1D-BENCH: A BENCHMARK FOR ITERATIVE UI CODE GENERATION WITH VISUAL FEEDBACK IN REAL-WORLD

Anonymous authors
Paper under double-blind review

ABSTRACT

Design-to-code translates high-fidelity UI designs into executable front-end implementations, but progress remains hard to compare due to inconsistent datasets, toolchains, and evaluation protocols. We introduce 1D-Bench, a benchmark grounded in real e-commerce workflows, where each instance provides a reference rendering and an exported intermediate representation that may contain extraction errors. 1D is short for one day, representing the efficient completion of design-to-code tasks in less than one day. Models take both as input, using the intermediate representation as structural cues while being evaluated against the reference rendering, which tests robustness to intermediate representation defects rather than literal adherence.

1D-Bench requires generating an executable React codebase under a fixed toolchain with an explicit component hierarchy, and defines a multi-round setting in which models iteratively apply component-level edits using execution feedback. Experiments on commercial and open-weight multimodal models show that iterative editing generally improves final performance by increasing rendering success and often improving visual similarity. We further conduct a pilot study on post-training with synthetic repair trajectories and reinforcement learning based editing, and observe limited and unstable gains that may stem from sparse terminal rewards and high-variance file-level updates. The data and scripts used in this study are available in an anonymized repository at `https://anonymous.4open.science/r/d2c-benchmark-A9C4`.

1 INTRODUCTION

Generating user interface code from design drafts is an important problem spanning HCI and software engineering(Khan et al., 2025). In practice, designers produce high-fidelity drafts in tools such as Sketch and Figma(Petridis et al., 2023), and engineers manually translate them into front-end code. This workflow is costly and error-prone, as small deviations in hierarchy, spacing, or constraints can cause visible inconsistencies and increased maintenance(Yang et al., 2025b; Wu et al., 2025a).

Recent multimodal large language models have advanced UI code synthesis(Zhou et al., 2025; Gui et al., 2025c; Wan et al., 2025; Liang et al., 2025). However, many benchmarks rely on synthetic or crawled data and use heterogeneous output targets and toolchains, limiting comparability and alignment with production. Real-world e-commerce UIs further involve deeply nested layouts and diverse constraints that remain challenging.

We present 1D-Bench, grounded in e-commerce development practices, with a standardized evaluation protocol. Each instance includes a reference rendering and an exported design IR that may contain extraction errors(Lu et al., 2024; You et al., 2024; Li & Li, 2023). Models receive the IR as structural cues while being evaluated against the reference rendering, which tests robustness to IR defects. 1D-Bench requires generating an executable React codebase under a fixed toolchain with an explicit component hierarchy, rather than isolated HTML.

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.

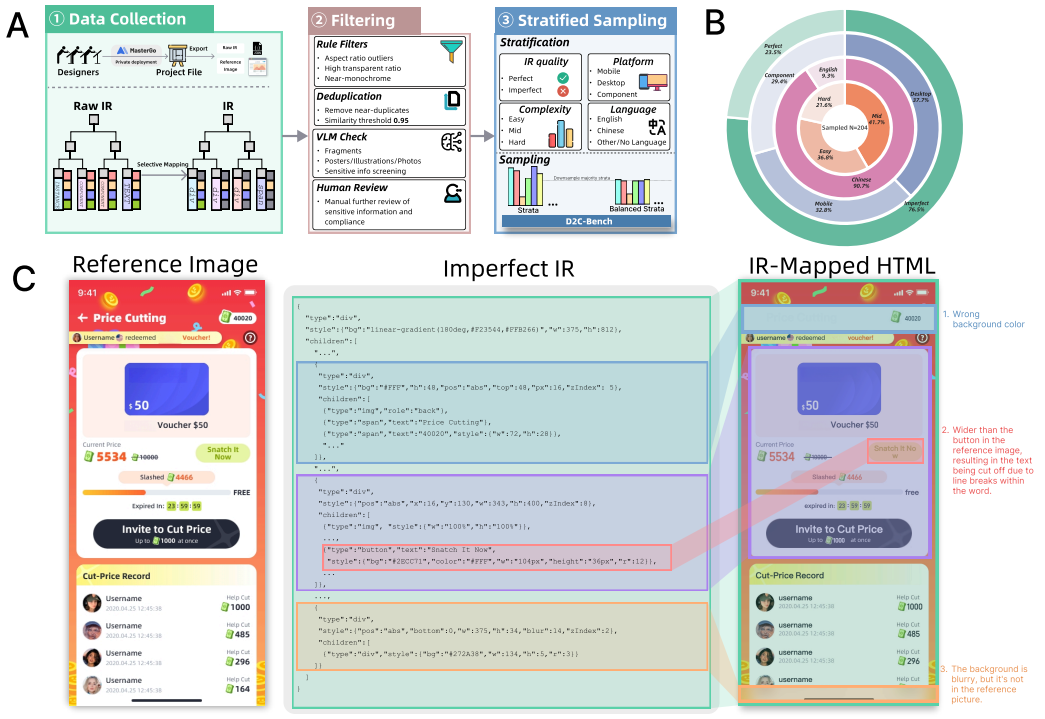


Figure 1: (A) Dataset construction pipeline. (B) Dataset distribution. (C) One dataset example showing the reference UI image and its exported intermediate representation (IR). The IR may contain extraction defects, which propagate to the IR-mapped HTML rendering.

We also define a multi-round setting where models iteratively apply component-level edits using execution feedback. In addition, we conduct an exploratory pilot post-training study using synthetic repair trajectories and reinforcement learning based editing(Jiang et al., 2025; Wang et al., 2025a). The observed gains are limited, and we discuss factors that may contribute.

Our contributions are as follows. (1) We introduce 1D-Bench, derived from production e-commerce workflows, that evaluates executable React generation under a fixed toolchain using both a reference rendering and an imperfect exported IR as input. (2) We define a multi-round protocol with execution feedback for iterative editing. (3) We report an exploratory pilot study on post-training for multi-round generation and discuss factors that may limit improvements.

2 1D-Bench

2.1 Data Construction

1D-Bench is derived from an internal static D2C platform used in our organization. For each MasterGo design file, the platform exports an IR and a reference rendering from the same specification. We collect paired IRs and reference images from employee uploads, apply automated filtering, and manually review the retained instances to remove incomplete, low-quality, duplicate, non-UI, or non-compliant records (Appendix B). This yields 984 instances from 5,856 candidates (Figure 1A).

This construction process is designed to preserve the practical uncertainty of design-to-code workflows rather than producing an artificially clean input representation. In real production pipelines, exported structural representations are often affected by imperfect parsing, missing attributes, incorrect grouping, or noisy style values. Directly treating such an IR as ground truth would overestimate the reliability of the design export process and would encourage models to copy potentially incorrect structures. We therefore use the IR as an auxiliary input and retain the reference rendering as the final target. This setting requires models to exploit useful structural cues from the IR while remaining robust to its defects.

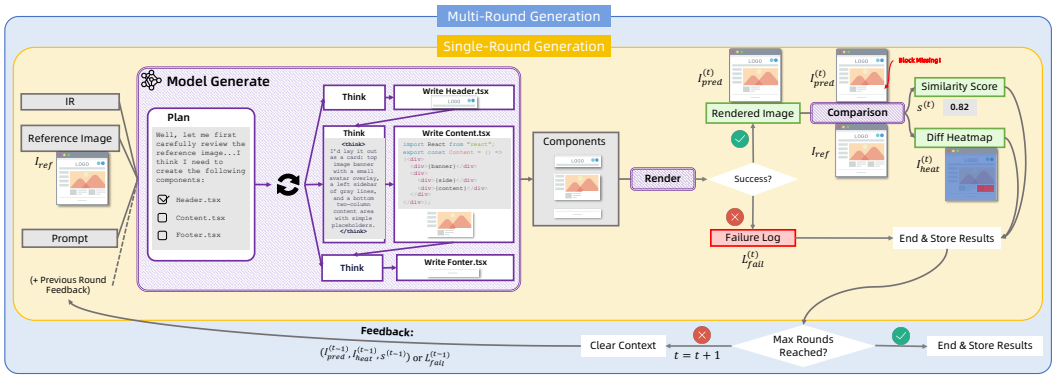


Figure 2: Task definition for single-round and multi-round generation.

The filtering and review process also aims to ensure that the benchmark focuses on implementable UI tasks. Automatically removed samples include incomplete artifacts, extreme aspect ratios, near-duplicate pages, visually degenerate images, non-UI content, and records with potential privacy risks. Since automatic filters may incorrectly reject complex but valid UI layouts or retain ambiguous cases, uncertain samples are further inspected manually. This combination of rule-based filtering, model-assisted screening, and human review reduces noise while avoiding an overly narrow dataset that only contains simple or visually regular pages.

We construct the evaluation set of size 204 via stratified sampling over IR reliability, platform type, primary language, and IR size. The IR is provided as an auxiliary, potentially noisy input, while the reference rendering defines the target. The task is to generate an executable React project that matches the reference (Figure 1A-B). Figure 1C shows an instance.

The stratified sampling step is important because raw industrial uploads are naturally imbalanced. Some categories, such as common mobile marketing pages or Chinese e-commerce components, appear much more frequently than others. Without balancing, evaluation results could be dominated by a small number of frequent design patterns and would provide limited evidence about model behavior on less common but still realistic cases. We therefore sample across IR reliability, platform type, primary language, and IR size to obtain a more diverse evaluation set. This design makes the benchmark better suited for comparing models under heterogeneous UI conditions, including both relatively clean inputs and cases where the exported IR contains visible errors.

2.2 Task Definition

1D-Bench evaluates executable design-to-code generation given an imperfect exported IR and a reference image that defines the target rendering. For each instance, the model writes a runnable React project under a fixed toolchain. An execution harness builds and renders the project in a standardized viewport and compares the screenshot I_{pred} with I_{ref} (Section 2.3). Inputs are the exported IR, the reference image, and a pre-initialized workspace with a fixed React scaffold and pinned dependencies (Appendix D). The MLLMs edit the workspace by creating or overwriting files via a WriteTool-only interface, and the final workspace file tree is evaluated. Specifically, the tasks can be divided into single-round generation and multi-round generation (see Figure 2).

