WEBCODER: SCALING GENERAL FRONTEND DEVELOPMENT WITH ARTIFACT-GROUNDED AGENTIC REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Real-world frontend development requires agents to compose reproduction, generation, modification, and repair over an evolving application state. Learning a general policy for this process requires translating heterogeneous execution feedback into task-aligned rewards and effective credit assignment across long sequences of interdependent actions. We introduce **WebCoder**, a general frontend development agent trained with artifact-grounded agentic reinforcement learning. Our key idea is to assess diverse development activities through their effects on a shared executable artifact and its satisfaction of task requirements. We build Webdev-World, an executable environment that integrates code editing, application execution, browser inspection, and on-demand skills while preserving project state across interactions. An independent verifier evaluates the final application against task requirements to provide terminal rewards, while changes in intermediate artifact progress redistribute the resulting rollout advantage across development segments. Context compaction and message-tree sample construction support training on trajectories beyond a single context window while avoiding repeated loss computation on shared generated history. We further introduce Webdev-Bench to evaluate composed development requirements within a single project. On Webdev-Bench, WebCoder-27B raises requirement satisfaction from 64.8 for its Qwen3.8-27B base model to 72.8, exceeding Claude-Opus-4.8 (65.1) and GPT-5.4 (59.6); on Vision2Web, it improves the average visual and functional score from 41.1 to 62.4, on par with Claude-Opus-4.8. Ablation studies confirm the contributions of artifact-grounded credit assignment and long-horizon training.

1 INTRODUCTION

Recent advances in multimodal code generation have expanded the capabilities of frontend agents, from reproducing webpage designs (Si et al., 2025; Xu et al., 2025; 2026) to refining interfaces through rendered feedback (Yang et al., 2026; Deng et al., 2026) and generating interactive websites (Lu et al., 2026). In practice, these capabilities often need to work together within the same application. A user may ask an agent to reproduce a reference design, add a product section, connect it to an existing shopping cart, and preserve the navigation behavior. These requirements share code and application state: implementing a new feature can affect the layout, while a visual revision can disrupt an existing interaction. As illustrated in Figure 1, completing such a request requires reproduction, generation, modification, and repair to be coordinated throughout development. This raises a central question: *how can we train a general frontend agent to compose these capabilities over an evolving application state?*

Training such an agent requires coordinating activities whose local goals differ but whose effects are coupled through shared code and application state. Satisfying one requirement can violate another, making it necessary to assess the combined effects of these activities on the application. Progress may become observable only after several edits and executions, while later changes can introduce regressions, making it difficult to distinguish the contributions of different development stages. The interaction history can also outgrow the model's context window as the project evolves. Therefore,

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.

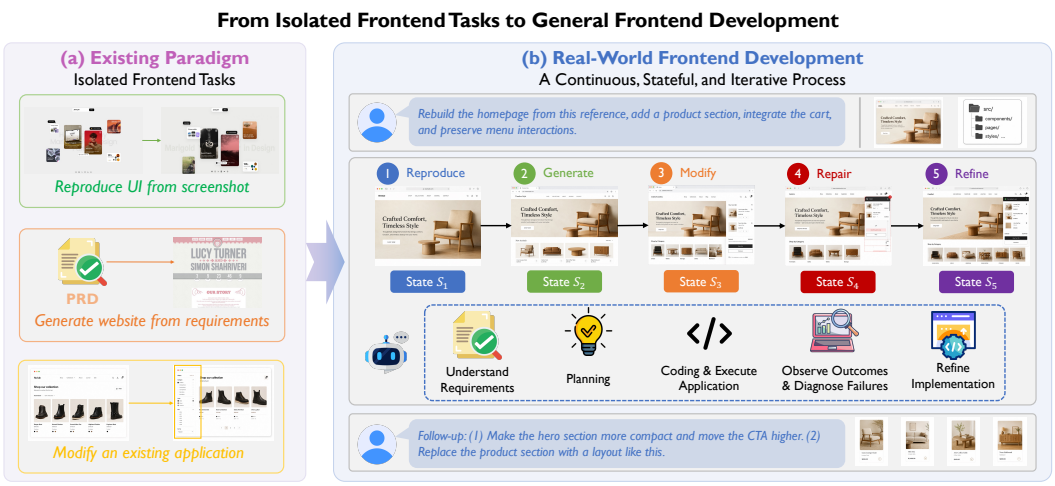

Figure 1: **From isolated frontend tasks to general frontend development.** (a) Existing work typically studies frontend reproduction, generation, and modification as separate task categories. (b) In real-world development, a single user request often requires multiple capabilities to be composed within a continuous, stateful, and iterative development process, followed by multi-turn and multimodal refinement over the evolving application state.

learning a general policy requires a common basis for evaluating heterogeneous activities, credit assignment across interdependent stages, and support for long-horizon training.

To address these challenges, we introduce **WebCoder**, a general frontend development agent trained with *artifact-grounded agentic reinforcement learning*. We unify heterogeneous frontend tasks within **Webdev-World**, an executable environment that provides a shared, task-agnostic interface for code editing, execution, browser inspection, and on-demand skills while preserving project state across interactions. This common interaction space allows a single policy to interleave reproduction, generation, modification, and repair as the application evolves. Our key idea is to ground policy learning in the evolution of an executable application, using task requirements to evaluate both the final outcome and intermediate development progress. Evaluating build results, rendered pages, and interaction outcomes against the same requirements provides a common basis for assessing the combined effects of development activities and optimizing the policy from its own trajectories.

Specifically, we construct a weighted checklist from the task requirements and use an independent verifier to build, run, and inspect the final application. Deterministic checks and visual assessments measure requirement satisfaction, with the aggregated score gated by successful execution and rendering to form the terminal reward. This terminal reward evaluates the final outcome but does not distinguish which development stages advanced the task or introduced regressions. Thus, we estimate intermediate requirement satisfaction from available project and execution evidence using the same checklist. Changes in these estimates provide stage-level progress and regression signals, which guide credit assignment across the trajectory and help the policy learn from the development process. For training across context windows, we combine context compaction with message-tree sample construction. Compaction lets development continue while project state persists in Webdev-World. A message tree records histories across context rewrites, enabling training on different stages of the same trajectory. Shared generated history remains available as context but contributes to the policy loss at most once.

To evaluate capability composition, we introduce **Webdev-Bench**, which combines development requirements within a single project, including requested changes and the preservation of specified existing behavior. On Webdev-Bench, WebCoder-27B reaches a requirement satisfaction score of 72.8 against 64.8 for its Qwen3.8-27B base model and 65.1 for Claude-Opus-4.8, and delivers a running application on all 40 tasks. On Vision2Web (He et al., 2026), it achieves an average visual and functional score of 62.4, compared with 41.1 for the base model and 62.5 for Claude-Opus-4.8.

Our main contributions are summarized as follows:

- We study **general frontend development** as capability composition over a persistent project state, and introduce **Webdev-World** as a shared environment for training **WebCoder**, a general frontend development agent.
- We develop **artifact-grounded agentic reinforcement learning**, using task requirements to connect final application verification with progress-based credit assignment, and supporting training over trajectories that span multiple context windows.
- We introduce **Webdev-Bench** to assess capability composition and preservation of specified existing behavior, and demonstrate improved visual and functional performance on Vision2Web.

2 METHOD

We present an artifact-grounded agentic reinforcement learning framework for training WebCoder to compose frontend development capabilities over a persistent project state (Figure 2). We formulate the task, introduce Webdev-World as the shared environment, describe how one requirement decomposition connects terminal rewards with progress-based credit assignment, and detail training on trajectories that span multiple context windows.

2.1 PROBLEM STATEMENT

We study general frontend development as a sequential process in which a single agent coordinates reproduction, generation, modification, and repair within an evolving project. Given a task input x, which may include reference screenshots, natural-language requirements, and an initial repository, the agent iteratively edits, executes, and inspects the application; success is determined by whether the final application jointly satisfies the task requirements, including constraints on existing behavior.

We model this process as a partially observable decision process. At time t, the policy π_θ generates an action a_t (text output and tool calls) conditioned on the context $c_t = \mathcal{C}(h_t)$, where $h_t = [p, x, a_0, o_0, \ldots, a_{t-1}, o_{t-1}]$ is the interaction history under system instructions p, o_t is the observation returned after executing a_t (command output, file contents, or a page screenshot), and $\mathcal{C}$ is a context management mechanism that keeps the information needed for the task within a finite window. A trajectory τ terminates at time T with a final project state $\mathcal{A}_T(\tau)$. A verifier R independently builds, runs, and inspects the resulting application and returns a scalar terminal reward $R(\mathcal{A}_T(\tau), x)$, using only evidence from the application and its execution rather than the agent's conversation or self-reported claims. We train π_θ to maximize the expected terminal reward:

$$\max_\theta \ \mathbb{E}_{x\sim\mathcal{D}}\mathbb{E}_{\tau\sim\pi_\theta(\cdot|x)}\left[R\big(\mathcal{A}_T(\tau), x\big)\right], \tag{1}$$

where $\mathcal{D}$ denotes the task distribution.

2.2 WEBDEV-WORLD

Webdev-World is an executable environment with a common interface for code editing, command execution, browser inspection, and on-demand skills. Each rollout runs in an isolated container initialized with a React/Vite scaffold or an existing project, and project files persist across interactions, so a single policy can interleave development activities over the same evolving application. The policy edits the repository through file and shell tools and invokes `webshot` to build and serve the application and obtain rendered screenshots and DOM statistics, which show how preceding changes affect the application; on-demand skills provide procedural guidance for page decomposition, asset preparation, code generation, validation, and repair.

2.3 ARTIFACT-GROUNDED AGENTIC REINFORCEMENT LEARNING

We ground policy learning in the application's satisfaction of task requirements, using final verification to compute terminal rewards and intermediate progress estimates to guide credit assignment.

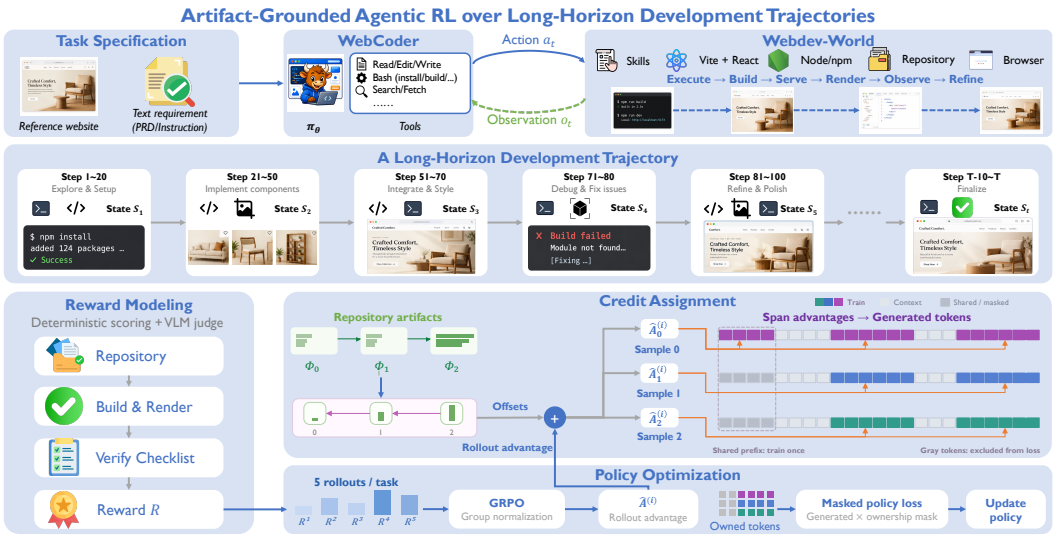

Figure 2: **Overview of our artifact-grounded agentic reinforcement learning framework.** WebCoder builds and refines a runnable project in Webdev-World through development tools and on-demand skills; an independent verifier scores the final application against the task checklist, artifact-based potentials redistribute the GRPO rollout advantage across trajectory segments, and an ownership mask assigns each generated token to one training sample.

Reward Modeling. Frontend tasks combine visual, content, functional, and interaction requirements that the final application must satisfy jointly. From the requirements, reference images, and project context in x, we construct a weighted checklist $\{(w_i, v_i)\}_{i=1}^{m}$, where w_i is the weight of requirement i and v_i is its verification function, chosen among deterministic checks for directly testable requirements, visual comparison against the reference for reproduction requirements, and rubric-guided judgments by a VLM (GLM-5V-Turbo (Hong et al., 2026)) for quality requirements that resist deterministic verification. An independent verifier R builds, runs, and inspects the final project state $\mathcal{A}_T$ under runtime conditions consistent with the development environment and aggregates the checklist as

$$Q(\mathcal{A}_T, x) = \sum_{i=1}^{m} w_i \, v_i(\mathcal{A}_T, x), \qquad v_i(\mathcal{A}_T, x) \in [0,1], \quad w_i \geq 0, \quad \sum_{i=1}^{m} w_i = 1. \tag{2}$$

To ensure that the delivered application can be executed and inspected, we gate this quality score by execution validity. Let $G(\mathcal{A}_T) \in \{0,1\}$ indicate whether the project passes all build, entry-point resolution, and independent browser-rendering checks. The terminal reward returned by the verifier is

$$R(\mathcal{A}_T, x) = G(\mathcal{A}_T)\, Q(\mathcal{A}_T, x). \tag{3}$$

Appendix C.2 lists the evidence, verification method, and weight of each requirement type.

Credit Assignment. The terminal reward measures task completion in the final project but does not directly distinguish the contributions of different stages in a long development trajectory. Thus, we estimate intermediate requirement satisfaction using the same checklist as final verification and use changes in these estimates to adjust stage-level advantages. The effect of a code change becomes observable only after the application is built, run, and inspected, so progress cannot be attributed to individual tool calls. We therefore assign credit at the level of segments, taking context compaction boundaries as natural segment boundaries, and measure progress by the change in project state between consecutive boundaries.

To align progress measurement with task evaluation, we retain the checklist decomposition and estimate requirement satisfaction at segment boundaries. Let $\phi_{k,i} \in [0,1]$ denote the estimated satisfaction of requirement i at boundary k. For tasks with a requirement checklist, $\phi_{k,i}$ is the fraction of the requirement's expected content present in the most recent page the agent fetched

from its own running application at or before boundary k; the estimate is updated only by such fetches, carries the previous value forward when no new fetch occurred, and records any decrease in full when a new fetch shows regression. For reproduction tasks without a checklist, a structural authoring score over the source files present at boundary k takes its place (Appendix C.2). We aggregate these estimates with the checklist weights into an artifact potential, whose difference across consecutive boundaries represents observable progress within a segment:

$$\Phi_k = \sum_{i=1}^{m} w_i \, \phi_{k,i}, \qquad \Delta\Phi_k = \Phi_k - \Phi_{k-1}. \tag{4}$$

We combine this process signal with the rollout-level advantage derived from terminal rewards. Let $\widehat{A}_{\text{rollout}}$ denote the GRPO advantage computed from terminal rewards for the same task. The segment adjustment b_k is the discounted return of potential differences,

$$b_k = \sum_{j \geq k} \gamma^{\,j-k} \, r_j, \qquad r_j = \gamma \, \Phi_{j+1} - \Phi_j \;\; (j < K-1), \qquad r_{K-1} = 0, \tag{5}$$

with $\gamma = 0.995$; the last segment carries no shaped reward because the terminal reward already enters through $\widehat{A}_{\text{rollout}}$. To redistribute credit across segments while preserving the mean advantage determined by terminal evaluation, we apply token-weighted centering and clipping:

$$\delta_k = \text{clip}\left(b_k - \frac{\sum_{j \in \mathcal{K}} n_j b_j}{\sum_{j \in \mathcal{K}} n_j}, \, \pm c \right), \qquad \widehat{A}_k = \widehat{A}_{\text{rollout}} + \delta_k, \tag{6}$$

where $\mathcal{K}$ is the set of segments selected for training, n_k is the number of generated tokens in segment k that contribute to the policy loss, and $c = 5$; the offsets are re-centered after clipping so that $\sum_{k \in \mathcal{K}} n_k \delta_k = 0$ holds exactly. Process signals thus redistribute credit across segments, while the terminal reward remains the trajectory-level evaluation baseline.

