LEARNING THE LOOP, NOT JUST THE PAGE: EXECUTION-GROUNDED LOOP LEARNING FOR WEB GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Functional Web generation is increasingly optimized with executable rewards, yet existing methods largely focus on the quality of the final page and leave the process of diagnosing and repairing imperfect implementations underexplored. We identify a central challenge in this setting: the Generator and Refiner produce executable artifacts with direct environment rewards, whereas the intermediate Critic influences downstream behavior without a directly executable outcome. We introduce **WebLoop**, an execution-grounded framework that jointly learns generation, critique, and refinement within a shared policy. WebLoop trains an execution-free Critic with complementary signals for requirement-level discriminability and downstream helpfulness, first establishing reliable diagnosis and then introducing consequence-aware credit, while all three roles are jointly optimized with group-relative policy learning. With Qwen3.5-9B, WebLoop reaches 41.5 Overall on WebRise and 38.9% accuracy on WebGen-Bench, improving the base model by 11.3 and 15.4 points, respectively. The gains transfer to first-pass generation, persist at 27B scale, and generalize from text-only training to multimodal inputs. Controlled analyses further show that the improvement cannot be explained by an additional refinement pass alone, highlighting the importance of learning the Critic and the loop itself.

1 INTRODUCTION

Large language models are rapidly moving Web generation from page-level code synthesis toward *functional Web generation*, where models must not only produce complete HTML/CSS/JavaScript but also satisfy natural-language requirements through correct interactions, state transitions, and persistent behaviors (Lu et al., 2025b; Meng et al., 2026). Unlike conventional code or text generation, generated Web pages are directly executable, allowing functional correctness to be verified through browser interactions and further optimized with reinforcement learning (Jiang et al., 2026; Chen et al., 2026). However, existing execution-based learning remains largely outcome-centric: it optimizes what the model ultimately generates, while paying considerably less attention to how the model should diagnose and repair an already imperfect implementation. In practical Web development, first-pass generation is rarely the end of the process, and *reliable diagnosis and revision are equally critical to producing a functional page*.

A natural direction is self-critique and refinement, where models inspect their own outputs and use the resulting feedback to guide subsequent revisions (Madaan et al., 2023; Shinn et al., 2023; Gou et al., 2024; Chen et al., 2024), with related paradigms increasingly adopted in Web generation (Lu et al., 2025a; Li et al., 2026). As shown in Fig. 1, execution supervision is typically used either to directly optimize generated Web pages, as in generation-only RL (Fig. 1(a)), or as external feedback for iterative editing (Fig. 1(b)). Neither formulation directly addresses how to optimize an intermediate Critic that guides subsequent refinement. From a reinforcement learning perspective, this introduces an *intermediate credit-assignment problem*: the Generator and Refiner produce executable artifacts and receive direct environment rewards, whereas the Critic influences downstream behavior without a directly executable outcome. An effective Critic must therefore learn both *discriminability*, which identifies satisfied and violated requirements, and *helpfulness*, which reflects whether its feedback improves subsequent refinement. Since accurate diagnosis does not necessarily

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.

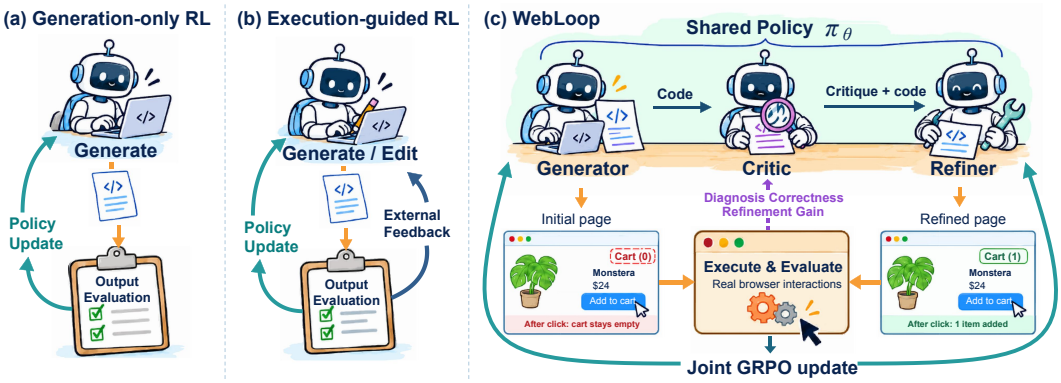


Figure 1: **Conceptual comparison of execution-based learning paradigms for functional Web generation.** (a) Generation-only RL optimizes executable outputs directly. (b) Execution-guided RL feeds external feedback back into the code policy. (c) WebLoop jointly learns Generator, Critic, and Refiner, assigning execution-derived credit to the intermediate Critic.

lead to effective repair, the central challenge is how to assign reliable RL credit to the Critic from verifiable execution outcomes.

To address this challenge, we introduce **WebLoop**, an execution-grounded loop learning framework shown in Fig. 1(c). WebLoop jointly optimizes Generator, Critic, and Refiner as interdependent roles of a shared policy. The Generator first produces a candidate page and receives functional rewards from browser execution. The Critic then evaluates the task requirements and candidate source code without access to test results or execution traces. Conditioned on this critique, the Refiner revises the implementation and is evaluated in the same browser environment. WebLoop trains the Critic with two complementary signals: requirement-level execution labels supervise its discriminability, while the execution improvement induced by subsequent refinements measures its helpfulness. We first establish reliable diagnostic behavior and then introduce downstream improvement for consequence-aware optimization, while all three roles form separate comparison groups for group-relative advantage estimation and are jointly optimized through GRPO. The Critic therefore remains *execution-free* during inference while receiving execution-derived retrospective credit during training, turning a fixed self-refinement pipeline into a learnable RL loop.

We evaluate WebLoop on functional Web generation benchmarks across model scales and input modalities. With Qwen3.5-9B, WebLoop reaches 41.5 Overall on WebRise and 38.9% accuracy on WebGen-Bench, improving the corresponding base model by 11.3 and 15.4 points, respectively. The gains transfer to first-pass generation, persist at the 27B scale, and generalize from text-only training to Markdown, Sketch, Image, and Video inputs. Controlled ablations further show that the improvement cannot be attributed to an additional refinement pass alone: joint loop optimization consistently outperforms staged alternatives, while the Critic benefits from complementary discriminability and downstream helpfulness signals with staged credit assignment.

Our contributions are threefold. (1) We formulate functional Web self-refinement as a structured reinforcement learning problem and identify intermediate Critic credit assignment as a central challenge under asymmetric reward observability. (2) We propose WebLoop, which learns an execution-free Critic through complementary discriminability and helpfulness signals and jointly optimizes the Generator, Critic, and Refiner through group-relative policy learning. (3) We conduct controlled evaluations across Web benchmarks, model scales, and input modalities, demonstrating consistent gains over generation-only and refinement-only alternatives and validating the role of Critic learning and loop-level optimization.

2 Related Work

Functional Web Generation and Execution-Based Learning. Web generation has evolved from reconstructing interfaces from screenshots or visual specifications toward generating complete and interactive Web applications. Early and concurrent work studies large-scale screenshot-to-code

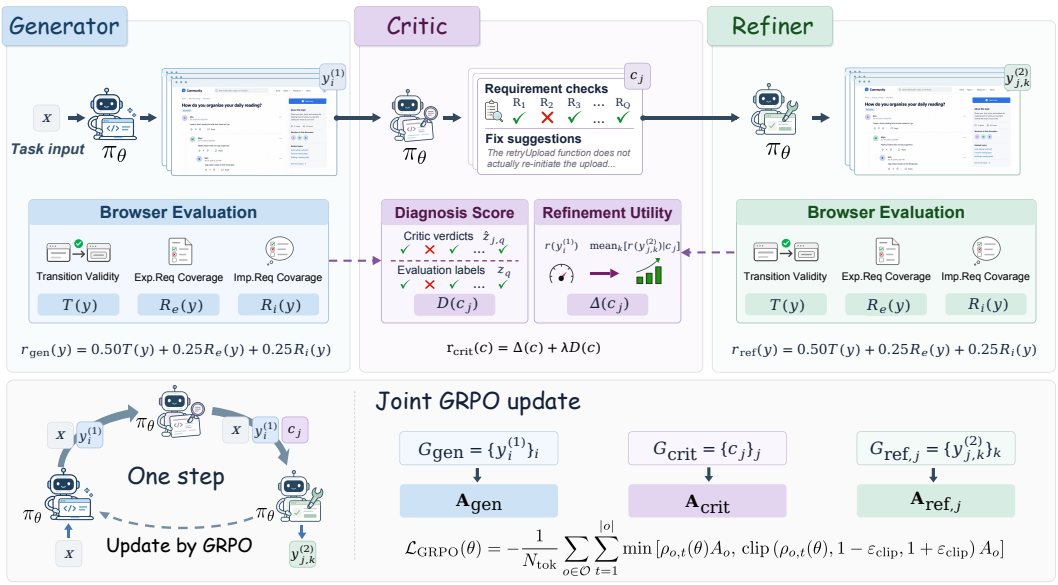


Figure 2: **Overview of our WebLoop framework.** A shared policy sequentially acts as Generator, Critic, and Refiner. The Generator and Refiner produce executable Web pages evaluated in the same browser environment, while the execution-free Critic guides subsequent refinement and receives training credit from its diagnosis and downstream effects.

learning and front-end reconstruction through datasets and benchmarks such as WebSight, Design2Code, and Web2Code (Laurençon et al., 2024; Si et al., 2025; Yun et al., 2024), while FrontendBench further emphasizes executable evaluation for realistic front-end development (Zhu et al., 2025). More recent benchmarks shift the focus from visual fidelity toward functional correctness, using interaction traces, state transitions, and requirement-level tests to evaluate generated websites (Lu et al., 2025b; Meng et al., 2026). This verifiability has also enabled reinforcement learning for Web generation. WebGen-R1 and WebGrader optimize Web policies with executable rewards or programmatic graders (Jiang et al., 2026; Chen et al., 2026), while WebGen-Agent incorporate iterative interaction or rendering feedback into step-wise Web optimization (Lu et al., 2025a). These methods establish execution as an effective signal for optimizing Web artifacts. In contrast, WebLoop further uses executable outcomes to assign learning credit to the intermediate critique that determines how an imperfect artifact is subsequently revised.

Critique-Guided Refinement and Self-Improvement. A broad line of work improves model outputs through self-generated feedback, reflection, verification, or tool-assisted correction (Madaan et al., 2023; Shinn et al., 2023; Gou et al., 2024; Chen et al., 2024). Since intrinsic self-correction is often unreliable, later work explicitly trains correction, evaluation, or critic behavior with supervised or reinforcement signals (Huang et al., 2024; Kumar et al., 2025; McAleese et al., 2024; Wang et al., 2024). More recent methods optimize critiques according to the quality of their induced revisions, with RCO (Yu et al., 2025) and Critique-RL (Xi et al., 2025) directly connecting critic learning to downstream refinement and staged critic optimization. Similar ideas have entered Web generation, where WebGen-Agent (Lu et al., 2025a) and ReLook (Li et al., 2026) use external or multimodal feedback to support iterative revision. WebLoop instead learns an execution-free Critic from requirement-level diagnosis and downstream execution gains, while jointly optimizing Generator, Critic, and Refiner within a shared policy.

3 WebLoop: Learning the Generation–Critique–Refinement Loop

As shown in Fig. 2, WebLoop organizes generation, critique, and refinement into a sequential loop implemented by a shared policy. Executable outputs are directly grounded by browser evaluation, while the intermediate critique conditions subsequent refinement.

3.1 Loop Formulation and Execution Rewards

Generator–Critic–Refiner loop. Let $x = (d, \mathcal{R}_x)$ denote a Web generation task, where d is the natural-language description and $\mathcal{R}_x = \{r_q\}_{q=1}^{Q_x}$ contains its functional requirements. WebLoop uses a shared policy π_θ to instantiate three roles under different conditioning contexts. The Generator maps the task to a complete initial HTML implementationThe Critic inspects an imperfect implementation against the task requirements and produces requirement-level judgments and repair guidance. The Refiner conditions on the task, the current implementation, and the critique to generate a complete revised HTML implementation.

