{t+1}^{(k)} = \mathcal{R}\left(\widetilde{H}_{t+1}^{(k)}\right). \tag{7}$$

A verifier compares the candidate and current renders against the reference and determines whether to accept the edit:

$$\begin{aligned} \Delta_t^{(k)} &= \mathcal{M}_{\mathrm{verifier}}\left(I_{\mathrm{ref}}, I_t, \widetilde{I}_{t+1}^{(k)}, b_{\mathrm{ref}}^{(k)}, b_t^{(k)}, f_t^{(k)}\right), \\ H_{t+1} &= \begin{cases} \widetilde{H}_{t+1}^{(k)}, & \Delta_t^{(k)} > 0, \\ H_t, & \text{otherwise}. \end{cases} \end{aligned} \tag{8}$$

Refinement stops when the page-level score reaches a threshold or the editing budget is exhausted.

IterUI requires only editable source code, browser renders, and the rendered DOM, enabling refinement without access to the base generator's parameters. It can therefore operate as a plug-in for existing VLMs or as a specialized editor trained from IterUI trajectories.

### 3.4 Learning the Local Visual Code Editor

The local visual code-editing formulation defines a conditional repair task: given a localized discrepancy, its grounded DOM component, and layout context, predict an executable update.

**Trajectory Construction.** To bootstrap this capability without manual trajectory annotation, we use Gemini-3-Flash to execute the IterUI loop and retain 2,000 successful refinement trajectories from Common Crawl screenshots, with no overlap with evaluation datasets. Each trajectory provides demonstrations for initial generation, discrepancy detection, component editing, and visual verification. For component editing, we render and evaluate multiple candidate updates and retain the highest-scoring accepted candidate:

$$r^* = \arg\max_{r:a_r=1} R_r, \tag{9}$$

where $a_r \in \{0, 1\}$ indicates acceptance and $R_r$ is the verifier-assigned visual score. This yields executable, verifier-improving supervision targets. We initialize IterUI-Coder-8B with supervised fine-tuning on demonstrations from all four roles; training details are provided in Appendix A.2.

### 3.5 Group-Scored Relative Policy Optimization

Supervised training does not directly optimize rendered visual quality (Zhang et al., 2018; Song et al., 2025). Pixel- or feature-based rewards may miss fine-grained visual differences (Chen et al., 2024), while independently scoring $N$ candidates with a VLM requires $N$ calls and produces absolute scores that may be weakly calibrated across states. We therefore introduce **Group-Scored Relative Policy Optimization (GSRPO)**, which jointly evaluates rendered candidates for the same state and converts their scores into group-relative learning signals.

**Rendered Candidate Groups.** For each training state, the policy produces $N$ candidate code actions. Initial-generation candidates are rendered directly, whereas component-editing candidates are reintegrated into the original page before full-page rendering. The reward therefore evaluates the executed visual consequence of each action rather than the code in isolation.

**Group-scored Visual Rewards.** We construct a joint multimodal prompt containing the reference screenshot and all candidate renders:

$$\mathbf{s} = f_{\mathrm{VLM}}(I_{\mathrm{ref}}, I_1, \ldots, I_N)\,. \tag{10}$$

Here, $\mathbf{s} = [s_1, \ldots, s_N]^\top$ contains the visual scores for candidates generated from the same state. Joint scoring uses one evaluator call per group rather than $N$ independent calls, although its visual context grows with $N$. We adopt group scoring primarily for its higher-contrast comparative signal and lower evaluation cost; alternative reward and scoring designs are analyzed in Appendix D. We convert these scores into group-relative advantages:

$$\widehat{A}_i = \frac{s_i - \mu(\mathbf{s})}{\sigma(\mathbf{s}) + \epsilon}. \tag{11}$$

Here, $\mu(\mathbf{s})$ and $\sigma(\mathbf{s})$ denote the mean and standard deviation of the group scores. We optimize these advantages with the token-normalized GRPO objective and asymmetric clipping used in DAPO:

$$\mathcal{J}_{\mathrm{GSRPO}}(\theta) = \mathbb{E}_{x,\{y_i\}\sim\pi_{\theta_{\mathrm{old}}}}\left[\frac{1}{\sum_i |y_i|}\sum_i\sum_j \min\Big(\rho_{i,j}(\theta)\widehat{A}_i,\ \mathrm{clip}(\rho_{i,j}(\theta), 1-\epsilon_{\mathrm{low}}, 1+\epsilon_{\mathrm{high}})\,\widehat{A}_i\Big)\right]. \tag{12}$$

Here, $\rho_{i,j}(\theta) = \pi_\theta(y_{i,j} \mid x, y_{i,<j})/\pi_{\theta_{\mathrm{old}}}(y_{i,j} \mid x, y_{i,<j})$. The update otherwise follows standard GRPO (Shao et al., 2024) with asymmetric clipping from DAPO (Yu et al., 2026). Full optimization details are provided in Appendix A.3, while the protocol analysis and scoring prompts appear in Appendices D and G, respectively. GSRPO's task-specific contribution remains the executed, jointly scored reward construction rather than a new policy optimizer.

# 4 EXPERIMENTS

## 4.1 EXPERIMENTAL SETUP

**Datasets.** We evaluate on four UI-to-code benchmarks: Design2Code (Si et al., 2025), Flame-React (Ge et al., 2025), Web2Code (Yun et al., 2024), and UIHTML. We use up to 500 examples per benchmark, yielding 484, 80, 500, and 500 examples, respectively, and average results over three runs. The benchmarks range from standard templates to complex webpages; our custom-curated UIHTML uses public webpage screenshots from Common Crawl to capture dense layouts and diverse styles.

**Baselines.** We compare IterUI-Coder against several strong baselines with different paradigms and model sizes: (1) **Open-Source VLMs**: Models that use one-shot generation without iterative refinement. We test Qwen3-VL-32B-Instruct (Team, 2025), Qwen3-VL-8B-Instruct (Team, 2025), and Qwen3.5-9B (Qwen Team, 2026a). (2) **Open-Source UI-to-Code Systems**: We include UI2Code$^N$ (Yang et al., 2025), which performs test-time interactive refinement, and Screen-Coder (Jiang et al., 2025), a modular screenshot-to-code pipeline evaluated with the GLM-5.3-Flash (GLM-5-Team et al., 2026) backend. (3) **Closed-Source Commercial APIs**: Leading proprietary models with strong generalist capabilities. We evaluate GPT-5.6-Terra, Kimi-K2.5 (Team et al., 2026), Gemini-3-Flash, Qwen3.8-Max (Qwen Team, 2026b), and GLM-5V-Turbo.

**Implementation Details.** We run IterUI for at most $T = 3$ page-level refinement rounds, selecting up to five localized regions per round and rewriting each selected component once; Appendix C.6 reports an ablation over iteration budgets. Candidate code is sampled at temperature $1.0$ with a 13,000-token output limit. GPT-5.2 provides VLM rewards for reinforcement learning, while Claude Sonnet 5 serves as the test-time VLM judge. We report DINOv3 Score and VLM Score; DINOv3 Score is 100 times the cosine similarity between global features of the reference and exact-size browser rendering, extracted with pretrained DINOv3 ConvNeXt-Tiny (Siméoni et al., 2025). Claude Sonnet 5 achieves 76.4% pairwise agreement with human preference and an Elo Pearson correlation of 0.9837 (Appendix B.2). Evaluation parameters are provided in Appendix B, judge prompts in Appendix G, and additional training and inference details in Appendix A.

## 4.2 OVERALL UI-TO-CODE PERFORMANCE

Table 2 establishes the overall performance of the trained model and its refinement framework. IterUI-Coder-8B obtains the highest VLM scores among the evaluated open-source models across all four datasets. Applying IterUI further improves both metrics on every dataset: the UIHTML VLM score rises from 66.80 to 69.12. These results show that the refinement loop can improve pages produced by an already strong task-specific generator. See Appendix F for qualitative examples.

The two metrics provide complementary assessments of rendered visual fidelity: DINOv3 measures global image-feature similarity, while the VLM judge evaluates visual correspondence. On Design2Code, IterUI-Coder-8B improves over Qwen3-VL-32B-Instruct on both metrics, increasing DINOv3 similarity from 91.58 to 94.35 and the VLM score from 58.98 to 66.94. These results support improved visual fidelity under both evaluation measures. To contextualize these comparisons, a 10,000-repeat sample-bootstrap analysis gives typical 95% confidence margins of approximately ±0.62 VLM points and ±0.58 DINOv3 points.

## 4.3 EXECUTING LOCAL EDITING TASKS IN A REFINEMENT LOOP

On the same 400 Qwen3-VL-32B Direct outputs, we evaluate full-page refinement, plug-in editing, and human-aligned verification over at most three rounds. *Iterative* rewrites the full page; *IterUI (All Roles)* uses the evaluated model for diagnosis, localization, patch generation, and acceptance; and *IterUI (Model as Editor)* replaces only patch generation model while retaining other components.

Starting from the same 400 pages, Table 3 shows that: **(1) Local refinement is more effective and token-efficient than full-page revision.** IterUI (All Roles) improves over the iterative baseline by 4.00–5.39 VLM points across the three editors while reducing token usage by roughly 28–34%, despite requiring more calls. **(2) IterUI supports effective plug-in refinement with external editors.** Replacing only the component editor yields positive gains over the shared Direct starting point (+1.23

Table 2: Results on four UI-to-code datasets. Gray rows mark our methods; the light green row marks Qwen3-VL-8B-Instruct, the base model used to train IterUI-Coder-8B.

| Method | Design2Code | | Flame-React | | Web2Code | | UIHTML | |
|---|---|---|---|---|---|---|---|---|
| | VLM | DINOv3 | VLM | DINOv3 | VLM | DINOv3 | VLM | DINOv3 |
| *Direct Generation* | | | | | | | | |
| Qwen3-VL-8B | 53.05 | 90.22 | 49.27 | 85.48 | 52.60 | 89.71 | 37.52 | 78.50 |
| Qwen3-VL-32B | 58.98 | 91.58 | 57.91 | 89.43 | 59.09 | 90.05 | 43.58 | 82.37 |
| Qwen3.5-9B | 52.25 | 87.98 | 51.35 | 87.53 | 48.60 | 86.94 | 39.63 | 79.13 |
| GLM-5V-Turbo | 67.64 | 91.14 | 58.69 | 86.41 | 58.77 | 89.27 | 60.53 | 85.85 |
| Qwen3.8-Max | 64.72 | 84.59 | 74.80 | 92.64 | 68.76 | 93.23 | 66.28 | 84.50 |
| Gemini-3-Flash | 67.72 | 92.64 | 66.90 | 89.32 | 61.98 | 90.07 | 67.78 | 87.02 |
| IterUI-Coder-8B | 66.94 | 94.35 | 65.53 | 92.80 | 65.10 | 94.22 | 66.80 | 88.55 |
| Kimi-K2.5 | 77.02 | 93.81 | 74.40 | **93.14** | 74.28 | 93.76 | 65.29 | 86.92 |
| GPT-5.6-Terra | **80.61** | **94.81** | **79.79** | 92.28 | **80.07** | **96.60** | **75.57** | **91.55** |
| *Multi-stage Generation* | | | | | | | | |
| ScreenCoder (2025) | 40.42 | 82.96 | 18.04 | 79.27 | 42.76 | 86.73 | 35.33 | 74.98 |
| UI2Code$^N$ (2025) | 57.21 | 87.80 | 64.76 | 89.21 | 62.79 | 86.66 | 56.75 | 82.90 |
| IterUI-Coder-8B + IterUI | 68.90 | 94.58 | 66.41 | 92.97 | 67.13 | 94.58 | 69.12 | 89.13 |