Development segments define the units of credit assignment, while message-tree paths define the training samples: each retained generated token receives the advantage of the segment in which it was generated, and an ownership mask assigns it to exactly one training sample, so tool responses and repeated history serve only as context and shared history is never optimized twice.

Policy Optimization. We optimize the policy with a masked, clipped objective that applies segment-level advantages to generated tokens in the constructed training samples. For each task x, we sample N trajectories from the behavior policy $\pi_{\theta_{\text{old}}}$ and obtain terminal rewards $r_j = R(\mathcal{A}_T(\tau^{(j)}), x)$. Following GRPO (Shao et al., 2024), we compute the rollout-level advantage as

$$\widehat{A}^{(i)}_{\text{rollout}} = \frac{r_i - \mu_x}{\sigma_x + \epsilon}, \qquad \mu_x = \frac{1}{N} \sum_{j=1}^{N} r_j, \tag{7}$$

where σ_x is the standard deviation of rewards within the group and ϵ is a numerical stability constant; credit assignment then yields segment-level advantages.

Let u index token positions in the training batch, y_u denote a generated token, and c_u denote its generation context. The probability ratio between the current and behavior policies is

$$\rho_u(\theta) = \frac{\pi_\theta(y_u \mid c_u)}{\pi_{\theta_{\text{old}}}(y_u \mid c_u)}. \tag{8}$$

Let $\widehat{A}_u$ be the advantage of the segment containing token u, and let $M_u \in \{0, 1\}$ be the joint generation and ownership mask. The policy loss is

$$\mathcal{L}_{\text{policy}}(\theta) = -\frac{1}{\sum_u M_u} \sum_u M_u \min\Big(\rho_u(\theta) \widehat{A}_u, \, \text{clip}(\rho_u(\theta), 1-\eta, 1+\eta) \, \widehat{A}_u \Big), \tag{9}$$

where η is the clipping threshold.

Training Beyond the Context Window. To support trajectories beyond the context window, we apply tool-output truncation, tool-result folding, and summary compaction. After compaction, the agent continues from the summarized history while project state persists in the environment.

For training, we record conversation histories in a message tree that preserves branches across context rewrites. Root-to-leaf paths form candidate training samples. Each rollout retains its initial and terminal candidates and a bounded number of additional samples selected by artifact progress, falling back to evenly spaced selection when measurements are unavailable. Ownership masks based on message position and content ensure that repeated generated tokens contribute to the loss at most once. Samples receive the segment advantages defined above, while each rollout contributes only one terminal reward to GRPO group statistics. A path exceeding the 65,536-token training budget keeps its most recent tokens so that the trainable segment is preserved, and the behavior-policy log-probabilities in ρ_u are recomputed on the assembled sequence (Appendix C.2). The algorithm is detailed in Appendix C.1. Rollout infrastructure, including speculative decoding, prefix caching, and log-probability monitoring between rollout and training engines, is described in Appendix C.3.

3 Experiments

3.1 Experimental Setup

WebCoder-27B is initialized from Qwen3.8-27B and trained with GRPO in Webdev-World, with 16 tasks per step, 8 rollouts per task, a learning rate of 2×10^{-6}, a 116,736-token rollout context, and a 65,536-token training context. Training proceeds in two stages: the first uses single-type tasks (text-to-project generation and single- and long-screenshot replication), and the second adds composite tasks that replicate a page from scratch and then extend and modify it. All hyperparameters are listed in Table 3.

We evaluate on two benchmarks. **Vision2Web** (L2-frontend, 66 titles) provides reference screenshots of a multi-page site and asks for a runnable frontend project; its official scorer reports a visual score (VS), a functional test pass rate (FS), and their average (Avg). **Webdev-Bench** is our benchmark of 40 composite tasks (188 requirements, 1,125 hidden checklist items) that start from a reference screenshot, a textual specification, or an existing repository; requirements are revealed and gated round by round, and an independent acceptance agent operates the deliverable and grades every item, giving the requirement satisfaction score (RS) and its visual, functional, and interaction components (VS, FS, IS). Both benchmarks also report the delivery success rate (DSR), and undelivered tasks count as 0; definitions and protocols are given in Appendix B.2.

We compare against six proprietary models (Claude-Opus-4.8 (Anthropic, 2026), GPT-5.4 (OpenAI, 2026), GPT-5 (OpenAI, 2025), Gemini-3-Pro (Google DeepMind, 2025b), Gemini-3-Flash (Google DeepMind, 2025a), Seed-1.8 (ByteDance Seed, 2026)) and eight open-source models (Kimi-K2.5 (Kimi Team et al., 2026), MiMo-V2.5 (Xiaomi, 2026), Step-3.7-Flash (StepFun, 2026), Qwen3.8-27B (Qwen Team, 2026b), Qwen3.5-397B-A17B and Qwen3.5-122B-A10B (Qwen Team, 2026a), Qwen3-VL-32B/8B-Instruct (Yang et al., 2025)); see Appendix A. All models are evaluated with the same agent framework, tools, prompts, container environment, time budget, and judges. Judge consistency is analyzed in Appendix E.

3.2 Main Results

Table 1 reports the main results. On Webdev-Bench, WebCoder-27B reaches an RS of 72.8, above the base model Qwen3.8-27B (64.8) and the strongest baseline Claude-Opus-4.8 (65.1); the remaining proprietary models lie between 48.0 and 59.6. All three components are also the highest: VS 81.9, FS 73.8, and IS 90.8, where the interaction component leads the runner-up (Gemini-3-Flash, 85.0) by 5.8 points, indicating that the gains come mainly from interactive behavior that works correctly after multi-round development. WebCoder delivers all 40 tasks, against 95% for the base model. Larger open-source models do not translate scale into results: Qwen3.5-397B-A17B and Qwen3.5-122B-A10B reach RS 28.7 and 24.6, and the two Qwen3-VL-Instruct models rarely complete a task within the agent framework. Appendix G splits the results by task starting point: the gains of WebCoder concentrate on text-to-project tasks (RS 70.0 versus 57.2 for the base model) and replication tasks (58.9 versus 51.7).

On Vision2Web, WebCoder-27B reaches an Avg of 62.4, on par with Claude-Opus-4.8 (62.5) and above GPT-5.4 (57.9) and all open-source models; its FS of 67.1 is the highest (tied with Qwen3.5-397B-A17B) and its VS of 57.7 is second to Claude-Opus-4.8's 59.1. Relative to the base model, Avg improves from 41.1 by 21.3 points, VS and FS improve by similar margins, and the delivery rate rises from 48.5% to 80.3%.

Table 1: **Main results on Vision2Web and Webdev-Bench.** Vision2Web (L2-frontend, 66 titles) reports the visual score (VS), functional score (FS), their average (Avg), and the delivery success rate (DSR); Webdev-Bench (40 tasks) reports requirement satisfaction (RS) together with its visual (VS), functional (FS), and interaction (IS) components, their average (Avg), and DSR. Undelivered tasks count as 0 in all score columns. Best in **bold** and second best underlined per column across all models.

Model	Vision2Web				Webdev-Bench					
	VS	FS	*Avg*	*DSR*	RS	VS	FS	IS	*Avg*	*DSR*
Proprietary Models										
Claude-Opus-4.8	**59.1**	65.9	**62.5**	98.5%	65.1	75.0	70.7	84.1	73.7	100.0%
GPT-5.4	51.3	64.4	57.9	97.0%	59.6	72.7	70.0	81.0	70.8	100.0%
GPT-5	23.3	64.7	44.0	98.5%	53.5	64.0	65.8	74.6	64.5	100.0%
Gemini-3-Pro	13.3	14.6	14.0	65.2%	48.0	65.8	58.5	68.5	60.2	100.0%
Gemini-3-Flash	13.1	27.9	20.5	78.8%	56.6	65.8	64.8	85.0	68.1	95.0%
Seed-1.8	7.0	13.3	10.2	36.4%	52.6	66.5	63.8	71.5	63.6	100.0%
Open-Source Models										
Kimi-K2.5	35.1	61.0	48.1	92.4%	46.8	61.5	55.0	67.7	57.7	82.5%
MiMo-V2.5	38.2	66.5	52.4	87.9%	14.9	25.2	21.2	22.6	21.0	65.0%
Step-3.7-Flash	22.1	47.3	34.7	95.5%	10.8	20.0	14.0	19.0	15.9	60.0%
Qwen3.8-27B	36.4	45.7	41.1	48.5%	64.8	77.9	68.8	79.8	72.8	95.0%
Qwen3.5-397B-A17B	35.6	**67.1**	51.4	100.0%	28.7	39.0	31.9	36.5	34.0	80.0%
Qwen3.5-122B-A10B	20.2	39.0	29.6	87.9%	24.6	30.4	28.9	33.1	29.3	72.5%
Qwen3-VL-32B-Instruct	0.0	0.0	0.0	0.0%	2.5	6.3	5.4	4.4	4.6	50.0%
Qwen3-VL-8B-Instruct	0.0	0.0	0.0	0.0%	0.0	0.8	0.0	0.0	0.2	30.0%
WebCoder-27B	57.7	**67.1**	62.4	80.3%	**72.8**	**81.9**	**73.8**	**90.8**	**79.8**	100.0%

3.3 Ablation Studies

We compare the full model with two ablated variants that share its initialization, training data, terminal reward, and hyperparameters, evaluated on Vision2Web L2 with the base model as reference. ***w/o Compaction (linear truncation)*** treats each rollout as a single linear token sequence and trains only on its first 65,536 tokens. ***w/o Artifact-Grounded Credit Assignment*** keeps the sample-construction procedure but gives all segments of a rollout the same rollout-level GRPO advantage.

Figure 3 shows that the full model outperforms both variants on all four metrics. Removing compaction reduces Avg from 62.4 to 54.9, a drop of 7.5 points, while removing artifact-grounded credit assignment reduces it to 52.6, a drop of 9.8 points. Both ablations also lower visual and functional scores: VS and FS decrease by 5.8 and 9.3 points, respectively, without compaction, and by 11.0 and 8.7 points without artifact-grounded credit assignment. Both variants remain above the base model's Avg of 41.1, but neither matches the full training recipe. DSR also decreases from 80.3% to 72.7% and 66.7% for the compaction and credit-assignment ablations, respectively.

3.4 In-depth Analysis

Figure 4 reports evaluation scores on a fixed held-out set of 28 anchor tasks and rewards from on-policy training rollouts; the fixed set gives a common reference across checkpoints, whereas training rewards depend on the tasks sampled at each step. The anchor score increases across all evaluated checkpoints, from 0.346 at step 4 to 0.575 at step 29 (a relative gain of 66.2%), with improvements within both training stages. Training rewards, in contrast, show an early decline followed by a recovery in Stage 1; behavioral statistics (Appendix F) associate the dip on replication

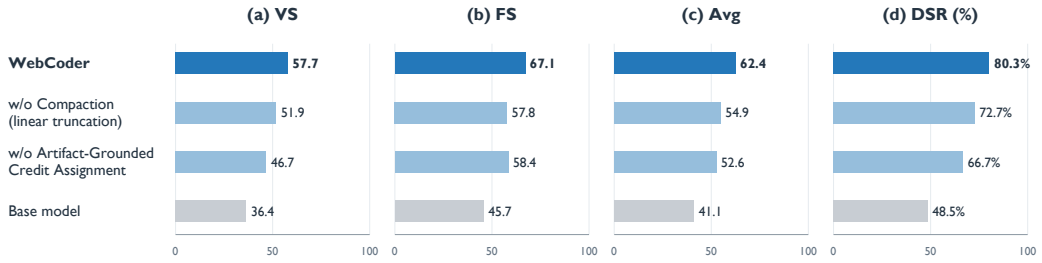

Figure 3: **Ablation studies of WebCoder on Vision2Web L2.** Full model, the two ablated variants, and the base model on (a) VS, (b) FS, (c) Avg, and (d) DSR. Higher is better.



Figure 4: **Training dynamics of WebCoder**. (a) Evaluation scores on the fixed anchor set. (b) Per-step mean training rewards and centered five-step moving averages, computed separately within each stage. (c) Early and late reward distributions: steps 1–4 versus 11–14 in Stage 1, and steps 18–21 versus 26–29 in Stage 2. Blue and orange shading mark reward thresholds above which the late or the early window, respectively, has the larger fraction of rollouts.

tasks with more redundant operations and less frequent skill invocation, both of which reverse during the recovery. Interaction volume alone therefore does not capture development progress, which motivates credit assignment based on changes in the artifact.

Figure 4(c) shows how reward distributions change within each stage. In Stage 1, the late window (steps 11–14) contains both more high-reward rollouts than the early window (steps 1–4; reward ≥ 0.8: 33.8% to 39.7%) and more zero-reward rollouts (2.6% to 6.3%). In Stage 2, the zero-reward fraction decreases from 6.5% (steps 18–21) to 4.9% (steps 26–29) while the mean reward increases from 0.604 to 0.633, so the later gains come with fewer failures rather than only more high-reward outcomes.

3.5 Case Studies

Figure 5 shows WebCoder on one task from each benchmark. On the Vision2Web title *caplugs*, a six-page e-commerce site, the agent reproduces the reference's hero banner, category grid, and promotional band and serves all six pages (VS 79.1, FS 87.5). On the Webdev-Bench task c22, a five-round modification of an existing website, the simulated-user gate rejects the first attempt at requirement three, the agent rearranges the section and passes on the second attempt, and terminal acceptance grades all 27 checklist items 3 (RS 1.0, against 0.6 for GPT-5.4). Appendix I adds baseline comparisons, full trajectories, and a task on which WebCoder falls short.