- **Single-Round Generation.** The model writes the full codebase in one pass. Instances that fail to build or render are counted as rendering failures and receive no similarity score. We report mean similarity over successful renders and the rendering success rate.
- **Multi-Round Generation.** The model iteratively edits the same workspace for up to a fixed number of rounds. After each round t, the harness attempts to build and render. On success, it returns $I_{\text{pred}}^{(t)}$, a diff heatmap $I_{\text{heat}}^{(t)}$, and a similarity score $s^{(t)}$; on failure, it returns runtime logs $L_{\text{fail}}^{(t)}$. The process stops at the round limit or when $s^{(t)}$ reaches a preset threshold. We track rendering success across rounds and report final-round results following Section 3.

	Size	Input	Output	Multi-Round	Type	Real Source
IW-Bench(Guo et al., 2025)	1200	Image	HTML	✗	Synthetic	-
pix2code(Beltramelli, 2018)	1742	Image	HTML	✗	Synthetic	-
WebCode2M(Gui et al., 2025a)	768	Image	HTML	✗	Real	Crawl
CC-HARD(Gui et al., 2025b)	128	Image	HTML	✗	Real	Crawl
MRWeb(Wan et al., 2024)	500	Image	HTML	✓	Mixed	Crawl
FullFront(Sun et al., 2025)	400	Image	HTML	✗	Synthetic	-
Flame-React-Eval(Ge et al., 2025)	109	Image	React	✗	Real	Hand-designed
Web2Code(Yun et al., 2024)	5990	Image	HTML	✗	Mixed	Crawl
Sketch2Code(Li et al., 2025b)	731	Image	HTML	✗	Real	Hand-designed
Design2Code(Si et al., 2025)	484	Image	HTML	✗	Real	Crawl
1D-Bench	**204**	**Image + Metadata**	**React**	✓	**Real**	**Industry**

Table 1: Public design-to-code benchmarks.

Table 1 summarizes public D2C benchmarks. Compared with prior work, 1D-Bench supports multi-round evaluation, uses IR-guided inputs, targets executable React projects under a fixed toolchain, and is sourced from industrial e-commerce data.

The distinction between the two settings is important for evaluating practical design-to-code systems. Single-round generation measures whether a model can directly infer a complete implementation from the visual target and the exported IR. This setting is simple and reproducible, but it does not reflect the way front-end implementation is often completed in practice, where developers inspect intermediate renderings, identify local mismatches, and revise the code accordingly. Multi-round generation therefore evaluates an additional capability: the ability to use execution feedback as an external signal for localized repair.

The workspace-based formulation also prevents the task from being reduced to independent image-to-code attempts across rounds. Since each round modifies the same persistent project, later actions depend on earlier architectural choices, file organization, and component boundaries. This makes the benchmark sensitive to both visual fidelity and engineering consistency. A model that produces visually plausible code in the first round may still perform poorly if its component structure is difficult to edit, while a model with a cleaner decomposition can benefit more from feedback in later rounds. In this sense, 1D-Bench evaluates not only generation quality, but also the editability of the generated implementation under realistic execution constraints.

2.3 METRICS

1D-Bench follows an execution-based protocol: the generated React project is built and rendered in a controlled environment, and the resulting screenshot is compared with the reference image. We report (i) visual similarity on successfully rendered instances and (ii) rendering success rate to capture executability. Implementation details are provided in Appendix C.

Visual similarity. Given the rendered screenshot I_{pred} and the reference image I_{ref}, we compute a similarity score $S \in [0, 1]$ using a composite metric. The metric is primarily based on LPIPS, with auxiliary structural signals (e.g., SSIM) and text/layout cues to better reflect both perceptual fidelity and layout completeness. We report the mean similarity over successfully rendered instances. In

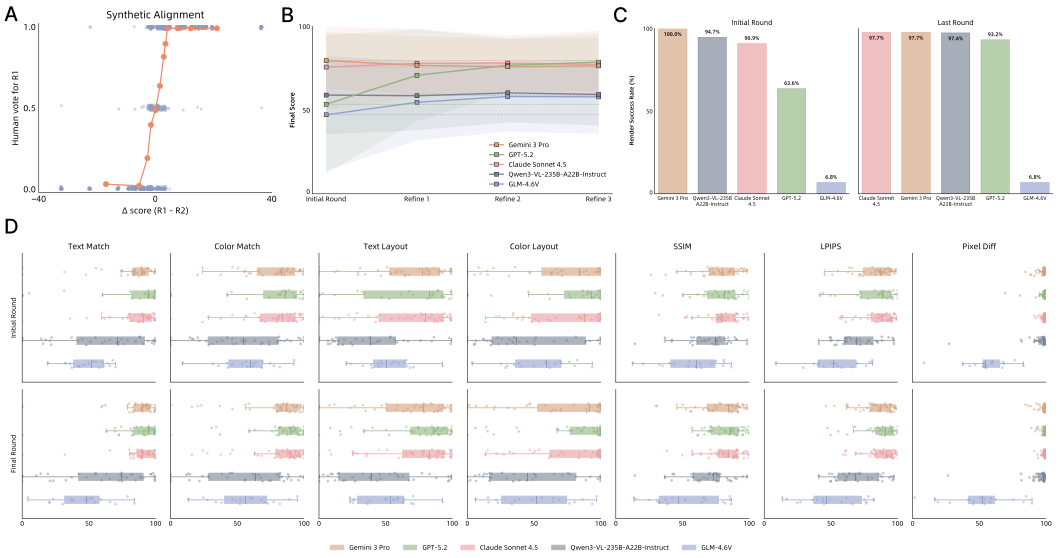


Figure 3: (A) Synthetic preference data relating the score difference between R1 and R2 to human preference, shown as jittered scatter and binned means. (B) Multi-round final score trends with mean curves and ± 1 standard deviation bands. (C) Render success rates for initial and final rounds shown as paired bar charts. (D) Boxplots of metric breakdowns for initial and final rounds.

the multi-round setting, we use the similarity score from the final round if it renders; otherwise the instance receives no similarity score.

Rendering success rate. We record a binary outcome per instance: a run is successful if the project builds and produces a valid screenshot within a fixed timeout, and unsuccessful otherwise. We report the fraction of successful runs over all test instances. In the multi-round setting, failure logs are returned to the model as feedback.

We separate visual similarity from rendering success because these two aspects capture different failure modes. A generated project may be visually accurate when it renders successfully, but still be unreliable if it frequently fails to build, times out, or produces an invalid screenshot. Conversely, a model may achieve high rendering success by producing conservative layouts that compile reliably but fail to match the reference design. Reporting both metrics makes these trade-offs explicit and avoids conflating perceptual quality with executability.

This design is particularly important for design-to-code evaluation. In front-end engineering, an implementation that cannot be built or rendered is unusable regardless of its intended visual structure. At the same time, executability alone is insufficient because a valid React project may still omit important content, distort layout relationships, or ignore the design hierarchy. The composite protocol therefore follows an execution-first evaluation principle: only successfully rendered outputs are compared visually, while rendering failures are still reflected through the rendering success rate and the final reported score.

Final reported score. To jointly reflect fidelity and executability, we report

$$\mathit{FinalScore} = \bar{S} \cdot \mathit{RSR},$$

where $\bar{S} = \mathbb{E}[S(I_{\text{pred}}, I_{\text{ref}}) \mid \text{render succeeds}]$ is the mean similarity over successfully rendered instances, and *RSR* is the rendering success rate. This design prevents methods that overfit visual similarity while frequently failing to build or render from being over-credited.

We also validate metric–human alignment with a small preference study (Figure 3A): on a 50-instance subset, we generate paired renderings $(R1, R2)$ via different IR perturbations and ask 20 annotators to rank the pair. Human preference correlates with the metric score difference $\Delta S = S(R1) - S(R2)$.

Model	Single-Round Generation			Multi-Round Generation		
	$\bar{S}$	*RSR*	*FinalScore*	$\bar{S}$	*RSR*	*FinalScore*
GLM-4.6V(Team et al., 2025)	54.0	6.8%	3.7	77.0	6.8%	5.2
Qwen3-VL-235B-A22B-Instruct(Bai et al., 2025)	62.3	94.7%	59.0	63.5	97.4%	61.9
Claude Sonnet 4.5	**81.4**	90.9%	74.0	82.2	**97.7**%	**80.4**
GPT-5.2	78.3	63.6%	49.8	**84.9**	93.2%	79.1
Gemini 3 Pro	79.6	**100.0**%	**79.6**	81.3	**97.7**%	79.5

Table 2: Model performance on single-round and multi-round generation.

3 BENCHMARKING

3.1 MODELS

We evaluate a set of commercial and open-weight multimodal models under a unified protocol, including GPT-5.2, Gemini 3 Pro, Claude Sonnet 4.5, Qwen3-VL-235B-A22B-Instruct(Bai et al., 2025), and GLM-4.6V(Team et al., 2025). These models cover both closed-source and open-source models, allowing us to compare different levels of multimodal understanding, code generation, and tool-use capability within the same design-to-code setting.

For each model, we provide the same input information, consisting of the exported IR and the reference image. The IR serves as a structural cue, while the reference image defines the target rendering. All models are evaluated using the same execution harness described in Section 2.3. We also provide an identical React scaffold, pinned dependencies, and a shared WriteTool-only interface for file creation and modification. This controlled setup reduces confounding factors from differences in project structure, package versions, or execution environments, and makes the resulting comparison more directly reflect model behavior under the 1D-Bench task definition.

In both the single-round and multi-round settings, the generated workspace is built and rendered by the same harness before metric computation. For multi-round generation, models interact with the same persistent workspace across rounds and receive feedback through the standardized execution interface.