Table 3: Executing refinement tasks from the same pages (initial score 59.92).

| Target model | Method | Final ↑ | $\Delta_{\text{Direct}}$ ↑ | Tokens/page | Calls/page |
|---|---|---|---|---|---|
| Qwen3-VL-8B | Iterative Full-Page | 55.85 | -4.07 | 5.6K | 4.00 |
| | IterUI (All Roles) | 59.85 | -0.07 | 3.7K | 11.92 |
| | IterUI (Model as Editor) | 61.15 | +1.23 | 4.4K | 10.94 |
| GLM-5.3-Flash | Iterative Full-Page | 59.64 | -0.28 | 35.8K | 4.00 |
| | IterUI (All Roles) | 64.81 | +4.89 | 25.9K | 6.56 |
| | IterUI (Model as Editor) | 63.67 | +3.75 | 56.0K | 11.44 |
| IterUI-Coder-8B | Iterative Full-Page | 58.32 | -1.60 | 7.2K | 4.00 |
| | IterUI (All Roles) | 63.71 | +3.79 | 5.0K | 12.80 |

and +3.75) and outperforms full-page iterative revision in both cases, while keeping IterUI's feedback, DOM grounding, and acceptance modules fixed. Performance relative to the all-roles setting varies by editor, showing that the framework provides a reusable component-editing interface rather than depending on a single editor configuration.

### 4.4 Stable Improvements Validated by Human Preference

The refinement loop retains a candidate only after reintegrating its patch into the full page and evaluating the rendered result. We compare this page-level acceptance decision with human visual preferences on 1,000 full-page UIHTML pairs. The IterUI-Coder verifier reaches 76.0% agreement with human decisions, close to Claude Sonnet 5 at 76.4% and above GPT-5.2 and Qwen3.5-35B at 74.3% and 66.2%, respectively. Its false-accept and false-reject rates are 26.1% and 21.8%. This alignment supports the stability of the improvement decisions used by the refinement loop: retained updates are selected by a verifier whose page-level judgments agree closely with human visual preference.

### 4.5 Automatically Grounding Visual Feedback to Executable Edits

The preliminary study in Section 3.1 uses oracle visual targets to isolate explicit visual-to-code correspondence; in the complete IterUI loop, these targets must be constructed automatically from the current rendering. We evaluate this transition on the same 171 single-edit cases, measuring discrepancy-box coverage with IoGT (intersection over ground-truth-box area) and DOM grounding

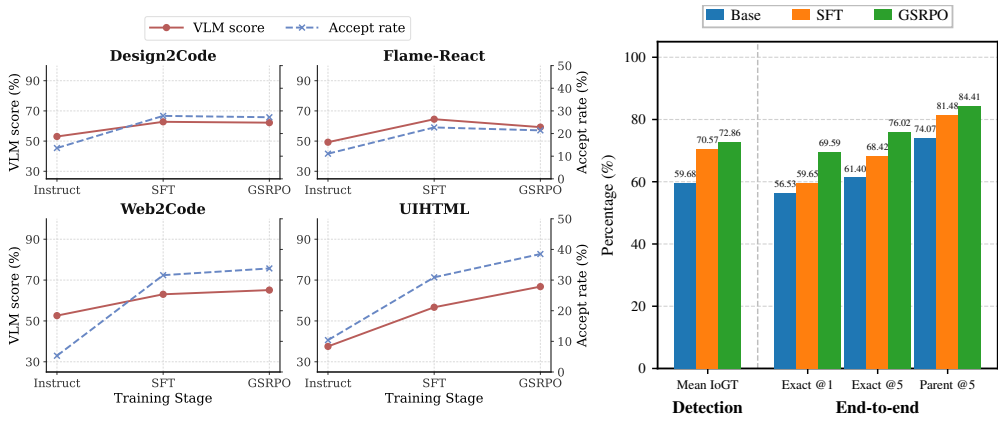



Figure 3: Training-stage performance (left) and discrepancy localization and DOM-grounding accuracy (right).

with Exact@1, Exact@5, and Exact-or-parent@5, which additionally accepts the modified node's immediate parent. Figure 3 shows that the final model achieves 72.86% mean IoGT, with 73.68% of predictions covering at least half of the causal DOM-node box. Highest-IoU grounding remains above 91% exact-node accuracy under 20% shifts or rescaling, while the end-to-end detection-to-grounding chain reaches 84.41% Exact-or-parent@5. **Thus, the oracle correspondence motivating the method can be approximated automatically with substantial accuracy and robustness.** Further analyses of localization, grounding, and masking are provided in Appendices C.2–C.5.

### 4.6 Effect of Supervised and Reinforcement Learning

We train IterUI-Coder-8B from trajectories and further optimize the code-generating actions with rendered visual rewards through Group-Scored Relative Policy Optimization (GSRPO). Figure 3 shows the effect of supervised fine-tuning and reinforcement learning. SFT substantially improves the base model, showing that trajectory supervision effectively teaches the local editing workflow. Starting from the SFT checkpoint, GSRPO brings further gains by directly optimizing code-generating actions for their rendered visual outcomes, indicating that **outcome-based optimization complements trajectory supervision.** Interestingly, Figure 3 shows that although GSRPO directly optimizes only code-generating actions, it also improves grounding accuracy, suggesting that rendered-outcome optimization can further strengthen the capabilities supporting localized editing.

To provide this rendered-outcome signal efficiently, we jointly score eight candidates in a single call to obtain within-state comparative feedback. **Compared with independent scoring, group scoring increases the normalized advantage range from 1.08 to 1.34 while reducing estimated evaluation tokens by ∼45%.** See Appendix D for full analysis.

## 5 Conclusion

We formulate visual refinement for UI-to-code as *local visual code editing*, where each refinement action couples a grounded DOM component with an executable code update and is evaluated through its full-page rendered effect. Based on this formulation, we propose IterUI, a closed-loop framework that connects discrepancy detection, DOM grounding, localized component editing, and render-based verification. IterUI restricts each refinement action to an explicit code scope while evaluating its potentially non-local effects after full-page rendering. We further train IterUI-Coder-8B on refinement trajectories to learn localized visual feedback and component-level repair. In addition, we propose Group-Scored Relative Policy Optimization (GSRPO), which jointly evaluates groups of rendered code candidates and converts their scores into comparative learning signals for visual policy optimization. Experiments across four UI-to-code benchmarks show that IterUI improves multiple base VLMs over full-page iterative refinement, while IterUI-Coder-8B achieves the strongest VLM scores among the open-source models evaluated under our protocol. These results demonstrate the effectiveness of DOM-grounded local code editing for high-fidelity UI generation and refinement.

## AI USE STATEMENT

Generative AI models were used as components of our research pipeline. Specifically, Gemini-3-Flash was used to construct refinement trajectories, and GPT-5.2 was used to provide visual reward signals for reinforcement learning. Generative AI tools were also used for language polishing and improving the clarity of the manuscript. All experimental results, analyses, and conclusions were reviewed and verified by the authors.

## ETHICS STATEMENT

Our human evaluation involved three professional frontend developers who assessed the visual fidelity of generated webpages. Human judgments were used solely for evaluating model outputs and validating automated evaluation metrics.

## REPRODUCIBILITY STATEMENT

Section 3 specifies the IterUI pipeline, the local editing and verification procedure, and the GSRPO objective. Section 4 describes the evaluated benchmarks, baselines, metrics, and test-time inference settings. Appendix A documents the provenance, construction, task format, and splits of the SFT and RL data, while Appendices A.2 and A.3 provide the corresponding optimization configurations. Appendix B defines the browser-rendering and automated evaluation protocol, including repeated evaluation and the separation between the training reward model and the test-time judge. The exact prompts for generation, discrepancy localization, component editing, verification, group scoring, and final evaluation are provided in Appendices G and G. Finally, Appendix B.2 reports the human validation of the automated judge, and Appendices C.6 and D document the iteration-budget and reward-scoring protocol analyses relevant to reproducing our design choices.

## REFERENCES

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.

Tanvir Bhathal and Asanshay Gupta. Websight: A vision-first architecture for robust web agents, 2025.

Dongping Chen, Ruoxi Chen, Shilin Zhang, Yinuo Liu, Yaochen Wang, Huichi Zhou, Qihui Zhang, Yao Wan, Pan Zhou, and Lichao Sun. Mllm-as-a-judge: Assessing multimodal llm-as-a-judge with vision-language benchmark. *arXiv preprint arXiv:2402.04788*, 2024.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025.

Jie Deng, Kaichun Yao, and Libo Zhang. Visrefiner: Learning from visual differences for screenshot-to-code generation, 2026.

Tong Ge, Yashu Liu, Jieping Ye, Tianyi Li, and Chao Wang. Advancing vision-language models in front-end development via data synthesis, 2025.

GLM-5-Team, :, Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, Chenzheng Zhu, Congfeng Yin, Cunxiang Wang, Gengzheng Pan, Hao Zeng, Haoke Zhang, Haoran Wang, Huilong Chen, Jiajie Zhang, Jian Jiao, Jiaqi Guo, Jingsen Wang, Jingzhao Du, Jinzhu Wu, Kedong Wang, Lei Li, Lin Fan, Lucen Zhong, Mingdao Liu, Mingming Zhao, Pengfan Du, Qian Dong, Rui Lu, Shuang-Li, Shulin Cao, Song Liu, Ting Jiang, Xiaodong Chen, Xiaohan Zhang, Xuancheng Huang, Xuezhen Dong, Yabo Xu, Yao Wei, Yifan An, Yilin Niu, Yitong Zhu, Yuanhao Wen, Yukuo Cen, Yushi Bai, Zhongpei Qiao, Zihan Wang, Zikang Wang, Zilin Zhu, Ziqiang Liu, Zixuan Li, Bojie Wang,

Bosi Wen, Can Huang, et al. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

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*, KDD '25, pages 721–732, 2025.