4 Related Work

Frontend Development. Automated frontend development spans reproducing interfaces from visual references, generating applications from textual requirements, and modifying or repairing existing projects. Datasets such as WebSight (Laurençon et al., 2024), Web2Code (Yun et al., 2024),



Figure 5: **WebCoder on Vision2Web (caplugs) and Webdev-Bench (c22).** Left: the reference homepage of caplugs and WebCoder's render. Right: c22 states after each round, each cropped to the region the requirement changes. R2 tightens the hero and removes the investor row; the first attempt at R3 keeps the five "Why SEDA" parts in the old staggered layout and is rejected by the gate, while the second attempt compacts them into two rows and passes; R4 adds a filterable newsroom below the hero.

and WebCode2M (Gui et al., 2025a) support screenshot-to-code learning, and benchmarks including Design2Code (Si et al., 2025), Sketch2Code (Li et al., 2025), Figma2Code (Gui et al., 2026), and Interaction2Code (Xiao et al., 2025a) evaluate generation from screenshots, sketches, design files, and interactive prototypes. WebGen-Bench (Lu et al., 2025) and Vision2Web (He et al., 2026) add browser-based functional testing, while DesignBench (Xiao et al., 2025b) and MT-Web2Code (Li et al., 2026a) cover editing and repair with preservation of unaffected content. On the modeling side, UICopilot (Gui et al., 2025b) and ScreenCoder (Jiang et al., 2025) generate code coarse-to-fine, JanusCoder (Sun et al., 2026) unifies generation and editing, UI2Code^N (Yang et al., 2026) and VisRefiner (Deng et al., 2026) refine from rendered feedback, and WebWorld (Xiao et al., 2026) selects browser-verified transitions for supervised training. Our work instead trains a single policy to compose all four activities within the same evolving application through artifact-grounded agentic reinforcement learning.

Agentic Reinforcement Learning. Agentic reinforcement learning optimizes policies through multi-step interactions with tools and environments (Wang et al., 2025; Luo et al., 2025); SWE-RL (Wei et al., 2025) learns from software evolution data with rule-based rewards, and GiGPO (Feng et al., 2025) combines episode-level advantages with step-level estimates from repeated environment states. In frontend development, WebGen-Agent (Lu et al., 2026) uses screenshot and GUI-agent scores as step-level rewards, ReLook (Li et al., 2026b) trains refinement with a multimodal critic, and WebGrader (Chen et al., 2026) derives executable interaction checks from requirements. For trajectories beyond the context window, ReSum (Wu et al., 2025) broadcasts advantages across summarized segments, Context-Folding (Sun et al., 2025) learns branching and folding, and CompactionRL (Li et al., 2026c) jointly optimizes execution and summarization. Our approach grounds terminal rewards and intermediate progress estimates in one checklist, redistributes the rollout advantage across segments by changes in these estimates, and trains across context rewrites with message-tree samples and ownership masks.

5 Conclusion

We introduce WebCoder, a general frontend development agent trained with artifact-grounded agentic RL in Webdev-World. A single policy coordinates reproduction, generation, modification, and repair over a persistent project state; one requirement decomposition grounds both terminal rewards and credit assignment across development stages; and compaction with message-tree samples supports training beyond a single context window. On Webdev-Bench, which we introduce to evaluate capability composition, and on Vision2Web, WebCoder improves substantially over its base model and matches or exceeds the strongest proprietary models, supporting artifact evolution under shared requirements as a basis for training general frontend agents.

USE OF LARGE LANGUAGE MODELS

We used LLM-based coding assistants during this project for code development, data-processing scripts, figure generation, and editing of the manuscript, in the sense described by the ICLR policy. LLMs are also components of the method and the evaluation, and these uses are documented in the paper rather than being incidental: a vision–language judge (GLM-5V-Turbo) forms one leg of the training reward (Section 2.3, Appendix C.2); an LLM acts as the simulated user and the acceptance judge in the Webdev-Bench protocol (GLM-5.3-Flash, Appendix B.2.3); LLM-based author, reviewer, and hardening stages assist the construction of Webdev-Bench tasks and hidden checklists, whose outputs were reviewed by the authors (Appendix B.2.2); and the official Vision2Web scorer uses LLM judges (Appendix B.1). The reliability of the judges is measured in Appendix E and their agreement with human annotators in Appendix H. We did not use LLMs to generate experimental results, fabricate citations, or produce claims without support. The authors reviewed all LLM-assisted text and code and take full responsibility for the content of the paper.

ETHICS STATEMENT

This work trains and evaluates agents that write frontend code. Training tasks are constructed from synthetic product specifications and from screenshots of publicly accessible web pages used as visual references; Webdev-Bench tasks are likewise seeded by public-page screenshots, synthetic specifications, or repositories prepared by the authors. No user data, account-restricted content, or non-public documents were collected, and reference pages were selected to avoid personal information. Agents run in isolated containers with no access to external services beyond package registries, and the verifiers inspect only the applications the agents build. Models that reproduce web pages from screenshots could be misused to clone protected designs; our benchmark uses reproduction only as the first step of composite development tasks, and the released materials are intended for research on frontend development agents.

REPRODUCIBILITY STATEMENT

The method is specified in Section 2: the environment, the terminal reward and its checklist decomposition, the credit-assignment computation, and the policy objective. Appendix C gives the implementation details needed to reproduce training: context management and message-tree sample construction with pseudocode (Algorithm 1), the evidence, verification method, and weight of each requirement type in the reward, the definition of the intermediate potentials and segment offsets, the rollout infrastructure, and all hyperparameters (Table 3). Appendix J reproduces the prompts and skill definitions supplied to the agent, the task-construction prompts, and the acceptance prompt. Appendix B.2 describes the benchmarks, the Webdev-Bench construction pipeline, the evaluation protocol with time budgets and retry rules, the judge models, and the exact scoring formulas; Appendix A lists every baseline and its serving configuration. Every number in the tables and figures is computed programmatically from archived per-task reports and judge outputs by scripts that will be released with the benchmark. Upon acceptance, we will open-source the WebCoder model, the Webdev-Bench benchmark, and the training and evaluation code. Judge stochasticity is controlled by median-of-three grading during training and quantified by repeated acceptance passes in Appendix E.

REFERENCES

Anthropic. Introducing Claude Opus 4.8. `https://www.anthropic.com/news/claude-opus-4-8`, 2026.

ByteDance Seed. Seed1.8 model card: Towards generalized real-world agency. *arXiv preprint arXiv:2603.20633*, 2026.

Boshui Chen, Huiping Liu, and Shaolei Zhang. WebGrader: Training LLMs for web development with self-evolving programmatic grader. *arXiv preprint arXiv:2608.06474*, 2026.

Jie Deng, Kaichun Yao, and Libo Zhang. VisRefiner: Learning from visual differences for screenshot-to-code generation. *arXiv preprint arXiv:2602.05998*, 2026.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for LLM agent training. *Advances in Neural Information Processing Systems*, 38:46375–46408, 2025.

Google DeepMind. Gemini 3 Flash: Frontier intelligence built for speed. https://blog.google/products-and-platforms/products/gemini/gemini-3-flash/, 2025a.

Google DeepMind. Gemini 3 Pro: The frontier of vision AI. https://blog.google/innovation-and-ai/technology/developers-tools/gemini-3-pro-vision/, 2025b.

Yi Gui, Zhen Li, Yao Wan, Yemin Shi, Hongyu Zhang, Bohua Chen, Yi Su, Dongping Chen, Siyuan Wu, Xing Zhou, et al. WebCode2M: A real-world dataset for code generation from webpage designs. In *Proceedings of the ACM on Web Conference 2025*, pp. 1834–1845, 2025a.

Yi Gui, Yao Wan, Zhen Li, Zhongyi Zhang, Dongping Chen, Hongyu Zhang, Yi Su, Bohua Chen, Xing Zhou, Wenbin Jiang, et al. UICopilot: Automating UI synthesis via hierarchical code generation from webpage designs. In *Proceedings of the ACM on Web Conference 2025*, pp. 1846–1855, 2025b.

Yi Gui, Jiawan Zhang, Yina Wang, Tianran Ma, Yao Wan, Shilin He, Dongping Chen, Zhou Zhao, Wenbin Jiang, Xuanhua Shi, et al. Figma2Code: Automating multimodal design to code in the wild. In *International Conference on Learning Representations*, 2026.

Zehai He, Wenyi Hong, Zhen Yang, Ziyang Pan, Mingdao Liu, Xiaotao Gu, and Jie Tang. Vision2Web: A hierarchical benchmark for visual website development with agent verification. *arXiv preprint arXiv:2603.26648*, 2026.

Wenyi Hong, Xiaotao Gu, Ziyang Pan, Zhen Yang, Yuting Wang, Yue Wang, Yuanchang Yue, Yu Wang, Yanling Wang, Yan Wang, et al. GLM-5V-Turbo: Toward a native foundation model for multimodal agents. *arXiv preprint arXiv:2604.26752*, 2026.

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

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Ziwei Chai, Y Charles, HS Che, Cheng Chen, et al. Kimi K2.5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

Hugo Laurençon, Léo Tronchon, and Victor Sanh. Unlocking the conversion of web screenshots into HTML code with the WebSight dataset. *arXiv preprint arXiv:2403.09029*, 2024.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023.

Qiming Li, Shujie Hu, Haohan Liu, Xiaocheng Feng, Songxiang Liu, and Guanglu Wan. MT-Web2Code: Benchmarking coding agents on multi-turn regional reconstruction and localized modification. *arXiv preprint arXiv:2608.03474*, 2026a.

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, 2025.

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 *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 25471–25485, 2026b.

Yujiang Li, Zhenyu Hou, Yi Jing, Jie Tang, and Yuxiao Dong. CompactionRL: Reinforcement learning with context compaction for long-horizon agents. *arXiv preprint arXiv:2607.05378*, 2026c.

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. *Advances in Neural Information Processing Systems*, 38, 2025.

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. In *International Conference on Learning Representations*, 2026.

Xufang Luo, Yuge Zhang, Zhiyuan He, Zilong Wang, Siyun Zhao, Dongsheng Li, Luna K Qiu, and Yuqing Yang. Agent Lightning: Train any AI agents with reinforcement learning. *arXiv preprint arXiv:2508.03680*, 2025.

OpenAI. Introducing GPT-5. https://openai.com/index/introducing-gpt-5/, 2025.

OpenAI. Introducing GPT-5.4. https://openai.com/index/introducing-gpt-5-4/, 2026.

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

Qwen Team. Qwen3.8. https://qwen.ai/blog?id=qwen3.8, 2026b.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

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, 2025.

StepFun. Step 3.7 Flash. https://static.stepfun.com/blog/step-3.7-flash/, 2026.

Qiushi Sun, Jingyang Gong, Yang Liu, Qiaosheng Chen, Lei Li, Kai Chen, Qipeng Guo, Ben Kao, and Fei Yuan. JanusCoder: Towards a foundational visual-programmatic interface for code intelligence. In *International Conference on Learning Representations*, 2026.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon LLM agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, et al. RAGEN: Understanding self-evolution in LLM agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. *Advances in Neural Information Processing Systems*, 38:78500–78525, 2025.

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*, 2025.

Jingyu Xiao, Yuxuan Wan, Yintong Huo, Zixin Wang, Xinyi Xu, Wenxuan Wang, Zhiyao Xu, Yuhang Wang, and Michael R Lyu. Interaction2Code: Benchmarking MLLM-based interactive webpage code generation from interactive prototyping. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 241–253. IEEE, 2025a.

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

Zikai Xiao, Jianhong Tu, Chuhang Zou, Yuxin Zuo, Zhi Li, Peng Wang, Bowen Yu, Fei Huang, Junyang Lin, and Zuozhu Liu. WebWorld: A large-scale world model for web agent training. *arXiv preprint arXiv:2602.14721*, 2026.

Xiaomi. MiMo-V2.5. `https://mimo.xiaomi.com/mimo-v2-5/`, 2026.

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

Mingde Xu, Zhen Yang, Yan Wang, Yu Wang, Xijun Liu, Zijun Dou, Wenyi Hong, Xiaotao Gu, Bin Xu, and Jie Tang. Video2Code: Generating interactive webpages from UI videos via action-aware revisit. *arXiv preprint arXiv:2606.20711*, 2026.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Zhen Yang, Wenyi Hong, Mingde Xu, Xinyue Fan, Weihan Wang, Jiale Cheng, Xiaotao Gu, and Jie Tang. UI2Code^N: UI-to-code generation as interactive visual optimization. In *Forty-third International Conference on Machine Learning*, 2026.

Feng Yao, Liyuan Liu, Dinghuai Zhang, Chengyu Dong, Jingbo Shang, and Jianfeng Gao. On the rollout-training mismatch in modern RL systems. In *OPT 2025: Optimization for Machine Learning*, 2025.

Sukmin Yun, Haokun Lin, Rusiru Thushara, Mohammad Q Bhat, Yongxin Wang, Zutao Jiang, Mingkai Deng, Jinhong Wang, Tianhua Tao, Junbo Li, et al. Web2Code: A large-scale webpage-to-code dataset and evaluation framework for multimodal LLMs. *Advances in Neural Information Processing Systems*, 37:112134–112157, 2024.

Z.ai. GLM-4.6V: Open source multimodal models with native tool use. `https://z.ai/blog/glm-4.6v`, 2026a.

Z.ai. GLM-5.3-Flash: Frontier intelligence, flash cost. `https://z.ai/blog/glm-5.3-flash`, 2026b.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. SGLang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems*, 37:62557–62583, 2024.

APPENDIX CONTENTS

A Baselines **15**

B Benchmarks **15**

B.1 Vision2Web 15

B.2 Webdev-Bench 15

C Long-Horizon Training Implementation **19**

C.1 Context Management and Sample Construction 19

C.2 Reward Modeling and Credit Assignment Details 21

C.3 Rollout Infrastructure 23

C.4 Training Configuration 23

D Trajectory Scale and Rollout Efficiency **23**

E Verifier Reliability **24**

E.1 Training Reward Judge: Self-Consistency and Signal-to-Noise 25

E.2 Webdev-Bench Acceptance Judge: Repeated-Grading Consistency 26

F Analysis of Agent Behavior **27**

G Full Results **29**

H Human Evaluation **31**

I Additional Case Studies **31**

I.1 Comparison with Baselines on a Shared Task 31

I.2 Complete Trajectories of WebCoder 32

I.3 A Trajectory That Does Not Pass Acceptance 33

J Prompts and Skills **36**

J.1 Prompts 36

J.2 On-Demand Skills 50

A BASELINES

We compare WebCoder-27B with fourteen models, all evaluated under the same agent framework, tools, prompts, container environment, time budget, and judges as WebCoder.

Proprietary models. **Claude-Opus-4.8** (Anthropic, 2026) is Anthropic's flagship model. **GPT-5.4** (OpenAI, 2026) and **GPT-5** (OpenAI, 2025) are OpenAI's two most recent flagship models. **Gemini-3-Pro** (Google DeepMind, 2025b) and **Gemini-3-Flash** (Google DeepMind, 2025a) are Google's flagship and efficiency-oriented models of the Gemini 3 family. **Seed-1.8** (ByteDance Seed, 2026) is ByteDance's multimodal model designed for real-world agentic tasks.

Open-source models. **Kimi-K2.5** (Kimi Team et al., 2026) is Moonshot AI's visual agentic model. **MiMo-V2.5** (Xiaomi, 2026) is Xiaomi's multimodal model. **Step-3.7-Flash** (StepFun, 2026) is StepFun's efficiency-oriented model. **Qwen3.8-27B** (Qwen Team, 2026b) is the dense model from which WebCoder-27B is initialized and serves as the base-model reference. **Qwen3.5-397B-A17B** and **Qwen3.5-122B-A10B** (Qwen Team, 2026a) are mixture-of-experts models of the Qwen3.5 family with 17B and 10B activated parameters. **Qwen3-VL-32B-Instruct** and **Qwen3-VL-8B-Instruct** (Yang et al., 2025) are vision–language instruction models of the Qwen3 family.