3.2 MAIN RESULTS

Table 2 summarizes single-round and multi-round results using the metrics in Section 2.3. For multi-round generation, $\bar{S}$ and RSR are computed on the final round. In single-round generation, Gemini 3 Pro achieves the best FinalScore of 79.6 with 100.0% RSR, while Claude Sonnet 4.5 attains the highest similarity with $\bar{S} = 81.4$. In multi-round generation, iterative editing generally improves RSR, most notably for GPT-5.2 from 63.6% to 93.2, and yields the largest similarity gain for GPT-5.2 from 78.3 to 84.9. Claude Sonnet 4.5 achieves the best multi-round FinalScore of 80.4, followed by Gemini 3 Pro at 79.5 and GPT-5.2 at 79.1.

Figure 3B-D shows that multi round interaction improves both rendering success rates and similarity scores in most cases.

The comparison between single-round and multi-round generation suggests that execution feedback is particularly useful for models that initially suffer from build or rendering failures. For example, a model may produce an incomplete project in the first round, but subsequent feedback can expose missing files, invalid imports, or layout inconsistencies. The improvement in rendering success therefore indicates that multi-round interaction does not only refine visual details, but also helps models recover from engineering-level errors. This observation supports the need for an evaluation protocol that measures iterative repair rather than only one-pass generation.

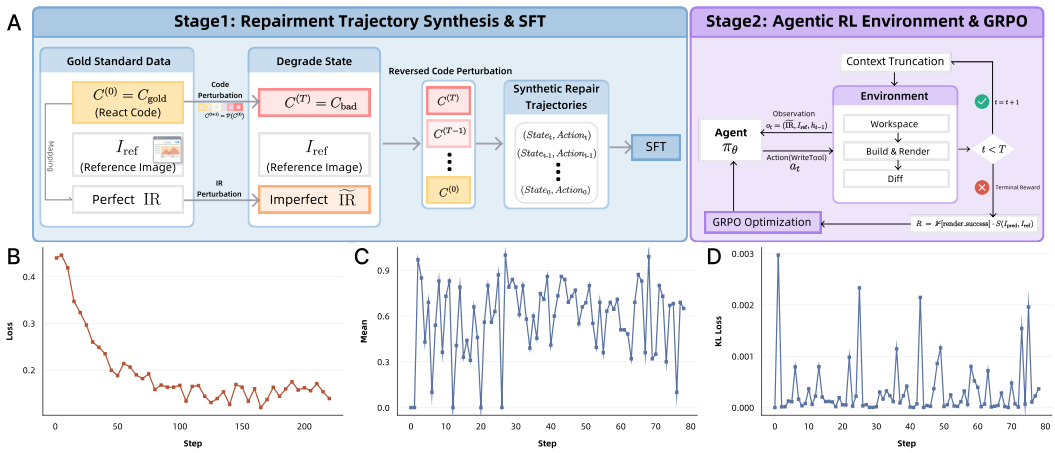


Figure 4: Overview of the pilot post-training study. (A) Synthetic trajectory construction for SFT and the segmented-rollout GRPO setup. (B) SFT training loss. (C) Mean similarity score during GRPO training. (D) KL divergence to the reference policy during GRPO training.

4 Synthetic Repair Data and Agentic RL

4.1 Motivation

Under a fixed toolchain, each output is executable and can be rendered for direct comparison with the reference image (Section 2.3), which makes multi-round editing a natural alternative to single-pass generation. In 1D-Bench, the exported IR provides structural cues but is imperfect, so models may benefit from iterative revision using execution feedback. Since Section 3.2 shows that multi-round editing improves final-round performance in most cases, we conduct a pilot post-training study under the same evaluation interface, with access to the IR, the reference image, and a persistent workspace via WriteTool.

This setting also exposes limitations of purely prompt-based generation. In single-pass D2C, the model must simultaneously infer the intended layout, resolve errors in the exported IR, choose an implementation structure, and produce executable code. Errors made in any of these steps can propagate to the final rendering. In contrast, the multi-round interface decomposes the problem into generation and repair, where later rounds can use concrete execution feedback to correct missing elements, layout shifts, style mismatches, or build failures. This makes iterative D2C closer to a practical front-end development process.

However, effective repair is not guaranteed by instruction following alone. A model must decide which discrepancies are important, identify the corresponding files or components, and update the workspace without damaging previously correct parts. These decisions require a form of interactive policy learning, since the quality of an edit can only be observed after the project is rebuilt and rendered. This motivates our pilot study on post-training: SFT is used to teach the model the format and basic behavior of generation and repair, while reinforcement learning is explored as a way to optimize editing decisions under execution-based feedback.

4.2 Synthetic Repair Trajectories

Figure 4A summarizes the construction of synthetic trajectories for both initial generation and repair under the WriteTool-only interface. We first obtain high-quality reference workspaces C_{gold} from two sources: high-scoring projects collected from an internal platform and projects synthesized by Qwen3-Coder-Plus (Qwen Team, 2026). For each C_{gold}, we render the workspace to produce the reference image I_{ref}, export a corresponding IR, and apply mild perturbations to produce an imperfect IR.

To create repair supervision, we derive a degraded workspace C_{bad} by applying component-level code perturbations that mimic common errors, such as small numeric drifts in style values and localized structural changes in JSX. We render C_{bad} to obtain I_{bad}, together with a similarity score and a diff heatmap relative to I_{ref}. From each pair $(C_{\text{gold}}, C_{\text{bad}})$, we generate two trajectories that share the same endpoint C_{gold}. The first is an initial-generation trajectory conditioned on $(\widetilde{\text{IR}}, I_{\text{ref}})$ that writes the workspace through file overwrites. The second is a repair trajectory conditioned on $(\widetilde{\text{IR}}, I_{\text{ref}}, C_{\text{bad}}, I_{\text{bad}}, \text{score}, \text{diff})$ that restores the perturbed files. The difficulty of repair is controlled by the perturbation types and their coverage.

4.3 Post-training: SFT and Segmented-rollout GRPO

We adopt a two-stage post-training recipe for the multi-round interface of 1D-Bench. Stage 1 performs supervised fine-tuning on synthetic generation and repair traces. Inspired by ReSum(Wu et al., 2025c), Stage 2 applies GRPO with segmented rollouts under context resetting.

Segmented rollouts. Each instance maintains a persistent workspace. A multi-round run is executed as K segments. In segment k, the model observes

$$o_k = \Phi\big(IR, I_{\text{ref}}, C_k, f_{k-1}\big),$$

where C_k is the workspace state and f_{k-1} is the previous segment feedback with $f_0 = \varnothing$. Within the segment, the model emits a sequence of WriteTool actions

$$u_{k,1:m_k} \sim \pi_\theta(\cdot \mid o_k),$$

which deterministically updates the workspace,

$$C_{k+1} = T\big(C_k, u_{k,1:m_k}\big).$$

The harness then builds and renders C_{k+1} to produce feedback f_k. Before segment $k+1$, the dialogue context is cleared while the workspace C_{k+1} is kept. We set $K = 3$ to match evaluation.

Terminal reward. A run yields a segmented trajectory $\tau = \{(o_k, u_{k,1:m_k}, f_k)\}_{k=1}^{K}$. We use a terminal-only reward from the final segment,

$$R(\tau) = \mathbb{1}[\text{render_success}]\; S(I_{\text{pred}}, I_{\text{ref}}) \in [0, 1].$$

and treat intermediate scores only as feedback.

Segmented GRPO with advantage broadcasting. For each instance we sample G trajectories $\{\tau_g\}_{g=1}^{G}$ and compute $R_g = R(\tau_g)$. We form a trajectory-level advantage (Shao et al., 2024)

$$\hat{A}_g = \frac{R_g - \text{mean}(\{R_1, \ldots, R_G\})}{\text{std}(\{R_1, \ldots, R_G\})},$$

and broadcast it to all segments in the same trajectory. Let $r_{g,k}(\theta)$ be the GRPO importance ratio for tokens generated in segment k of trajectory g. The objective is

$$J_{\text{seg-GRPO}}(\theta) \quad = \quad \mathbb{E}\Big[\frac{1}{GK}\sum_{g=1}^{G}\sum_{k=1}^{K}\min\big(r_{g,k}(\theta)\hat{A}_g, \text{clip}\big(r_{g,k}(\theta),\, 1 \;-\; \epsilon,\, 1 \;+\; \epsilon\big)\hat{A}_g\big)\Big].$$

This formulation matches the evaluation protocol by resetting context between segments while learning from a single final-round score.

4.4 Results

Figure 4B shows the SFT loss decreasing and then plateauing in Qwen3-VL-32B, indicating successful fitting of synthetic trajectories (tool-call format and edit patterns). However, this does not necessarily translate to better 1D-Bench performance, since the supervision mainly encourages reproducing trajectory structure rather than improving multi-round strategies.

During GRPO in Qwen3-VL-32B, Figure 4C shows that the mean similarity score fluctuates without a sustained upward trend, suggesting no consistent improvement. Figure 4D reports near-zero KL divergence for most updates, implying limited deviation from the reference policy. Together, these results indicate weak and unstable learning.

A plausible explanation is that the learning signal is dominated by sparse terminal rewards and discontinuous failure modes. In addition, WriteTool actions are implemented as full-file overwrites, which introduces high-variance macro edits that make credit assignment across rounds difficult. These properties can reduce the effective signal-to-noise ratio of policy gradients, leading to unstable optimization and limited policy change.

These findings indicate that post-training for iterative D2C repair requires more fine-grained learning signals than the current pilot setup provides. In our setting, a single terminal reward must explain a sequence of file edits across multiple rounds, although the final score may be affected by only a small subset of changes. This mismatch makes it difficult to identify which edit improves or harms the rendering. Future training methods may benefit from denser component-level rewards, more localized edit actions, or reward decomposition that separately accounts for executability, layout alignment, text correctness, and style consistency.