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, 2025.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *Advances in neural information processing systems*, 36:34892–34916, 2023.

Tuan Anh Nguyen and Christoph Csallner. Reverse engineering mobile application user interfaces with remaui (t). In *2015 30th IEEE/ACM international conference on automated software engineering (ASE)*, pages 248–259. IEEE, 2015.

Zhiliang Peng, Wenhui Wang, Li Dong, Yaru Hao, Shaohan Huang, Shuming Ma, and Furu Wei. Kosmos-2: Grounding multimodal large language models to the world. *arXiv preprint arXiv:2306.14824*, 2023.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026a. URL https://qwen.ai/blog?id=qwen3.5.

Qwen Team. Qwen3.8-max: A new bar for coding and cowork, August 2026b. URL https://qwen.ai/blog?id=qwen3.8.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2code: Benchmarking multimodal code generation for automated front-end engineering, 2025.

Oriane Siméoni, Huy V Vo, Maximilian Seitzer, Federico Baldassarre, Maxime Oquab, Cijo Jose, Vasil Khalidov, Marc Szafraniec, Seungeun Yi, Michaël Ramamonjisoa, et al. Dinov3, 2025.

Yiren Song, Xiaokang Liu, and Mike Zheng Shou. Diffsim: Taming diffusion models for evaluating visual similarity. In *2025 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 16904–16915. IEEE, 2025.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, S. H. Cai, Yuan Cao, Ziwei Chai, Y. Charles, H. S. Che, Cheng Chen, Guanduo Chen, Huarong Chen, Jia Chen, Jianlong Chen, Jun Chen, Kefan Chen, Liang Chen, Ruijue Chen, Xinhao Chen, Yanru Chen, Yanxu Chen, Yicun Chen, Yimin Chen, Yingjiang Chen, Yuankun Chen, Yujie Chen, Yutian Chen, Zhirong Chen, Ziwei Chen, Dazhi Cheng, Yean Cheng, Minghan Chu, Jialei Cui, Jiaqi Deng, Muxi Diao, Hao Ding, Mengfan Dong, Mengnan Dong, Yuxin Dong, Yuhao Dong, Angang Du, Chenzhuang Du, Dikang Du, Lingxiao Du, Yulun Du, Yu Fan, Shengjun Fang, Qiulin Feng, Yichen Feng, Garimugai Fu, Kelin Fu, Hongcheng Gao, Tong Gao, Yuyao Ge, Shangyi Geng, Chengyang Gong, Xiaochen Gong, Zhuoma Gongque, Qizheng Gu, Xinran Gu, Yicheng Gu, Longyu Guan, Shuhao Guan, Yuanying Guo, Xiaoru Hao, Dailan He, Tianhong He, Weiran He, Wenyang He, Yibo He, Yunjia He, Chao Hong, Hao Hu, Jiaxi Hu, Yangyang Hu, Zhenxing Hu, Ke Huang, Ruiyuan Huang, Weixiao Huang, Zhiqi Huang, Chaobo Jia, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yu Jing, Guokun Lai, Aidi Li, C. Li, Cheng Li, Fang Li, Guanghe Li, Guanyu Li, Haitao Li, Haoyang Li, Jia Li, Jingwei Li, Junxiong Li, Lincan Li, Mo Li, Weihong Li, et al. Kimi k2.5: Visual agentic intelligence, 2026. URL https://arxiv.org/abs/2602.02276.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

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.

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.

Weiyun Wang, Zhangwei Gao, Lixin Gu, Hengjun Pu, Long Cui, Xingguang Wei, Zhaoyang Liu, Linglin Jing, Shenglong Ye, Jie Shao, et al. Internvl3. 5: Advancing open-source multimodal models in versatility, reasoning, and efficiency, 2025.

Tianyi Xiong, Zhengyuan Yang, Xiaofei Wang, Chung-Ching Lin, Ruichun Ma, Kevin Lin, Zhendong Wang, Linjie Li, Chenxi Liu, Ruibo Chen, et al. Rubrics as visual-repair context for self-evolving ui-to-code generation. *arXiv preprint arXiv:2608.24138*, 2026.

Mingde Xu, Zhen Yang, Wenyi Hong, Lihang Pan, Xinyue Fan, Yan Wang, Xiaotao Gu, Bin Xu, and Jie Tang. Webvia: A web-based vision-language agentic framework for interactive and verifiable ui-to-code generation. *arXiv preprint arXiv:2511.06251*, 2025.

Zhen Yang, Wenyi Hong, Mingde Xu, Xinyue Fan, Weihan Wang, Jiale Cheng, Xiaotao Gu, and Jie Tang. Ui2code$\Theta n$: A visual language model for test-time scalable interactive ui-to-code generation, 2025.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale, 2026.

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, 2024.

Houston H Zhang, Tao Zhang, Baoze Lin, Yuanqi Xue, Yincheng Zhu, Huan Liu, Li Gu, Linfeng Ye, Ziqiang Wang, Xinxin Zuo, et al. Widget2code: From visual widgets to ui code via multimodal llms. pages 20293–20302, 2026.

Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *2018 IEEE/CVF conference on computer vision and pattern recognition*, pages 586–595. IEEE, 2018.

Yaozhi Zheng, Yilei Jiang, Manyuan Zhang, Yuxuan Wan, Kaituo Feng, Tianshuo Peng, Bo Zhang, and Xiangyu Yue. Unicoder: Unified visual-to-code generation via symbolic rewards and reference-guided code optimization. *arXiv preprint arXiv:2606.31732*, 2026.

# A Data and Training Details

## A.1 Training Data and Task Format

The training data follows the operational decomposition of the IterUI framework. In the supervised fine-tuning stage, we train the model on demonstrations from all major stages of the iterative UI refinement loop: initial HTML generation, discrepancy detection, localized component editing, and visual verification. Each example is serialized in a multimodal ShareGPT-style format, consisting of message turns and one or more associated images.

The SFT corpus contains 20,160 multimodal examples derived from the 2,000 successful IterUI refinement trajectories generated by Gemini-3-Flash, as described in Section 3. For component refinement demonstrations, candidate edits are integrated into the page, rendered, and verified; when multiple candidates are available for the same localized region, we retain the highest-scoring accepted candidate.

The RL corpus is restricted to the two stages that directly synthesize code: initial HTML generation and component refinement. The corpus contains 6,067 examples: 3,378 for initial HTML generation and 2,689 for component refinement. Training uses 5,460 examples, and periodic validation evaluates 100 examples.

## A.2 Supervised Fine-Tuning

We initialize IterUI-Coder from Qwen3-VL-8B-Instruct and perform supervised fine-tuning on the multimodal IterUI demonstrations. Training runs for 4 epochs with a global batch size of 16 and a maximum sequence length of 22,000 tokens. We use a learning rate of $5 \times 10^{-6}$, a warmup ratio of 0.01, and a weight decay of 0.1. Sequence parallelism is set to 2. In total, we train for 7,000 steps, and the resulting checkpoint is used to initialize the subsequent RL stage.

## A.3 Group-Scored Reinforcement Learning

After supervised fine-tuning, we further optimize the model with Group-Scored Relative Policy Optimization (GSRPO). The RL stage focuses exclusively on initial HTML generation and component-level refinement. For each prompt, the policy samples a group of candidate outputs. Initial-generation candidates are rendered directly, whereas component edits are integrated into the original page before full-page rendering. GPT-5.2 jointly scores the resulting candidate renders against the same reference to construct the group-relative rewards described in Section 3. We initialize RL from the SFT checkpoint and train for 1 epoch on 4 GPUs with a global batch size of 16. For each prompt, we sample 8 rollouts with a temperature of 1.0. The maximum prompt length is 12,000 tokens, and the rollout model length is capped at 20,000 tokens. The actor learning rate is $1 \times 10^{-6}$. We use asymmetric clipping with lower and upper clip ratios of 0.2 and 0.3, respectively. In the final run, both the KL reward coefficient and the actor KL loss coefficient are set to 0.0, and the entropy coefficient is also set to 0.0.

# B Evaluation Protocol and Human Validation

## B.1 Automated Evaluation Protocol

We evaluate IterUI-Coder on four UI-to-code benchmarks: Design2Code, Flame-React, Web2Code, and UIHTML. We cap each benchmark at 500 examples and use the complete benchmark when fewer examples are available, resulting in 484 Design2Code, 80 Flame-React, 500 Web2Code, and 500 UIHTML examples (1,564 total). For each example, the generated HTML is rendered in a browser at the exact target resolution and compared against the reference screenshot. We report both automated DINOv3 visual similarity and VLM-based visual scores. DINOv3 Score is 100 times the cosine similarity between global image features extracted by pretrained DINOv3 ConvNeXt-Tiny. Each evaluation is repeated three times and the resulting scores are averaged. The test-time VLM judge is Claude Sonnet 5, distinct from GPT-5.2 used for RL rewards. It uses the strict full-page evaluation prompt provided in Appendix G, which requires a score in the range $[0, 100]$ together with a detailed explanation of the visual differences.

### B.2 HUMAN–VLM JUDGE CONSISTENCY

To evaluate the reliability of our automated metrics, we sampled 1,000 comparison pairs from UIHTML generation outputs, with ten pairs per reference image. Three independent professional frontend developers annotated each pair for visual fidelity, and majority preference served as the human reference. We compared these labels with Claude Sonnet 5, GPT-5.2, Qwen3.5-35B, DINOv3-based rules, and the IterUI acceptance logic. We additionally compare DINOv3 pairwise decisions directly against GPT-5.2 and Claude Sonnet 5 on the same pairs. Every row in the unified audit has 1,000 valid comparisons.

To place pairwise judges and acceptance policies on the same decision scale, we treat a reference preference for Rendering B as a positive decision (accept B) and a preference for Rendering A as a negative decision (reject B). Reference ties are mapped to rejection. Predictor ties are mapped to rejection for strict rules and to acceptance for non-decrease rules, including the IterUI-Coder verifier setting reported in the main text. Table 4 reports the tested predictor or rule and its actual reference separately, followed by accuracy, precision, recall, false-accept rate (FAR), false-reject rate (FRR), and binary Cohen's kappa. FAR and FRR correspond to FPR and FNR, respectively, in Table 6.