B BENCHMARKS

B.1 VISION2WEB

Vision2Web (He et al., 2026) is a benchmark for evaluating multimodal coding agents on website development, covering static webpages, interactive frontends, and full-stack websites. We evaluate on the Interactive Frontend level (L2), which contains 66 tasks. Each task provides multiple UI prototypes, textual requirements, and supporting assets, and requires an interactive frontend with visual consistency and coherent navigation across pages. This level assesses both visual reproduction and functional implementation, aligning with our focus on frontend development.

We allow two hours of development per task and independently deploy and evaluate the resulting projects. During deployment, the project's startup script is executed in a fresh container, and the application is checked for accessibility at the designated address. Once deployment succeeds, evaluators interact with the application according to predefined workflows, checking functional outcomes and collecting page screenshots. Functional evaluation uses Claude Sonnet 4.6 with `playwright-cli` to execute interactions and record test-case outcomes. Visual evaluation uses GLM-4.6V (Z.ai, 2026a) to compare the captured screenshots against the reference prototypes. The same evaluation configuration is used for all models.

We report Visual Score (VS), Functional Score (FS), their average (Avg), and Deployment Success Rate (DSR). VS aggregates visual scores across reference prototypes, while FS measures the proportion of functional test cases passed; both are reported on a 0–100 scale. Avg is the arithmetic mean of VS and FS, and DSR is the proportion of the 66 tasks whose applications are successfully deployed. All tasks are included in the reported results, with failed deployments contributing zero scores for their associated prototypes and functional test cases. These metrics jointly characterize deployability, visual fidelity, and functional correctness.

B.2 WEBDEV-BENCH

Webdev-Bench evaluates an agent's ability to develop a real front-end project continuously. The benchmark comprises 40 tasks spanning three starting conditions: replicating a page from a reference screenshot, building an application from a textual product description, and continuing development on an existing repository, including defect repair. Every task is a composite task: four to five natural-language development requirements are revealed one round at a time, and the agent completes them sequentially within a single session on a single project. Each requirement exercises one concrete development operation (replication, data wiring, interaction, layout restructuring, re-theming, and so on), and together they form one sustained, multi-requirement delivery. After all requirements have been revealed and development ends, the final application undergoes a single terminal acceptance pass that re-examines not only the last requirement but the behavior promised by

every earlier one. The central question the benchmark asks is therefore explicit: *can the agent satisfy each newly revealed requirement while preserving the behavior the project has already established?* Figure 6 summarizes the construction and evaluation pipeline; the following subsections cover task composition (§B.2.1), construction (§B.2.2), the evaluation protocol and metrics (§B.2.3), and two representative tasks (§B.2.4).

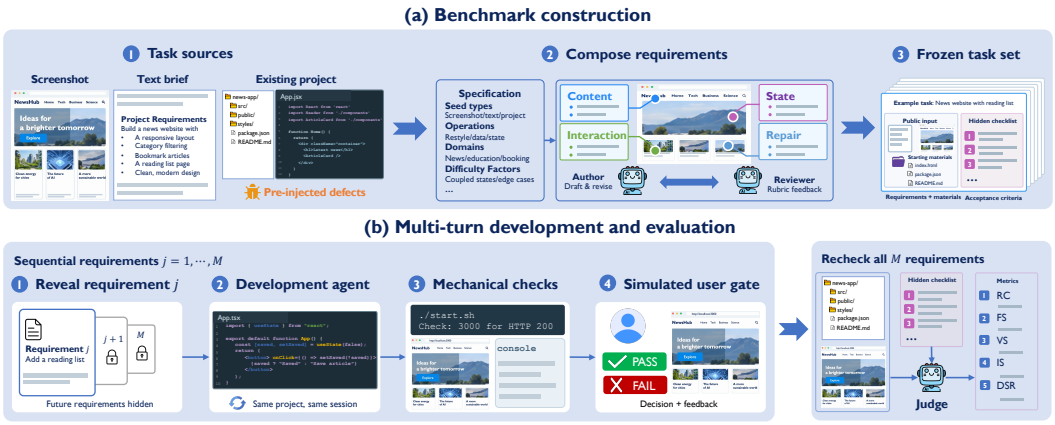

Figure 6: **Construction (top) and multi-round evaluation (bottom) of Webdev-Bench.** During construction, a specification drives a two-agent authoring loop that produces each task together with its hidden checklist. During evaluation, requirements are revealed round by round under a simulated-user gate, and an independent acceptance agent scores the complete application at the end.

B.2.1 Task Composition and Checklist Statistics

All 40 tasks are composite and fall into three starting conditions (Figure 7a). Nine reference-screenshot tasks ask the agent to rebuild a page from a full-page screenshot of a real website and then keep developing on top of it; twenty text-requirement tasks build an application from scratch against a product description; eleven existing-project tasks start from a running repository, and nine of these contain a pre-injected, reproducible defect whose diagnosis and repair constitute the first requirement. The 40 tasks contain 188 requirements in total (five in each of 28 tasks, four in each of the remaining 12). The requirement vocabulary covers eleven operation types, ranging from visually oriented replication and re-theming, through functionally oriented data wiring, stateful behavior, and interaction, to structurally oriented layout restructuring and section addition or removal; task subjects span ten domains including e-commerce, booking services, editorial content, and data dashboards.

During construction, each requirement is further decomposed into checklist items: observable, operationally verifiable descriptions of application behavior, each annotated with its parent requirement, an evaluation dimension (functional, interaction, visual, or coverage), a weight (1.5 for core items, 1.0 otherwise), and an acceptance criterion. The items are hidden from the agent under evaluation throughout; we refer to them collectively as the hidden checklist. Requirement counts and item counts measure scale at two different levels: one requirement typically constrains page content, functional results, and interaction states simultaneously, unfolding into roughly six items on average. Each task carries 22–33 items, and the full benchmark carries 1,125 (Figure 7b): 408 functional, 309 visual, 299 interaction, and 108 coverage items (plus one data-consistency item). Coverage items explicitly test whether behavior delivered for earlier requirements still holds after later modifications, realizing the "preserve established behavior" objective at the acceptance level. In sum, the task level characterizes sustained development through progressively revealed requirements, while the acceptance level concretizes those requirements into individually checkable application behaviors.

B.2.2 Benchmark Construction

Tasks and their hidden checklists are produced jointly by one pipeline, so that requirement text and acceptance criteria correspond item by item. Construction starts from a specification that fixes a

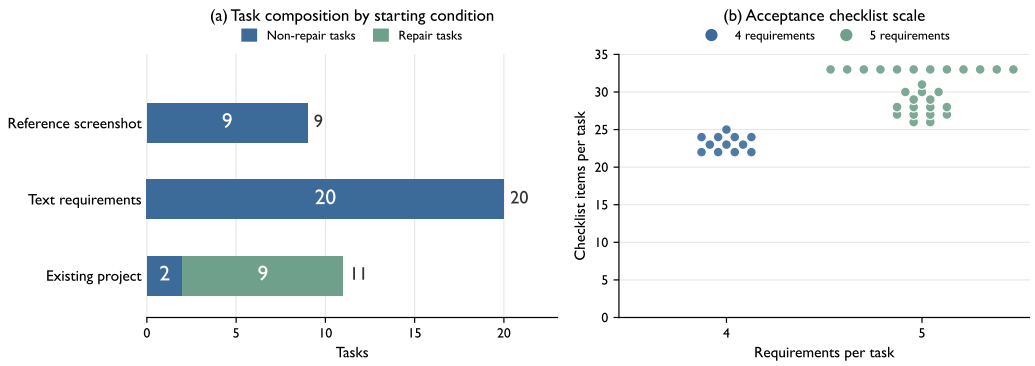

Figure 7: **Task composition and acceptance scale of Webdev-Bench.** (a) Tasks per starting condition, 40 composite tasks in total; the nine repair tasks (green segment) are contained within the existing-project category. (b) Each point is one task, with the number of requirements on the horizontal axis and the number of checklist items on the vertical axis; small horizontal jitter reduces overlap. The benchmark contains 1,125 checklist items in total.

task's starting condition, operation sequence, and difficulty tier (15 regular, 25 challenge), controlling the composition of the exam at the source. Seed material is then drawn from a curated pool according to the specification: replication tasks receive a full-page screenshot of a real website, and existing-project tasks receive a runnable repository; text-requirement tasks need no external seed. The specification and seed are given to an **author agent**, which drafts the round-by-round natural-language requirements together with the matching hidden checklist; a **reviewer agent** reads the draft independently and returns revision suggestions concerning feasibility, ambiguity, and acceptance blind spots, which the author incorporates. The purpose of the two-agent loop is to couple "how a user would phrase the next request" with "what acceptance should later check."

B.2.3 Evaluation Protocol and Metrics

The protocol deliberately separates two roles: driving development forward and producing the final score. Development proceeds in rounds. When requirement j is revealed, the agent continues working in the same session on the same project, with all later requirements still unknown. At the end of a round, the harness first performs a mechanical health check—launching the application with the task's own script, probing the service port, and collecting a full-page screenshot, the page text, and console output. A simulated user (GLM-5.3-Flash (Z.ai, 2026b), an LLM independent of the agent) then judges, from this evidence alone, whether the checklist items of the current requirement appear satisfied and issues a pass-or-return verdict with a short critique in the user's voice that must not quote the items: a pass reveals the next requirement, while a return sends the agent back to rework the current one, with at most three attempts per requirement. All rounds share a time budget proportional to the number of requirements (1,200 seconds per requirement), with gating and feedback overhead charged against it. This gate exists to push development forward in a manner close to a real user; it never sees the hidden checklist and produces no score.

All scores come from terminal acceptance. After development ends, an acceptance agent (also GLM-5.3-Flash) independent of the simulated user operates the delivered application in a browser and grades every item of every requirement against the hidden checklist on a 0–3 scale; requirements already passed by the gate during development are re-examined all the same. Aggregation follows a per-requirement conjunctive rule: a requirement counts as satisfied only if all items under it reach the top grade, and the per-task satisfaction fraction, macro-averaged over the 40 tasks, is the requirement satisfaction score (RS). Restricting the same grade records to the items of each dimension yields the functional (FS), interaction (IS), and visual (VS) scores; the delivery success rate (DSR) is the fraction of tasks that deliver a runnable application within budget, with all items of undelivered tasks graded 0. The two halves of the protocol thus map exactly onto the objective stated in §B.2: round-by-round feedback drives new requirements toward completion, while the terminal full re-examination checks whether later modifications broke earlier results.

Formally, let R_t denote the requirement set of task t and I_r the checklist-item set of requirement r; each item i carries a dimension $d_i \in \{\text{functional}, \text{interaction}, \text{visual}, \text{coverage}\}$ and a weight $w_i \in \{1.0, 1.5\}$, and terminal acceptance assigns it a grade $g_i \in \{0, 1, 2, 3\}$, with 3 the top grade. Under the per-requirement conjunctive rule, a requirement is satisfied if and only if every item under it reaches the top grade:

$$\text{sat}(r) \;=\; \mathbf{1}[\, g_i = 3,\;\; \forall\, i \in I_r \,], \tag{10}$$

and RS macro-averages requirement satisfaction within each task and then across the 40 tasks:

$$\text{RS} \;=\; \frac{1}{40} \sum_{t} \frac{1}{|R_t|} \sum_{r \in R_t} \text{sat}(r). \tag{11}$$

Dimension scores restrict each requirement's item set to a dimension subset D and apply the same conjunctive rule: with $I_r^D = \{\, i \in I_r : d_i \in D \,\}$ and $\text{sat}_D(r) = \mathbf{1}[\, g_i = 3,\; \forall\, i \in I_r^D \,]$, counting only requirements with $I_r^D \neq \varnothing$,

$$\text{Score}_D \;=\; \frac{1}{40} \sum_{t} \frac{1}{|\{r \in R_t : I_r^D \neq \varnothing\}|} \sum_{\substack{r \in R_t \\ I_r^D \neq \varnothing}} \text{sat}_D(r), \tag{12}$$

where FS takes $D = \{\text{functional}, \text{coverage}\}$—coverage items test whether behavior delivered for earlier requirements still holds after later changes and therefore count toward functional correctness—while IS takes $D = \{\text{interaction}\}$ and VS takes $D = \{\text{visual}\}$. DSR is the number of delivered tasks divided by 40, where a task counts as delivered if its own start script brings the application up on the designated port and passes the launch check; all items of undelivered tasks enter with $g_i = 0$, so their task-level RS and dimension scores are all 0. As a reference scale, the weighted four-grade score of requirement r on subset D is $\sum_{i \in I_r^D} w_i \,(g_i/3) \big/ \sum_{i \in I_r^D} w_i$, aggregated exactly as in Eq. equation 12.

B.2.4 Representative Tasks

Two tasks from the official set serve as case studies of task design: each is presented through its own initial material, the round-by-round requirement texts (abridged), and an excerpt of its hidden checklist, showing how requirements depend on one another and how the checklist concretizes natural-language requests into state-level checks. Requirement and checklist texts in the figures are abridged and translated from the Chinese originals.

A screenshot-seeded composite task (c38). The only initial material is a full-page screenshot of a consulting-firm website, and four requirements are revealed in order (Figure 8a–b): R1 replicates the landing page from the screenshot; R2 converts the static services section into a bookable expert-session schedule with city and topic filters, a seats-only toggle, sorting, and seat counts that decrement on booking; R3 removes the first-screen privacy-consent overlay and the featured-projects section; R4 re-themes the whole site from deep blue to dark green with warm sand-gold accents on a cream background. The requirements depend on one another: the R2 schedule is built on the layout replicated in R1, and the R4 re-theme must also cover the interactive section newly added by R2. The matching hidden checklist goes beyond "the section exists" (Figure 8c): a functional item under R2 requires that "with no filters the schedule lists 45 sessions, 8 per page over 6 pages, dropping to 39 sessions over 5 pages when the seats-only toggle is on," and an interaction item requires that "booking three named sessions in sequence yields a subtotal of £740 with the correct discount applied"—both verifying post-interaction application state rather than the mere presence of controls. A coverage item under R3 requires that no title, body text, or button of the overlay survives anywhere on the page, turning "remove" into a criterion that can be checked verbatim.

An existing-project task with defect repair (c22). The task starts from a runnable website of a web3 data protocol (Figure 9a, the starting repository's home page) whose repository carries a pre-injected *selector drift* defect: the anchor identifier of the white card section has been altered, so the PRIMER link in the navigation bar, the PRIMER button in the hero, and "Why SEDA" in the Learn menu no longer respond to clicks, while the neighbouring entries still scroll smoothly. Five requirements follow: R1 repairs the three entries and confirms that the others are unaffected; R2 removes the hero's investor row and the funding stat and tightens the hero to 780–900 pixels;

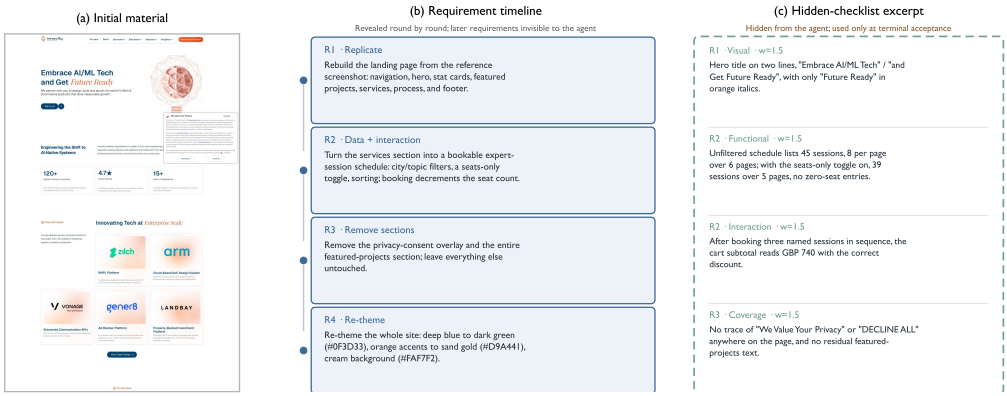

Figure 8: **Anatomy of representative task c38, a screenshot-seeded composite task.** (a) Initial material: the task's own full-page reference screenshot of a real website. (b) The four requirements revealed round by round (abridged; later requirements are invisible to the agent when requirement j is revealed). (c) Hidden-checklist excerpt with parent requirement, dimension, and weight tags; hidden from the agent throughout and used only at terminal acceptance. Texts are translated from the Chinese originals.