5 Conclusion

We present 1D-Bench, a real-world benchmark derived from e-commerce development workflows for evaluating design-to-code under practical constraints. Each instance provides both a reference rendering as the target, and an exported, potentially noisy IR as structural cues. It requires generating an executable React codebase under a fixed toolchain with an explicit component hierarchy. We further standardize a multi-round setting where models iteratively apply component-level edits using execution feedback.

Across commercial and open-weight MLLMs, we find that multi-round generation generally increases final performance. Finally, we conduct a pilot study for iterative D2C using synthetic repair trajectories and agentic GRPO. While SFT fits synthetic edit patterns, RL yields limited improvements, highlighting challenges from sparse terminal rewards and high-variance file-level updates. We hope 1D-Bench enables more comparable progress and motivates learning methods better aligned with iterative, executable front-end generation.

AI use statement

The authors used AI-assisted tools solely for limited assistance in writing a small portion of the experimental code. All such code was carefully reviewed, tested, and revised by the authors before use. The experimental designs, results, analyses, and conclusions were independently produced and verified by the authors.

Ethics statement

This work does not involve human subjects, personal data collection, or experiments on users. The benchmark was constructed from internal design-to-code workflow data after anonymization, automated filtering, and manual review to remove incomplete, duplicate, non-UI, and potentially privacy-sensitive or non-compliant records. We do not release any raw proprietary content that could identify individuals or expose sensitive information. The proposed benchmark and evaluation protocol are intended for research on executable UI code generation and iterative editing, and we do not anticipate direct harmful use beyond the general dual-use risks common to code-generation research. We have followed the ICLR Code of Ethics in the preparation of this work.

Reproducibility statement

The data and scripts used in this study are available in an anonymized repository at https://anonymous.4open.science/r/d2c-benchmark-A9C4/.

REFERENCES

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, Wenbin Ge, Zhifang Guo, Qidong Huang, Jie Huang, Fei Huang, Binyuan Hui, Shutong Jiang, Zhaohai Li, Mingsheng Li, Mei Li, Kaixin Li, Zicheng Lin, Junyang Lin, Xuejing Liu, Jiawei Liu, Chenglong Liu, Yang Liu, Dayiheng Liu, Shixuan Liu, Dunjie Lu, Ruilin Luo, Chenxu Lv, Rui Men, Lingchen Meng, Xuancheng Ren, Xingzhang Ren, Sibo Song, Yuchong Sun, Jun Tang, Jianhong Tu, Jianqiang Wan, Peng Wang, Pengfei Wang, Qiuyue Wang, Yuxuan Wang, Tianbao Xie, Yiheng Xu, Haiyang Xu, Jin Xu, Zhibo Yang, Mingkun Yang, Jianxin Yang, An Yang, Bowen Yu, Fei Zhang, Hang Zhang, Xi Zhang, Bo Zheng, Humen Zhong, Jingren Zhou, Fan Zhou, Jing Zhou, Yuanzhi Zhu, and Ke Zhu. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*, 2025.

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

Yunnong Chen, Xinyu Yu, Shixian Ding, Yingying Zhang, Chengwei Shi, Jingzhou Du, and Liuqing Chen. Designcoder: Hierarchy-aware and self-correcting ui code generation with large language models. *Inf. Softw. Technol.*, 198(C), August 2026. ISSN 0950-5849. doi: 10.1016/j.infsof.2026.108214. URL https://doi.org/10.1016/j.infsof.2026.108214.

Sidong Feng, Mingyue Yuan, Jieshan Chen, Zhenchang Xing, and Chunyang Chen. Designing with language: Wireframing ui design intent with generative large language models, 2023. URL https://arxiv.org/abs/2312.07755.

Tong Ge, Yashu Liu, Jieping Ye, Tianyi Li, and Chao Wang. Advancing vision-language models in front-end development via data synthesis, 2025. URL https://arxiv.org/abs/2503.01619.

Yi Gui, Zhen Li, Yao Wan, Yemin Shi, Hongyu Zhang, Yi Su, Bohua Chen, Dongping Chen, Siyuan Wu, Xing Zhou, Wenbin Jiang, Hai Jin, and Xiangliang Zhang. Webcode2m: A real-world dataset for code generation from webpage designs. In *THE WEB CONFERENCE*, 2025a.

Yi Gui, Zhen Li, Zhongyi Zhang, Guohao Wang, Tianpeng Lv, Gaoyang Jiang, Yi Liu, Dongping Chen, Yao Wan, Hongyu Zhang, Wenbin Jiang, Xuanhua Shi, and Hai Jin. 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, pp. 721–732, 2025b.

Yi Gui, Yao Wan, Zhen Li, Zhongyi Zhang, Dongping Chen, Hongyu Zhang, Yi Su, Bohua Chen, Xing Zhou, Wenbin Jiang, and Xiangliang Zhang. Uicopilot: Automating ui synthesis via hierarchical code generation from webpage designs. In *Proceedings of the ACM on Web Conference 2025*, WWW '25, pp. 1846–1855, 2025c.

Hongcheng Guo, Wei Zhang, Junhao Chen, Yaonan Gu, Jian Yang, Junjia Du, Shaosheng Cao, Binyuan Hui, Tianyu Liu, Jianxin Ma, et al. Iw-bench: Evaluating large multimodal models for converting image-to-web. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 6449–6466, 2025.

Dongfu Jiang, Yi Lu, Zhuofeng Li, Zhiheng Lyu, Ping Nie, Haozhe Wang, Alex Su, Hui Chen, Kai Zou, Chao Du, et al. Verltool: Towards holistic agentic reinforcement learning with tool use. *arXiv preprint arXiv:2509.01055*, 2025.

Abidullah Khan, Atefeh Shokrizadeh, and Jinghui Cheng. Beyond automation: How designers perceive ai as a creative partner in the divergent thinking stages of ui/ux design. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25, 2025.

Gang Li and Yang Li. Spotlight: Mobile UI understanding using vision-language models with a focus. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=9yE2xEj0BH7.

Guochang Li, Yuchen Liu, Zhen Qin, Yunkun Wang, Jianping Zhong, Chen Zhi, Binhua Li, Fei Huang, Yongbin Li, and Shuiguang Deng. Empowering repoqa-agent based on reinforcement learning driven by monte-carlo tree search, 2025a. URL https://arxiv.org/abs/2510.26287.

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)*, pp. 3921–3955, April 2025b.

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)*, pp. 24786–24802, 2025.

Qiqiang Lin, Muning Wen, Qiuying Peng, Guanyu Nie, Junwei Liao, Jun Wang, Xiaoyun Mo, Jiamu Zhou, Cheng Cheng, Yin Zhao, Jun Wang, and Weinan Zhang. Robust function-calling for on-device language model via function masking. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=yVQcr4qjD6.

Weiwen Liu, Xu Huang, Xingshan Zeng, xinlong hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong WANG, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Wang Xinzhi, Yong Liu, Yasheng Wang, Duyu Tang, Dandan Tu, Lifeng Shang, Xin Jiang, Ruiming Tang, Defu Lian, Qun Liu, and Enhong Chen. ToolACE: Winning the points of LLM function calling. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=8EB8k6DdCU.

Yadong Lu, Jianwei Yang, Yelong Shen, and Ahmed Awadallah. Omniparser for pure vision based gui agent, 2024. URL https://arxiv.org/abs/2408.00203.

Oorja Majgaonkar, Zhiwei Fei, Xiang Li, Federica Sarro, and He Ye. Understanding code agent behaviour: An empirical study of success and failure trajectories, 2025. URL https://arxiv.org/abs/2511.00197.

Savvas Petridis, Michael Terry, and Carrie Jun Cai. Promptinfuser: Bringing user interface mockups to life with large language models. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI EA '23. Association for Computing Machinery, 2023.

Qwen Team. Qwen3-coder-next technical report. Technical report, Alibaba Group, 2026. URL https://github.com/QwenLM/Qwen3-Coder/blob/main/qwen3_coder_next_tech_report.pdf. Accessed: 2026-02-03.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

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)*, pp. 3956–3974, April 2025.

Haoyu Sun, Huichen Will Wang, Jiawei Gu, Linjie Li, and Yu Cheng. Fullfront: Benchmarking mllms across the full front-end engineering workflow, 2025. URL https://arxiv.org/abs/2505.17399.

5 Team, Aohan Zeng, Xin Lv, and Others. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models, 2025. URL https://arxiv.org/abs/2508.06471.

FAIR CodeGen team, Jade Copet, Quentin Carbonneaux, and Others. Cwm: An open-weights llm for research on code generation with world models, 2025. URL https://arxiv.org/abs/2510.02387.

Kimi Team. Kimi k2: Open agentic intelligence, 2025. URL https://arxiv.org/abs/2507.20534.

V Team, Wenyi Hong, Wenmeng Yu, Xiaotao Gu, Guo Wang, Guobing Gan, Haomiao Tang, Jiale Cheng, Ji Qi, Junhui Ji, Lihang Pan, Shuaiqi Duan, Weihan Wang, Yan Wang, Yean Cheng, Zehai He, Zhe Su, Zhen Yang, Ziyang Pan, Aohan Zeng, Baoxu Wang, Bin Chen, Boyan Shi, Changyu Pang, Chenhui Zhang, Da Yin, Fan Yang, Guoqing Chen, Jiazheng Xu, Jiale Zhu, Jiali Chen, Jing Chen, Jinhao Chen, Jinghao Lin, Jinjiang Wang, Junjie Chen, Leqi Lei, Letian Gong, Leyi Pan, Mingdao Liu, Mingde Xu, Mingzhi Zhang, Qinkai Zheng, Sheng Yang, Shi Zhong, Shiyu Huang, Shuyuan Zhao, Siyan Xue, Shangqin Tu, Shengbiao Meng, Tianshu Zhang, Tianwei Luo, Tianxiang Hao, Tianyu Tong, Wenkai Li, Wei Jia, Xiao Liu, Xiaohan Zhang, Xin Lyu, Xinyue Fan, Xuancheng Huang, Yanling Wang, Yadong Xue, Yanfeng Wang, Yanzi Wang, Yifan An, Yifan Du, Yiming Shi, Yiheng Huang, Yilin Niu, Yuan Wang, Yuanchang Yue, Yuchen Li, Yutao Zhang, Yuting Wang, Yu Wang, Yuxuan Zhang, Zhao Xue, Zhenyu Hou, Zhengxiao Du, Zihan Wang, Peng Zhang, Debing Liu, Bin Xu, Juanzi Li, Minlie Huang, Yuxiao Dong, and Jie Tang. Glm-4.5v and glm-4.1v-thinking: Towards versatile multimodal reasoning with scalable reinforcement learning, 2025. URL https://arxiv.org/abs/2507.01006.