Let p_{gen}, p_{crit}, and p_{ref} denote the corresponding prompt construction functions, and let $\pi_{\theta_{\text{old}}}$ denote the rollout policy. For each task, WebLoop samples the three stages as

$$\begin{aligned} y_i^{(1)} &\sim \pi_{\theta_{\text{old}}}\big(\cdot \mid p_{\text{gen}}(x)\big), && i = 1, \dots, n_g, \\ c_j &\sim \pi_{\theta_{\text{old}}}\big(\cdot \mid p_{\text{crit}}(x, y_b)\big), && j = 1, \dots, n_c, \\ y_{j,k}^{(2)} &\sim \pi_{\theta_{\text{old}}}\big(\cdot \mid p_{\text{ref}}(x, y_b, c_j)\big), && k = 1, \dots, n_r, \end{aligned} \tag{1}$$

where y_b denotes the imperfect base implementation selected from the initial generations. The Generator and Refiner therefore produce executable artifacts, whereas the Critic produces an intermediate feedback decision that changes the conditional distribution of subsequent refinements.

Execution reward. Both initial and refined implementations are evaluated by the same browser environment. Following WebRISE (Meng et al., 2026), each task is associated with an Interaction Contract Graph $\mathcal{G}_x$ that specifies browser interactions and functional assertions. Executing a page y returns

$$\mathcal{E}(y; \mathcal{G}_x) = \big(T(y), R_{\text{e}}(y), R_{\text{i}}(y), \mathbf{z}(y)\big), \tag{2}$$

where $T(y)$ denotes transition validity, $R_{\text{e}}(y)$ and $R_{\text{i}}(y)$ denote explicit and implicit requirement coverage, and $\mathbf{z}(y) \in \{0,1\}^{Q_x}$ contains requirement-level execution labels. We define the page-level execution reward as

$$r_{\text{web}}(y) = 0.50T(y) + 0.25R_{\text{e}}(y) + 0.25R_{\text{i}}(y). \tag{3}$$

Both Generator and Refiner outputs are optimized with r_{web}, placing the two executable endpoints in the same reward space. This shared objective allows the behavioral change before and after critique to be measured directly.

Critique-conditioned branching. After evaluating the initial generations, we select the median-reward candidate among valid but imperfect implementations as the shared base draft y_b. All critiques are sampled from the same (x, y_b) context, which removes variation caused by different starting implementations. Importantly, the Critic observes only the task requirements and the source code of y_b, without access to browser results or execution traces. Each critique c_j then conditions n_r independent refinements, and every refined page is evaluated under the same execution objective. Multiple refinements under the same critique provide a more robust estimate of how effectively the current Refiner can act on that feedback. If no valid imperfect draft exists, the task contributes only its initial-generation samples. Further details are provided in Appendix A.2.

This construction gives direct environment rewards to the Generator and Refiner, while the intermediate Critic has no directly executable outcome. We next assign Critic credit using requirement-level diagnosis and downstream refinement effects.

3.2 Credit Assignment for the Critic

Unlike the executable Generator and Refiner outputs, a critique has no direct browser outcome from which to obtain an environment reward. Following the critic decomposition in Critique-RL (Xi et al., 2025), we distinguish two complementary properties of a critique: *discriminability*, which measures whether it correctly diagnoses the current implementation, and *helpfulness*, which measures whether acting on it improves downstream execution. WebLoop grounds these two properties with requirement-level browser labels and executable refinement outcomes, respectively.

Execution-free critique. For the selected base draft y_b, the Critic receives only the task requirements and source code through $p_{\text{crit}}(x, y_b)$. It predicts whether each explicit requirement is satisfied and provides corresponding repair guidance. Importantly, browser results, requirement labels, and

execution traces are never exposed to the Critic as input. Execution is instead used only to construct training signals, requiring the Critic to infer functional failures directly from the specification and implementation.

Diagnostic discriminability. Let $z_q \in \{0, 1\}$ denote the execution-grounded label for requirement r_q, where $z_q = 1$ only if all corresponding assertions are satisfied. For critique c_j, let $\hat{z}_{j,q}$ denote its predicted requirement status. We define the failed and satisfied requirement sets as

$$\mathcal{F} = \{q : z_q = 0\}, \qquad \mathcal{P} = \{q : z_q = 1\}. \tag{4}$$

To avoid bias toward the majority class, we measure diagnostic discriminability using balanced class-wise accuracy,

$$D(c_j) = a_{\text{fail}}(c_j) + a_{\text{pass}}(c_j) - 1, \tag{5}$$

where a_{fail} and a_{pass} denote the prediction accuracies over $\mathcal{F}$ and $\mathcal{P}$, respectively. When only one class is present, we linearly rescale its accuracy to the same $[-1, 1]$ range. Thus, $D(c_j)$ measures whether the Critic correctly distinguishes satisfied from violated requirements independently of downstream refinement.

Retrospective helpfulness. Diagnostic correctness alone does not indicate whether a critique can actually guide an effective repair. For each critique c_j, we therefore estimate its downstream utility from the n_r refinements conditioned on it:

$$H(c_j) = \frac{1}{n_r} \sum_{k=1}^{n_r} r_{\text{web}}\big(y_{j,k}^{(2)}\big) - r_{\text{web}}(y_b). \tag{6}$$

A positive $H(c_j)$ indicates that refinements conditioned on the critique outperform the original implementation on average. Using multiple refinements reduces sensitivity to a single stochastic regeneration and measures the practical utility of the critique under the current Refiner. While $D(c_j)$ captures what the Critic understands, $H(c_j)$ captures what happens when its feedback is acted upon.

Staged critic credit. Inspired by the two-stage optimization principle of Critique-RL (Xi et al., 2025), we first establish reliable discriminability before introducing downstream helpfulness. This separation is particularly important in WebLoop because helpfulness is a delayed signal that depends jointly on the critique and the current Refiner, whereas discriminability can be directly grounded by requirement-level execution labels. At update step m, we define the Critic reward as

$$r_{\text{crit}}^{(m)}(c_j) = \begin{cases} D(c_j), & m \le 15, \\ H(c_j) + \lambda D(c_j), & m > 15, \end{cases} \tag{7}$$

where λ controls the contribution of diagnostic discriminability in the second stage. Retaining $D(c_j)$ prevents downstream improvement from serving as a substitute for correct diagnosis, while $H(c_j)$ encourages feedback that can be effectively acted upon by the Refiner. This schedule prevents downstream improvement from serving as a substitute for correct diagnosis while gradually introducing consequence-aware credit. The resulting scalar reward allows critique samples to participate in policy optimization while preserving their distinct reward semantics from executable Web outputs. Complete reward definitions are provided in Appendix A.2.

3.3 Joint Group-Relative Optimization

WebLoop produces three types of policy outputs with different conditioning contexts and reward semantics. We therefore construct role specific comparison groups for advantage estimation, and then jointly optimize all outputs under the shared policy.

Role specific comparison groups. For each task, Generator outputs are compared under the same task, Critic outputs under the same base draft, and Refiner outputs under the same critique. Specifically, we define

$$G_{\text{gen}}(x) = \{y_i^{(1)}\}_{i=1}^{n_g}, \quad G_{\text{crit}}(x, y_b) = \{c_j\}_{j=1}^{n_c}, \quad G_{\text{ref},j}(x, y_b, c_j) = \{y_{j,k}^{(2)}\}_{k=1}^{n_r}. \tag{8}$$

The corresponding rewards are r_{web} for Generator and Refiner outputs and r_{crit} for Critic outputs. For any output o with comparison group $g(o)$, we compute the group relative advantage as

$$A_o = \frac{r(o) - \mu_{g(o)}}{\sigma_{g(o)} + \epsilon_{\text{norm}}}, \tag{9}$$

where $\mu_{g(o)}$ and $\sigma_{g(o)}$ are the reward mean and standard deviation within its group. This grouping keeps each comparison local to outputs generated under the same decision context, while preserving the distinct credit semantics of the three roles.

Joint policy update. After advantage estimation, Generator, Critic, and Refiner samples are merged into a single optimization batch $\mathcal{O}$. For an output o with conditioning context u_o, the token level probability ratio is

$$\rho_{o,t}(\theta) = \frac{\pi_\theta(o_t \mid u_o, o_{<t})}{\pi_{\theta_{\text{old}}}(o_t \mid u_o, o_{<t})}. \tag{10}$$

We optimize the shared policy with the clipped GRPO objective

$$\mathcal{L}_{\text{GRPO}}(\theta) = -\frac{1}{N_{\text{tok}}} \sum_{o \in \mathcal{O}} \sum_{t=1}^{|o|} \min\Big(\rho_{o,t}(\theta)A_o, \text{clip}\left(\rho_{o,t}(\theta), 1-\epsilon_{\text{clip}}, 1+\epsilon_{\text{clip}}\right)A_o\Big), \tag{11}$$

where N_{tok} is the number of valid output tokens in the batch. Therefore, group structure assigns role-specific relative credit, while all three roles jointly update the shared policy. WebLoop is therefore sequential in rollout, role-specific in credit assignment, and joint in optimization.

3.4 Executable Training Data Construction

We construct executable Web generation tasks that pair natural-language specifications with browser-based functional supervision for WebLoop training.

Task scope. The corpus covers diverse Web applications, including e-commerce, office productivity, education, developer tools, and data analytics. Across these domains, we instantiate functional scenarios involving core operations, management functions, user settings, data presentation, and multi-step workflows.

Construction pipeline. For each scenario, we construct explicit requirements that specify functions and interaction entry points, together with implicit requirements covering behavioral constraints such as state consistency, operation ordering, boundary handling, and reset behavior. Motivated by WebRISE, we convert these requirements into an initial-state contract and an executable Interaction Contract Graph (ICG) containing browser interactions and functional assertions. We then generate a reference implementation, execute the associated tests in a real browser, and retain tasks with transition validity $T \geq 0.60$. Further construction details are provided in Appendix B.

Dataset statistics. The resulting corpus contains 2,880 tasks across 21 domains and 1,442 scenarios, with 11.87 requirements per task on average: 5.47 explicit and 6.39 implicit, with implicit requirements accounting for 53.9%. The ICGs average 10.64 states, 11.40 transitions, and 28.15 executable assertions, enabling fine-grained functional supervision.

4 Experiments

4.1 Experimental Setup

Benchmarks and metrics. We evaluate functional correctness on WebRise (Meng et al., 2026) and WebGen-Bench (Lu et al., 2025b). WebRise contains 442 tasks and reports transition validity T, explicit requirement coverage R_{e}, implicit requirement coverage R_{i}, and their average Overall score. For consistent comparison, all controlled training uses Text-conditioned tasks, and the main WebRise results are evaluated under the Text modality. WebGen-Bench contains 101 website tasks with 647 functional test items, and we use the Bolt.diy framework with automatic starter-template selection disabled to generate websites from an empty workspace. Following the official protocol, we report Yes, Partial, No, Start-Fail, and Accuracy. "N/A" indicates that model outputs could not be parsed by the harness due to incompatible output formatting.

Baselines. We compare against representative general-purpose models, including Qwen3.5-122B-A10B and Qwen3.5-397B-A17B (Qwen Team, 2026), Kimi-K2.6 (Moonshot AI, 2026), GLM-5.3 (GLM-5 Team, 2026), Claude Opus 4.7 (Anthropic, 2026), Gemini 3.1 Pro (Google DeepMind, 2026), and GPT-6 Astra (OpenAI, 2026). We also include Web-generation specialized models, including WebGen-LM (Lu et al., 2025b), WebGen-Agent (Lu et al., 2025a), and UIGEN-T2 (tesslate, 2024). For controlled same-backbone comparisons, we evaluate the original Qwen3.5

Table 1: **Main results on WebRise and WebGen-Bench.** General-purpose and Web-generation specialized models are included as capability references, while the Qwen3.5 block provides controlled comparisons. **Bold** denotes the best result within each reference group or controlled backbone size according to the metric direction. † denotes proprietary models.