R3 re-lays the five "Why SEDA" blocks in two rows while keeping numerals, original texts, and images; R4 adds a 48-article newsroom before the ecosystems section, with categories, sorting, paging, and a read-later toggle; R5 wires a Newsroom entry into the navigation and turns the third stat cell into the library size (Figure 9b). The requirements are layered: the R5 entry reuses the scrolling mechanism repaired in R1, its number must agree with the actual article count from R4, and the content removed in R2 must not resurface in R5. The hidden checklist turns these into state-level checks (Figure 9c): a functional item under R1 requires that "after clicking PRIMER the card section is in view with its top within 120 pixels of the viewport top," accompanied by an explicitly regression-tagged item requiring that the other four entries keep working; a coverage item under R2 requires that no institution logo is rendered in the former investor area and that no investor name appears anywhere in the page text; and a coverage item under R5 searches again for "+$22M / Raised" and investor logos, verifying that the removal still holds after three further rounds of modification rather than merely that it happened.

C Long-Horizon Training Implementation

C.1 Context Management and Sample Construction

This section describes how we support development trajectories beyond the context window and construct training samples from them. During rollout, context management allows the agent to continue development within a bounded window. For training, we select samples from the recorded conversation histories, handle their length limits, and determine which generated tokens contribute to the loss in each sample. We describe these steps below and summarize the procedure in Algorithm 1.

As tool calls accumulate, the conversation history grows. We control its length by applying individual tool-output truncation, historical tool-result folding, and summary compaction in sequence. After compaction, the agent continues from the shortened history while the project files remain in the environment. Figure 10a illustrates this process in one rollout: prompt length increases as interactions accumulate and decreases when the context is reduced.

We record the conversation in a message tree to preserve histories before and after context rewrites. Shared messages form a common prefix, while rewritten contexts form separate branches. As illustrated in Figure 11, the history before compaction remains available, and subsequent responses extend the branch containing the compacted context. Each root-to-leaf path provides a candidate training sequence containing the responses along that path and their context.

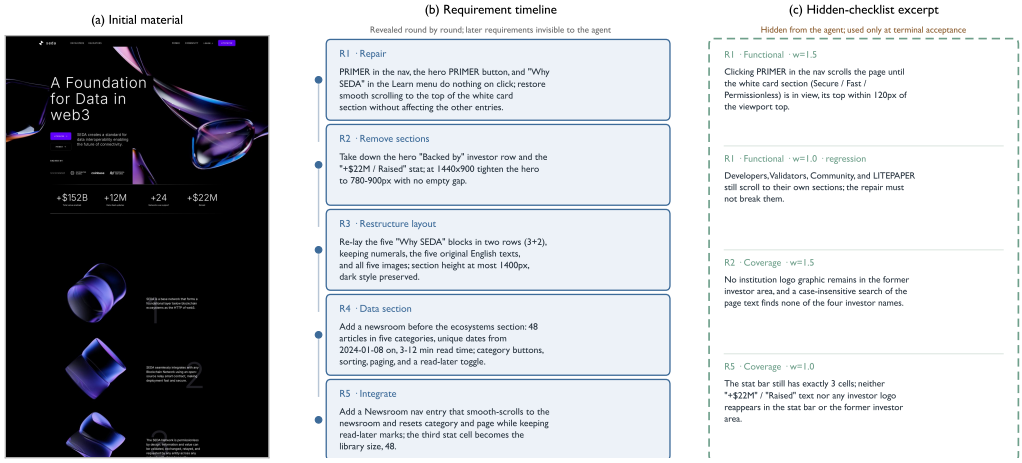

Figure 9: **Anatomy of representative task c22, an existing-project task with defect repair.** (a) Initial material: the above-the-fold view of the task's own starting repository (the injected selector-drift defect disables the three PRIMER entries and is invisible in a static screenshot). (b) The five requirements revealed round by round (abridged). (c) Hidden-checklist excerpt, including the repair item, an explicitly regression-tagged item, and coverage items that check that the removed content does not resurface after later modifications. Texts are translated from the Chinese originals.

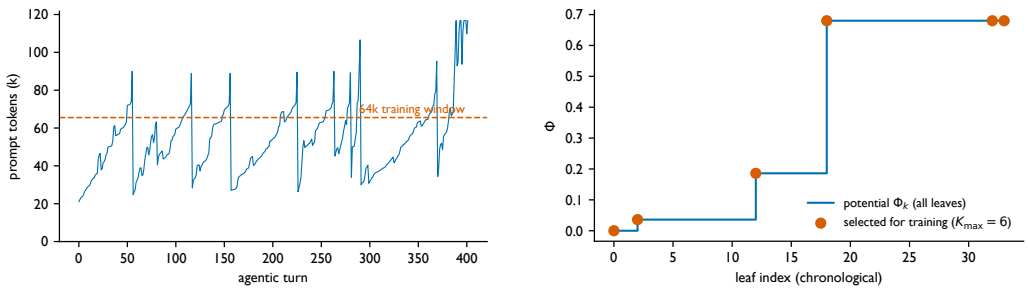

Figure 10: **Context management and training sample selection.** (a) Prompt length across interaction turns in an example rollout. Context reduction allows development to continue as the history grows; the dashed line marks the 64k training window. (b) Artifact potential Φ over chronologically ordered candidate paths. Orange markers indicate the six candidates selected for training from 34 paths, retaining the initial and terminal candidates along with selected intermediate candidates.

To limit training cost, we retain at most $K_{\max}$ candidate paths from each rollout. We always retain the initial and terminal candidates, then select the remaining candidates according to measured increases in artifact potential. Figure 10b shows an example in which six of 34 candidate paths are retained. The curve shows the recorded artifact potential, and the orange markers indicate the selected candidates.

Only eligible model-generated tokens contribute to the training loss; tool responses and other context tokens are excluded. Since different selected paths may contain the same generated history, we identify repeated generated

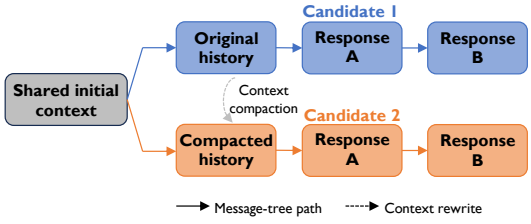

Figure 11: **Message-tree representation**. Blue and orange paths show candidates before and after compaction. The dashed arrow denotes the context rewrite.

Algorithm 1 Message-Tree Sample Construction for Long-Horizon Training

Require: A group of N rollouts for task x, $\{(\mathcal{T}^{(i)}, R^{(i)}, \mathcal{P}^{(i)})\}_{i=1}^{N}$; sample cap $K_{\max} \geq 2$; length limit $L_{\max}$
Ensure: Training batch $\mathcal{B}$

$\mathcal{B} \leftarrow \varnothing$
$\mu_x \leftarrow \mathrm{Mean}(\{R^{(i)}\}_{i=1}^{N})$
$\sigma_x \leftarrow \mathrm{Std}(\{R^{(i)}\}_{i=1}^{N})$
for $i = 1, \ldots, N$ **do**
  $\widehat{A}^{(i)} \leftarrow (R^{(i)} - \mu_x)/(\sigma_x + \epsilon)$
  $\mathcal{L} \leftarrow \textsc{ChronologicalLeafPaths}(\mathcal{T}^{(i)})$
  $K \leftarrow \min(K_{\max}, |\mathcal{L}|)$
  $\mathcal{S} \leftarrow \{\mathrm{First}(\mathcal{L}), \mathrm{Last}(\mathcal{L})\}$
  if $\textsc{SelectionScoresAvailable}(\mathcal{P}^{(i)})$ **then**
    $\mathcal{S} \leftarrow \mathcal{S} \cup \textsc{TopProgress}(\mathcal{L} \setminus \mathcal{S}, K - |\mathcal{S}|, \mathcal{P}^{(i)})$
  else
    $\mathcal{S} \leftarrow \mathcal{S} \cup \textsc{EvenlySpaced}(\mathcal{L} \setminus \mathcal{S}, K - |\mathcal{S}|)$
  end if
  for each path $k \in \mathcal{S}$ **do**
    $(z_k, e_k) \leftarrow \textsc{Assemble}(k, L_{\max})$ ▷ Tokens and eligibility mask
  end for
  $\mathcal{S} \leftarrow \{k \in \mathcal{S} : \sum_u e_{k,u} > 0\}$
  $O \leftarrow \textsc{AssignOwnership}(\mathcal{S}, \{z_k, e_k\}_{k \in \mathcal{S}})$
  for each $k \in \mathcal{S}$ **do**
    $M_{k,u} \leftarrow e_{k,u} \mathbb{1}[O(q_{k,u}) = k]$
    $n_k \leftarrow \sum_u M_{k,u}$
  end for
  $\mathcal{S} \leftarrow \{k \in \mathcal{S} : n_k > 0\}$
  if $\mathcal{S} = \varnothing$ **then**
    continue
  end if
  $\{\delta_k\}_{k \in \mathcal{S}} \leftarrow \textsc{CreditOffsets}(\mathcal{P}^{(i)}, \mathcal{S}, \{n_k\}_{k \in \mathcal{S}})$ ▷ $\sum_k n_k \delta_k = 0$; Appendix C.2
  for each $k \in \mathcal{S}$ **do**
    $\widehat{A}_k \leftarrow \widehat{A}^{(i)} + \delta_k$
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{(i, z_k, M_k, \widehat{A}_k)\}$
  end for
end for
return $\mathcal{B}$

tokens using message position and content and assign each token to one sample for loss computation. Its occurrences in other samples serve only as context. The final loss mask is $M_{k,u} = e_{k,u}\,\mathbf{1}[O(q_{k,u}) = k]$, where $e_{k,u}$ indicates whether token u in sample k is eligible for training, $q_{k,u}$ identifies the original generated token, and O specifies its owning sample.

C.2 Reward Modeling and Credit Assignment Details

This section specifies the verifiers used during training and the credit-assignment computation, complementing Section 2.3 and Algorithm 1.

Terminal reward. The verifier builds the project, resolves its entry point, and renders it in a browser; the product of these three gates is $G \in \{0, 1\}$. The quality term combines a deterministic leg and a vision–language judge leg (GLM-5V-Turbo (Hong et al., 2026)), $R = G \cdot (0.6\, f_{\mathrm{det}} + 0.4\, f_{\mathrm{vlm}})$, with a penalty of 0.1 when the wall-clock budget is exhausted. Table 2 lists, for each requirement type, the evidence the verifier collects, how it is verified, and its weight. The judge grades a rendered screenshot on a five-level anchored scale per dimension (mapped to $1.0/0.75/0.5/0.25/0$), and its score is the median of three draws.

Table 2: **Evidence, verification method, and weight of each requirement type in the training reward.** Weights of judge dimensions are given as the judge-leg weight 0.4 times the within-leg dimension weight.

Requirement type	Evidence	Verification	Weight
Reproduction tasks (reference screenshot)			
Pixel-level fidelity	rendered page vs. reference	deterministic image comparison (f_{det})	0.6
Visual fidelity / layout	screenshot	judge, anchored scale	0.4×0.30 / 0.22
Content coverage / text match	screenshot	judge, anchored scale	0.4×0.24 / 0.10
Render health	screenshot	judge, anchored scale	0.4×0.14
Generation tasks (textual requirements)			
Required sections and counts	DOM census of the served page	keyword presence and graded counts per required section (f_{det})	0.6
Design quality / content quality	screenshot	judge, anchored scale	0.4×0.38 / 0.26
Functionality (visual cues)	screenshot	judge, anchored scale; interactions are not executed	0.4×0.18
Render health	screenshot	judge, anchored scale	0.4×0.18
Build, entry point, render	build log, served port, browser	hard gates, multiplicative	G

Intermediate potentials. Two instantiations of $\phi_{k,i}$ are used. For tasks with a requirement checklist, $\phi_{k,i}$ is a ledger value: whenever the agent fetches a page from its own running application, the fetched text is scored against requirement i's expected content with the same matching function as the terminal f_{det}, and the ledger takes the new value; $\phi_{k,i}$ is the ledger state at the last fetch at or before boundary k. Missing measurements carry the previous value forward, and a lower measurement is recorded in full. File reads and writes are not observations, and the checklist is hidden from the agent. Count-based checks require the terminal DOM census and are excluded from the ledger. For reproduction tasks, which have no checklist, a source-authoring score over the files present at boundary k is used: whether the application root is no longer the scaffold placeholder (0.30), authored source lines (0.20), component files (0.20), local modules actually imported (0.15), and styling (0.15), each saturating; a file is attributed to the earliest segment whose messages name it.

Segment offsets. For a rollout with segments $k = 0, \ldots, K-1$ and potentials Φ_k, the shaped interior rewards are $r_k = \gamma\Phi_{k+1} - \Phi_k$ for $k < K-1$ and $r_{K-1} = 0$; the offsets are the discounted returns $b_k = \sum_{j \geq k} \gamma^{j-k} r_j$ with $\gamma = 0.995$, centered to a token-weighted mean of zero, clipped to ± 5, and re-centered. The rollout's total advantage contribution under the token-weighted loss therefore equals $\widehat{A}_{\text{rollout}}$, and potentials only redistribute credit within a rollout. If any segment lacks a potential, the rollout falls back to equal broadcast.

Segment selection and ownership. Each rollout keeps at most $K_{\max} = 6$ segments for training: the first, which trains the shared trunk; the last, which holds the terminal artifact; and the interior segments with the largest $\Delta\Phi_k$, falling back to evenly spaced segments when potentials are unavailable. Ownership is keyed by message position and content: a generated turn is trained by the first retained path that contains it and appears as loss-masked context in later paths, so every retained token is trained exactly once with the advantage of the segment in which it was generated. Tokens of a dropped segment are trained by the next retained path and take that path's advantage. Rewards are split as R/K across the K retained samples and re-summed per rollout before group statistics, so each rollout is one data point in GRPO.

Training window and log-probabilities. Rollouts use a 116,736-token window and training samples are capped at 65,536 tokens. A path longer than the cap keeps its most recent 65,536 tokens, so the trainable segment is preserved and the earliest history is dropped. Behavior-policy log-probabilities are recomputed by the training engine on the assembled sequence; rollout-engine log-probabilities are recorded to monitor the discrepancy between the two engines. Other settings follow Table 3, with the clipping threshold $\eta = 0.2$.