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, 2024. URL https://arxiv.org/abs/2412.15310.

Yuxuan Wan, Chaozheng Wang, Yi Dong, Wenxuan Wang, Shuqing Li, Yintong Huo, and Michael Lyu. Divide-and-conquer: Generating ui code from screenshots. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025.

Huanting Wang, Jingzhi Gong, Huawei Zhang, Jie Xu, and Zheng Wang. Ai agentic programming: A survey of techniques, challenges, and opportunities, 2025a. URL https://arxiv.org/abs/2508.11126.

Weixun Wang, Shaopan Xiong, et al. Reinforcement learning optimization for large-scale learning: An efficient and user-friendly scaling library. *arXiv preprint arXiv:2506.06122*, 2025b.

Hao Wen, Shizuo Tian, Borislav Pavlov, Wenjie Du, Yixuan Li, Ge Chang, Shanhui Zhao, Jiacheng Liu, Yunxin Liu, Ya-Qin Zhang, and Yuanchun Li. *AutoDroid-V2: Boosting SLM-based GUI Agents via Code Generation*, pp. 223–235. Association for Computing Machinery, 2025.

Fan Wu, Cuiyun Gao, Shuqing Li, Xin-Cheng Wen, and Qing Liao. Mllm-based ui2code automation guided by ui layout information. *Proc. ACM Softw. Eng.*, 2(ISSTA), June 2025a.

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 *NAACL*, 2025b. URL https://arxiv.org/abs/2406.07739.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, et al. Resum: Unlocking long-horizon search intelligence via context summarization. *arXiv preprint arXiv:2509.13313*, 2025c.

Bang Xiao, Lingjie Jiang, Shaohan Huang, Tengchao Lv, Yupan Huang, Xun Wu, Lei Cui, and Furu Wei. Code aesthetics with agentic reward feedback. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=Q87kwGI6bx.

Yang Xiao, Mohan Jiang, Jie Sun, Keyu Li, Jifan Lin, Yumin Zhuang, Ji Zeng, Shijie Xia, Qishuo Hua, Xuefeng Li, Xiaojie Cai, Tongyu Wang, Yue Zhang, Liming Liu, Xia Wu, Jinlong Hou, Yuan Cheng, Wenjie Li, Xiang Wang, Dequan Wang, and Pengfei Liu. Limi: Less is more for agency, 2025. URL https://arxiv.org/abs/2509.17567.

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, 2025. URL https://arxiv.org/abs/2511.06251.

An Yang, Anfeng Li, and Others. Qwen3 technical report, 2025a. URL https://arxiv.org/abs/2505.09388.

Bo Yang and Shanping Li. Uisgpt: Automated mobile ui design smell detection with large language models. *Electronics*, 13(16), 2024.

Zhen Yang, Wenyi Hong, Mingde Xu, Xinyue Fan, Weihan Wang, Jiele Cheng, Xiaotao Gu, and Jie Tang. Ui2code^{N}: A visual language model for test-time scalable interactive ui-to-code generation, 2025b. URL https://arxiv.org/abs/2511.08195.

Keen You, Haotian Zhang, Eldon Schoop, Floris Weers, Amanda Swearngin, Jeffrey Nichols, Yinfei Yang, and Zhe Gan. Ferret-ui: Grounded mobile ui understanding with multimodal llms. In *Computer Vision – ECCV 2024: 18th European Conference, Milan, Italy, September 29–October 4, 2024, Proceedings, Part LXIV*, pp. 240–255, 2024.

Chengyue Yu, Siyuan Lu, Chenyi Zhuang, Dong Wang, Qintong Wu, Zongyue Li, Runsheng Gan, Chunfeng Wang, Siqi Hou, Gaochi Huang, Wenlong Yan, Lifeng Hong, Aohui Xue, Yanfeng Wang, Jinjie Gu, David Tsai, and Tao Lin. Aworld: Orchestrating the training recipe for agentic ai, 2025. URL https://arxiv.org/abs/2508.20404.

Sukmin Yun, Haokun Lin, Rusiru Thushara, Mohammad Qazim Bhat, Yongxin Wang, Zutao Jiang, Mingkai Deng, Jinhong Wang, Tianhua Tao, Junbo Li, Haonan Li, Preslav Nakov, Timothy Baldwin, Zhengzhong Liu, Eric P. Xing, Xiaodan Liang, and Zhiqiang Shen. Web2code: a large-scale webpage-to-code dataset and evaluation framework for multimodal llms. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, 2024.

Zizheng Zhan, Ken Deng, and Others. Kat-coder technical report, 2025. URL https://arxiv.org/abs/2510.18779.

Guibin Zhang, Hejia Geng, Xiaohang Yu, Zhenfei Yin, Zaibin Zhang, Zelin Tan, Heng Zhou, Zhongzhi Li, Xiangyuan Xue, Yijiang Li, Yifan Zhou, Yang Chen, Chen Zhang, Yutao Fan, Zihu Wang, Songtao Huang, Francisco Piedrahita-Velez, Yue Liao, Hongru Wang, Mengyue Yang, Heng Ji, Michael Littman, Jun Wang, Shuicheng Yan, Philip Torr, and Lei Bai. The landscape of agentic reinforcement learning for llms: A survey, 2025. URL https://arxiv.org/abs/2509.02547.

Bingxi Zhao, Lin Geng Foo, Ping Hu, Christian Theobalt, Hossein Rahmani, and Jun Liu. Llm-based agentic reasoning frameworks: A survey from methods to scenarios, 2025. URL https://arxiv.org/abs/2508.17692.

Ting Zhou, Yanjie Zhao, Xinyi Hou, Xiaoyu Sun, Kai Chen, and Haoyu Wang. Declarui: Bridging design and development with automated declarative ui code generation. *Proc. ACM Softw. Eng.*, 2(FSE), June 2025.

Siyu Zhu, Anastasiya Karpovich, Albert Chen, Jessica Koscheka, Shailesh Jannu, Di Wen, Yuqing Zhu, Rohit Jain, and Alborz Geramifard. Agentic reinforcement learning for real-world code repair, 2025. URL https://arxiv.org/abs/2510.22075.

Appendix

A Related Works

A.1 Design-to-Code

With the advancements in the programming capabilities of code LLMs, the design-to-code (D2C) task has garnered renewed interest from researchers in both academia and industry(Khan et al., 2025). Feng used LLMs to generate mid-fidelity wireframes in the UI design process(Feng et al., 2023). Petridis developed a Figma plugin that enables designers to author LLM-infused mock-ups(Petridis et al., 2023). Zhou proposed an automated approach that synergizes computer vision, MLLMs, and iterative compiler-driven optimization to generate and refine declarative UI code from designs(Zhou et al., 2025). Wu used a code compiler and a pre-trained VLM to finetune LLMs to

generate UI code from user-provided textual descriptions(Wu et al., 2025b). Li used LLMs to detect UI design smells and explain each violation of specific design guidelines in natural language(Yang & Li, 2024). Wen presented a document-guided, script-based, end-to-end system named AutoDroid-V2 to support mobile task automation using on-device SLMs(Wen et al., 2025). Chen proposed a hierarchy-aware and vision-guided self-correcting approach for generating high-quality UI code from design mockups(Chen et al., 2026). Wan proposed a divide-and-conquer approach to automate the translation of webpage design to UI code based on MLLMs(Wan et al., 2025). Gui generated UI code from webpage designs by coarse DOM tree prediction and fine-grained code synthesis(Gui et al., 2025c). Xu built a web-based vLLM-based agentic framework for interactive and verifiable UI-to-code generation(Xu et al., 2025). Liang proposed a fine-tuning pipeline for UI-to-HTML code generation based on transformer-based MLLMs(Liang et al., 2025). However, LLMs struggle to consistently generate UI code that compiles and produces visually relevant designs, and lack fine-grained editing capabilities. One reason is that base vLLMs are not trained on large-scale D2C related data, leading to insufficient coding and visual comprehension abilities. Another reason is that existing training paradigms typically rely on single-round interactions and therefore do not adequately support D2C tasks that require iterative interactions with multiple turns(Yang et al., 2025b; Wu et al., 2025a).