Table 4: Unified pairwise consistency and human audit on 1,000 full-page UIHTML pairs. The Reference column identifies the actual comparison target. Rendering B is the positive class.

| Predictor / Rule | Reference | Accuracy | Precision | Recall | FAR | FRR | $\kappa$ |
|---|---|---|---|---|---|---|---|
| *Pairwise judges* | | | | | | | |
| Claude Sonnet 5 | Human | **76.4%** | 76.4% | 74.5% | 21.8% | 25.5% | **0.5274** |
| GPT-5.2 | Human | 74.3% | 72.2% | 76.8% | 28.1% | 23.2% | 0.4865 |
| Qwen3.5-35B | Human | 66.2% | 66.4% | 62.0% | 29.8% | 38.0% | 0.3224 |
| *DINOv3 pairwise rule* | | | | | | | |
| DINOv3 | Human | 67.3% | 65.5% | 69.4% | 34.7% | 30.6% | 0.3465 |
| DINOv3 | GPT-5.2 | 68.2% | 69.4% | 69.1% | 32.8% | 30.9% | 0.3633 |
| DINOv3 | Claude Sonnet 5 | 65.7% | 62.8% | 68.2% | 36.6% | 31.8% | 0.3151 |
| *Page-level acceptance rules* | | | | | | | |
| IterUI-Coder verifier | Human | 76.0% | 74.0% | **78.2%** | 26.1% | **21.8%** | 0.5204 |
| DINOv3 | Human | 67.9% | 66.1% | 70.0% | 34.1% | 30.0% | 0.3585 |

From 1,000 aligned per-pair records, the original three-label DINOv3–GPT-5.2 statistics are 68.2% exact agreement, $\kappa = 0.3639$, discrete Pearson $r = 0.3644$, and DINOv3-margin Pearson $r = 0.3588$; the corresponding DINOv3–Claude Sonnet 5 statistics are 65.6%, 0.3151, 0.3175, and 0.3395. The table maps reference ties to rejection, yielding binary accuracies of 68.2% and 65.7%, respectively.

Among the generic pairwise judges, Claude Sonnet 5 has the highest accuracy at 76.4%, followed by GPT-5.2 at 74.3%. We separately aggregate pairwise preferences into model-level Elo ratings: the Elo Pearson correlations with human-derived ratings are 0.9837 for Claude Sonnet 5, 0.9760 for GPT-5.2, 0.9036 for Qwen3.5-35B. These model-level correlations are not mixed into Table 4 because they summarize seven model-level observations rather than 1,000 pair-level binary decisions.

Table 5 reports the model-level Elo scores, win rates, and rankings for humans and GPT-5.2. Their Elo rankings have Spearman correlation 0.9643, distinct from the Elo Pearson correlation reported above. IterUI-Coder and Kimi-K2.5 occupy the top two positions under both evaluations, while GPT-5.2 reverses the human ordering of Gemini-3-Flash and IterUI-Coder-SFT. This supports aggregate ranking consistency for GPT-5.2 on the evaluated models, rather than exact agreement on every model ordering.

### B.3 HUMAN AUDIT OF THE ITERUI ACCEPTANCE GATE

We further audit the actual page-level acceptance logic using the same 1,000 human-annotated full-page pairs. For each pair, we treat Rendering A as the current page (before) and Rendering B as the proposed candidate (after). A human preference for B is therefore a positive decision (the candidate should be accepted), whereas a preference for A is a negative decision (the candidate should be rejected). The candidate is deterministically placed in Candidate A or Candidate B with near-balanced frequency (519/481) before inference, so the verifier cannot infer the candidate identity from its

Table 5: Comparison of model rankings, Elo scores, and win rates (WR) evaluated by experienced frontend developers versus the GPT-5.2 automated judge.

| Model | Human Rank | VLM Rank | Human Elo | VLM Elo | Human WR | VLM WR |
|---|---|---|---|---|---|---|
| IterUI-Coder | 1 | 1 | 1687.1 | 1730.3 | 76.1% | 79.2% |
| kimi-k2.5 | 2 | 2 | 1661.9 | 1657.5 | 72.2% | 69.4% |
| gemini-3-flash | 3 | 4 | 1532.8 | 1564.5 | 55.3% | 58.1% |
| IterUI-Coder-SFT | 4 | 3 | 1500.8 | 1579.5 | 50.3% | 60.1% |
| glm-5v-turbo | 5 | 5 | 1457.7 | 1469.5 | 45.2% | 45.5% |
| Qwen3-VL-32B-Instruct | 6 | 6 | 1403.9 | 1342.9 | 37.7% | 31.3% |
| Qwen3-VL-8B-Instruct | 7 | 7 | 1255.7 | 1155.8 | 18.5% | 11.0% |

display position. The human labels contain 487 positive and 513 negative decisions; IterUI-Coder returns a valid A/B/TIE verdict on all 1,000 pairs. We report the false-accept rate (FAR) among human-negative pairs and the false-reject rate (FRR) among human-positive pairs.

The learned verifier provides the highest accuracy among the acceptance rules: under non-decrease semantics, its accuracy is 76.0% (Wilson 95% CI: 73.3%–78.5%), with FAR 26.1% (22.5%–30.1%) and FRR 21.8% (18.3%–25.6%). The deployed dual gate reduces FAR to 12.7% and raises precision to 80.6%, but its recall falls to 55.4% and FRR rises to 44.6%..

### B.4 Comparison of Page-Level Verifiers

Because a local code edit can alter the full-page layout, the verifier must reject updates that appear locally plausible but degrade the rendered page. We validate this decision against human preferences on the same 1,000 full-page UIHTML pairs used to evaluate three general-purpose judges. We treat Rendering B as the candidate update and acceptance as the positive class. Besides agreement accuracy, we report the false-positive rate (FPR), where a human-rejected candidate is incorrectly accepted, and the false-negative rate (FNR), where a human-preferred candidate is incorrectly rejected. These correspond to the false-accept rate (FAR) and false-reject rate (FRR), respectively, in Appendix B.2.

Table 6: Verifier agreement and error rates against human preferences on 1,000 full-page UIHTML pairs. The general-purpose judges use the strict rule from Table 4.

| Verifier / Rule | Accuracy ↑ | FPR ↓ | FNR ↓ |
|---|---|---|---|
| Claude Sonnet 5 | **76.4%** | **21.8%** | 25.5% |
| GPT-5.2 | 74.3% | 28.1% | 23.2% |
| Qwen3.5-35B | 66.2% | 29.8% | 38.0% |
| IterUI-Coder verifier | 76.0% | 26.1% | **21.8%** |

Table 6 shows that the IterUI-Coder verifier reaches 76.0% human-agreement accuracy, within 0.4 points of Claude Sonnet 5 and above GPT-5.2 and Qwen3.5-35B by 1.7 and 9.8 points. For the key safety failure of accepting an incorrect edit, IterUI-Coder has a 26.1% FPR, lower than GPT-5.2 (28.1%) and Qwen3.5-35B (29.8%), but higher than Claude Sonnet 5 (21.8%). Its 21.8% FNR is the lowest of the four, showing that it does not obtain its false-positive performance by simply rejecting more candidates. The deployed dual gate prioritizes safety more aggressively: it lowers FPR to 12.7%, while increasing FNR to 44.6%. Thus, the learned verifier provides a stronger accuracy–retention balance, whereas the dual gate is preferable when preventing harmful commits is the primary objective. Because these are generic full-page comparison pairs rather than localized component-repair candidates, the results validate page-level acceptance behavior but do not directly measure component-repair error rates. Full results are provided in Appendix B.2.

# C Supplementary Analyses of Local Visual Editing

## C.1 Effect of Localization and Local Edit Constraints

This analysis uses the same 171 single-edit cases introduced in Section 3.1 and evaluated for discrepancy detection in Appendix C.2. Each example provides a clean reference, a corrupted implementation with one recorded modification, and a candidate rendering produced by the editor.

**Input and output configurations.** Table 7 summarizes the progressively added inputs and output constraint. All four configurations retain full screenshots and full HTML. A–B tests the addition of bbox-masked visual evidence, B–C tests explicit target-fragment context, and C–D tests requiring a local patch rather than full-page output while retaining the same visual and code inputs. D does not add a local-crop input or remove the full HTML; the crop described below is used for evaluation.

Table 7: Configurations for the 171-example single-edit repair study. Every configuration receives full screenshots and full HTML. Numerical results are reported in Table 1.

| Config. | Bbox-masked screenshots | Target fragment | Output |
|---|---|---|---|
| A | No | No | Full HTML |
| B | Yes | No | Full HTML |
| C | Yes | Yes | Full HTML |
| D | Yes | Yes | Local patch |

**Full-page and local similarity.** Let $I_{\text{clean}}$, $I_{\text{corrupt}}$, and $I_{\text{candidate}}$ denote the clean reference, corrupted input render, and candidate render, respectively, and let $f_{\text{DINO}}$ extract DINO features. We define

$$s_{\text{page}}(I) = 100 \cos(f_{\text{DINO}}(I_{\text{clean}}), f_{\text{DINO}}(I)) . \tag{13}$$

The full-page delta is $s_{\text{page}}(I_{\text{candidate}}) - s_{\text{page}}(I_{\text{corrupt}})$. For local similarity, we use a fixed bounding-box union expanded by 16 pixels to define the crop region $U$. The same crop is used for the clean, corrupted, and candidate renders:

$$s_{\text{local}}(I) = 100 \cos(f_{\text{DINO}}(I_{\text{clean}}[U]), f_{\text{DINO}}(I[U])) . \tag{14}$$

The local delta is $s_{\text{local}}(I_{\text{candidate}}) - s_{\text{local}}(I_{\text{corrupt}})$. Both deltas are reported in DINO score points, with cosine similarities multiplied by 100, matching Table 1.

**Local repair and outside-bbox damage.** A local repair is counted when $s_{\text{local}}(I_{\text{candidate}}) > s_{\text{local}}(I_{\text{corrupt}})$. Let $\Omega_{\text{out}}$ be the fixed pixel region outside the evaluation bbox, and define

$$e_{\text{out}}(I) = \text{MAE}(I[\Omega_{\text{out}}], I_{\text{clean}}[\Omega_{\text{out}}]) . \tag{15}$$

Outside-bbox damage is counted when $e_{\text{out}}(I_{\text{candidate}}) > e_{\text{out}}(I_{\text{corrupt}}) + 10^{-6}$. This criterion measures an increase in pixel error relative to the clean reference, not merely any change outside the target. The repair and damage rates summarize the respective event frequencies under the evaluation aggregation.

**Interpretation.** Table 1 reports positive full-page and local DINO deltas for both editors in all four configurations. Adding the oracle region (A–B) improves Qwen3.8-27B but reduces GLM-5.3-Flash's repair rate and increases its outside-bbox damage. Providing the corresponding code fragment (B–C) improves both editors' repair rates and reduces outside-bbox damage. Restricting the output to a local patch (C–D) yields smaller additional changes: Qwen3.8-27B's repair rate rises from 98.8% to 99.4% and its outside-bbox damage falls from 17.5% to 17.0%; GLM-5.3-Flash's repair rate rises from 97.7% to 98.2%, while outside-bbox damage remains at 20.5% and its local DINO delta decreases slightly. These results support making the visual-to-code correspondence explicit, with a smaller additional benefit from local output constraints. Outside-bbox damage remains nonzero with local patches, consistent with the distinction between source-code locality and visual locality in Section 3.