Model / Method	WebRise				WebGen-Bench				
	$T\uparrow$	$R_e\uparrow$	$R_i\uparrow$	**Overall** ↑	Yes (%) ↑	Partial (%) ↑	No (%) ↓	Start-Fail (%) ↓	**Acc.** ↑
General-Purpose Models (Open-Source & Proprietary)									
Qwen3.5-122B-A10B	38.0	48.9	35.2	40.7	9.9	6.2	78.1	5.9	13.0
Kimi-K2.6	44.6	54.2	41.8	46.9	25.7	11.1	55.3	7.9	31.2
Qwen3.5-397B-A17B	45.7	57.2	42.8	48.6	16.1	9.1	70.0	4.8	20.6
GLM-5.3	59.5	77.1	54.9	63.8	20.6	11.9	67.5	**0.0**	26.5
Claude Opus 4.7†	48.8	57.6	45.8	50.7	**47.4**	**15.8**	**35.5**	1.2	**55.3**
Gemini 3.1 Pro†	50.7	61.1	47.5	53.1	40.2	12.5	47.0	0.3	46.4
GPT-6 Astra†	**62.5**	**78.9**	**57.7**	**66.4**	37.2	11.9	49.5	1.4	43.2
Web-Generation Specialized Models									
WebGen-LM-7B	5.5	17.7	6.1	9.8	2.8	2.0	94.1	1.1	3.8
WebGen-Agent-7B (Step-GRPO)	5.8	23.7	6.9	12.1	1.9	**7.1**	**90.6**	0.5	**5.4**
WebGen-LM-32B	6.0	13.4	5.4	8.3	2.6	1.4	95.8	**0.2**	3.3
UIGEN-T2-7B	12.1	33.0	11.1	18.7	N/A	N/A	N/A	N/A	N/A
WebGen-LM-14B	**15.3**	**35.5**	**14.5**	**21.8**	**2.9**	1.1	94.3	1.7	3.5
Controlled Comparisons: Qwen3.5									
Base (Qwen3.5-9B)	23.2	45.3	22.1	30.2	18.4	10.2	67.1	4.3	23.5
Generate–Critique–Refine (w/o RL)	25.8	47.9	24.6	32.8	26.3	9.6	59.2	4.9	31.1
Critique-GRPO	27.6	50.0	26.2	34.6	13.4	8.3	76.7	**1.5**	17.6
Critique-RL	26.9	48.8	25.6	33.8	24.1	8.8	62.0	5.1	28.5
WebLoop (Gen)	31.7	55.4	30.0	39.1	28.7	10.8	56.9	3.6	34.2
WebLoop (Refine)	**34.6**	**57.9**	**32.0**	**41.5**	**31.7**	**14.4**	**48.4**	5.6	**38.9**
Base (Qwen3.5-27B)	36.3	47.3	34.3	39.3	24.3	11.0	62.1	2.6	29.8
Generate–Critique–Refine (w/o RL)	40.5	63.1	37.6	47.1	32.6	11.0	53.8	2.6	38.1
WebLoop (Gen)	43.8	66.6	40.1	50.2	26.0	13.0	60.9	**0.2**	32.5
WebLoop (Refine)	**46.1**	**68.0**	**42.7**	**52.2**	**36.0**	**13.6**	**50.2**	**0.2**	**42.8**

model, Generate–Critique–Refine without RL, and alternative critique-learning methods Critique-GRPO (Zhang et al., 2025) and Critique-RL (Xi et al., 2025). Throughout the paper, "Gen" denotes the first-pass code produced by the Generator of the trained checkpoint, while "Refine" denotes the revised code after one complete Generator–Critic–Refiner loop from the same checkpoint.

Models and training. We instantiate WebLoop with Qwen3.5-9B and Qwen3.5-27B, with all controlled ablations conducted on the 9B model. Training uses 3K text-conditioned Web generation tasks with a task batch size of 64. For each task, we sample three initial generations, three critiques for the selected imperfect draft, and three refinements per critique, yielding 12 executable Web candidates per rollout. The Critic is trained with discriminability alone for the first 15 updates and then with the combined discriminability and helpfulness reward using $\lambda = 0.2$. Unless otherwise stated, Generator, Critic, and Refiner samples are jointly optimized with GRPO. Further training, rollout, and optimization details are provided in Appendix A.

4.2 Main Results

Functional Web generation. Table 1 shows that WebLoop consistently improves functional correctness across both backbone sizes and benchmarks. With Qwen3.5-9B, WebLoop (Refine) reaches 41.5 Overall on WebRise and 38.9% accuracy on WebGen-Bench, improving the base model by 11.3 and 15.4 points and surpassing Qwen3.5-122B-A10B on both benchmarks. At 27B, WebLoop further reaches 52.2 and 42.8%, outperforming Qwen3.5-397B-A17B and the 1T-parameter Kimi-K2.6 despite a substantially smaller total parameter footprint. It also approaches proprietary frontier models, trailing Gemini 3.1 Pro by only 0.9 points on WebRise and 3.6 points on WebGen-Bench, and GPT-6 Astra by 0.4 points on WebGen-Bench.

First-pass generation and model scaling. As shown in Table 1, WebLoop improves both first-pass generation and final refinement. WebLoop (Gen) raises Qwen3.5-9B from 30.2 to 39.1 on WebRise and from 23.5% to 34.2% on WebGen-Bench. These gains indicate that loop-level training

Table 2: **Cross-modal generalization on WebRise.** WebLoop is trained only on Text-conditioned tasks and evaluated on all five input modalities using the same checkpoint. **Avg.** averages the modality-level $(T + R_e + R_i)/3$ scores.

Model / Stage	Text			Markdown			Sketch			Image			Video			Avg. ↑
	T ↑	R_e ↑	R_i ↑	T ↑	R_e ↑	R_i ↑	T ↑	R_e ↑	R_i ↑	T ↑	R_e ↑	R_i ↑	T ↑	R_e ↑	R_i ↑	
Base (Qwen3.5-9B)	23.2	45.3	22.1	28.2	51.1	26.0	25.2	48.8	21.9	23.0	46.2	22.8	24.7	45.7	23.9	31.9
WebLoop (Gen)	31.7	55.4	30.0	30.5	55.0	28.4	28.7	53.4	26.2	27.7	51.9	26.1	31.8	54.1	30.9	37.5
WebLoop (Refine)	**34.6**	**57.9**	**32.0**	**34.4**	**57.5**	**32.1**	**31.1**	**55.0**	**28.1**	**30.6**	**53.1**	**29.0**	**35.9**	**57.3**	**33.6**	**40.1**

Table 3: **Where does the gain come from?** Controlled comparison of training schemes on WebRise and WebGen-Bench. All variants use the same Qwen3.5-9B backbone, training data, and rollout budget, and differ only in which roles are optimized and whether they are trained jointly or sequentially. **Bold** denotes the best result according to the metric direction.

Training Scheme / Output	Optimized Roles	WebRise				WebGen-Bench				
		T ↑	R_e ↑	R_i ↑	**Overall** ↑	Yes (%) ↑	Partial (%) ↑	No (%) ↓	Start-Fail (%) ↓	**Acc.** ↑
Base	–	23.2	45.3	22.1	30.2	18.4	10.2	67.1	4.3	23.5
Generation Only (Gen)	Gen	29.5	54.1	27.3	36.9	17.8	10.5	67.9	3.9	23.0
Refine w/o Critique (Gen)	Gen + Refine	28.2	53.3	25.2	35.6	16.7	11.9	70.5	0.9	22.6
Refine w/o Critique (Refine)		32.9	57.3	29.7	40.0	21.6	10.0	68.3	**0.0**	26.7
Staged Training (Gen)	Gen → Critic + Refine	27.6	52.0	25.9	35.2	20.7	11.1	66.0	2.2	26.3
Staged Training (Refine)		32.9	56.4	30.9	40.1	29.8	12.5	55.5	2.2	36.1
WebLoop (Gen)	Gen + Critic + Refine	31.7	55.4	30.0	39.1	28.7	10.8	56.9	3.6	34.2
WebLoop (Refine)		**34.6**	**57.9**	**32.0**	**41.5**	**31.7**	**14.4**	**48.4**	5.6	**38.9**

strengthens the underlying generation policy itself, rather than benefiting only critique-conditioned revision. At 27B, the corresponding gains are 39.3 to 50.2 and 29.8% to 32.5%, respectively. The complete loop further reaches 52.2 and 42.8%, showing that the benefits persist with model scaling and continue to compound through refinement.

Generalization across input modalities. Table 2 evaluates whether Text-only WebLoop training transfers to other input modalities. The average WebRise score improves from 31.9 for the base model to 37.5 for WebLoop (Gen) and 40.1 after the complete loop, with consistent gains across Text, Markdown, Sketch, Image, and Video. The improvements also span all three functional metrics, demonstrating that the learned generation and refinement capabilities generalize beyond the text-conditioned training distribution.

5 Analysis and Discussion

5.1 Where Does the Gain Come From?

Beyond additional refinement. Table 3 shows that WebLoop's gains cannot be explained by an extra refinement pass alone. Generation Only reaches 36.9 on WebRise and 23.0% on WebGen-Bench, while Refine w/o Critique improves the final output to 40.0 and 26.7%, respectively. WebLoop further reaches 41.5 and 38.9%, indicating that learned critique provides additional value beyond second-pass generation. Joint optimization is also important: Staged Training reaches 40.1 and 36.1% after refinement, while WebLoop achieves 41.5 and 38.9%, with a larger gap already visible in first-pass generation. These results show that WebLoop benefits from jointly learning generation, critique, and refinement rather than treating refinement or Critic learning as isolated stages.

5.2 What Makes a Useful Critique?

Complementary critic signals and staged learning. Table 4 shows that discriminability and helpfulness capture complementary aspects of critique quality. Optimizing either signal alone underperforms WebLoop, and their relative ranking differs across benchmarks: discriminability performs better on WebRise, while helpfulness is stronger on WebGen-Bench. Moreover, mixing the two

Table 4: **Critic reward ablation.** All variants use the same Qwen3.5-9B backbone, training data, and rollout structure and differ only in the Critic reward. D denotes discriminability and H denotes downstream helpfulness. **Bold** denotes the best result according to the metric direction.

Critic Training / Output	$\mathbf{R}_{\text{crit}}$	WebRise				WebGen-Bench				
		$T\uparrow$	$R_e\uparrow$	$R_i\uparrow$	**Overall** $\uparrow$	Yes (%) $\uparrow$	Partial (%) $\uparrow$	No (%) $\downarrow$	Start-Fail (%) $\downarrow$	**Acc.** $\uparrow$
Base	–	23.2	45.3	22.1	30.2	18.4	10.2	67.1	4.3	23.5
Discriminability Only (Gen)	D	30.0	53.6	27.2	36.9	22.3	8.7	65.8	3.2	26.6
Discriminability Only (Refine)	D	33.4	57.3	30.7	40.5	27.8	10.7	59.4	**2.2**	33.2
Helpfulness Only (Gen)	H	27.6	50.7	24.7	34.3	18.2	9.0	68.2	4.6	22.7
Helpfulness Only (Refine)	H	31.3	54.1	28.7	38.0	28.1	13.0	56.3	2.6	34.6
Mixed from Start (Gen)	$H + \lambda D$	30.2	54.0	27.8	37.3	24.4	9.3	62.9	3.4	29.1
Mixed from Start (Refine)	$H + \lambda D$	32.7	56.5	29.4	39.5	28.0	10.5	58.3	3.2	33.2
WebLoop (Gen)	$D \to H + \lambda D$	31.7	55.4	30.0	39.1	28.7	10.8	56.9	3.6	34.2
WebLoop (Refine)	$D \to H + \lambda D$	**34.6**	**57.9**	**32.0**	**41.5**	**31.7**	**14.4**	**48.4**	5.6	**38.9**

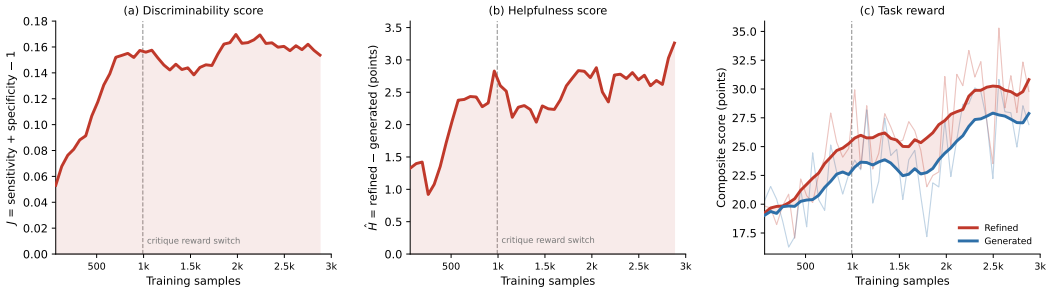


Figure 3: **Critic learning dynamics.** (a) Discriminability D. (b) Downstream helpfulness H. (c) Generator and Refiner execution rewards. Thin and thick curves denote per-step values and moving averages; the dashed line marks the reward switch after 15 updates (around 1K samples).

signals from the beginning still falls short of the staged objective, indicating that both the reward composition and its scheduling matter. WebLoop reaches 41.5 on WebRise and 38.9% on WebGen-Bench, supporting a curriculum that first establishes reliable diagnosis and then introduces downstream utility. This allows the Critic to learn feedback that is both correct and actionable.