Table 3: Model initialization and hyperparameter settings for long-horizon RL training.

Item	Configuration
Policy initialization	Qwen3.8-27B
Training algorithm	GRPO
Tasks per training step	16
Rollouts per task	8
Learning rate	2×10^{-6} (constant)
Training precision	BF16
Training context limit	65,536 tokens
Rollout context limit	116,736 tokens
Maximum output per turn	16,000 tokens
Evaluation interval	Every 5 training steps
Evaluation set	28 fixed tasks

C.3 Rollout Infrastructure

Agent containers execute rollouts concurrently, while rollout collection and policy optimization alternate on shared GPU devices. Updated policy weights are synchronized before the next rollout phase. We use speculative decoding with the model's native MTP head and multi-turn prefix caching to improve serving efficiency (Leviathan et al., 2023; Zheng et al., 2024).

We monitor the log-probability differences between the rollout and training engines on the same generated tokens and support truncated importance sampling (TIS) as an optional correction (Yao et al., 2025). Transient trajectory-decoding failures are retried with backoff. Rollouts invalidated by infrastructure failures are excluded from training, while valid task failures are scored by the terminal verifier.

C.4 Training Configuration

We initialize the policy from Qwen3.8-27B and train it with GRPO. Each training step samples 16 tasks with eight rollouts per task. We use a constant learning rate of 2×10^{-6} and BF16 precision. Evaluation is performed every five training steps on a fixed set of 28 tasks. Table 3 summarizes these settings and the sequence length limits.

D Trajectory Scale and Rollout Efficiency

We collect 1,556 rollout sessions from Stage 1 (steps 1–14) to characterize the trajectory scale of long-horizon web development. Because context compaction rewrites the interaction history, per-turn token counts alone do not determine the exact length of the full deduplicated transcript. We therefore summarize session scale using the proxy $L = \max_t P_t + \sum_t O_t$, where P_t and O_t denote the prompt and completion token counts at interaction turn t, respectively. The median L is 364k tokens, the 90th percentile is 903k, and the maximum reaches 2.02M. As shown in Figure 12a, 97.9% of sessions exceed the 64k training window under this measure. This mismatch is also evident without the prompt-length term: cumulative generated tokens alone, $\sum_t O_t$, exceed the window in 94.5% of sessions. These statistics motivate context management and the construction of multiple training samples from each long rollout.

Long trajectories also incur repeated-input overhead during inference. The median session spans 153 interaction turns, with successive requests repeatedly including retained history. Let C_t denote the cached prompt-token count. We compare total requested tokens, $U_{\text{all}} = \sum_t (P_t + O_t)$, with cache-adjusted tokens, $U_{\text{cache}} = \sum_t (P_t - C_t + O_t)$. The median per-session ratios U_{all}/L and U_{cache}/L are 23 and 3.8, respectively (Figure 12b). Prefix caching reuses matching history prefixes, while compaction limits the context retained across interactions (Figure 13). These mechanisms address repeated-prefill overhead and context growth, respectively.

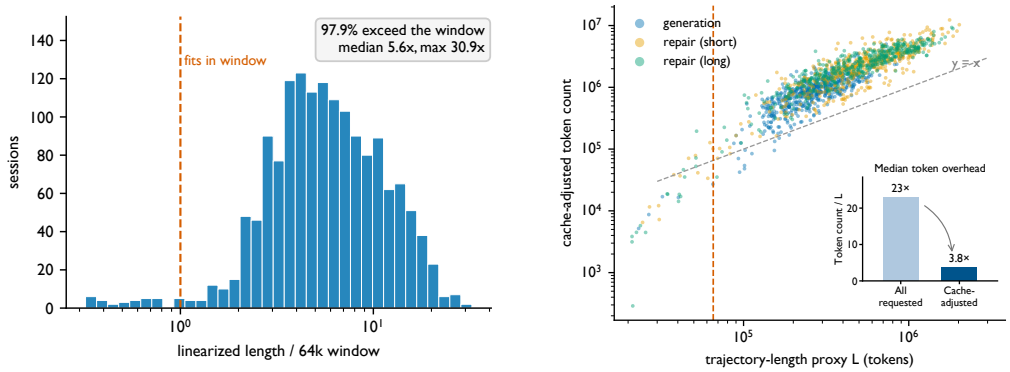

Figure 12: **Trajectory scale and inference token overhead.** (a) Distribution of the trajectory-length proxy normalized by the 64k training window across 1,556 rollout sessions. The dashed line marks one window; 97.9% of sessions exceed it. (b) Cache-adjusted token count versus L, with one point per session and colors indicating task families. The inset compares the median per-session ratios U_{all}/L and U_{cache}/L before and after subtracting cached input tokens (23× and 3.8×, respectively).

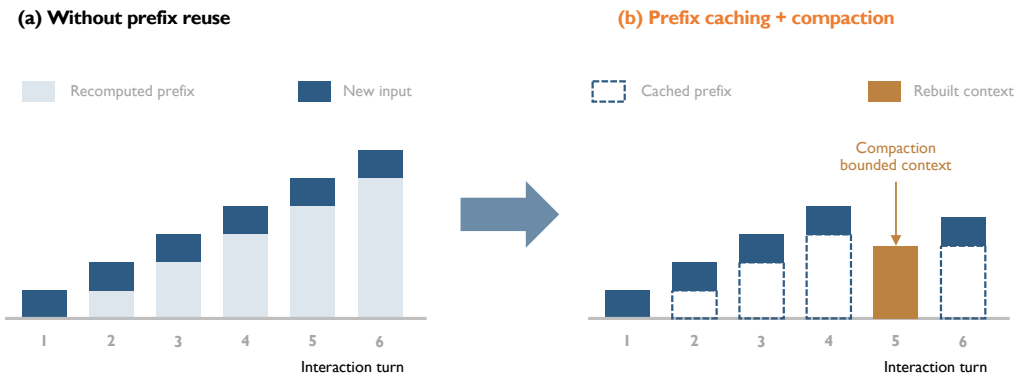

Figure 13: **Context compaction and prefix caching.** Left: historical inputs are repeatedly prefilled without prefix reuse. Right: matching prefixes are cached, and compaction rebuilds a bounded context. Filled regions indicate input requiring prefill; dashed outlines indicate reused prefixes. The two panels illustrate mechanisms rather than a controlled timing comparison. Turns and bar heights are schematic, not measured; decoding and cache-access costs are not depicted.

E VERIFIER RELIABILITY

Two model-based verifiers are used in this work. The *reward judge* scores the final artifact of every rollout during reinforcement learning and forms the vision–language-model leg of the terminal reward. The *acceptance judge* operates the running application at the end of each Webdev-Bench session and assigns every item of a hidden checklist an integer grade from 0 to 3, with 3 denoting full satisfaction. This appendix measures the repeated-grading error of both and shows, on concrete cases, that their scores correspond to visible differences in the artifacts. Notation follows Appendix B.2.3: a requirement counts as satisfied only when all of its items receive grade 3, and the mean over requirements is the requirement satisfaction score (RS); restricting items to the function-and-coverage, interaction, and visual dimensions gives FS, IS, and VS.

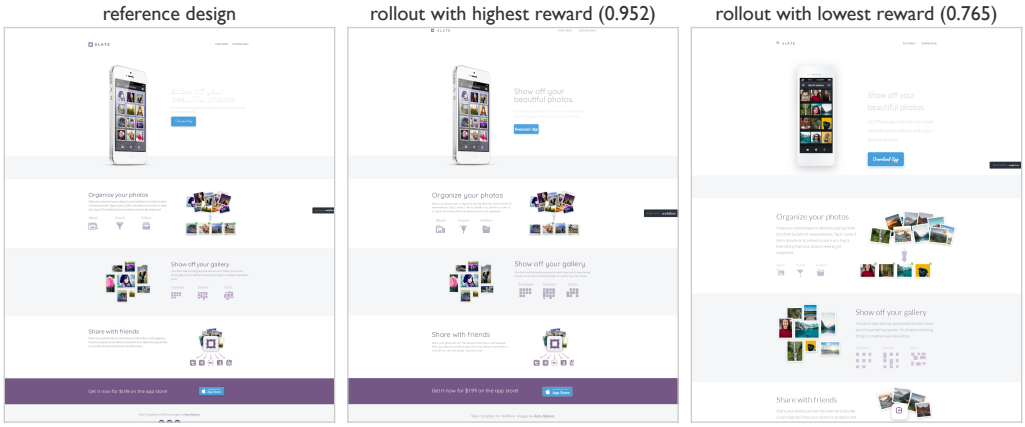

Figure 14: **Training-side case: renders of the highest- and lowest-reward rollouts of one reproduction task.** Panels show the reference design, the rollout with reward 0.952, and the rollout with reward 0.765, each cropped to the same height from the top of the page and scaled at native aspect ratio.

E.1 Training Reward Judge: Self-Consistency and Signal-to-Noise

The reward judge (GLM-5V-Turbo (Hong et al., 2026)) grades a rendered screenshot on a five-level anchored ordinal scale per dimension (excellent, good, fair, poor, broken, mapped to 1.0, 0.75, 0.5, 0.25, 0). The dimensions are fixed by task type—visual fidelity, content coverage, layout, text match, and render health for reproduction tasks; design quality, content quality, functionality, and render health for generation tasks—and their weighted sum is the judge-leg score $f_{\text{vlm}} \in [0, 1]$. The terminal reward is $R = g \cdot \left(w_{\text{det}} f_{\text{det}} + w_{\text{vlm}} f_{\text{vlm}}\right)$, where g is the product of the build, entry, and render gates, f_{det} is the programmatic deterministic leg (pixel comparison for reproduction, DOM-structure checks for generation), and $w_{\text{vlm}} = 0.4$.

Grading the same rendered screenshot repeatedly gives a per-draw standard deviation of 0.074 on the unit score range. The median of three draws reduces this to 0.019, and five draws lower it by only a further 7%. Training uses the median of three throughout, which bounds the judge leg's measurement error at about 2% of the range.

Whether this error is small enough depends on the differences it must resolve. GRPO builds its gradient from relative advantages among rollouts of the same task, so the relevant comparison is the within-group spread. In the Stage 1 training groups themselves (eight rollouts per task; 220 groups and 1,280 gate-passing rollouts from steps 1–14), the pooled within-group standard deviation of the reward is 0.078, against a judge noise of roughly 0.008 once propagated through its weight in the reward: genuine within-group differences are about ten times the measurement noise, and the gradient is driven by policy differences rather than grading jitter.

What a within-group spread corresponds to in the artifacts can be seen from the highest- and lowest-reward rollouts of a reproduction task with a wide within-group range (Figure 14, Table 4). The highest-reward artifact matches the reference in layout, section order, and the horizontal placement of the phone mock-up; the judge gives 1.0 on both visual fidelity and layout, and the pixel leg scores 0.920. The lowest-reward artifact turns the mock-up into a tall vertical image, widens section spacing, and stretches the page by about 18%; the judge lowers visual fidelity to 0.75 and the pixel leg falls to 0.659. Both legs move in the same direction, and the judge's higher score corresponds to a visibly closer reproduction.

Table 4: **Judge dimension scores and deterministic-leg score for the two rollouts in Figure 14.** Dimension scores are medians of three draws.

Rollout	Visual fidelity	Coverage	Layout	Text match	Render health	Pixel leg f_{det}	Reward R
Highest	1.00	1.00	1.00	1.00	1.00	0.920	0.952
Lowest	0.75	1.00	1.00	1.00	1.00	0.659	0.765

Table 5: **Evaluation-side consistency case: per-requirement outcome of three acceptance passes on the same artifact.** Cells give the number of items graded 3 over the number of items in the requirement.

	Req. 1 dashboard	Req. 2 search and filters	Req. 3 watch list	Req. 4 sidebar and views
Pass 1	5/5	6/6	5/6	5/5
Pass 2	5/5	6/6	5/6	5/5
Pass 3	5/5	6/6	5/6	5/5

E.2 WEBDEV-BENCH ACCEPTANCE JUDGE: REPEATED-GRADING CONSISTENCY

For the 77 tasks from six model runs on which three independent terminal acceptance passes were executed and all three per-item outputs parse, we align the three grades of every item, giving 999 item triples; each pass is a separate session in which the judge restarts and operates the application before grading. At the item level, each item is classified by the range of its three grades: identical in all passes, differing by at most one grade, or differing by two or more. At the task level, RS is computed for each pass and compared across the three passes. Figure 15 shows the result: 89.8% of items receive identical grades in all three passes, 8.8% differ by one grade, and 1.4% by two or more. The three passes agree closely at both the item-grade and the task-level RS layer, indicating that the acceptance judge's grading is stable and reproducible.

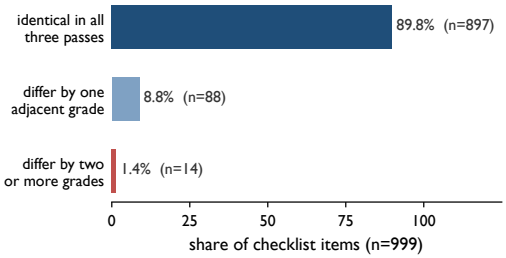

Figure 15: **Agreement of the acceptance judge across three independent passes.** Share of 999 checklist items whose three grades are identical, differ by one adjacent grade, or differ by two or more grades.

The three-pass record of one task shows what this agreement looks like in practice (Table 5). The task is a store-operations dashboard with four requirements and 22 items, built by Claude Opus 4.8. The number of items graded 3 under each requirement is identical across the three passes, and the same item is the one deducted every time: requirement three asks that, once six stores are on the watch list, clicking a further store shows a visible message that the limit has been reached. In all three passes the judge filled the list to six, confirmed that the seventh click was blocked, and recorded that no visible message appeared, assigning grade 2 each time. Three independent sessions deducting at the same place for the same reason indicates that the verdict reflects the artifact rather than grading randomness.