A.2 Agentic Learning

Agentic learning aims to identify effective training methods that enable LLM-based agents to learn from interaction processes and their associated feedback, thereby enhancing their ability to solve real-world problems. The distinction between agent learning and LLM learning lies mainly in two aspects(Zhu et al., 2025; Zhang et al., 2025; Zhao et al., 2025). First, agent learning involves not only processing text tokens but also handling rich and dynamic contextual environmental information. Second, it requires multi-round interactions with the environment through external tools and API calls(Liu et al., 2025; Lin et al., 2025). In the area of AI programming, the agentic training paradigm has been widely adopted in foundational LLMs, yet its application to concrete programming tasks remains relatively uncommon(Wang et al., 2025a). KAT-Coder adopted agentic RL to achieve efficient multi-trajectory optimization, enhanced exploration, and robust policy diversity(Zhan et al., 2025). GLM-4.5 was trained on agent data in the mid-training stage to increase its capabilities in agentic tasks and coding(Team et al., 2025). Kimi K2 synthesized large-scale agentic data and used agentic RL in the post-training(Team, 2025). Qwen3 was also trained on agentic tasks in the post-training(Yang et al., 2025a). CWM released by Meta is an open-weights LLM for research on code generation with world models(team et al., 2025). Beyond base LLMs, Yu proposed AWorld to orchestrate the training recipe for agentic AI(Yu et al., 2025). Jiang introduced VERLTOOL to address key limitations of agentic RL with tool use in model training(Jiang et al., 2025). Xiao's work demonstrated the less is more paradigm for intelligent agency by using only 78 carefully designed training samples(Xiao et al., 2025). Xiao built a dataset and benchmark of code aesthetic and introduced an agentic reward framework for code generation(Xiao et al., 2026). Majgaonkar analysed trajectories from three state-of-the-art code agents (OpenHands, SWE-agent, and Prometheus) on the SWE-Bench benchmark(Majgaonkar et al., 2025). Li introduced RepoSearch-R1, a novel agentic RL framework driven by Monte-carlo tree search for repository-level software engineering tasks(Li et al., 2025a). It can be observed that agentic learning is still in its early exploratory phase, with very few studies specifically applying it to D2C tasks, and training data for agentic learning in D2C remain extremely scarce.

B Implementation of Data Construction

We implement data construction as a two-stage pipeline. We first perform automatic cleaning to remove invalid, non UI, privacy risk, and near duplicate records. We then stratify the cleaned pool to form a balanced evaluation subset. Each raw record contains a reference rendering treated as reference target paired with an exported structural IR that may be imperfect.

B.1 Automatic cleaning

We drop records with missing or unreadable artifacts. The remaining records are filtered in a fixed order:

1. aspect ratio in $[1/4.5, 4.5]$
2. near-duplicate removal using CLIP (ViT-B-32), remove if cosine similarity ≥ 0.95
3. transparent background ratio ≤ 0.1
4. low-variation images, remove if `std_mean` < 10 and (`unique_ratio` < 0.05 or `entropy` < 1.5)
5. VLM full-page screen with decision in keep, remove, and review
6. VLM poster or illustration screen with decision in keep, remove, and review
7. privacy compliance via OCR plus pattern-based PII detection, followed by a VLM privacy screen

All `review` cases are manually inspected and we retain only records confirmed to be UI relevant and safe.

B.2 IR QUALITY LABELING AND STRATIFIED SAMPLING

We compute IR statistics and define a coarse complexity label by the 33rd/66th percentiles of node counts (easy/mid/hard). We then conduct a manual review to confirm and correct the complexity label for each task. We estimate IR reliability by rendering the exported IR in a deterministic HTML renderer, capturing an IR-based screenshot with a headless browser, and comparing it against the reference image. We label IR quality as `perfect` if similarity is ≥ 0.95 and `imperfect` otherwise. Platform (`desktop`/`mobile`/`component`) and primary language (`Chinese`/`English`/`other`) are labeled by a VLM. All VLM-based screening and labeling steps use a locally deployed Qwen3-VL-30B-A3B-Instruct model running on 4 H20 GPUs.

B.3 FILTERING PROMPTS

We use a small set of VLM screening prompts to filter non page samples and obvious privacy risks (Figure 5). Each prompt requests a minimal JSON object with fields `decision` and `reason`. The `decision` is one of keep, remove, review. Review cases are sent to manual inspection. All screening is performed by the locally deployed Qwen3-VL-30B-A3B-Instruct model on 4 H20 GPUs, and no data are transmitted outside the local environment.

C IMPLEMENTATION OF METRICS

We adopt an execution-based evaluation protocol. For each generated React project, we build and render it to obtain a deterministic screenshot. We then compute a similarity score against the reference image. The scoring function follows the composite design deployed in our production evaluation to ensure stable and reproducible measurements.

Rendering success rate. We treat executability as a first-class metric. A run is successful if the harness produces a nonempty screenshot within a fixed timeout. Otherwise the run is a rendering failure. We denote the rendering success rate by *RSR*. The harness performs the following steps.

1. **Deterministic build**. Clean stale build artifacts, install dependencies under a fixed toolchain, and build the project with a timeout.
2. **Static serving**. Serve the built static assets via a lightweight HTTP server on a randomly selected free port.
3. **Headless screenshot**. Use Playwright with a Chromium backend to capture a screenshot of the root container. We wait for `networkidle`, then resize the browser viewport to the tight bounding box of root container and its descendants to reduce cropping caused by fixed or absolute positioning.

We retry screenshot capture up to three times. The rendering success rate is the fraction of tasks that are successful.

Prompt for VLM UI Filtering

```
You are the UI Screenshot filtering assistant. Your task is to review the current page screenshots and filter
out the obviously incomplete page segments.
Your response format is JSON, containing only the "decision" and "reason" fields.

Only when you are absolutely certain that the following conditions are met will decision = keep:
1) It is clearly a complete page layout or a complete component (including multiple
areas/modules/navigation/lists/forms/buttons, etc.);
2) It is not dominated by a single large image/poster/photography/illustration;
3) It is not a partial cropped fragment (such as only cutting to a button, a corner, a background, an icon,
etc.).
If it is clearly not met, then decision = remove; if unsure, then decision = review.
Output in JSON format: {\"decision\":\"keep|remove|review\",\"reason\":\"... \"}
You can enclose it with the ```json code block.

Current UI Screenshot:
<IMAGE>
```

Prompt for VLM Poster Filtering

```
You are the UI Screenshot filtering assistant. Determine whether the image is a poster, illustration,
photography, or other non-UI page. Only filter when the entire page is almost entirely a poster, illustration,
or photography and cannot be implemented as a page or component. The response must be in JSON format,
containing only the "decision" and "reason" fields.

Please determine whether this image is a screenshot of a UI page.
1) If it is definitely a screenshot of a UI page, then decision = keep;
2) If the entire page is almost entirely composed of posters, marketing materials, photography images, or
illustrations and cannot be used as a page or component, then decision = remove;
3) If it is a marketing promotion component, product introduction page, product promotion homepage, flowchart
explanatory diagram, product demonstration diagram, etc. in the page, it should still have decision = keep.
5) If it is uncertain, then decision = review.
Output in JSON format: {"decision":"keep|remove|review","reason":"..."}
You can wrap it with a ```json code block.

Current UI Screenshot:
<IMAGE>
```

Prompt for VLM Compliance Review

```
You are a compliance and desensitization review assistant responsible for an initial privacy-risk screening
of images. Filter out information that is very obvious and directly identifies a real
individual/company/business. You must return JSON with only two fields: decision and reason.

Note: this step is only an initial screening. There will be strict manual review later, so your filtering
principle is to avoid false positives. Set decision=remove when there is very obvious and directly real
sensitive information.

Examples that must be removed: national ID / certificate numbers, full phone numbers, real email addresses,
real names, addresses precise down to a door number, real bank card / payment account numbers, real license
plates, full legal names of real companies, real business statistics (e.g., GMV/revenue/user count).

Examples that do NOT need to be removed: addresses only down to district level, virtual nicknames or
placeholder names/titles (Zhang San / Mr. Wang / nick / Shanshan / Sun Wukong), virtual IDs (e.g., 123456),
dummy emails (example@example.com), occupations (software engineer), code snippets, partially redacted
company info (*** Co., Ltd.), product names (e.g., "Watermelon Bobo" / "cookie biscuits"), etc. These should
not be filtered.

You must output one of decision=keep/remove/review and provide a brief reason. Output in JSON format:
{"decision":"keep|remove|review","reason":"..."} (you may wrap it in a ```json code block).

Current UI Screenshot:
<IMAGE>
```

Figure 5: Prompt for VLM UI check

Visual similarity normalization. Given the rendered screenshot I_{pred} and reference image I_{ref}, the scorer outputs a similarity $S \in [0, 1]$. We also generate a diff heatmap that visualizes pixel level discrepancies. The scoring pipeline is modular. If OCR, SSIM, or LPIPS cannot be computed, the corresponding term is omitted and the remaining weights are renormalized over the available signals.

Image preprocessing and size alignment. Before detection and matching, we apply transparency-aware preprocessing. If an image has an alpha channel, pixels with $\alpha < \tau_k$ are treated as background and set to white in grayscale space. For transparent images, we evaluate multiple alpha thresholds $\tau_k \in \{20, 40, 60, 80, 100, 150\}$ and merge detections across thresholds; for non-transparent/general processing, we use a default alpha threshold of 200. For very dark foreground regions, we apply brightness amplification and contrast stretching. To avoid upscaling, we set $w = \min(w_{\text{ref}}, w_{\text{pred}})$ and $h = \min(h_{\text{ref}}, h_{\text{pred}})$, and resize both images to (w, h). We run all detectors on these aligned images.

Element-level completeness for text blocks. We detect text blocks in both images using an OCR pipeline that combines a DBNet text detector and a CRNN recognizer. We set `max_side_len=960` and a detection score threshold of `0.6`. Each detection yields a bounding box and a recognized string. We match each reference text block to at most one screenshot text block using a weighted similarity:

$$s_{\text{text}} = 0.6\, s_{\text{content}} + 0.3\, s_{\text{pos}} + 0.1\, s_{\text{size}},$$

where s_{content} is a normalized edit distance similarity over lowercased strings, s_{pos} is a center distance similarity normalized by the image diagonal, and s_{size} is an area ratio similarity. We accept matches when $s_{\text{text}} \geq 0.5$ and resolve spatial conflicts to enforce a one to one assignment. We avoid hard binarization by mapping the match confidence through a piecewise saturation function $f(\cdot)$, and we average $f(s_{\text{text}})$ over all reference text blocks. Unmatched blocks contribute zero.