Critic learning dynamics. Fig. 3 tracks discriminability D, downstream helpfulness H, and executable reward throughout training. We use the first 15 updates as a diagnostic warm-up, optimizing only D so that the Critic first learns reliable requirement-level judgments from direct execution labels. We then switch to $H + \lambda D$, introducing the delayed downstream signal while retaining diagnostic supervision. This staged schedule avoids relying on refinement-dependent utility before the Critic has learned a stable notion of correctness. Refined outputs remain consistently stronger than first-pass generations, indicating that improved Critic behavior translates into executable gains. Additional training-dynamics and co-adaptation analyses are provided in Appendix C.1.

6 Conclusion

We presented **WebLoop**, an execution-grounded framework that jointly learns generation, critique, and refinement for functional Web generation. WebLoop treats critique as a learnable intermediate decision, assigning execution-derived credit through complementary discriminability and downstream helpfulness signals while jointly optimizing all three roles within a shared policy. Across WebRise and WebGen-Bench, WebLoop improves both first-pass generation and final refinement, with gains that persist across model scales and transfer from text-only training to multimodal inputs. Controlled ablations show that these gains cannot be explained by an additional refinement pass alone, suggesting that executable supervision can be used not only to optimize final artifacts, but also to learn how they should be diagnosed and improved.

AI USE STATEMENT

Generative AI tools, including ChatGPT and AI coding assistants such as Codex, were used during the preparation of this work. They assisted with language editing, improving clarity and organization, and the implementation, debugging, and refinement of selected code components. AI models were also used to assist the generation and refinement of portions of the executable training data under predefined data-construction protocols. AI-assisted training data were subsequently filtered and validated through the authors' specified quality-control and browser-execution procedures. The authors made and verified the final decisions regarding the research questions, methodological design, experimental protocol, analysis, and conclusions. Generative AI was not used to fabricate, alter, or post hoc select reported experimental results; all reported metrics were computed from actual model outputs using the stated evaluation pipelines. The authors reviewed all AI-assisted content and take full responsibility for the final manuscript, implementation, data, and reported findings.

REPRODUCIBILITY STATEMENT

We provide the formulation, training procedure, data construction pipeline, and evaluation protocol required to reproduce WebLoop. The main paper describes the Generator–Critic–Refiner loop, execution-grounded Critic credit assignment, joint group-relative optimization, and experimental protocol. Appendix A provides training and rollout configurations, complete objective definitions, browser evaluation details, and controlled-baseline adaptations, while Appendix B details executable task construction and filtering. The complete role-specific prompts are provided in Appendix D. Unless otherwise stated, controlled comparisons use the same backbone initialization, training-task distribution, browser evaluation infrastructure, and metric computation. The core code is provided in the supplementary material accompanying this paper. Source code, configurations, and data-construction assets will be publicly released upon acceptance to facilitate reproducibility and future research.

REFERENCES

Anthropic. Claude Opus 4.7 System Card. Technical report, Anthropic, April 2026. URL https://www.anthropic.com/claude-opus-4-7-system-card.

Boshui Chen, Huiping Liu, and Shaolei Zhang. Webgrader: Training llms for web development with self-evolving programmatic grader. *CoRR*, abs/2608.06474, 2026.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *ICLR*. OpenReview.net, 2024.

GLM-5 Team. GLM-5: From vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Google DeepMind. Gemini 3.1 Pro Model Card. Technical report, Google DeepMind, February 2026. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-1-Pro-Model-Card.pdf.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: large language models can self-correct with tool-interactive critiquing. In *ICLR*. OpenReview.net, 2024.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *ICLR*. OpenReview.net, 2024.

Juyong Jiang, Chenglin Cai, Chansung Park, Jiasi Shen, Sung Hun Kim, Jianguo Li, and Yue Wang. Webgen-r1: Incentivizing large language models to generate functional and aesthetic websites with reinforcement learning. *CoRR*, abs/2604.20398, 2026.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D. Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei M. Zhang, Kay McKinney, Disha

Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal M. P. Behbahani, and Aleksandra Faust. Training language models to self-correct via reinforcement learning. In *ICLR*. OpenReview.net, 2025.

Hugo Laurençon, Léo Tronchon, and Victor Sanh. Unlocking the conversion of web screenshots into HTML code with the websight dataset. *CoRR*, abs/2403.09029, 2024.

Yuhang Li, Chenchen Zhang, Ruilin Lv, Ao Liu, Ken Deng, Yuanxing Zhang, Jiaheng Liu, and Bo Zhou. Relook: Vision-grounded RL with a multimodal LLM critic for agentic web coding. In *ACL (1)*, pp. 25471–25485. Association for Computational Linguistics, 2026.

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

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. Webgen-bench: Evaluating llms on generating interactive and functional websites from scratch. In *NeurIPS*, 2025b.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *NeurIPS*, 2023.

Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. LLM critics help catch LLM bugs. *CoRR*, abs/2407.00215, 2024.

Yuxin Meng, Yuhan Suo, Junjie Wang, Yuhan Sun, Yiyao Yu, Ruixu Zhang, Ruining Hu, Yubin Wang, Shouwei Ruan, Bin Wang, Yuxiang Zhang, and Yujiu Yang. Webrise: Requirement-induced state evaluation for mllm-generated web artifacts. *CoRR*, abs/2606.03220, 2026.

Moonshot AI. Kimi K2.6. Model Card, April 2026. URL `https://huggingface.co/moonshotai/Kimi-K2.6`.

OpenAI. GPT-6 Astra System Card. Technical report, OpenAI, September 2026. URL `https://deploymentsafety.openai.com/gpt-6-astra`.

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

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2code: Benchmarking multimodal code generation for automated front-end engineering. In *NAACL (Long Papers)*, pp. 3956–3974. Association for Computational Linguistics, 2025.

tesslate. Uigen-t2: Scaling ui generation with reasoning on qwen2.5-coder-7b, 2024. URL `https://huggingface.co/tesslate/UIGEN-T2`.

Tianlu Wang, Ilia Kulikov, Olga Golovneva, Ping Yu, Weizhe Yuan, Jane Dwivedi-Yu, Richard Yuanzhe Pang, Maryam Fazel-Zarandi, Jason Weston, and Xian Li. Self-taught evaluators. *CoRR*, abs/2408.02666, 2024.

Zhiheng Xi, Jixuan Huang, Xin Guo, Boyang Hong, Dingwen Yang, Xiaoran Fan, Shuo Li, Zehui Chen, Junjie Ye, Siyu Yuan, Zhengyin Du, Xuesong Yao, Yufei Xu, Jiecao Chen, Rui Zheng, Tao Gui, Qi Zhang, and Xuanjing Huang. Critique-RL: Training language models for critiquing through two-stage reinforcement learning. *CoRR*, abs/2510.24320, 2025.

Tianshu Yu, Chao Xiang, Mingchuan Yang, Pei Ke, Bosi Wen, Cunxiang Wang, Jiale Cheng, Li Zhang, Xinyu Mu, Chuxiong Sun, and Minlie Huang. Training language model to critique for better refinement. In *ACL (Findings)*, volume ACL 2025 of *Findings of ACL*, pp. 26760–26804. Association for Computational Linguistics, 2025.

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 *NeurIPS*, 2024.

Xiaoying Zhang, Hao Sun, Yipeng Zhang, Kaituo Feng, Chaochao Lu, Chao Yang, and Helen Meng. Critique-grpo: Advancing LLM reasoning with natural language and numerical feedback. *CoRR*, abs/2506.03106, 2025.

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

APPENDIX

A IMPLEMENTATION AND TRAINING DETAILS

We provide additional implementation details for WebLoop training, complete definitions of the objectives simplified in the main text, and the evaluation protocols used for controlled comparisons.

A.1 TRAINING AND ROLLOUT CONFIGURATION

Optimization setup. We instantiate WebLoop with Qwen3.5-9B and Qwen3.5-27B, and conduct all controlled ablations with Qwen3.5-9B. Training uses the 2,880 executable tasks described in Sec. 3.4, referred to as approximately 3K tasks in the main text, with a task batch size of 64. We optimize the shared policy with GRPO using a learning rate of 1×10^{-6} and no explicit KL penalty. Rollouts use a temperature of 0.7 and top-p of 1.0, with maximum prompt and response lengths of 16,384 tokens.

Rollout structure. For each task, the Generator samples $n_g = 3$ initial implementations. After selecting one valid imperfect draft, the Critic samples $n_c = 3$ critiques, each of which conditions $n_r = 3$ independent refinements. A complete rollout therefore contains 3 Generator outputs, 3 Critic outputs, and 9 Refiner outputs, including 12 executable Web pages. If no valid imperfect draft exists, the task contributes only its Generator samples. The three roles are sampled sequentially and jointly optimized in one GRPO update as described in Sec. 3.3.

Critic schedule. For the first 15 policy updates, the Critic is optimized only with diagnostic discriminability $D(c)$. Subsequent updates use $H(c) + \lambda D(c)$ with $\lambda = 0.2$, thereby introducing retrospective helpfulness while retaining requirement-level diagnostic supervision. Generator, Critic, and Refiner share all model parameters throughout training.

System implementation. Our implementation is built on `verl` with an FSDP2 actor and asynchronous vLLM rollout workers. Policy weights are synchronized to the rollout workers after each update to maintain on-policy sampling. Browser execution is parallelized independently from model rollout, with up to 72 concurrent evaluations and a per-example hard timeout of 2,400 seconds.

A.2 ADDITIONAL OBJECTIVE DETAILS

Candidate selection. Let $\mathcal{Y}_x^{(1)} = \{y_i^{(1)}\}_{i=1}^{n_g}$ denote the initial implementations sampled for task x. We retain candidates that receive a valid browser evaluation but do not fully solve the task:

$$\mathcal{Y}_x^- = \left\{ y_i^{(1)} \in \mathcal{Y}_x^{(1)} \mid y_i^{(1)} \text{ is valid},\ r_{\text{web}}\big(y_i^{(1)}\big) < 1 \right\}. \tag{12}$$

The shared base draft is the median-reward candidate in this set,

$$y_b = \text{Median}_{r_{\text{web}}}\left(\mathcal{Y}_x^-\right). \tag{13}$$

This choice keeps all critiques anchored to the same implementation while avoiding both solved examples and unusually poor failures. If $\mathcal{Y}_x^-$ is empty, no Critic or Refiner branch is expanded for that task.