### C.2 Discrepancy Bounding-Box Detection

To isolate discrepancy detection from DOM grounding, we evaluate the first model-returned render box against the corrupted causal DOM-node box on the same 171 single-edit cases and three seeds used in Section 4.5. We define $\mathrm{IoGT} = |B_p \cap B_{gt}|/|B_{gt}|$. This recall-oriented measure quantifies how much of the causal target is covered by the detector box without invoking any node-matching rule. Invalid detector responses remain in the denominator with zero score, giving 513 observations per model.

Table 8: Pure render-side bounding-box detection accuracy. IoGT≥0.5 is the fraction of predictions covering at least half of the causal ground-truth box.

| Model | Mean IoGT | IoGT≥0.5 |
|---|---|---|
| Base | 59.68% | 60.23% |
| SFT | 70.57% | 71.73% |
| GSRPO | **72.86%** | **73.68%** |

SFT improves mean IoGT by 10.90 percentage points over Base, while GSRPO adds 2.29 points over SFT. The corresponding IoGT≥0.5 gains are 11.50 and 1.95 points. Page-cluster bootstrap intervals exclude zero for SFT versus Base but include zero for GSRPO versus SFT. Thus, supervised training provides the clear detector improvement, whereas the final checkpoint's additional gain in exact-node grounding should not be attributed to a statistically resolved increase in box coverage.

### C.3 DOM Grounding Robustness

We complement the joint discrepancy-detection and DOM-grounding evaluation in Section 4.5 with a grounding robustness study. Ground-truth targets are derived from real webpage refinement trajectories, where the modified DOM node and its rendered bounding box are known from the editing process. We perturb the ground-truth bounding box to simulate localization noise.

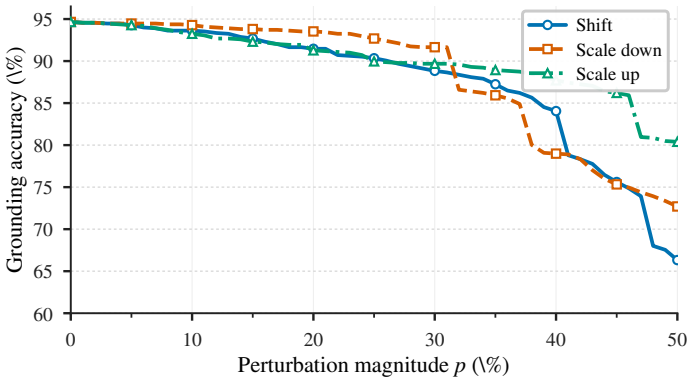



Figure 4: Highest-IoU grounding under box perturbations. Shift is $(pw/100, ph/100)$; scale multiplies both dimensions by $(1 \pm p/100)$. Exact-node accuracy is reported over 1,066 cases.

**DOM Grounding.** We map each target box to the rendered DOM node with the highest IoU. In practice, predicted boxes may deviate from the target in position and size, so we simulate these errors using deterministic shifts and center-preserving scaling from 0% to 50% in 1% increments before applying the same grounding rule. Figure 4 reports exact-node accuracy: at 20% perturbation, accuracy remains 91.46%, 93.53%, and 91.28% for shifts, shrinking, and enlargement, respectively, decreasing to 66.32%, 72.70%, and 80.39% at 50%. These results indicate robustness to moderate localization errors, while larger perturbations can change the highest-IoU match to a neighboring or enclosing element.

### C.4 CONTROLLED DOM STRUCTURAL STRESS

To complement the aggregate box-perturbation analysis in Appendix C.3, we construct a controlled structural stress suite in which the causal DOM node and its rendered box are known exactly. The suite contains 200 clean–corrupted render pairs, with 50 examples for each of four structures: nested elements, pseudo-elements, canvas content, and responsive containers. All target containers are absolutely positioned within a fixed stage, which controls their geometry while exercising absolute positioning throughout the suite. The changed-pixel ratio follows a deterministic grid spanning 0.11%–3.97%, with an overall mean of 2.10%; all 200 pairs produce a visible change and a valid causal box.

For nested cases, we edit an absolutely positioned parent containing an intermediate container and a leaf node. For pseudo-element cases, the visual change is applied to a `::before` rule and the owning element is used as the causal DOM target. For canvas cases, we modify a known drawn rectangle and evaluate localization only to the `<canvas>` owner. For responsive-container cases, we change a three-column grid into a single-column layout at a fixed viewport. Thus, the responsive condition tests container-layout propagation rather than a sweep across viewport breakpoints.

Table 9: Exact-node grounding accuracy (%) of pure highest-IoU on the controlled DOM structural stress suite. Each category contains 50 examples.

| DOM Structure | Accuracy |
|---|---|
| Nested | 96.0 |
| Pseudo-element | 100.0 |
| Canvas | 100.0 |
| Responsive | 94.0 |
| Average | 97.5 |

Given the exact causal box, pure highest-IoU identifies the correct node in 195 of 200 cases, corresponding to an average accuracy of 97.5%. The five errors are confined to nested and responsive-container examples, where a descendant can compete with the causal parent. The near-ceiling result indicates that hierarchical or absolutely positioned layouts do not inherently prevent geometry-based grounding when the target box is accurate, although parent–child ambiguity remains a source of error.

The pseudo-element and canvas results should be interpreted as owner-node localization: they do not identify an independent pseudo-element object or an internal canvas primitive. Likewise, the fixed-viewport responsive construction does not establish robustness across breakpoints. The suite is therefore a controlled diagnosis of bbox-to-DOM grounding under structural variation, rather than a substitute for end-to-end evaluation on naturally occurring complex webpages.

### C.5 VISUAL EVIDENCE FOR COMPONENT EDITING

The editing task combines a code target with visual evidence about the desired change. We compare masked and unmasked inputs while sampling one target component per page and performing one refinement step. Masked inputs preserve the original canvas and fill non-target regions with white pixels, following Section 3.

Masking increases the component DINO gain from 1.44 to 2.16 and the component VLM gain from 4.22 to 5.99. The page VLM gain also increases from 0.06 to 0.72, while acceptance rates remain similar (38.78% without masking and 37.76% with masking). These results support presenting focused visual evidence alongside the code target.

Table 10: Single-component masking ablation. The inputs differ in whether non-target pixels are masked; bold marks the larger reported improvement within a metric.

| Metric | Mask-Off | | | Mask-On | | |
|---|---|---|---|---|---|---|
| | Before | After | Δ | Before | After | Δ |
| *Component Level* | | | | | | |
| Component DINO | 69.38 | 70.81 | +1.44 | 69.38 | 71.54 | **+2.16** |
| Component VLM | 46.15 | 50.37 | +4.22 | 46.15 | 52.14 | **+5.99** |
| *Page Level* | | | | | | |
| Page DINO | 91.67 | 92.41 | +0.74 | 91.67 | 92.47 | +0.79 |
| Page VLM | 63.10 | 63.16 | +0.06 | 63.10 | 63.82 | **+0.72** |
| Accept Rate | 38.78% | | | 37.76% | | |

### C.6 Effect of the Number of Iteration Cycles

We analyze the relative visual score improvement $\Delta$ under different iteration budgets ($T \in \{1, 2, 3\}$). Table 11 reports the gain over the initial one-shot generation. The $T = 1$ and $T = 2$ rows retain the original iteration-budget measurements; the $T = 3$ row uses the corresponding three-round gains reported in the main experiments. The results show that refinement improves alignment across datasets, while the amount of gain depends on the dataset and the iteration budget.

Table 11: Ablation on the number of iteration cycles ($T$). We report the relative VLM score improvement ($\Delta_{\text{init}}$) over the initial one-shot generation across four datasets.

| Iteration ($T$) | Design2Code | Flame-React | Web2Code | UIHTML |
|---|---|---|---|---|
| $T = 1$ | +0.58 | +0.91 | +1.25 | +0.30 |
| $T = 2$ | +1.06 | +0.95 | +0.83 | +0.33 |
| $T = 3$ | +1.96 | +0.88 | +2.03 | +2.32 |

## D Reward and Scoring Design Analysis

### D.1 Reward Design

We compare four reward constructions for policy optimization, focusing on their downstream visual scores and evaluation trade-offs:

- **DINO only**: Computes the differential DINO similarity ($\Delta$) of the global page rendering before and after the code modification.
- **DINO + Spatial Penalty**: Augments the global DINO similarity delta with an $L_1$ spatial displacement penalty, explicitly penalizing unintended layout shifts.
- **Absolute VLM Judge**: Prompts a VLM to assign a continuous alignment score (0–100) based on absolute visual parity.
- **Group-scored VLM (Ours)**: Employs our GSRPO formulation (Section 3), using the VLM to score multiple candidates jointly.

We compute Page VLM Score and Page DINOv3 for the full page. Comp Accept Rate denotes the fraction of local edits that are accepted, and Avg Comp $\Delta$ is the average DINOv3 score difference on the cropped edited component region before and after each component edit.

Table 12: Comparison of different RL reward signals evaluated on the UIHTML dataset.

| **Reward Strategy** | **Page VLM Score ↑** | **Page DINOv3 ↑** | **Comp Accept Rate ↑** | **Avg Comp Δ ↑** |
|---|---|---|---|---|
| DINO only | $\underline{65.55 \pm 1.08}$ | $\underline{87.87 \pm 0.81}$ | 0.375 | 2.433 |
| DINO + Spatial Penalty | $62.72 \pm 1.38$ | $87.47 \pm 1.00$ | 0.389 | $\underline{2.455}$ |
| Absolute VLM Judge | $64.89 \pm 1.24$ | $87.41 \pm 0.86$ | $\underline{0.401}$ | **2.812** |
| Group-scored VLM (Ours) | $66.03 \pm 1.11$ | $87.99 \pm 1.00$ | **0.402** | 1.839 |

Table 12 reveals a distinction between local edit quality and full-page visual fidelity. Although the Absolute VLM Judge produces the largest average component improvement, its final Page VLM Score remains below that of Group-scored VLM. This suggests that stronger local improvements do not necessarily translate into better full-page results. Among the evaluated reward designs, Group-scored VLM achieves the highest mean page-level scores.

Beyond downstream performance, group scoring changes how the reward signal is constructed. By jointly comparing candidates from the same state, it produces a higher-contrast within-state signal while requiring substantially fewer evaluator calls than independent scoring. We therefore adopt Group-scored VLM in GSRPO for its favorable optimization–efficiency trade-off, rather than for superior visual-judging accuracy or stability. The following analysis reports the full protocol comparison.