Whether a deduction corresponds to a visible defect is checked on one task built by two model lines (Figure 16, Table 6). The task is c22 of Section 3.5: starting from the website of a blockchain project, the agent repairs dead navigation links, tightens the hero, compacts the five "Why SEDA" parts into two rows of three and two within a 1400-pixel height budget, adds a 48-article newsroom, and wires it into the navigation and the statistics bar. The WebCoder-27B artifact passes all 27 checklist items (RS 1.0). The Gemini-3-Pro artifact completes the first two rounds and the newsroom, but its five parts remain staggered—three in the first row and the remaining two offset one below the other—the block is 1613 pixels tall, and the numerals 1–5 are hidden behind the images. The judge records exactly these observations, measuring the card offsets (451 pixels between the fourth and fifth cards) and the block height and confirming by pixel comparison that the numerals are covered, and grades the layout and numeral items 1 and the height item 2; two further items are graded below 3 because

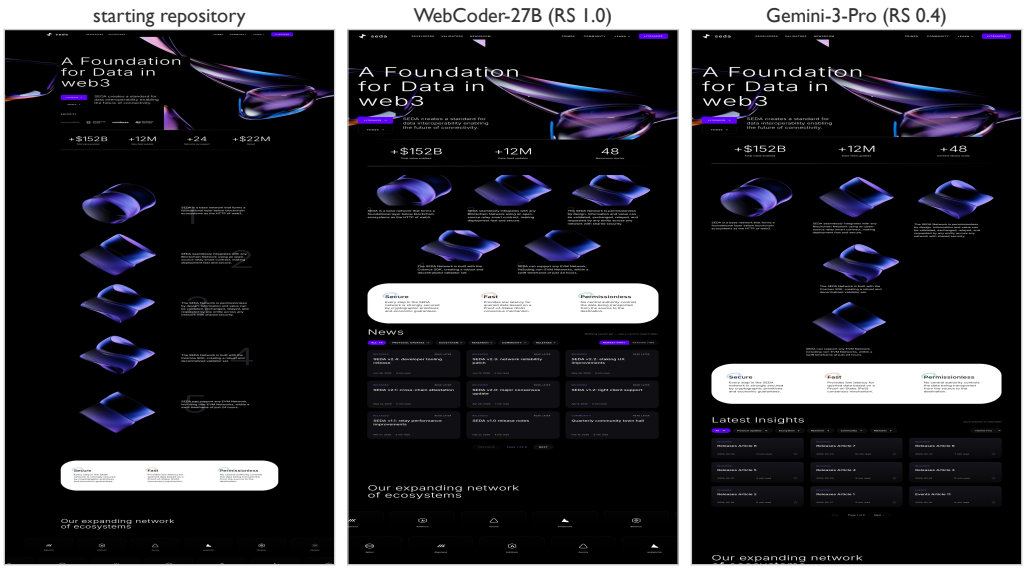

Figure 16: **Evaluation-side contrast case: final-state screenshots of the same task built by two model lines.** Panels show the starting repository's homepage, the WebCoder-27B artifact, and the Gemini-3-Pro artifact, each cropped to the top of the page at the same aspect ratio. WebCoder compacts the five "Why SEDA" parts into two rows of three and two; the Gemini-3-Pro artifact leaves the last two parts staggered below the first row, exceeding the height budget, with the numerals hidden behind the images.

Table 6: **Per-requirement acceptance outcome and items below grade 3 for the two artifacts in Figure 16.**

Model	Requirements 1–5	RS	Items below grade 3 (grade)
WebCoder-27B	1, 1, 1, 1, 1	1.0	none
Gemini-3-Pro	1, 1, 0, 0, 0	0.4	five parts in two rows (1), numerals visible with their texts (1), block height within 1400 px (2); pager shown for single-page categories (2); statistics cell linked to the newsroom (1), button style consistency (2)

the statistics cell is not a link and the newsroom filter pills do not follow the site's button style. Requirements three, four, and five are therefore unsatisfied (RS 0.4), and the two deductions that decide requirement three are visible in the screenshot.

F Analysis of Agent Behavior

To analyze the development behavior associated with changes in training reward, we collect 1,556 rollouts from Stage 1, steps 1–14, and examine their model-call records, saved conversation histories, and final-artifact evaluations. Based on the reward curve, we divide the analysis into three windows: Early (steps 1–4), Trough (steps 8–11), and Recovery (steps 12–14). The behavioral analysis focuses on replication tasks, where the reward decline is more pronounced. The three windows contain 291, 267, and 181 replication rollouts spanning 30, 27, and 28 tasks, respectively.

We measure three aspects of tool use from the observed conversation windows after deduplicating overlapping history:

- **Redundant operations.** The fraction of observed tool calls that repeat the same tool and arguments without an intervening call classified as modifying.

- **Observed skill use.** The fraction of rollouts with observable text that contain at least one skill invocation.
- **Observed webshot use.** The fraction of rollouts with observable text that contain at least one `webshot` command.

Figure 17 shows behavioral changes across the three windows for replication tasks. During the trough, the redundant-operation rate rises from 0.24% to 2.45%, while skill invocation falls from 24.1% to 15.7% and webshot invocation from 79.8% to 66.9%. The reward decline thus coincides with a larger fraction of tool calls repeating previous operations and fewer rollouts invoking Skills or webshot. During recovery, the redundant-operation rate decreases to 0.86%, while skill and webshot invocation rise to 20.8% and 78.7%, respectively. All three metrics move toward their early levels, suggesting a partial improvement in development behavior alongside reward recovery, with fewer redundant operations and renewed use of Skills and webshot. We further examine these changes through policy entropy, generated-word frequencies, and an observed operation sequence.

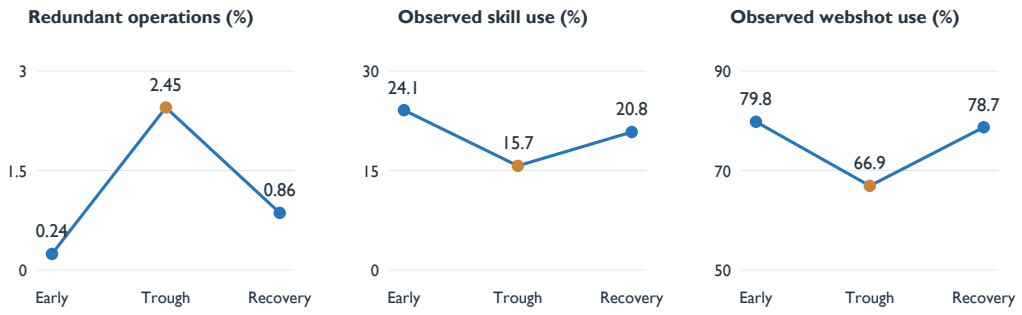

Figure 17: Tool-use patterns across training windows in replication tasks:redundant-operation rate, observed skill use, and observed webshot use. Early, Trough, and Recovery correspond to steps 1–4, 8–11, and 12–14, respectively.

To examine whether these behavioral changes coincide with changes in the model's predictive distribution, we analyze policy entropy during training. Entropy is computed over the full vocabulary and averaged over generated tokens in each training batch. As shown in Figure 18(a), policy entropy decreases from 0.452 nats at step 1 to 0.386 at step 9, then rises to 0.433 at step 14. Lower entropy during the trough reflects more concentrated predictions on the observed contexts and coincides with increased redundant operations. During recovery, entropy rises as redundant operations decrease.

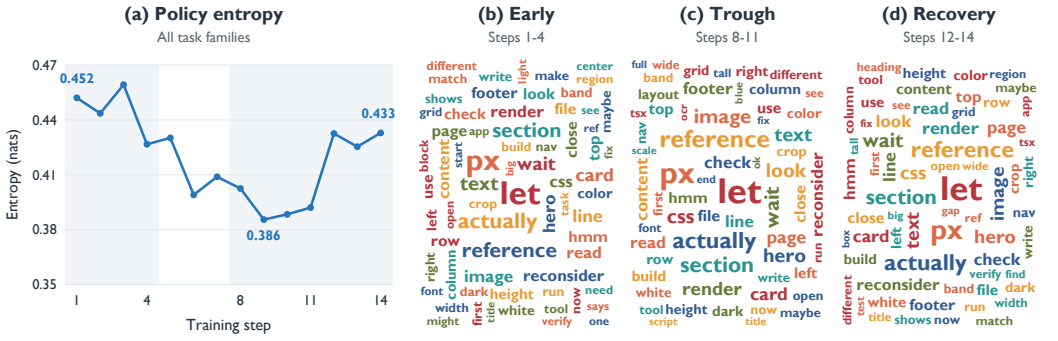

Figure 18: (a) Training-policy entropy. (b–d) Frequent words in replication tasks during Early, Trough, and Recovery. Font sizes use a shared square-root mapping of word frequency across all three clouds; color does not encode a statistic.

We further analyze generated language in replication trajectories to examine how the model's descriptions of development change during training. The corpus is drawn from deduplicated observed

conversations, retaining reasoning text and prose outside tool calls. We filter out code, tool arguments, stopwords, and task-specific terms, and merge word forms. Frequencies are normalized per 10,000 retained words within each window. Figure 18(b–d) shows the frequent-word distributions, and Table 7 reports frequencies for selected terms.

Table 7: Selected word frequencies per 10,000 retained words.

Word	Early	Trough	Recovery
`look`	63.2	125.3	89.1
`reference`	148.5	219.3	183.3
`check`	60.1	85.3	84.3
`edit`	19.3	8.1	10.3
`run`	42.2	31.1	37.6

During the trough, `check` increases from 60.1 to 85.3 occurrences per 10,000 words, and `reference` increases from 148.5 to 219.3, while webshot invocation declines. Inspection-related language becomes more frequent without a corresponding increase in webshot use. This divergence shows that emphasis on inspection in generated text and actual tool use can change in different directions. Operation records are therefore needed to determine whether these descriptions translate into development actions.



Figure 19: **A terminal-style excerpt from a recorded rollout.** Commands, paths, outputs, and image-omission notices are reproduced from the saved records; orange highlights repeated reads. `Read` is an agent tool, and the `harness` lines contain framework feedback. The figure is a typeset transcript, not an original terminal screenshot.

To illustrate repetitive behavior during the trough, we select a highly repetitive replication rollout from training step 11. Figure 19 presents turns 37–42 from a single continuous message chain: after cropping the reference image, the model reads the same file twice, then crops a larger region and again reads the resulting file twice.

The cropping commands successfully create files, but every `Read` call returns an image-omission notice, so the image content is not passed to the policy. No repository edits are observed in the saved windows, and the final artifact retains scaffold content, receiving a task reward of 0.031. This case illustrates that repeated calls to inspection tools do not necessarily provide usable feedback or advance the project. Assessing the contribution of long-horizon interactions therefore requires examining whether the operations advance the artifact, motivating segment-level credit assignment based on artifact progress.

G Full Results

Table 8 reports the complete Webdev-Bench metrics of every evaluated model, using the same models, ordering, and scoring as Table 1. The four right-hand columns give the item-level grade-3 rate per requirement dimension, which is unaffected by the conjunctive rule and shows the absolute degree of completion: WebCoder reaches 96% on interaction items, the highest of all models, and 85% on functional items, against 89% and 79% for the base model.

Table 8: **Full Webdev-Bench results.** The left six columns follow the metrics and ordering of Table 1; the right four columns give the item-level grade-3 rate, i.e., the fraction of checklist items of each dimension graded 3 at terminal acceptance (items of undelivered tasks count as grade 0). Best in bold, second best underlined.

Model	Webdev-Bench						Item grade-3 rate			
	RS	**VS**	**FS**	**IS**	*Avg*	*DSR*	Functional	Coverage	Interaction	Visual
Proprietary Models										
Claude-Opus-4.8	65.1	75.0	70.7	84.1	73.7	100.0%	83.6	83.3	91.6	86.1
GPT-5.4	59.6	72.7	70.0	81.0	70.8	100.0%	83.1	85.2	91.3	84.8
GPT-5	53.5	64.0	65.8	74.6	64.5	100.0%	81.4	80.6	86.3	77.0
Gemini-3-Pro	48.0	65.8	58.5	68.5	60.2	100.0%	75.2	75.9	80.3	77.7
Gemini-3-Flash	56.6	65.8	64.8	85.0	68.1	95.0%	80.4	84.3	91.6	78.6
Seed-1.8	52.6	66.5	63.8	71.5	63.6	100.0%	79.2	84.3	87.0	79.9
Open-Source Models										
Kimi-K2.5	46.8	61.5	55.0	67.7	57.7	82.5%	68.4	72.2	73.6	68.3
MiMo-V2.5	14.9	25.2	21.2	22.6	21.0	65.0%	28.4	46.3	27.8	37.5
Step-3.7-Flash	10.8	20.0	14.0	19.0	15.9	60.0%	18.6	34.3	23.1	28.2
Qwen3.8-27B	64.8	77.9	68.8	79.8	72.8	95.0%	79.2	85.2	88.6	86.7
Qwen3.5-397B-A17B	28.7	39.0	31.9	36.5	34.0	80.0%	42.2	56.5	47.2	49.2
Qwen3.5-122B-A10B	24.6	30.4	28.9	33.1	29.3	72.5%	36.3	44.4	40.8	41.7
Qwen3-VL-32B-Instruct	2.5	6.3	5.4	4.4	4.6	50.0%	11.8	6.5	10.4	15.5
Qwen3-VL-8B-Instruct	0.0	0.8	0.0	0.0	0.2	30.0%	0.5	1.9	1.0	1.9
WebCoder-27B	**72.8**	**81.9**	**73.8**	**90.8**	**79.8**	100.0%	**84.8**	**88.0**	**95.7**	**89.3**

Table 9: **Webdev-Bench results by task starting point.** Tasks start from an existing repository (brownfield), a reference screenshot (replication), or a textual specification (text-to-project); each cell is the mean over the tasks of that type, with undelivered tasks counted as 0. Best in bold, second best underlined.

Model	Brownfield (n=11)				Replication (n=9)				Text-to-project (n=20)			
	RS	**VS**	**FS**	**IS**	**RS**	**VS**	**FS**	**IS**	**RS**	**VS**	**FS**	**IS**
Proprietary Models												
Claude-Opus-4.8	79.5	90.9	90.0	90.9	**58.9**	61.1	**78.3**	77.8	60.0	72.5	56.7	83.2
GPT-5.4	71.8	75.0	87.7	95.5	55.0	**71.3**	66.7	74.1	55.0	72.1	61.7	76.1
GPT-5	59.5	65.9	86.8	74.2	37.2	50.9	46.7	61.1	57.5	68.8	62.9	80.8
Gemini-3-Pro	55.0	62.1	80.5	85.6	23.3	52.8	32.2	40.7	55.3	73.8	58.3	71.6
Gemini-3-Flash	58.2	65.2	85.5	90.2	44.4	66.7	55.6	**79.6**	61.3	65.8	57.5	84.6
Seed-1.8	52.3	62.1	80.5	66.7	42.8	63.9	55.6	55.6	57.3	70.0	58.3	81.3
Open-Source Models												
Kimi-K2.5	60.0	76.5	82.3	83.3	32.8	53.7	50.6	55.6	45.8	56.7	42.1	64.6
MiMo-V2.5	23.2	43.2	29.1	34.8	2.2	8.3	24.4	13.0	16.0	22.9	15.4	20.2
Step-3.7-Flash	15.5	34.1	21.8	27.3	0.0	5.6	7.8	3.7	13.0	18.8	12.5	21.3
Qwen3.8-27B	**89.1**	**97.7**	**97.7**	90.2	51.7	70.4	66.7	64.8	57.3	70.4	53.8	80.8
Qwen3.5-397B-A17B	32.7	47.7	41.8	37.9	17.2	22.2	30.6	18.5	31.8	41.7	27.1	43.8
Qwen3.5-122B-A10B	32.7	45.5	42.7	47.7	8.9	9.3	15.0	3.7	27.3	31.7	27.5	38.3
Qwen3-VL-32B-Instruct	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	5.0	12.5	10.8	8.8
Qwen3-VL-8B-Instruct	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.7	0.0	0.0
WebCoder-27B	**89.1**	**97.7**	93.2	**97.0**	**58.9**	68.5	70.6	74.1	**70.0**	**79.2**	**64.6**	**95.0**

Table 9 splits the results by task starting point. The gains of WebCoder over the base model concentrate on text-to-project tasks (RS 70.0 versus 57.2, with Claude-Opus-4.8 at 60.0) and on replication tasks (RS 58.9 versus 51.7); on text-to-project tasks WebCoder has the highest value in all four metrics.