Complete discriminability definition. For the selected draft y_b, let $z_q \in \{0, 1\}$ denote the execution-grounded status of requirement r_q, and let $\hat{z}_{j,q}$ be the corresponding prediction from critique c_j. We define

$$\mathcal{F} = \{q : z_q = 0\}, \qquad \mathcal{P} = \{q : z_q = 1\}, \tag{14}$$

and the class-wise accuracies

$$a_{\text{fail}}(c_j) = \frac{1}{|\mathcal{F}|} \sum_{q \in \mathcal{F}} \mathbb{I}[\hat{z}_{j,q} = 0], \qquad a_{\text{pass}}(c_j) = \frac{1}{|\mathcal{P}|} \sum_{q \in \mathcal{P}} \mathbb{I}[\hat{z}_{j,q} = 1], \tag{15}$$

whenever the corresponding set is non-empty. The full discriminability score is

$$D(c_j) = \begin{cases} a_{\text{fail}}(c_j) + a_{\text{pass}}(c_j) - 1, & |\mathcal{F}| > 0,\ |\mathcal{P}| > 0, \\ 2a_{\text{fail}}(c_j) - 1, & |\mathcal{F}| > 0,\ |\mathcal{P}| = 0, \\ 2a_{\text{pass}}(c_j) - 1, & |\mathcal{F}| = 0,\ |\mathcal{P}| > 0, \\ 0, & Q_x = 0. \end{cases} \tag{16}$$

When both classes are present, this reduces to Youden's J and assigns equal importance to satisfied and violated requirements. The single-class cases preserve the same $[-1, 1]$ range.

Expected refinement utility. The helpfulness of a critique is defined through the quality of refinements generated under that critique. For a fixed (x, y_b, c_j), the conditional refinement utility is

$$U(c_j \mid x, y_b) = \mathbb{E}_{y \sim \pi_{\theta_{\text{old}}}(\cdot \mid p_{\text{ref}}(x, y_b, c_j))} \left[r_{\text{web}}(y) \right]. \tag{17}$$

We estimate this expectation using the n_r sampled refinements,

$$\widehat{U}(c_j) = \frac{1}{n_r} \sum_{k=1}^{n_r} r_{\text{web}}\big(y_{j,k}^{(2)}\big), \tag{18}$$

and define retrospective helpfulness as

$$H(c_j) = \widehat{U}(c_j) - r_{\text{web}}(y_b). \tag{19}$$

Thus, $H(c_j)$ measures the expected execution gain obtained when the current Refiner acts on the critique rather than the linguistic form of the critique itself.

Role-specific rewards. For completeness, the scalar reward assigned to any policy output o is

$$r(o) = \begin{cases} r_{\text{web}}\big(y_i^{(1)}\big), & o = y_i^{(1)}, \\ r_{\text{crit}}(c_j), & o = c_j, \\ r_{\text{web}}\big(y_{j,k}^{(2)}\big), & o = y_{j,k}^{(2)}. \end{cases} \tag{20}$$

These heterogeneous rewards are normalized only within the role-specific comparison groups defined in Sec. 3.3. Critiques that cannot be parsed into the required requirement-level judgments receive the minimum Critic reward and do not instantiate downstream refinement branches.

A.3 Browser Evaluation and Baseline Adaptation

Browser evaluation. For WebRise (Meng et al., 2026), all generated pages are evaluated with the same Interaction Contract Graph and browser executor used to construct the execution rewards in Sec. 3.1. The evaluator executes the prescribed interactions and assertions and returns transition validity, explicit and implicit requirement coverage, and requirement-level execution labels. For WebGen-Bench (Lu et al., 2025b), we use the benchmark-provided evaluation harness and disable automatic starter-template selection (`autoSelectTemplate=false`), requiring each model to construct the website from an empty workspace. Pages that fail to launch are reported as Start-Fail separately from functional failures after initialization.

Same-backbone controls. The *refine (w/o RL)* baseline uses the original Qwen3.5 checkpoint for Generator, Critic, and Refiner and performs one generate–critique–refine cycle without parameter updates. Generation Only optimizes only the initial generation branch. Refine without Critique trains generation and a second-pass refinement without learned critique guidance. Staged Training first optimizes generation and subsequently trains critique and refinement, allowing direct comparison with WebLoop's joint optimization.

Critique-learning baselines. We additionally adapt Critique-GRPO (Zhang et al., 2025) and Critique-RL (Xi et al., 2025) to functional Web generation. For Critique-GRPO, a fixed teacher receives requirement-level execution information during training and provides feedback for policy optimization. For Critique-RL, the Generator and Refiner actor is kept fixed while only the Critic is optimized, following its actor–critic separation. At evaluation time, the learned Critic provides feedback that conditions refinement by the same underlying actor. All controlled variants use the same task distribution and browser evaluator as WebLoop wherever applicable.

Evaluation consistency. All same-backbone methods are initialized from the same Qwen3.5 checkpoints and evaluated on identical benchmark instances with the same browser infrastructure and metric computation. For WebLoop, "gen" denotes the first-pass output of the trained shared policy, while "refine" denotes the output after one critique-conditioned refinement cycle. External open-source, proprietary, and Web-specialized models use the same benchmark-specific execution harness while retaining the inference interfaces supported by their released implementations.

B EXECUTABLE TRAINING DATA CONSTRUCTION DETAILS

We provide additional details on the construction of the executable Web training corpus introduced in Sec. 3.4. The objective is to pair natural-language task specifications with browser-executable supervision that captures both whether a requested function is present and whether its behavior remains correct across interactions. This construction provides page-level execution signals for the Generator and Refiner and requirement-level supervision for Critic learning.

B.1 TASK AND REQUIREMENT CONSTRUCTION

Domain and scenario coverage. We construct the corpus around functional Web applications rather than isolated visual layouts. The task space spans 21 domains, including e-commerce, office productivity, education, developer tools, and data analytics. Within each domain, we instantiate concrete scenarios covering core operations, management functions, user settings, data presentation, and multi-step workflows. The resulting corpus contains 1,442 scenarios, which are further instantiated into 2,880 executable Web generation tasks.

Functional task instantiation. Each task specifies a target Web application through a natural-language description together with a set of functional requirements. Following the notation in the main text, we write a task as

$$x = (d, \mathcal{R}_x), \tag{21}$$

where d denotes the task description and $\mathcal{R}_x$ contains the requirements that define the expected behavior. Unlike page-generation data centered primarily on appearance or static content, these requirements describe interactive functionality that can subsequently be tested through browser execution. This makes each task suitable for evaluating both the initial implementation and its later refinement under the same functional specification.

Explicit and implicit requirements. We decompose the requirement set into explicit and implicit components,

$$\mathcal{R}_x = \mathcal{R}_x^{\text{exp}} \cup \mathcal{R}_x^{\text{imp}}. \tag{22}$$

Explicit requirements describe functions and interaction entry points directly requested by the task specification. They capture what operations the generated page should expose and provide the primary requirement-level targets inspected by the Critic. Implicit requirements instead describe behavioral constraints needed for these functions to operate consistently, including state consistency, operation ordering, boundary handling, and reset behavior. They therefore extend supervision beyond the presence of an interface element to the correctness of the behavior induced by interacting with it.

The final corpus contains an average of 11.87 requirements per task, including 5.47 explicit and 6.39 implicit requirements. Implicit requirements account for 53.9% of all requirements, reflecting the importance of interaction semantics that are not fully characterized by explicit function descriptions alone. This decomposition also supports the separate explicit- and implicit-requirement coverage terms used by the execution evaluator.

B.2 INTERACTION CONTRACT CONSTRUCTION AND EXECUTABILITY FILTERING

From requirements to executable contracts. Following the executable evaluation formulation of WebRISE (Meng et al., 2026), we convert each task specification into an initial-state contract and an Interaction Contract Graph (ICG). The initial-state contract defines the starting condition under which the subsequent browser interactions are executed. The ICG then operationalizes the functional requirements as executable interaction trajectories and corresponding assertions. In this way, natural-language requirements are transformed into programmatically verifiable behaviors rather than evaluated only through textual or visual similarity.

Interaction Contract Graph. For a task x, its ICG G_x organizes browser execution through states, transitions, and functional assertions. Transitions describe the interactions used to move between Web states, while assertions verify whether the resulting behavior satisfies the corresponding functional constraints. A single task can therefore contain multiple interaction transitions and multiple assertions, enabling evaluation of multi-step functionality rather than only isolated final states.

Table 5: **Human validation of functional evaluation.** Agreement between the automatic evaluator and two independent human reviewers on the same 300 sampled interaction cases. Human–human agreement is included as a reference.

Comparison	#Samples	Acc.↑	MAE↓	Spearman↑	Pearson↑	κ ↑
Human (A) vs. Human (B)	300	0.920	0.079	0.924	0.923	0.838
Evaluator vs. Human (A)	300	0.880	0.095	0.901	0.902	0.758
Evaluator vs. Human (B)	300	0.900	0.094	0.902	0.898	0.799

Across the corpus, each ICG contains 10.64 states, 11.40 transitions, and 28.15 executable assertions on average.

Executing a generated implementation y against G_x produces the execution signals used throughout WebLoop,

$$E(y; G_x) = (T(y), R_e(y), R_i(y), \mathbf{z}(y)), \tag{23}$$

where T measures transition validity, R_e and R_i measure explicit- and implicit-requirement coverage, and $\mathbf{z}$ provides requirement-level execution labels. The first three quantities form the page-level reward used for Generator and Refiner optimization, while the requirement-level labels provide execution-grounded supervision for Critic discriminability. Thus, the same executable contract supports both outcome-level optimization and intermediate Critic credit assignment.

Reference implementation and browser validation. To verify executability and reduce dependence on any single generator, we independently construct reference HTML implementations using Kimi-K2.6, Claude Opus 5, and GPT-5, and execute each against the associated ICG in a real browser. We retain a task only if all three implementations achieve transition validity of at least 0.60:

$$T_m \geq 0.60, \qquad \forall m \in \{1, 2, 3\}, \tag{24}$$

where T_m denotes the transition validity of the implementation generated by the m-th model. This cross-model criterion filters tasks with insufficiently reliable interaction contracts before training.

Human review and filtering. Browser-executable contracts may still be semantically misaligned with the intended requirements, for example by omitting necessary prerequisites or using assertions that execute successfully but do not faithfully test the target behavior. We therefore conduct a manual audit of sampled candidate tasks with two reviewers. For each audited task, the reviewers examine five aspects: (1) *requirement alignment*, whether each tested behavior is supported by an explicit requirement or a justified implicit constraint; (2) *interaction coherence*, whether the action sequence includes the necessary preconditions and follows a valid progression toward the target state; (3) *assertion validity*, whether the expected outcomes correctly reflect the intended functional semantics; (4) *structural consistency*, whether states, transitions, and requirement references are internally consistent with the ICG specification; and (5) *functional discriminability*, whether the checks distinguish actual functional completion from superficial interface presence. Audited tasks containing invalid or misaligned interaction contracts are excluded before corpus finalization.

Final corpus. After executable validation and manual filtering,, the training corpus contains 2,880 tasks across 21 domains and 1,442 scenarios. Together, their explicit and implicit requirements and associated ICGs provide fine-grained supervision over both functional outcomes and intermediate interaction behavior. This construction turns each natural-language Web task into an executable training unit that can evaluate initial generation, diagnose requirement-level failures, and measure the downstream effect of critique-conditioned refinement.

B.3 Human Consistency Validation

Annotation setup. We further assess the reliability of the automatic functional evaluator through human consistency analysis on 300 sampled interaction cases. For each case, two reviewers independently judge whether the expected behavior is satisfied based on the task requirements, tested interaction, and corresponding execution evidence.

Metrics. We report accuracy, mean absolute error (MAE), Spearman correlation, Pearson correlation, and Cohen's κ. Accuracy and Cohen's κ measure agreement on binary pass/fail judgments.