### D.2 Scoring Protocols

We isolate the scoring protocol itself and compare three prompt-based approaches:

- **Single:** The VLM evaluates each candidate image $I_i$ independently against the reference $I_{ref}$.
- **Pairwise:** The VLM conducts head-to-head comparisons for all candidate pairs to derive a definitive ranking.
- **N-way (Group-Scored, Ours):** The VLM simultaneously evaluates all $N$ candidates against $I_{ref}$ in a single prompt context.

We analyzed these protocols across 100 component pools, each containing $N = 8$ candidates produced by different models. For Single and N-way scoring, we repeated evaluation three times per pool and report score dispersion and repeat metrics in Table 13. Single scoring has better top-1 agreement with the Pairwise baseline (0.320 vs. 0.300), higher repeat Kendall's $\tau$ (0.385 vs. 0.227), and lower score standard deviation (4.30 vs. 8.12). N-way scoring instead has a larger normalized advantage range (1.34 vs. 1.08) and lower evaluation cost. Thus, this experiment supports higher contrast and efficiency, not greater repeat stability or agreement.

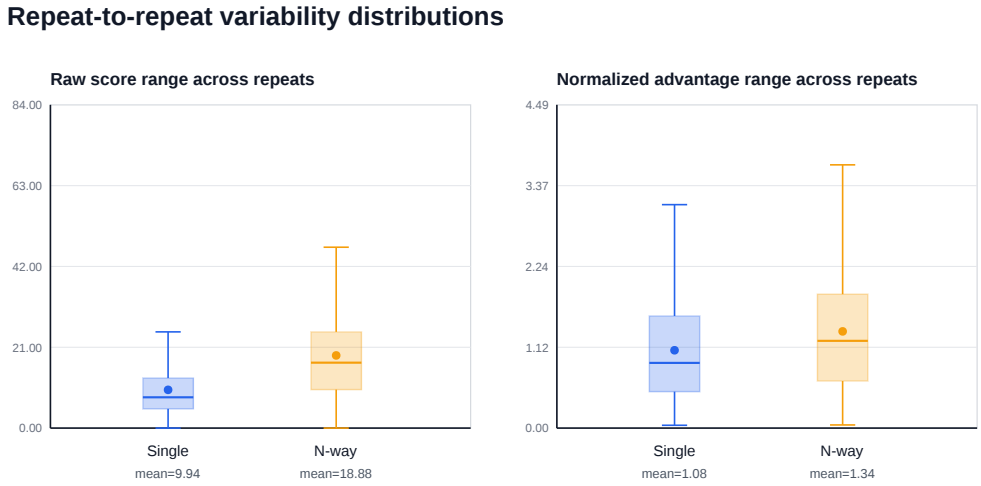



Figure 5: Repeat-to-repeat variability distributions

The repeat results expose a trade-off rather than a uniform improvement. N-way context produces wider separation among the normalized within-group advantages, but its scores vary more across repeated evaluations and agree slightly less with the Pairwise baseline. A larger advantage range can give GRPO a higher-contrast optimization signal because its update uses normalized group advantages $\hat{A}_i = \frac{s_i - \mu}{\sigma + \epsilon}$; by itself, however, it does not establish a more faithful ranking or a higher-quality gradient.

N-way scoring is substantially more evaluation-efficient in this experiment. Evaluating a pool of $N = 8$ candidates requires one API call and approximately 23K total tokens, compared with eight calls and approximately 42K tokens for Single scoring and 28 calls and approximately 214K tokens for the Pairwise baseline. N-way scoring uses $\mathcal{O}(1)$ calls per group, whereas Single requires $\mathcal{O}(N)$ and Pairwise requires $\mathcal{O}(N^2)$.

Table 13: Evaluation of VLM scoring protocols over 100 candidate pools, each containing $N = 8$ candidates from different models. For Single and N-way scoring, each pool is evaluated three times to compute standard deviation and repeat-consistency metrics.

| Protocol | Top-1 Agree vs. Pairwise | Kendall $\tau$ (Repeats) | Score Std. Dev. | Advantage Range | Calls / Pool | Est. Tokens / Pool |
|---|---|---|---|---|---|---|
| Pairwise | 1.000 | N/A | N/A | N/A | 28 | ~214K |
| Single | 0.320 | **0.385** | **4.30** | 1.08 | 8 | ~42K |
| N-way (Ours) | 0.300 | 0.227 | 8.12 | **1.34** | **1** | **~23K** |

In summary, N-way scoring provides a higher-contrast and substantially more evaluation-efficient comparative reward signal. These measured advantages motivate its use in GSRPO, while the lower repeat stability and agreement preclude claims that it is the uniquely optimal or more faithful scoring mechanism.

## E LIMITATIONS

Although IterUI improves high-fidelity UI-to-code generation through localized visual refinement, it has several limitations.

First, the current framework focuses on static webpage rendering. Our experiments evaluate HTML/CSS outputs rendered as screenshots, and do not consider dynamic JavaScript behavior, interactive states, animations, or scrolling behavior. Extending localized refinement to dynamic and interactive UI states is an important direction for future work.

Second, IterUI relies on visual critics and VLM-based verification to identify discrepancies and judge whether a localized edit improves the page. Although our human evaluation shows that the VLM judge correlates strongly with frontend-developer preferences, VLM scores can still be noisy, prompt-sensitive, and imperfectly calibrated across different webpage styles, domains, and visual densities. GSRPO provides higher within-group contrast and lower evaluation cost, but our protocol study does not show improved repeat stability or agreement, and group-relative scoring does not eliminate judge bias or occasional misranking.

Third, as an agentic framework, IterUI introduces additional computational overhead during inference. While our approach utilizes smaller language models and the localized refinement process is highly parallelizable to support concurrent execution, the iterative nature of the system still consumes a significantly larger number of tokens compared to single-pass generation. Consequently, this leads to an inherent increase in overall inference latency.

Finally, our training pipeline still depends on proprietary supervision: Gemini-3-Flash generates the initial 2,000 successful refinement trajectories, and GPT-5.2 provides the VLM rewards for reinforcement learning. This dependence limits full reproducibility and accessibility, as changes in the availability or behavior of these closed models may affect the resulting data and reward signals. Future work should replace them with open-source teachers and reward models.

## F Qualitative Examples

Figures 6–9 present eight qualitative UI-to-code examples. Each example compares the reference screenshot with the direct Qwen3-VL-8B output, our RL-trained model, and Gemini-3-Flash. DINOv3 scores are displayed within each figure.

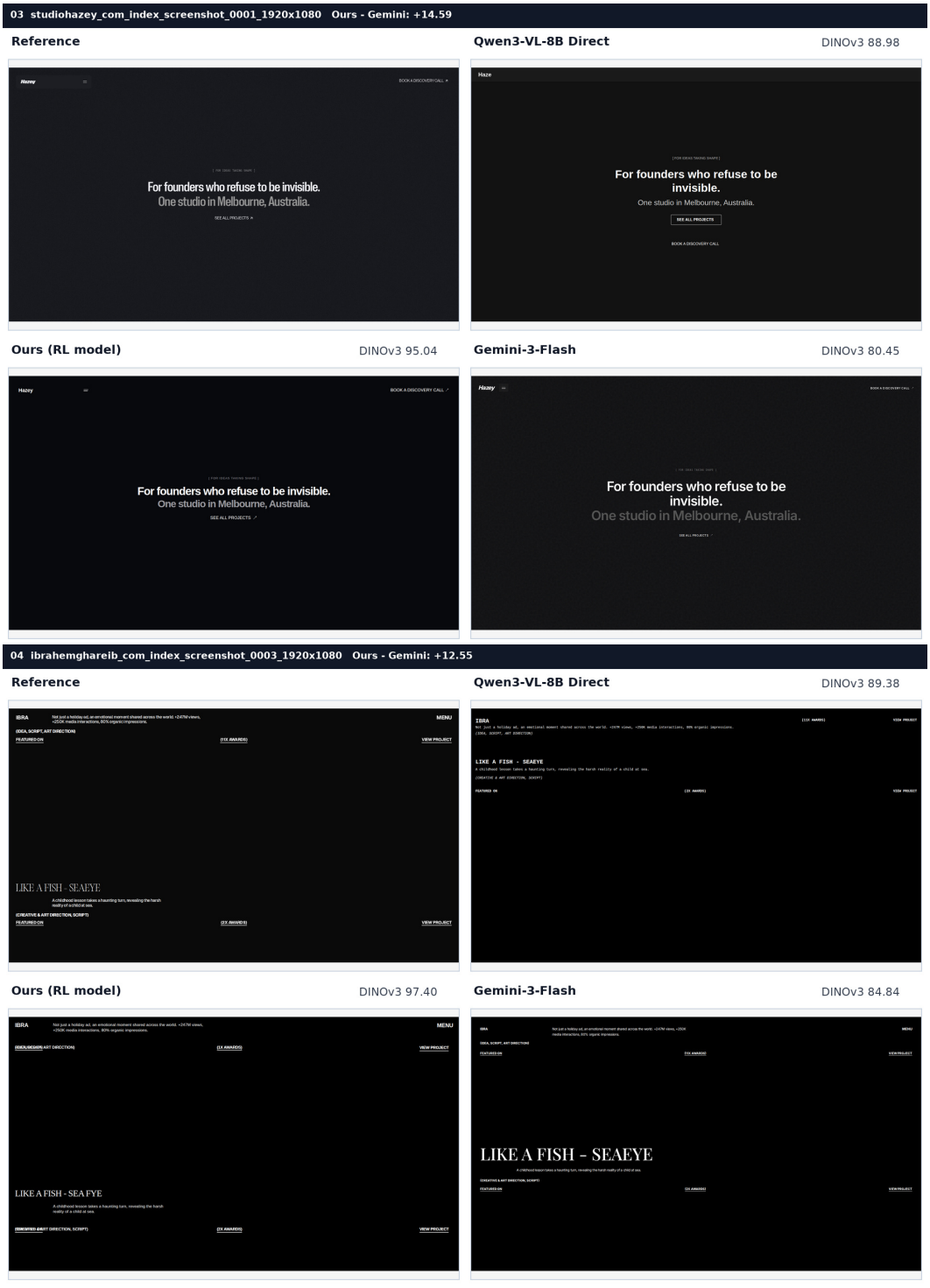



Figure 6: Qualitative comparisons for examples 03 and 04.