Element-level completeness for nontext blocks. To capture nontext visual regions, we detect blocks using multi scale Canny edge detection followed by contour extraction and bounding boxes. For transparent images, we use multiple Canny thresholds and exposure or gamma sweeps, then merge edge maps by pixelwise maximum and apply morphological closing. Candidate blocks are filtered by area, overlap is suppressed using non maximum suppression, and blocks overlapping OCR regions are removed to avoid double counting. For each reference block, we crop a grayscale template from the reference image and search it in the predicted image using normalized cross correlation template matching. We use a match threshold around 0.5, and we additionally check the original location with a confidence threshold of ≥ 0.8. As with text, we enforce a one to one assignment with spatial conflict handling, and we aggregate block completeness by averaging the saturated confidence.

Layout score. Presence alone can be insufficient when elements drift. Let $c(b)$ denote the center of a box b, and let $\delta(b)$ denote its diagonal length. We define the layout similarity as

$$\begin{aligned} c_i &= c(b_i), \\ \bar{\delta} &= \tfrac{1}{2}\big(\delta(b_1) + \delta(b_2)\big), \\ s_{\text{layout}}(b_1, b_2) &= \max\left(0, 1 - \frac{d(c_1, c_2)}{\bar{\delta}}\right). \end{aligned}$$

We average these scores over reference text blocks and over reference nontext blocks. When both types are present, we combine them with weights 0.6 for text and 0.4 for nontext. Otherwise the available type receives weight 1.0.

Perceptual image similarity. In addition to element cues, we compute image level similarity signals, each mapped to $[0, 1]$. These include SSIM, a normalized pixel difference score based on MSE, MAE, or RMSE, and LPIPS. LPIPS distances d are converted to similarity by $\exp(-d)$. All signals reuse the same preprocessing to improve robustness under transparency. We use weights $w_{\text{lpips}} = 0.8$, $w_{\text{ssim}} = 0.1$, and $w_{\text{pix}} = 0.1$, and renormalize them over the available methods.

Prompt for Initial React Code Generation

```
You are a professional frontend engineer. Your task is to generate React code based on the provided IR and
design image.

You will receive two types of input: a JSON-format IR (describing the page's component hierarchy and
properties) and a design image (showing the page's visual appearance). You need to generate a React project
source code that can compile successfully and strictly meet the following structure and constraints.

You must generate and only generate the following file structure:
- src/App.tsx - main application component
- src/components/{ComponentName}/index.tsx - component implementation file
- src/components/{ComponentName}/index.module.css - component style file

Technology and organization requirements:
- Use TypeScript and React for development, with 2 spaces for indentation; do not use any other tech stack
- Styles must use CSS Module (CSS Modules); inline styles and style attributes are forbidden
- Components must be in separate folders and written under src/components/{ComponentName}/, where
{ComponentName} is the component name; component names should be semantically clear but not overly complex,
and the number of components should be based on actual needs
- No need to write project configuration files, only write the files listed above; the number of components
must be correct, and if the code references a component, you must implement it
- Always write complete file contents; placeholders like "keep unchanged" or "omitted" are forbidden; you
must ensure the code is correct to compile successfully
- All <img> tags must use the real URL from the IR; do not replace with other URLs

Import path requirements (must match exactly):
- When App.tsx references components, write: import ComponentName from './components/ComponentName', and the
path must reference components under the current path
- When a component's index.tsx imports other components, write: import ComponentName from '../ComponentName',
and the path must reference other components under the parent path
- When a component's index.tsx imports its css, write and only write: import styles from './index.module.css'
- All tsx files must import React at the top, i.e., "import React from 'react';"

Before writing code, first analyze the IR and the design image and describe your thought process. You need to
describe the main nodes and component structure in the IR, note potential problems in the IR (such as
positional offsets or inaccurate properties; note that the IR may not necessarily have issues), and explain
how you will handle these potential problems. Then describe the overall layout and visual appearance of the
design image, identify the main UI elements and components, and note visual details such as colors, fonts,
and spacing. Finally, explain how you will organize the code structure, split components, and implement
styles to match the design image; if you determine there are problems in the IR, you also need to explain how
you will correct them in the code.

After analysis, do not wait for user instructions and directly call the write_file tool to write code. You
need to analyze the current situation step by step and what to do next, then use the write_file tool to
generate the complete code that meets all requirements above; if multiple files need to be written, you can
output multiple tool calls in one response and call write_file in order. Unless you believe you have already
written all required code, every response (including the first) must include your analysis and tool calls.

IR:
<IR>

The design image:
<IMAGE>
```

Figure 6: Prompt for Initial React Code Generation

Composite score and final reporting. The final similarity score is a weighted combination:

$$S = 0.5\, S_{\text{img}} + 0.3\, S_{\text{comp}} + 0.2\, S_{\text{layout}},$$

where S_{comp} averages text and nontext completeness when both exist, and otherwise uses the available type. If neither text nor nontext blocks are detected, we set $S = S_{\text{img}}$. As defined in Section 2.3, we report $\bar{S}$ over successfully rendered instances and the rendering success rate *RSR*. The summary score is $\textit{FinalScore} = \bar{S} \cdot \textit{RSR}$.

Preference-based calibration. To validate the automated score against human judgments, we run a pairwise preference study on synthetic perturbations. Starting from an IR to HTML rendering, we generate two perturbed variants, denoted R1 and R2, using controlled DOM and style edits such as sibling swaps, node moves, and numeric CSS drifts. We render both variants and score each against the same reference image. Annotators view the reference and the two variants side by side and select whether R1 or R2 is closer, or choose uncertain. With multiple annotators, we compute majority vote and consensus. We report agreement accuracy between the metric winner and the human majority, a calibration curve that relates $P(\text{human picks R1})$ to $\Delta\text{score} = \text{score(R1)} - \text{score(R2)}$, and disagreement as a function of $|\Delta\text{score}|$.

Prompt for React Code Refinement (with Render Success)

```
You are a self-driven frontend engineer with full decision-making authority. Your task is, after the user
provides initial instructions, to revise the current React code so that the rendered result is closer to the
real design image.

[Input Information Structure]
1) Render status
- Status: success
- Similarity score: <CURRENT_SCORE> / 100
2) IR data
<IR>
3) Project source code (split by file)
<FILE_CONTENT>
4) Image inputs (provided as multimodal messages)
- Ground truth (groundtruth)
- Current render
- Diff comparison image (if generated, red highlighted areas are the main differences)

Please make revisions following the steps below:

At the start, analyze the ground-truth design image, the current render, and the diff image to identify
visual defects: compare the overall visual effect of the ground truth and the current render, describe the
obvious differences, and, based on the red-highlighted areas in the diff image, analyze each specific
discrepancy (color deviation, incorrect spacing, font size, layout misalignment, etc.). Classify the issues
by type (layout, style, color, spacing, etc.) and assess the severity of each. Then analyze the code to
locate the modification scope. Based on the visual defects you identified, determine which component or file
each issue involves, explain your localization logic, list the specific files to modify, explain why each
file needs changes, describe how you plan to fix these issues, and explain the rationale for the modification
order.
After analysis and localization are complete, directly fix the code without waiting for user instructions.
Use the write_file tool to write the corrected code. Before each fix, think through the specific details to
change, analyze the current situation step by step and what to do next, then call the tool. If you need to
modify multiple files, you can output multiple tool calls in one response and call write_file in order.

When you believe the fixes are complete, output only your fix summary and do not call any tools.

Technology stack requirements:
- Use 2 spaces for code indentation
- Must use TypeScript and React for development; do not use any other tech stack or any open-source framework
- Must use CSS Module; inline styles and style attributes are forbidden
- Component names should be semantically clear but not overly complex

Import path requirements:
- When App.tsx references components, write: import ComponentName from './components/ComponentName' and the
path must reference components under the current path
- When a component's index.tsx imports other components, write: import ComponentName from '../ComponentName'
and the path must reference other components under the parent path
- When a component's index.tsx imports css, write and only write: import styles from './index.module.css'
- All tsx files must import React at the top, i.e., "import React from 'react';"

Fixing notes:
1. Focus on visual details such as layout, styles, spacing, colors, fonts
2. Carefully inspect the diff image (red highlighted areas are the main differences)
3. Ensure all changes comply with React and TypeScript conventions
4. Always write complete file contents; do not use ellipses
5. All <img> tags must use the real URL from the IR; do not replace with other URLs
6. Must ensure code correctness for successful compilation

Every response (including the first) must include your analysis and tool calls, unless you believe you have
finished all code changes you need to make.


The design image:
<IMAGE>

The rendered image:
<IMAGE_RENDERED>

The diff heatmap:
<IMAGE_HEATMAP>
```

Figure 7: Prompt for React Code Refinement (with Render Success)

Prompt for React Code Refinement (with Render Fail)

```
You are a self-driven frontend engineer with full decision-making authority. Your task, given that the
previous render failed, is to fix the current React + TypeScript code based on the failure log so it can
build successfully and render correctly, while matching the real reference image as closely as possible.

[Input Information Structure]
1) Render status
- Status: failed
- Render failure log (please first extract key errors and warnings from it, and determine whether they are
dependency, compilation, runtime, asset path, or other issues):
```text
<RENDER_LOG>
```
2) IR data
<IR>
3) Project source code (split by file)
<FILE_CONTENT>
4) Image inputs (provided as multimodal messages)
- Ground truth (groundtruth)
- Current render: none (render failed, not generated)
- Diff image: none (render failed, not generated)

Before starting fixes, you need to complete "Analysis and Localization":
1. Explain the failure cause, involved modules, modification scope, and modification order
2. Map log entries to code items and explain why you located a specific file/component
3. The top priority of the fix strategy is "resolve blocking build/render issues first"

After analysis is complete, do not wait for user instructions; directly start fixing the code. Use the
write_file tool to write the full corrected file contents; before each write, think through the specific
details and reasons for the changes. If multiple files need modification, you can output multiple tool calls
in one response and call write_file in order.

When you believe the fixes are complete, output only your fix summary and do not call any tools; the system
will automatically enter the next render and evaluation round.

Technology stack requirements:
- Use 2 spaces for code indentation
- Must use TypeScript and React for development; do not use any other tech stack or any open-source framework
- Must use CSS Module; inline styles and style attributes are forbidden
- Component names should be semantically clear but not overly complex

Import path requirements:
- When App.tsx references components, write: import ComponentName from './components/ComponentName'
- When a component's index.tsx imports other components, write: import ComponentName from '../ComponentName'
- When a component's index.tsx imports css, write and only write: import styles from './index.module.css'
- All tsx files must import React at the top, i.e., "import React from 'react';"

Fixing notes:
1. Prioritize fixing issues that cause build/render failure, and follow the key information in the log
2. Ensure all changes comply with React and TypeScript conventions and can compile successfully
3. Always write complete file contents; do not use ellipses
4. All <img> tags must use the real URL from the IR; do not replace with other URLs

Every response (including the first) must include your analysis and tool calls, unless you believe you have
finished all code changes you need to make.

The design image:
<IMAGE>
```

Figure 8: Prompt for React Code Refinement (with Render Fail)

D IMPLEMENTATION OF BENCHMARKING

We implement benchmarking as an execution-based harness that evaluates models by the quality of a runnable React project rather than a single static snippet. The harness supports both one-shot generation and multi-round refinement under identical toolchain and rendering settings.