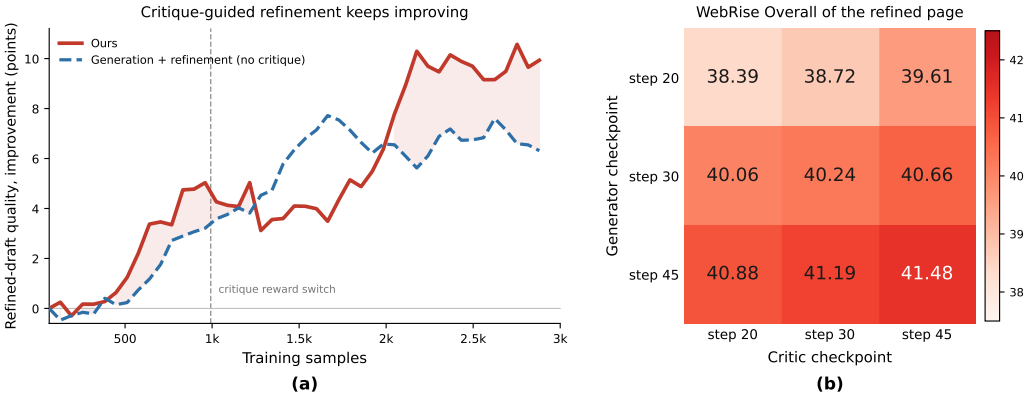


Figure 4: **Refinement learning and Generator–Critic checkpoint interaction.** (a) Refined-page quality over training, reported as improvement from the initial training point, with refinement without learned critique as a control. The dashed line marks the Critic reward switch. (b) WebRise Overall under cross-pairing Generator and Critic checkpoints from steps 20, 30, and 45. The Refiner uses the same checkpoint as the Generator in each pairing.

MAE and the correlation coefficients are computed over paired human and automatic graded scores normalized to $[0, 1]$. As shown in Table 5, the evaluator shows strong consistency with both reviewers across accuracy, MAE, rank and linear correlation, and Cohen's κ, while human–human agreement provides an upper-reference level. These results indicate that the automatic evaluator closely tracks human judgments and provides a reliable execution-based supervision signal for WebLoop training and evaluation.

C Additional Experimental Results and Analysis

C.1 Training Dynamics and Loop Co-Adaptation

Refinement gains over training. Fig. 4(a) compares WebLoop with generation and refinement trained without a learned Critic. Both improve with training, but critique-guided refinement develops a larger advantage, particularly after downstream helpfulness is introduced. This trend suggests that WebLoop increasingly exploits learned intermediate feedback rather than benefiting only from an additional generation pass.

Generator–Critic checkpoint interaction. Fig. 4(b) cross-pairs Generator and Critic checkpoints to separate their contributions over training. For every fixed Generator, later Critic checkpoints improve refinement quality; likewise, later Generators improve performance under every fixed Critic. The score rises from 38.4 for the step-20 pair to 41.5 for the step-45 pair, with the best result obtained when both use the latest checkpoint. These results indicate that both roles improve progressively and contribute jointly to the final refinement gain.

C.2 Fine-Grained Functional Evaluation

WebGen-Bench category breakdown. Table 6 decomposes WebGen-Bench along two independent taxonomies provided by the benchmark. Website categories group checks into Content Presentation, User Interaction, and Data Management, while check categories distinguish Functional Testing, Data Display Testing, and Design Validation. Each taxonomy covers all 647 checks, so the two groups provide complementary views of performance rather than six mutually exclusive categories. We use the same template-free evaluation protocol and scoring rule as in the main results.

Where does refinement help? As shown in Table 6, WebLoop improves functional behavior across a broad range of categories rather than only interface-level compliance. For Qwen3.5-9B, the final Refine output improves over the base model by 11.8 points on Functional Testing and 22.0 points on Data Display Testing, while also increasing all three website-category scores. The 27B model

Table 6: **Fine-grained WebGen-Bench results.** Accuracy under the same template-free setting as Table 1. The 647 interaction checks are independently grouped by website category and check category; each grouping covers the full benchmark, so the six columns are not additive. † denotes proprietary models.

Model / Method	By Website Category			By Check Category		
	Content Presentation	User Interaction	Data Management	Functional Testing	Data Display Testing	Design Validation
General-Purpose Models						
Qwen3.5-122B-A10B	21.3	9.6	10.6	5.2	18.0	27.0
Kimi-K2.6	42.5	26.4	28.4	20.9	37.9	49.6
Qwen3.5-397B-A17B	27.3	19.2	16.2	11.5	24.5	40.2
GLM-5.3	26.1	25.9	28.1	20.2	25.8	45.1
Claude Opus 4.7†	**67.0**	**45.8**	**61.3**	**41.3**	**69.1**	**73.4**
Gemini 3.1 Pro†	46.6	45.4	48.4	35.4	50.5	70.9
GPT-6 Astra†	48.6	39.0	45.6	30.5	51.1	66.4
Web-Generation Specialized Models						
WebGen-LM-7B	2.0	**5.4**	2.5	**1.6**	**4.8**	8.2
WebGen-Agent-7B (Step-GRPO)	**6.0**	5.0	**5.6**	0.4	3.8	**21.7**
WebGen-LM-32B	2.9	4.3	1.9	1.2	3.8	8.6
UIGEN-T2-7B	N/A	N/A	N/A	N/A	N/A	N/A
WebGen-LM-14B	1.7	5.0	2.5	1.3	**4.8**	7.4
Controlled Comparisons: Qwen3.5						
Base (Qwen3.5-9B)	29.0	17.1	30.0	15.5	25.0	43.4
Generate–Critique–Refine (w/o RL)	42.0	24.0	33.1	22.1	34.7	50.4
Critique-GRPO	12.6	17.1	24.1	11.1	16.9	36.9
Critique-RL	28.7	28.3	28.7	19.5	30.4	50.8
WebLoop (Gen)	41.1	**31.2**	32.5	23.0	37.9	**59.4**
WebLoop (Refine)	**50.3**	30.2	**43.4**	**27.3**	**47.0**	58.6
Base (Qwen3.5-27B)	45.1	23.6	25.0	17.1	39.5	50.0
Generate–Critique–Refine (w/o RL)	48.6	33.2	**36.2**	24.0	50.3	58.6
WebLoop (Gen)	43.4	27.6	30.0	19.0	41.7	55.7
WebLoop (Refine)	**59.2**	**37.1**	**36.2**	**29.5**	**51.3**	**66.8**

shows the same pattern, including gains of 12.4 points on Functional Testing and 11.8 points on Data Display Testing. Compared with first-pass WebLoop generation, refinement is more heterogeneous at 9B: it substantially improves Content Presentation, Data Management, Functional Testing, and Data Display Testing, while User Interaction and Design Validation decrease slightly by 1.0 and 0.8 points. At 27B, refinement improves all six categories, suggesting that larger models exploit critique-conditioned revision more consistently.

C.3 Critique Diagnostics and Repair Analysis

Critique length and refinement quality. Fig. 5 examines the association between critique length and downstream refinement quality under different Critic objectives. For WebLoop, refined WebRise Overall increases from 38.5 in the shortest bucket to 54.0 in the longest, whereas the alternative objectives show weaker or non-monotonic patterns. This contrast suggests that critique utility depends on how additional feedback is structured and used, rather than on verbosity alone. Because critique length is neither controlled nor independently randomized, we treat this analysis as descriptive rather than causal.

C.4 Qualitative Analysis of the Improvement Loop

We utilize qualitative examples to examine where the improvement loop succeeds or breaks down between requirement diagnosis and executable repair. We organize the analysis around successful repair and three distinct sources of failure: incorrect Critic judgments, insufficient repair guidance, and failures in Refiner implementation.

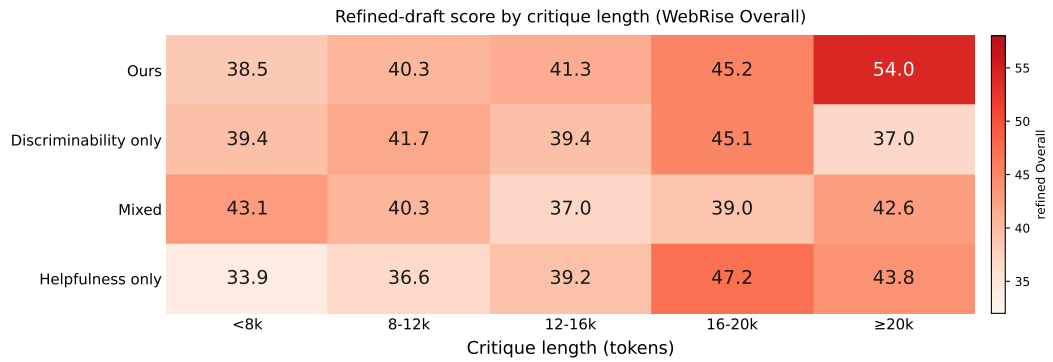


Figure 5: **Critique length versus refinement quality.** Each cell reports the mean WebRise Overall of refined pages within a critique-length bucket for a given Critic objective. WebLoop exhibits a stronger positive association, while alternative objectives show weaker or non-monotonic trends.

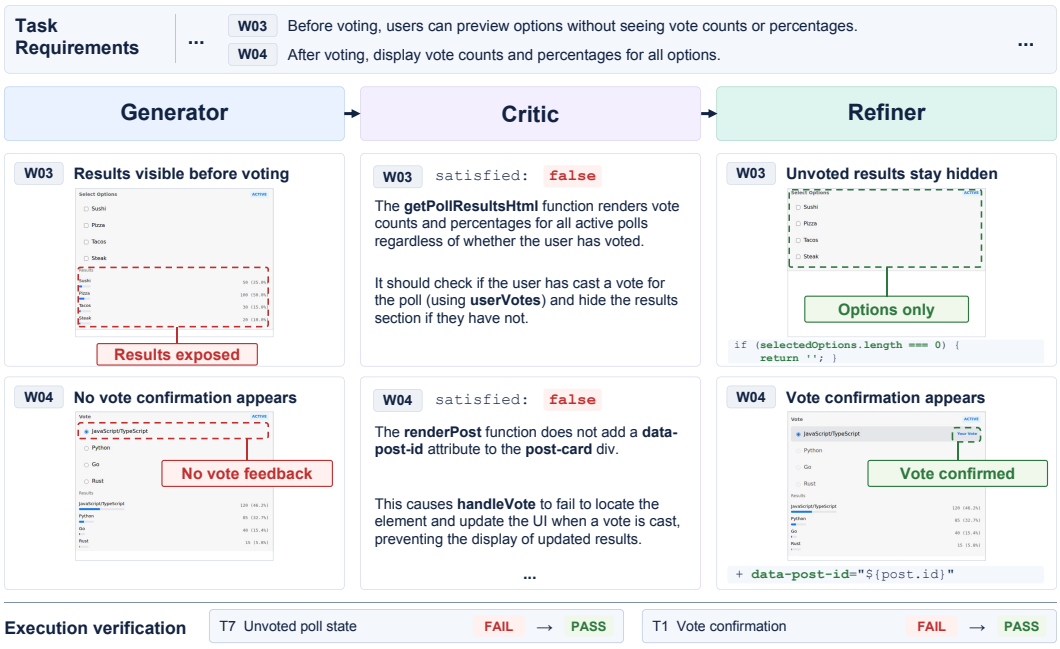


Figure 6: **Successful critique-guided repair on Poll Vote (WebRise).** The Critic localizes premature disclosure of voting results and a missing post identifier that blocks vote updates. The Refiner applies the corresponding fixes, changing both affected transition checks from FAIL to PASS.

Successful diagnosis and repair. Fig. 6 illustrates how an execution-free Critic can support targeted functional repair. For W03, the Critic identifies that voting results are exposed before a vote is cast and localizes the missing state-dependent guard. For W04, it traces the failed post-vote update to a missing `data-post-id` attribute. Conditioned on these diagnoses, the Refiner applies targeted code changes that turn both associated browser checks from FAIL to PASS. This example demonstrates how requirement-level diagnosis can yield actionable guidance and executable improvement.