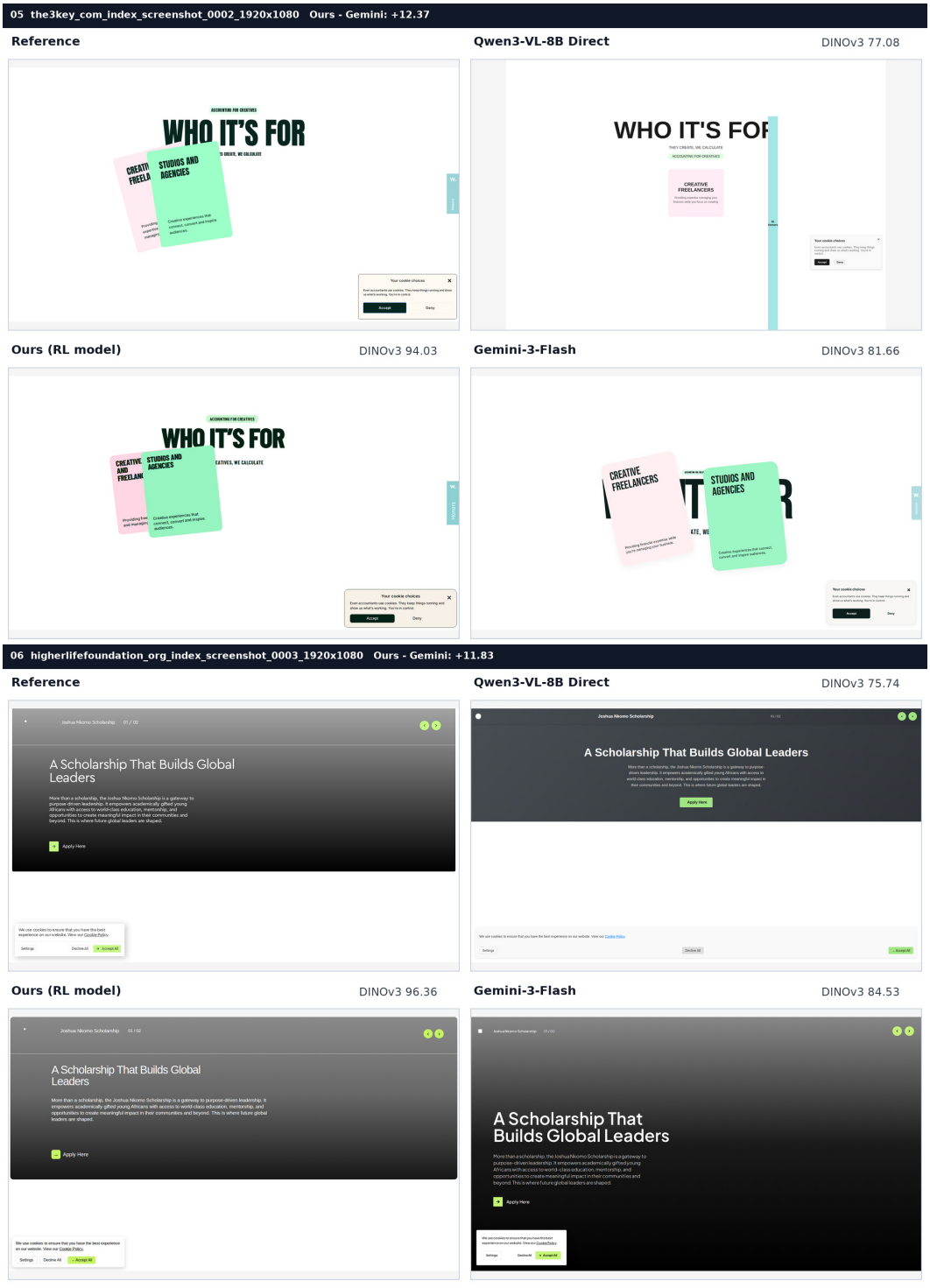



Figure 7: Qualitative comparisons for examples 05 and 06.

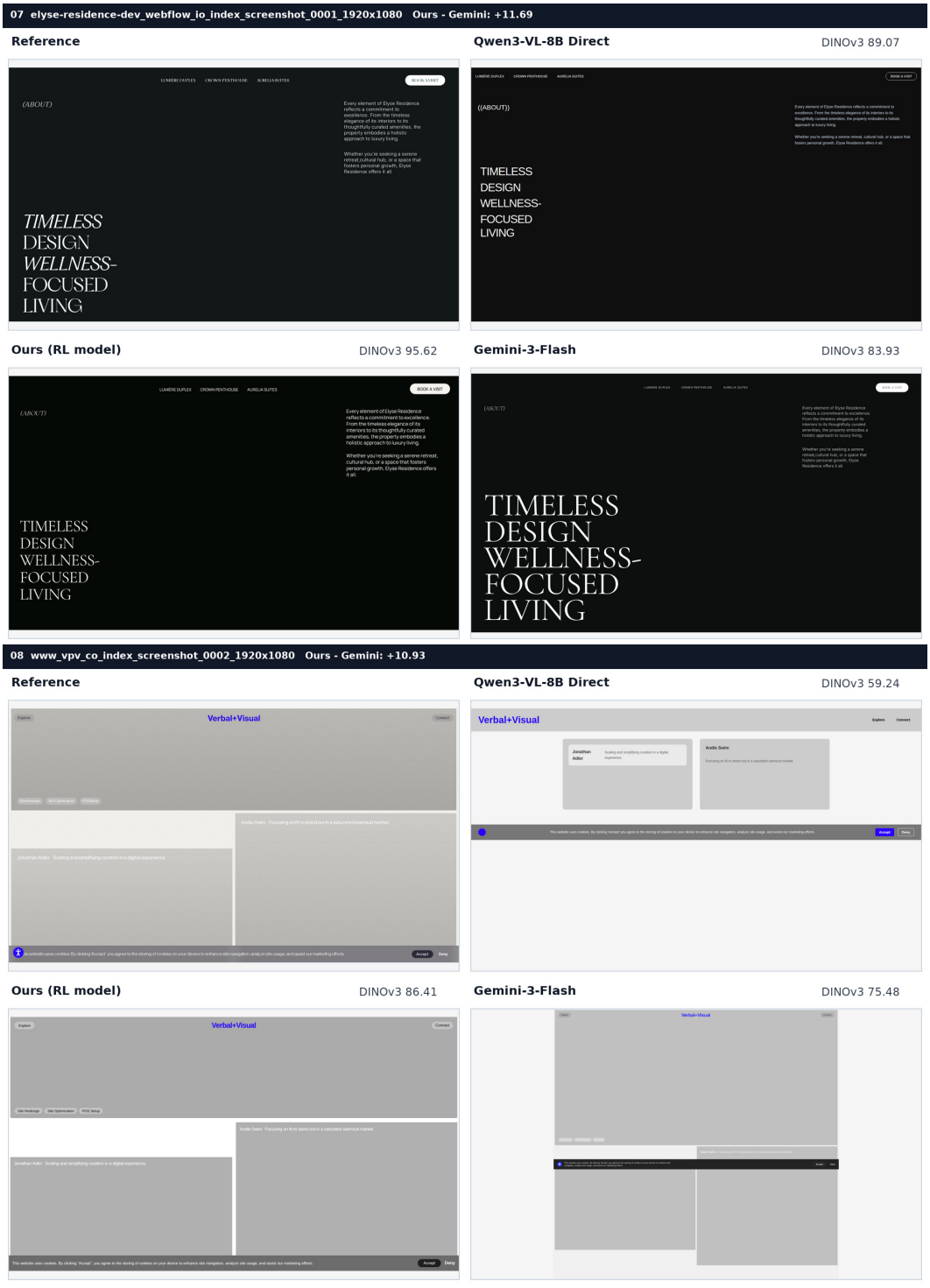



Figure 8: Qualitative comparisons for examples 07 and 08.

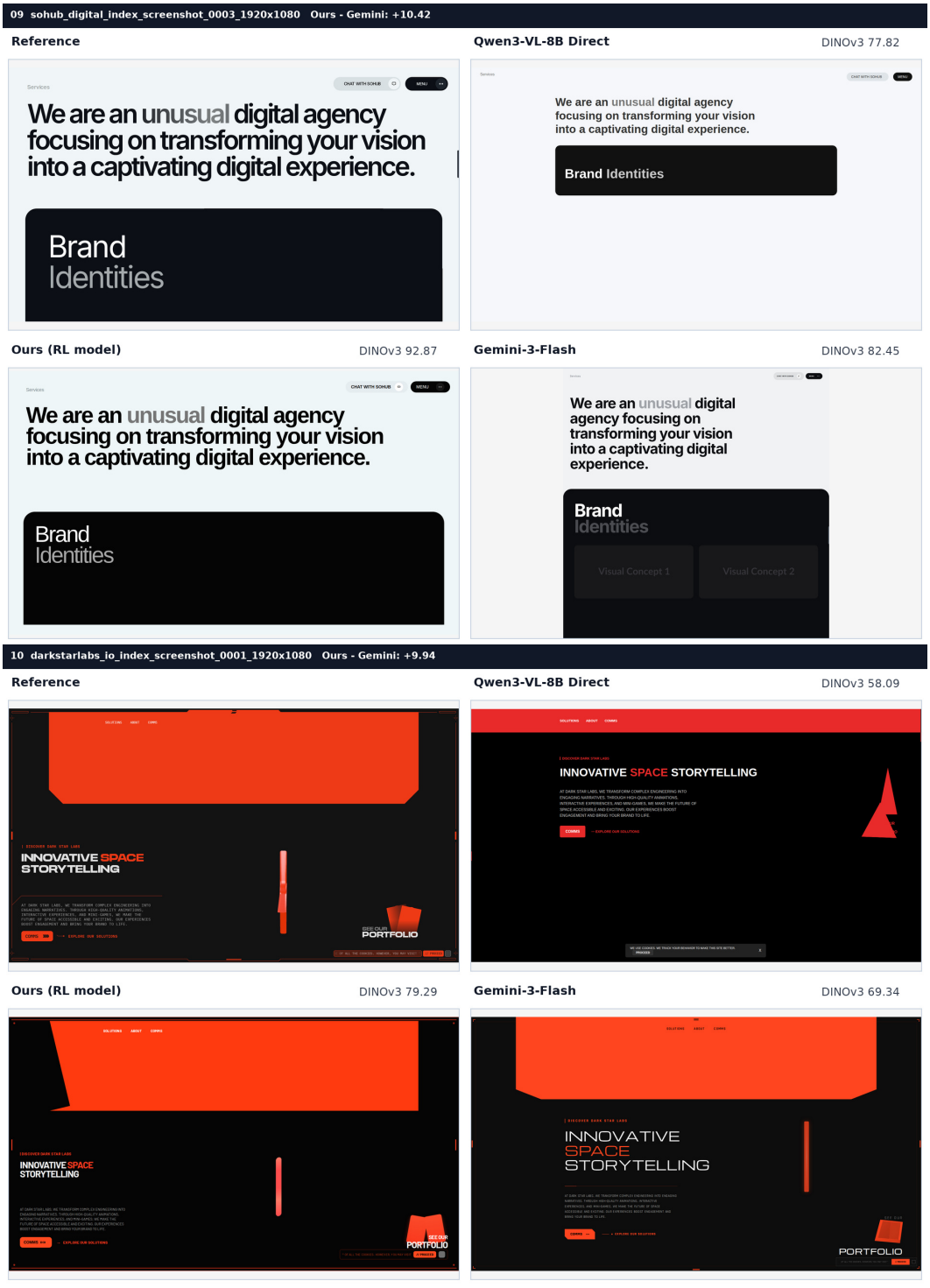



Figure 9: Qualitative comparisons for examples 09 and 10.