Fixed scaffold and controlled execution. For every instance, we start from the same dependency-locked React scaffold and build configuration. The harness installs dependencies, builds the project, and renders the resulting page in a headless browser with a fixed viewport and timeouts. This controls evaluation variance due to engineering choices and ensures comparability across models.

WriteTool-only interface. Models interact with the workspace through a single deterministic WriteTool API that can create or overwrite files. We support OpenAI-style structured tool calling when available, and otherwise parse tool-call payloads from plain-text outputs to remain compatible with servers that do not expose structured tool calls. When a model emits multiple writes to the same file within one turn, we execute only the last write to make action semantics well-defined.

Single-round generation. In the single-round setting, the model receives the exported IR together with the reference image and writes the full project in one pass. The harness then builds and renders the project and computes the visual similarity score and rendering success indicator (Section 2.3). The prompt is in Figure 6.

Multi-round refinement with visual feedback. In the multi-round setting, we iterate up to a fixed number of rounds. Each round begins by building and rendering the current workspace. If rendering succeeds, we compute (i) the scalar similarity score and (ii) a pixel-level diff visualization between the current render and the reference. If rendering fails, we return the build/runtime logs. The next model call is conditioned on the original inputs, the current code snapshot, and the round feedback. We stop early when the score reaches a configurable target, otherwise we continue until the maximum rounds. The prompt is in Figure 7 and Figure 8

E AGENTIC RL IMPLEMENTATION

We implement agentic RL on top of the ROLL framework with GRPO as the advantage estimator and a GEM-style environment interface(Wang et al., 2025b). Ray is used to coordinate dataset access and environment workers, while the scheduler/LLM proxy in ROLL handles batched generation. The D2C environment exposes a step-wise interaction protocol in which one *round* of editing (potentially containing multiple tool calls) is treated as a single RL step; inner tool invocations are micro-steps with zero reward, and the round reward is assigned after termination.

Each episode samples a task from a Ray-backed global dataset and initializes an isolated React workspace by copying a fixed scaffold. The environment returns the reference design image as the primary observation and provides the IR and metadata in the info dict. At the end of each round, the project is built and rendered via a capture script, then scored by an internal diff-based scorer that produces a percentage similarity and an optional heatmap. The scalar reward is the absolute score normalized to $[0, 1]$; rendering or scoring failures yield zero reward while preserving failure logs for feedback. A target score can trigger early termination.

F CASE STUDIES

Refer to Figures 9 and 10 for details on the Claude 4.5 Sonnet model.

Initial round Inputs



Reference Rendering

<IR>

Analysis and Reflection

IR Structure Analysis
The IR describes a restaurant review page, which mainly consists of the following sections:
1. **Top Navigation Bar**: Includes a back button, the store name "Xiao Yang Fried Rolls (Haitongyuan Branch)" and additional operation buttons
2. **Evaluation Reward Prompt**: Displays "Write an evaluation to earn 20 Credits"
3. **Overall Rating Area**: 5 star rating options (very bad, bad, average, satisfied, excellent)
4. **Detailed Rating Card**: Contains star ratings for three dimensions: taste, environment, and service
5. **Evaluation Guidance Area**: "What do people want to know" section, providing evaluation suggestions
6. **Text Input Area**: Text box for users to input evaluation content
7. **Image Upload Area**: Displays uploaded images and an add photo button
8. **Anonymous Evaluation Option**: Check box
9. **Submit Button**: Red "Submit Evaluation" button
10. **Bottom Security Area**
Design Diagram Analysis
From the design drawing, we can see:
- The overall background is a pink gradient (from #FDEAE9 to #F3F6F8)
- There is a status bar at the top showing time, signal, etc.
- The navigation bar includes a return arrow, store name, and a three-point menu
- The rating stars have two states: red (selected) and gray (unselected)
- The white cards carry the main content
- At the bottom, there is a black Home Indicator
Identification of Potential Issues
1. **Text content processing**: Some texts in the IR use array format (such as review content), which needs to be handled correctly.
2. **Star image status**: Different star images need to be displayed based on different rating statuses.
3. **Layout details**: The margins and padding of certain elements need to be carefully adjusted to match the design diagram.
4. **Gradient background**: The top area has a complex multi-layer gradient background.
5. **Absolute positioned elements**: The detailed rating cards use absolute positioning, which needs to be handled correctly.
Component Splitting Strategy
I will create the following components:
1. **Header** - Top navigation bar
2. **RewardBanner** - Evaluation reward prompt
3. **RatingSelector** - Overall rating selector
4. **DetailedRating** - Detailed rating card (taste, environment, service)
5. **ReviewGuide** - "What everyone wants to know" guidance area
6. **ReviewInput** - Evaluation input area
7. **PhotoUpload** - Image upload area
8. **SubmitButton** - Submit button
Style Implementation Strategy
- Manage styles using CSS Modules
- Implement layout with flexbox
- Achieve background effects using CSS gradients
- Handle overlapping elements using absolute positioning
Now let's start writing the code:

Generated component files

- src/components/Header/index.tsx
- src/components/Header/index.module.css
- src/components/RewardBanner/index.tsx
- src/components/RewardBanner/index.module.css
- src/components/RatingSelector/index.tsx
- src/components/RatingSelector/index.module.css
- src/components/DetailedRating/index.tsx
- src/components/DetailedRating/index.module.css
- src/components/ReviewGuide/index.tsx
- src/components/ReviewGuide/index.module.css
- src/components/ReviewInput/index.tsx
- src/components/ReviewInput/index.module.css
- src/components/PhotoUpload/index.tsx
- src/components/PhotoUpload/index.module.css
- src/components/AnonymousOption/index.tsx
- src/components/AnonymousOption/index.module.css
- src/components/SubmitButton/index.tsx

Initial round result

Render Failed Logs:

```
pnpm run build failed (exit code: 1)
Command: pnpm run build
--- STDOUT ---
ERROR in ./src/App.tsx 12:0-47
Module not found: Error: Can't resolve './components/BottomBar' in 'src'
ERROR in ./src/components/SubmitButton/index.tsx 2:0-40
Module not found: Error: Can't resolve './index.module.css' in '
src/components/SubmitButton'
webpack 5.104.1 compiled with 2 errors in 9729 ms
 ELIFECYCLE  Command failed with exit code 1.
--- STDERR ---
```

Figure 9: A Case of Initial Round

Refine round Inputs



Reference Rendering

<IR>

Render Failed Logs

Analysis and Reflection

From the build logs, two key errors are visible:

- ERROR 1: Module not found: Error: Can't resolve './components/BottomBar'
- Location: ./src/App.tsx line 12
- Cause: App.tsx references the BottomBar component, but the component file is missing

- ERROR 2: Module not found: Error: Can't resolve './index.module.css'
- Location: ./src/components/SubmitButton/index.tsx line 2
- Cause: The SubmitButton component imports a CSS module file, but that file is missing

Affected Modules
- src/components/BottomBar/ - Entire component directory missing
- src/components/SubmitButton/index.module.css - CSS file missing

Scope and Order of Changes
Priority 1 (build-blocking):
- Create SubmitButton/index.module.css - fix the SubmitButton style import
- Create BottomBar/index.tsx and BottomBar/index.module.css - add the missing BottomBar component

Code Dependency Analysis
- SubmitButton CSS missing:
- src/components/SubmitButton/index.tsx line 2 imports ./index.module.css
- The source code only provides index.tsx and no corresponding CSS file
- The submit button styles are defined in the IR data

- BottomBar component missing:
- src/App.tsx line 12 imports ./components/BottomBar
- The source code list has no BottomBar-related files
- The last IR element indicates the bottom bar is an image element

Edited component files

src/components/SubmitButton/index.module.css

```
@@ -0,0 +1,22 @@
+.button {
+  width: 718px;
+  height: 96px;
+  border-radius: 12px;
+  background: #FF352E;
+  display: flex;
+  flex-direction: column;
+  justify-content: center;
+  align-items: center;
+  position: relative;
+  margin-top: 14px;
+  z-index: 0;
+}
+
+.text {
+  font-size: 30px;
+  font-weight: 500;
+  line-height: 32px;
+  color: #FFFFFF;
+  text-align: center;
+  width: 120px;
+}
```

src/components/BottomBar/index.module.css

```
@@ -0,0 +1,10 @@
+.container {
+  width: 750px;
+  height: 78px;
+  margin-top: 4px;
+}
+
+.image {
+  width: 750px;
+  height: 78px;
+}
```

src/components/BottomBar/index.tsx

```
@@ -0,0 +1,14 @@
+import React from 'react';
+import styles from './index.module.css';
+
+function BottomBar() {
+  return (
+    <img
+      src="1.png"
+      alt="bottom bar"
+      className={styles.image}
+    />
+  );
+}
+
+export default BottomBar;
```

Refine round result



Renderd Image



Diff Heatmap

Score: 91

Figure 10: A Case of Refine Round