The file-filter example in Fig. 7 provides a complementary boundary-condition case. Although clearing the controls already works, empty size inputs are parsed as `NaN`, causing all files to be excluded. The Critic localizes this defect, and the Refiner adds guards that restore all 12 files while preserving the correct control-reset behavior. Across both examples, the diagnosed defect, implemented change, and browser-verified improvement are well aligned.

Critic judgment errors. The Critic may incorrectly classify already functional behavior as violating a requirement. In Fig. 8, it attributes broken message selection to a stale hidden-checkbox property. However, a CSS class controls the visible selection marker and a message-ID set controls the preview, and both behaviors already pass in the draft. The Refiner synchronizes the hidden checkbox while preserving these mechanisms, so the relevant predicates remain PASS. This case therefore reflects a false-positive Critic judgment rather than a failed Refiner implementation: preserving correct behavior after revision does not validate the preceding negative verdict.

Limitations of Critic repair guidance. Even when the requirement verdict is correct, the proposed repair may fail to address the defect responsible for the unmet behavior. In Fig. 9, the Critic identifies a genuine type-matching issue but does not address the required per-tab counts. The Refiner applies the suggested edit, yet the counts remain absent. The failure therefore reflects insufficient repair guidance rather than failure to follow the recommendation.

The paired critiques in Fig. 10 further isolate this distinction. They share the same draft, requirement verdicts, and discriminability score ($D = 0.75$), but differ in fault localization. Only Critique A identifies the conflation of petition-opening and signing states. Its corresponding refinement restores the disabled signing behavior, whereas Critique B's modal-close edit is implemented without removing the blocker. Thus, verdict-level correctness alone is insufficient to establish the correctness of the proposed repair direction.

Repair guidance may also be valid but incomplete. In Fig. 11, the Critic identifies a missing listener and the Refiner adds it, but an unaddressed selector error leaves the header checkbox unchecked. The prescribed edit is implemented correctly, yet the target behavior remains unsatisfied because the guidance does not cover all contributing defects.

Refiner implementation failures. Failure can instead arise when the Refiner does not faithfully realize an appropriate recommendation. In Fig. 12, both critiques correctly identify the missing event binding and explicitly recommend adding it, but one displayed refinement leaves the binding absent and fails. Another refinement conditioned on the same critique adds the binding and passes. Unlike the incomplete-guidance cases above, the required repair is already specified here; the failure lies in executing that guidance.

The Refiner may also introduce regressions while addressing genuine defects. In Fig. 13, the Critic correctly identifies missing liked-state feedback, but the revision conflates the user's like state with the aggregate like count, causing the count predicate to regress from PASS to FAIL. Thus, unsuccessful refinement may result either from failing to follow valid guidance or from introducing new implementation errors.

Together, these cases distinguish whether the Critic judges the requirement correctly, whether its guidance is sufficient, and whether the Refiner implements that guidance successfully. Discriminability evaluates the first, whereas helpfulness captures downstream outcomes jointly determined by feedback and implementation. A failed refinement therefore does not by itself identify the source of failure; the intermediate diagnosis and resulting code changes are required to localize where the improvement loop breaks down.

D PROMPT TEMPLATES

WebLoop utilizes role-specific prompts to instantiate the Generator, Critic, and Refiner within the shared policy.

Generator. The Generator prompt in Fig. 14 directly maps the task specification to a complete self-contained HTML implementation, restricting the output to executable code without auxiliary explanation.

Critic. The Critic prompt in Fig. 15 adopts an execution-free red-team perspective, requiring requirement-level judgments from the specification and source code alone and structured repair guidance for detected failures.

Refiner. The Refiner prompt in Fig. 16 places diagnosed defects before the prior implementation, encouraging targeted rewrites while preserving unflagged behavior.

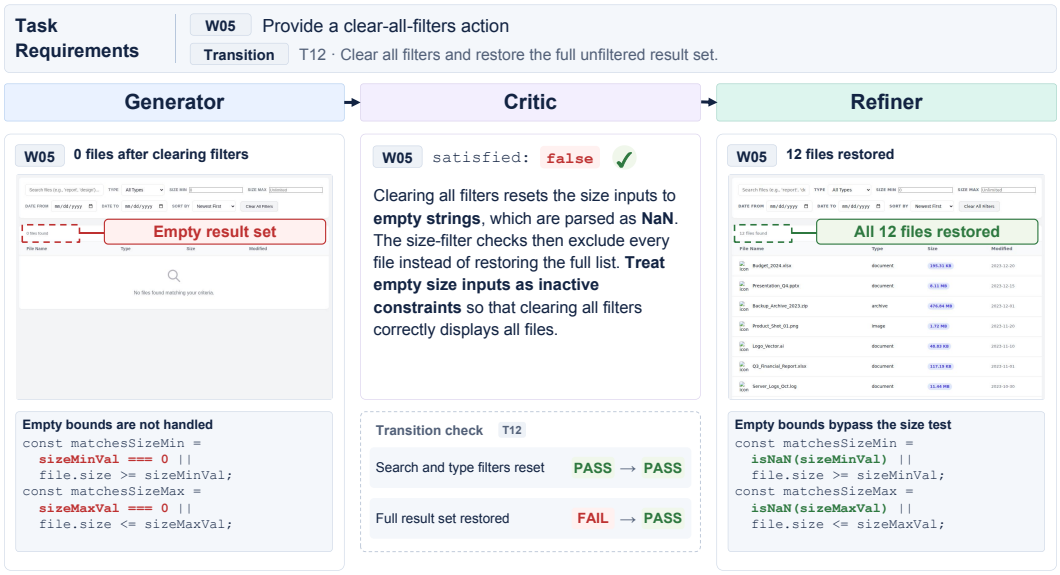


Figure 7: **Boundary-condition repair on File Search with Filters (WebRise).** The Critic traces the empty result set to size bounds parsed as `NaN` after clearing filters. The Refiner adds guards for empty bounds, restoring all 12 files while preserving the already-correct control reset.

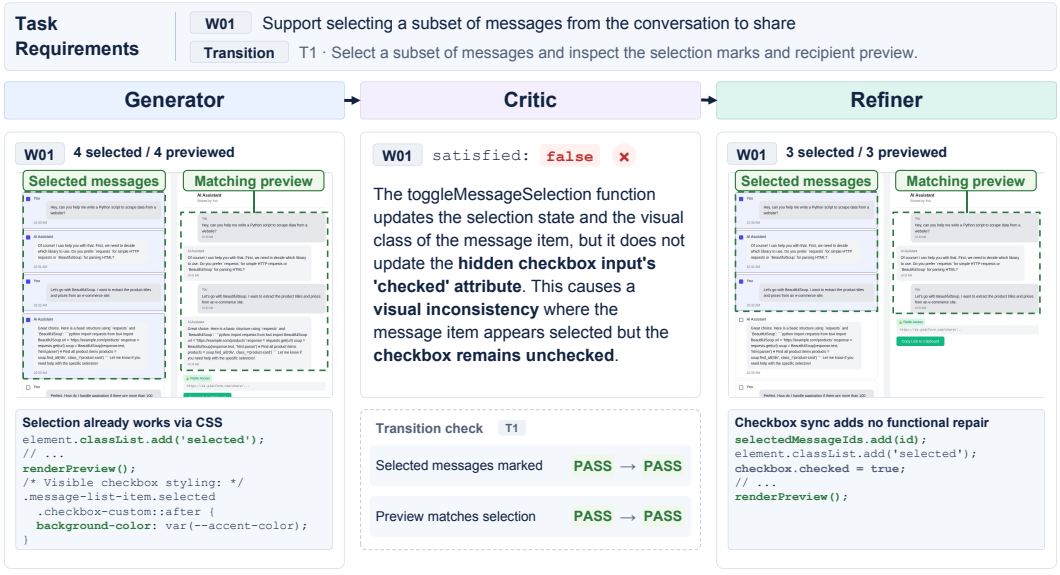


Figure 8: **False-positive Critic judgment on Share Chat Link Interface (WebRise).** The Critic attributes message-selection failure to an unsynchronized hidden checkbox, although the visible selection and preview are already controlled correctly. The Refiner synchronizes the checkbox without affecting either passing behavior.

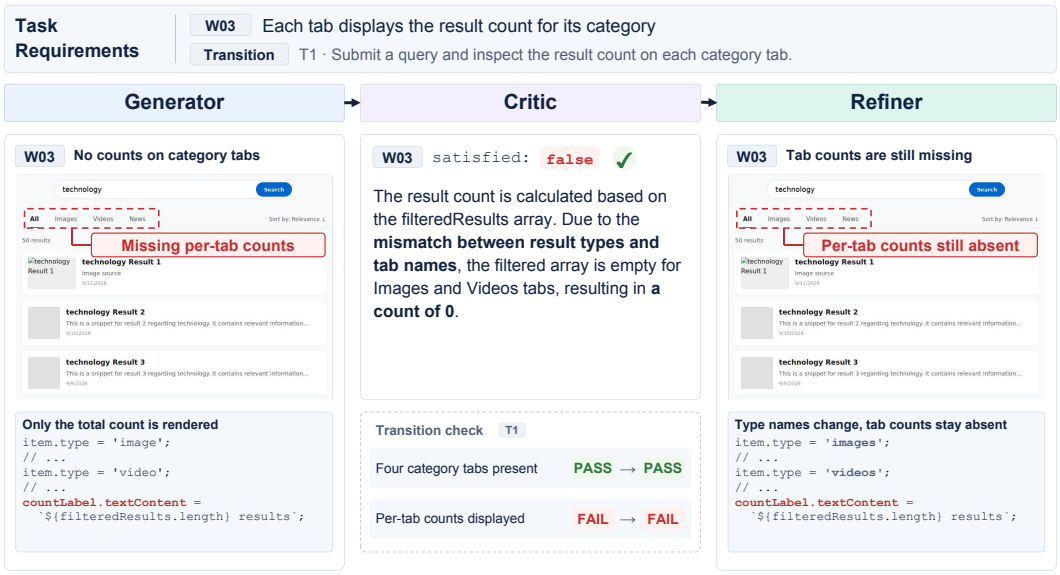


Figure 9: **Correct verdict but mislocalized repair guidance on Search Result Tabs (WebRise).** The Critic correctly marks the per-category count requirement as unmet but targets a type-matching issue rather than the missing count labels. The Refiner applies the proposed edit, while the target predicate remains FAIL.

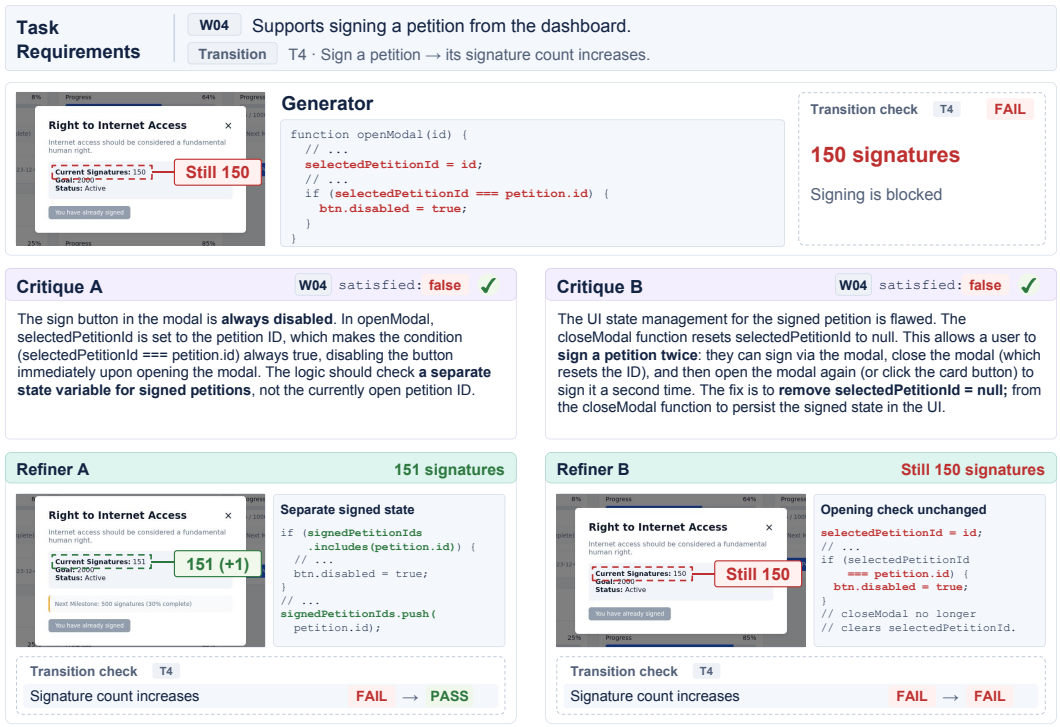


Figure 10: **Divergent fault localization on Petition Progress Dashboard (WebRise).** Both critiques correctly identify signing as broken, but only one localizes the state error that disables signing when the dialog opens. Although both suggested edits are implemented, only the correctly localized repair changes the transition from FAIL to PASS.