## G PROMPT TEMPLATES

This section provides prompt templates for initial generation, discrepancy localization, component refinement, and visual scoring. We separate system prompts from user prompts and mark dynamically inserted fields with braces, such as `{feedback}` or `{component_code}`. Images are passed as multimodal inputs in the order specified below.

### Initial HTML Generation Prompt

**Initial HTML generation.** For the initial full-page generation stage, the model receives one image: the target screenshot. The system prompt is:

```
Generate pixel-faithful HTML from screenshot.
```

The user prompt is:

```
You are an expert web developer
You take screenshots of a reference web page from the user, and
    then build single web page
using HTML and inline CSS.

- Make sure the app looks exactly like the screenshot.
- Pay close attention to background color, text color, font size,
    font family,
padding, margin, border, etc. Match the colors and sizes exactly.
- Use the exact text from the screenshot.
- Do not add comments in the code such as "<!-- Add other
    navigation links as needed -->" and "<!-- ... other news items
     ... -->" in place of writing the full code. WRITE THE FULL
    CODE.
- Repeat elements as needed to match the screenshot. For example,
    if there are 15 items, the code should have 15 items. DO NOT
    LEAVE comments like "<!-- Repeat for each news item -->" or
    bad things will happen.
- For images, use placeholder images from https://placehold.co and
     include a detailed description of the image in the alt text
    so that an image generation AI can generate the image later.

In terms of libraries,

Optional : you can use the following libraries:
- You can use this script to include Tailwind: <script src="https
    ://cdn.tailwindcss.com"></script>
- You can use Google Fonts: <script src="https://cdn.jsdelivr.net/
    npm/get-google-fonts@1.2.2/main.min.js"></script>
- Font Awesome for icons: <link rel="stylesheet" href="https://
    cdnjs.cloudflare.com/ajax/libs/font-awesome/5.15.3/css/all.min.
    css"></link>
- React: <script src="https://cdn.jsdelivr.net/npm/react@18.2.0/
    umd/react.development.js"></script>
- other libraries you need.

Output the full HTML code, and the code should be wrapped in <
    html_code> tags.
```

### Discrepancy Localization Prompt

**Discrepancy localization.** For discrepancy detection, the model receives two images: Image A is the target screenshot and Image B is the current rendered page. The system prompt is:

```
Compare paired screenshots and emit only machine-readable JSON
    coordinates.
```

The user prompt is:

```
Two screenshots are attached.
Image A is the desired target page. Image B is the current
    rendered page.

Find {num_regions} separate visual mismatch regions that a
    frontend patch should improve. Think like an image alignment
    tool creating a candidate inventory, not like a QA report
    repeating only the most obvious
issues.

Selection rules:
1. Prefer independent medium-sized UI/text/layout regions that can
     be compared side-by-side between Image A and Image B.
2. Ignore photographic/image/icon artwork details when possible.
3. Avoid single letters, tiny punctuation, blank whitespace, and
    the entire page or huge sections.
4. If a repeated group has the same mismatch, combine that group
    into one rectangle.
5. Use separate normalized coordinates for each image because
    matching regions may move or resize:
  - `target_region`: the mismatch region in Image A.
  - `current_region`: the corresponding mismatch region in Image B
      .
6. Silently consider more candidates than you need, then return
    the best non-overlapping regions.
{history_text}

Output raw JSON only, no markdown fence. Use exactly this schema:
[{"target_region":[x1,y1,x2,y2],"current_region":[x1,y1,x2,y2],"
    note":"short visual difference"}, ...].
All coordinates must be normalized numbers from 0 to 1000 relative
     to each respective image.
Return exactly {num_regions} objects, ordered by expected visual
    impact after applying the exclusion masks.
```

When previous regions have already been handled, we append the following exclusion-mask text as `{history_text}`:

```
Previously handled rectangles (exclusion masks, not examples):
1. target_image={previous_target_region} current_image={
    previous_current_region}
...

Treat every rectangle above as an opaque no-pick mask. Do not
    reuse it, slightly resize it, crop inside it, or choose a
    neighboring sliver of it. First build a broader mental
    inventory of alternative mismatch
regions, then return only regions outside these masks.
```

### Localized Component Refinement Prompt

**Localized component refinement.** For component refinement in the reported IterUI setting, the model receives two coordinate-preserving masked images: the reference screenshot followed by the current rendered page. Each image retains its original canvas, with non-target pixels masked in white, as described in Section 3. These editing inputs are distinct from the component crops used for scoring. The system prompt is:

```
Refine component HTML to better match reference image.
```

The base user prompt is:

```
You are a frontend page expert. You have two component screenshots
    : the first is the reference (target) component, and the
    second is your current rendered result.
In the 'think' section, please analyze in detail the differences
    between the two images in terms of layout, color, fonts,
    content, and fine details. List out the issues or
    inconsistencies point by point.
{alternative_instruction}
Then, based on these differences, rewrite the current component
    code and output the modified component code (including
    necessary CSS, and make it self-contained).
Make sure all code is placed inside <html_code> tags. Output only
    the code for this component, do not output the complete HTML
    page.
```

When multiple alternatives are generated for the same component, we insert the following text as `{alternative_instruction}`:

```
You are producing alternative {alternative_index} of {
    total_alternatives}. Make this attempt meaningfully different
    from earlier attempts: vary the implementation strategy, CSS
    values, layout method, or structure
instead of making a near-duplicate.
```

The refinement prompt additionally includes localized feedback and layout context. In the standard editing mode, the following suffix is appended:

```
- Suggestion: {feedback}

## Layout Context Information:
- Target component bbox (current render): {target_bbox_current}
- Target component bbox (reference): {target_bbox_reference}
- Parent chain (from body to this component):
  <{tag}> [x:{x}, y:{y}, w:{width}, h:{height}] {{display: {
      display}; position: {position}; flex-direction: {
      flex_direction}; justify-content: {justify_content}; align-
      items: {align_items}; gap: {gap};
grid-template-columns: {grid_template_columns}; grid-template-rows
    : {grid_template_rows}}}
  ...

- Current component code:
<target_region>
{component_code}
</target_region>
Please fix the issues.
Make sure all code is placed inside <html_code> tags. Output only
    the code for this component, do not output the complete HTML
    page.
```

In the from-scratch component-editing mode, the current component code is omitted and the following suffix is used instead:

```
- Suggestion: {feedback}

## Layout Context Information:
- Target component bbox (current render): {target_bbox_current}
- Target component bbox (reference): {target_bbox_reference}
- Parent chain (from body to this component):
  <{tag}> [x:{x}, y:{y}, w:{width}, h:{height}] {{display: {
      display}; position: {position}; flex-direction: {
      flex_direction}; justify-content: {justify_content}; align-
      items: {align_items}; gap: {gap};
grid-template-columns: {grid_template_columns}; grid-template-rows
    : {grid_template_rows}}}
```

```
  ...

Please write the component code from scratch based on the
    reference image.
Make sure all code is placed inside <html_code> tags. Output only
    the code for this component, do not output the complete HTML
    page.
```

When component masking is enabled together with from-scratch refinement, the model is additionally given the surrounding full-page HTML with the target component replaced by a placeholder. The suffix becomes:

```
- Suggestion: {feedback}

## Layout Context Information:
- Target component bbox (current render): {target_bbox_current}
- Target component bbox (reference): {target_bbox_reference}
- Parent chain (from body to this component):
  <{tag}> [x:{x}, y:{y}, w:{width}, h:{height}] {{display: {
      display}; position: {position}; flex-direction: {
      flex_direction}; justify-content: {justify_content}; align-
      items: {align_items}; gap: {gap};
grid-template-columns: {grid_template_columns}; grid-template-rows
    : {grid_template_rows}}}
  ...

## Full Page HTML (with component placeholder):
{full_html_with_component_placeholder}

Please write the component code from scratch based on the
    reference image to replace the placeholder.
Make sure all code is placed inside <html_code> tags. Output only
    the code for this component, do not output the complete HTML
    page.
```

Full-Page Visual Scoring Prompt

**Full-page visual scoring.** For full-page evaluation, the VLM judge receives two images: the reference screenshot and the rendered screenshot. The prompt is:

```
You will be given two images:
The first image is the reference image (design draft or target
    rendering).
The second image is the code rendering, which is generated based
    on the
first image using HTML/CSS/frontend code.
Your task is as follows:
Compare the overall similarity between the two images, on a scale
    from 0
to 100:
- 0 means completely dissimilar.
- 100 means perfectly identical.
When scoring, you should comprehensively consider the following
    aspects:
- Layout (whether the structural positions are consistent)
- Color scheme (whether the colors are faithfully reproduced)
- Typography (font, font size, line spacing, etc.)
- Spacing and alignment (whether element spacing and alignment are
     accurate)
- Fine details (button styles, icons, shadows, borders, etc.)
Strictly follow the output format below:
```

```
First, provide the final score, where the value must be enclosed
    in LaTeX
\boxed{}. Then, provide a justification for the score, explaining
    which
aspects are similar, which aspects differ, and the main factors
    influencing
the score.
```

Component Visual Scoring Prompt

**Component visual scoring.** For component-level scoring, the VLM judge receives two cropped component images: the reference component and the rendered component. The prompt is:

```
You will be given two cropped component images:
The first image is the reference component (from the target design
    ).
The second image is the rendered component (from the generated
    HTML).
Your task is as follows:
Compare the overall similarity between the two component images,
    on a scale from 0
to 100:
- 0 means completely dissimilar.
- 100 means perfectly identical.
When scoring, you should comprehensively consider the following
    aspects:
- Layout (whether the structural positions are consistent)
- Color scheme (whether the colors are faithfully reproduced)
- Typography (font, font size, line spacing, etc.)
- Spacing and alignment (whether element spacing and alignment are
     accurate)
- Fine details (button styles, icons, shadows, borders, etc.)
Strictly follow the output format below:
First, provide the final score, where the value must be enclosed
    in LaTeX
\boxed{}. Then, provide a justification for the score, explaining
    which
aspects are similar, which aspects differ, and the main factors
    influencing
the score.
```