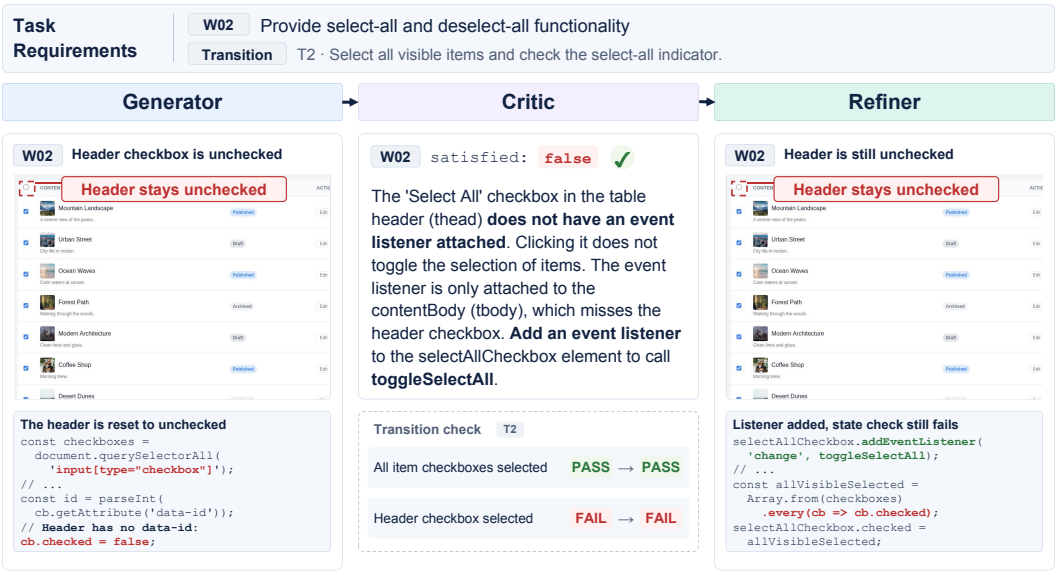


Figure 11: **Incomplete repair guidance on Bulk Content Operations (WebRise).** The Refiner adds the missing listener as recommended, but an unaddressed selector error still clears the header-checkbox state. The prescribed edit is implemented, yet the target requirement remains unsatisfied.

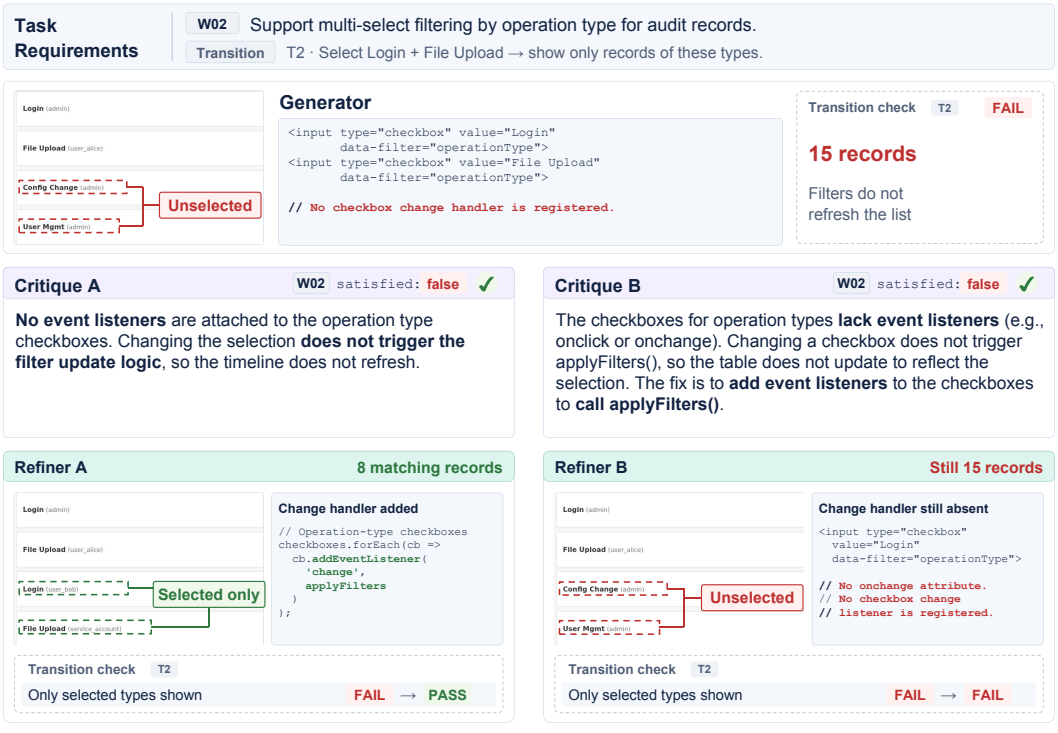


Figure 12: **Divergent Refiner outcomes under equivalent diagnoses on Audit Timeline (WebRise).** Both critiques identify the same missing event binding, but only one refinement implements the recommended listener and passes the transition check. The contrast isolates implementation fidelity from Critic diagnosis.

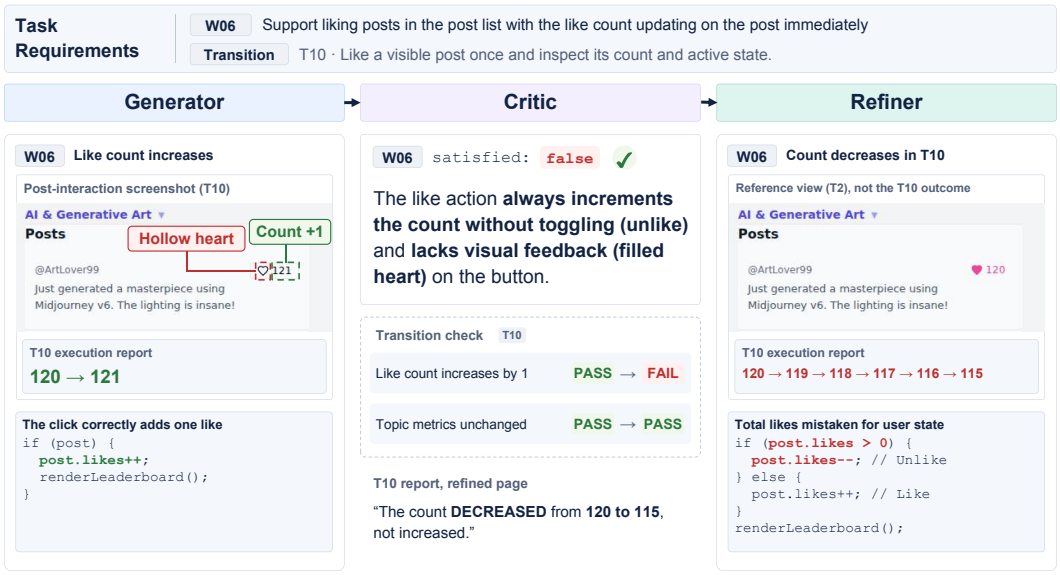


Figure 13: **Refiner-induced regression on Topic Trending (WebRise).** While addressing missing liked-state feedback, the Refiner conflates the user's like state with the aggregate like count, causing the count-update predicate to regress from PASS to FAIL.

Generator

You are an expert frontend developer. Given a task description and a set of functional requirements, produce a single, complete, self-contained HTML file that fulfills all requirements.

Rules:

1. Output only the HTML code, starting with `<!DOCTYPE html>` and ending with `</html>`.
2. Do not include any explanation, markdown fences, or commentary.
3. Inline all CSS and JavaScript; no external files.
4. The page must be fully functional on first load, with realistic placeholder data.
5. Implement every explicit requirement completely.

Input: the task statement {`input_text`}, which already carries the prose requirements and the test data contract, so the generator builds against exactly the statement the evaluation is derived from.

Output: the complete HTML page, with no surrounding prose.

Figure 14: Generator prompt for first-pass Web generation.

Critic

You are a red-team QA engineer. Your explicit goal is to break this self-contained HTML page. You are given the task it was built for, its explicit requirements (each with an id), and the page. You have no automated test results — find breakage by reading the JavaScript. Assume every requirement is broken until the code proves otherwise; a polished-looking UI is not proof.

Where the attacks come from: the strongest attacks come from the *implicit constraints* each explicit requirement entails — the sub-behaviours a careful spec author derives from it: validation of invalid inputs (negative, zero, empty, non-numeric, out-of-range); persistence across save, re-selection, and re-render; consistency while a filter or sort is active or after the data changes; reversibility (add ↔ delete, enable ↔ clear); and robust DOM handling under repeated or out-of-order use.

Procedure, for each explicit requirement:

1. Derive the implicit constraints it entails.
2. Find the input or action sequence most likely to violate one, and confirm the failure in the code.
3. Mark `satisfied` true only if you could not break the explicit behaviour or any entailed constraint.

Output rules:

1. Keep any internal reasoning extremely brief (at most a few sentences per requirement). Do not re-list task data, never re-enumerate every element of an input array, never quote every field, and never repeat a check already made.
2. Do not loop on the same doubt; after one brief re-check, commit to a verdict and move on.
3. Respond with a single JSON object and nothing after it: no reasoning after the JSON, no markdown fences around it.
4. Emit exactly one entry per requirement id given, in the same order, each with a boolean `satisfied`.
5. Include `fix` only where `satisfied` is false. It must state the breaking case, the entailed constraint it violates, the code defect, and the concrete change — describing behaviour and code without mentioning any requirement id.
6. Inside `fix` strings use plain prose; do not quote code with backticks or backslashes, which break JSON parsing.

Output: Return exactly one JSON object:

```
{"requirements": [
  {"id": "<requirement id>", "satisfied": true},
  {"id": "<requirement id>", "satisfied": false, "fix": "..."}
]}
```

Figure 15: Critic prompt for requirement-level diagnosis and repair guidance.

Refiner

You are an expert frontend developer fixing a web page that failed automated testing. A reviewer has flagged specific broken features, each with the defect and the fix.

Treat the existing code for every flagged feature as broken — do not trust or copy it. For each flagged defect, delete the current implementation of that feature and rewrite it from scratch so the described interaction works correctly: correct input validation, state updates, persistence, and DOM handling. Keep the code only for features the reviewer did *not* flag.

Rules:

1. For every flagged defect, rewrite that feature's JavaScript and DOM logic fresh; do not merely tweak the old code.
2. Preserve unflagged, working features as-is.
3. Output only the corrected HTML, from `<!DOCTYPE html>` to `</html>`. No commentary, no markdown fences. Inline all CSS and JavaScript.

Input layout: the prompt presents, in this order, the defect list to apply (`{critique}`), what the page must do (`{input_text}`), and the current page to repair (`{prev_html}`). Foregrounding the defects before the previous page is deliberate: the refiner otherwise inherits a strong prior to copy its own earlier output and ignore the critique.

Output: the corrected, complete HTML page, with every flagged feature rewritten.

Figure 16: Refiner prompt for critique-conditioned Web repair.