H HUMAN EVALUATION

The acceptance judge is itself a model, so we ask professional annotators with front-end development experience to check, by pairwise comparison on the 40-task Webdev-Bench suite, whether its preference order agrees with theirs. Each item pairs the final deliverable of WebCoder-27B with that of one of four opponents on the same task: the Qwen3.8-27B base model, Claude Opus 4.8, GPT-5.4, and Gemini-3-Flash, giving $40 \times 4 = 160$ items. The two deliverables are shown anonymously side by side as columns A and B, with the column-to-model assignment drawn at random for every item; each column shows the full-page screenshot captured at the terminal acceptance pass and the per-round screenshots, and the task header lists the requirements exactly as they were revealed to the model together with the starting material. The annotator chooses Win A, Tie, or Win B.

The evaluation policy is requirement-driven: for every requirement the annotator checks whether the required content is present, whether functional results are consistent with the specification, and whether interaction state is visibly evidenced; visual fidelity and overall usability are considered only afterwards. A deliverable that fails to start loses outright, and a tie is reserved for pairs that satisfy essentially the same requirements with no clear visual gap.

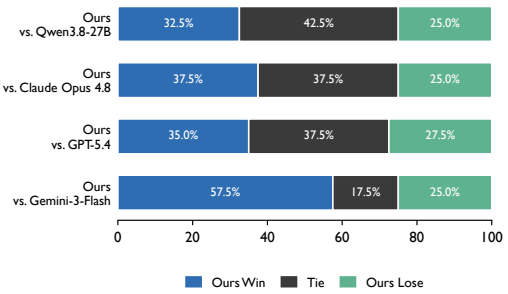

Figure 20: **Pairwise human preference for WebCoder-27B against four opponents on the 40-task Webdev-Bench suite.** Each row gives the share of the 40 items in which WebCoder wins, ties, or loses.

Figure 20 reports the results. Against all four opponents the win-or-tie share of WebCoder-27B lies between 72.5% and 75%: against the Qwen3.8-27B base it wins 32.5%, ties 42.5%, and loses 25.0% of the items; against Claude Opus 4.8 it wins 37.5%, ties 37.5%, and loses 25.0%; against GPT-5.4 it wins 35.0%, ties 37.5%, and loses 27.5%; against Gemini-3-Flash it wins 57.5%, ties 17.5%, and loses 25.0%. Annotator comments show that disagreements concentrate on interaction requirements, such as whether a filter narrows the list correctly or whether an over-limit action is blocked with a message; these items can only be confirmed by operating the application, which is exactly what the acceptance judge verifies item by item on the running deliverable.

I ADDITIONAL CASE STUDIES

This appendix complements Section 3.5 with three views of the same two benchmarks: how WebCoder compares with other model lines on a task that all of them completed, what its full trajectory looks like on the two tasks of the main text, and a task on which its deliverable does not pass acceptance.

I.1 COMPARISON WITH BASELINES ON A SHARED TASK

Figure 21 shows the Vision2Web title *caplugs*, on which every evaluated line delivered a running site, so the comparison is between complete deliverables rather than between delivery and failure. WebCoder leads on both metrics (VS 79.1, FS 87.5, Avg 83.3). The base model keeps the layout but fails three of eight functional workflows; several larger lines replace the photographic hero with a flat colour band or alter the header and product grid of the catalogue page. The difference is not one of delivery rate: on this title WebCoder is ahead by 14.5 points over the second-best line with every deliverable running.

Figure 22 shows the Webdev-Bench task c03, a five-requirement modification of an existing school website, on which thirteen of the fifteen lines delivered a final application. The final pages look nearly identical because the starting repository is shared, yet conjunctive acceptance spreads the lines from RS 1.0 to 0.2. WebCoder, Claude Opus 4.8, and GPT-5.4 satisfy all five requirements. The base model and Gemini-3-Flash each miss one requirement: the base model fails the interaction

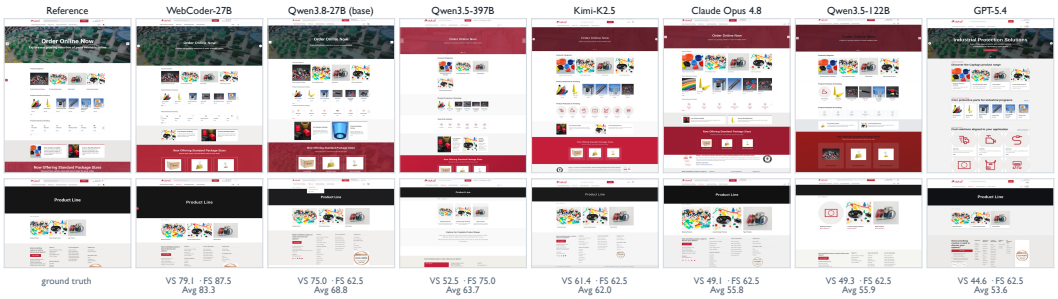

Figure 21: **Vision2Web, caplugs: reference and final renders of seven model lines.** Top row shows the homepage, bottom row the product-catalogue page, both cropped from the top. Scores below each column are VS, FS, and their average for the whole six-page title.

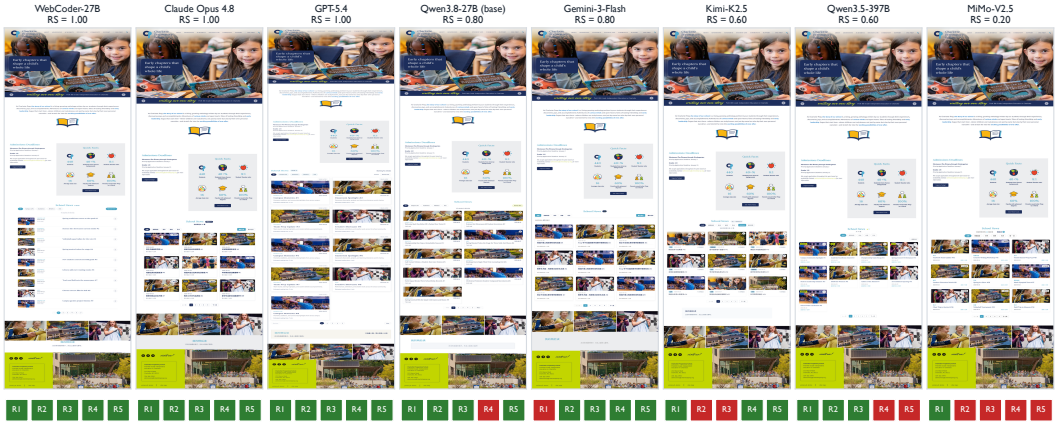

Figure 22: **Webdev-Bench, c03: final applications of eight model lines and their per-requirement acceptance.** Each column shows the full final page after the fifth requirement; the strip below marks each requirement as satisfied (green, all items grade 3) or not (red).

item that asks for an empty-state message and a zero counter after all saved articles are removed, and Gemini-3-Flash fails a layout item of requirement one, leaving a blank gap where the promotional card was removed. The weaker lines lose two to four requirements although their pages contain the same sections. The case illustrates why the benchmark grades by requirement rather than by appearance.

I.2 Complete Trajectories of WebCoder

Figure 23 gives the full c22 trajectory summarised in the main text. The starting repository is the single-page website of SEDA. Requirement one repairs three dead PRIMER entries so that they scroll smoothly to their target block like the other links. Requirement two removes the investor row and the fundraising figure, tightens the hero from 1085 to at most 900 pixels, and redistributes the remaining statistics. Requirement three compacts the five staggered "Why SEDA" parts into two rows of three and two within 1400 pixels while keeping every text and image. Requirement four adds a newsroom of 48 articles in five categories with per-category counts on the filter buttons, two sort orders, nine articles per page with pagination and a page indicator, and a read-later mark with a counter and an empty-state message. Requirement five adds a Newsroom entry to the navigation that scrolls to the section and resets category and page while preserving the marks, and replaces one statistics cell with the article count. The simulated-user gate, which judges the current requirement's checklist items from the running page without revealing them, returns the feedback reproduced under the figure after the first attempt at requirement three: the five parts are still in the old layout and exceed the height budget. The second attempt rearranges them into two rows and passes. The

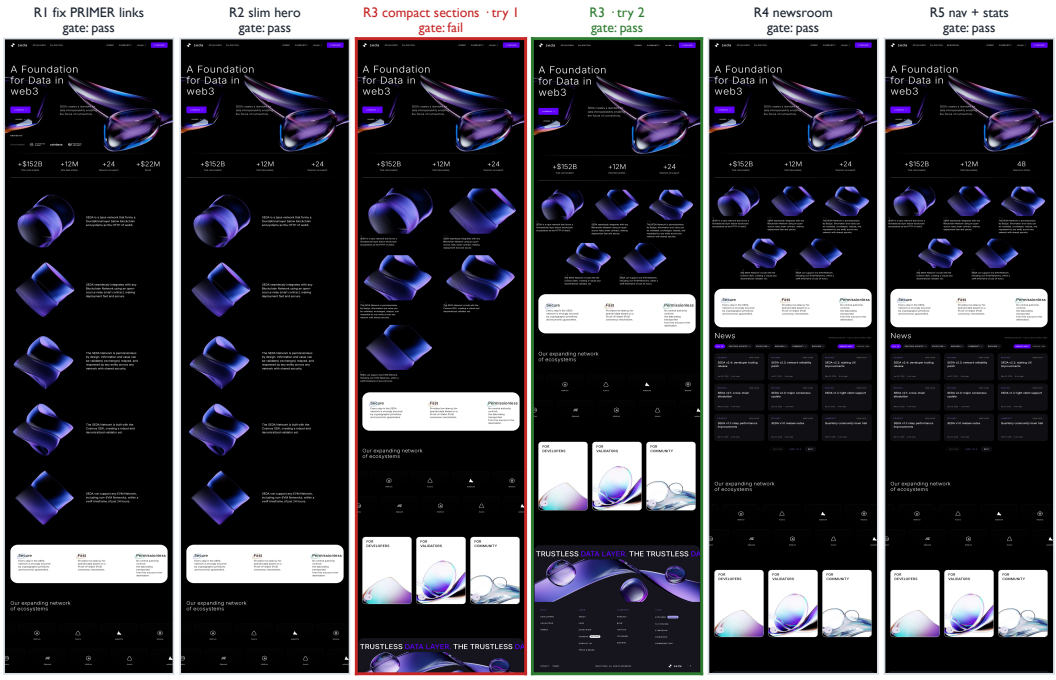

Figure 23: **Webdev-Bench, c22: every attempt of WebCoder.** The first attempt at requirement three is rejected by the simulated-user gate (red frame); the second attempt passes (green frame) and the remaining rounds pass at the first attempt. The box gives the gate's feedback after the rejected attempt, translated from the original.

other four rounds pass at the first attempt, and the terminal acceptance grades all 27 checklist items 3. The trajectory shows the intended division of labour between the two verifiers: the gate drives development with natural-language feedback and the terminal acceptance checks every item on the finished application.

Figure 24 shows all six pages of the caplugs deliverable against their references. The homepage and catalogue pages score highest (0.92 and 0.95 mean section score), and the material-specification and contact pages reproduce the dense table and the form; the industry-search page scores lowest because its search block and chat widget are missing, and the product-detail page is penalised for a header and share buttons that the cropped reference does not show. Of the thirteen functional test cases in six workflows, one fails: the footer link for an industry opens a page headed by the industry name instead of the expected product-line heading.

I.3 A Trajectory That Does Not Pass Acceptance

Figure 25 shows c15, which first reproduces the homepage of the handwritten-mail platform Simply Noted from a reference, then rearranges the template area into two columns with category counts, removes the video section and part of the statistics, and finally makes filtering, search, pagination, a plan counter, and a live price quote on two product cards work. The gate rejects the first two attempts at requirement one and passes the third; requirements two to four pass at the first attempt. Terminal acceptance grades 21 of the 22 checklist items 3. The single deduction is a grade of 2 on the price-quote item: for a postcard quantity of 499 the page should show $344.31 but renders

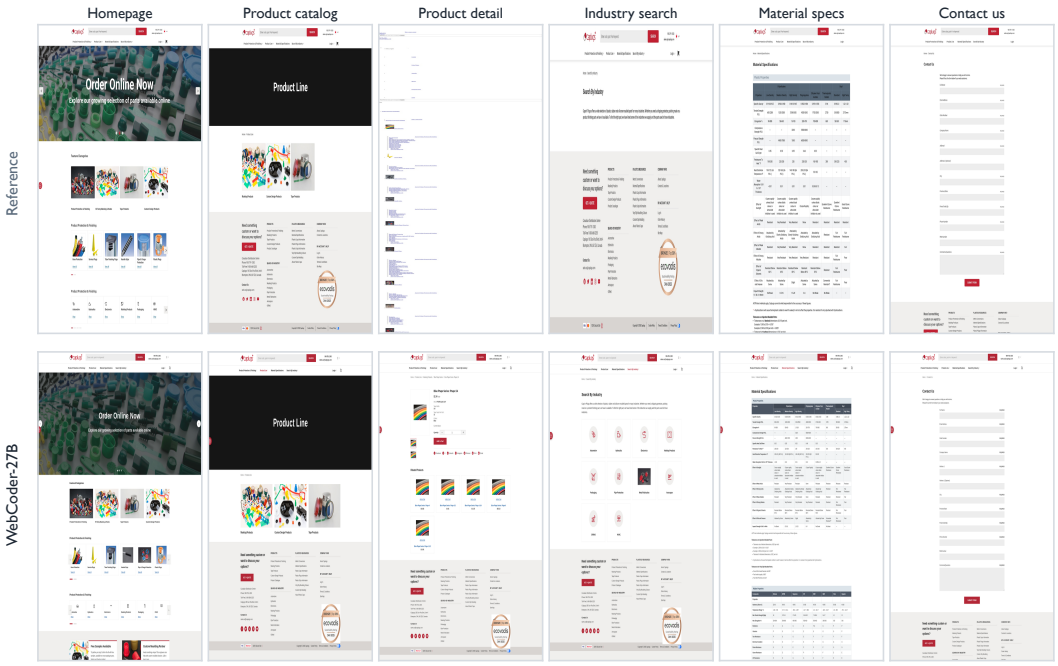

Figure 24: **Vision2Web, caplugs: all six pages of the WebCoder deliverable.** Top row shows the reference screenshots, bottom row the judge-time renders of the served application, cropped from the top.

"$344.30.999999999992724", a floating-point remainder that the truncation logic does not cut off; the other four quantities specified by the item (100, 500, 505, and 800 folded cards) are correct. This one formatting defect makes requirement four unsatisfied and leaves RS at 0.75, which is the intended strictness of grading by requirement.

The same item is lost across model lines: on c15, twelve of the fourteen delivered lines fall below grade 3 on the price quote, with Kimi-K2.5 and GPT-5 at grade 1 and Qwen3.5-397B at 0, while GPT-5.4, Gemini-3-Pro, Gemini-3-Flash, Seed-1.8, and WebCoder share grade 2. Currency arithmetic with discount and truncation rules is where current agents most often fall one grade short on tasks of this kind; on c15 WebCoder's RS of 0.75 ties with Kimi-K2.5 and Seed-1.8 for the highest, with Claude Opus 4.8 and GPT-5.4 at 0.5.



Figure 25: **Webdev-Bench, c15: the WebCoder trajectory on a task graded RS** 0.75**.** Panels show the reference design, the first passing state of requirement one (after two gate rejections), the state after requirement two, and the terminal state after requirement four, all cropped to the top 4:1 region and shown at the same scale.

J PROMPTS AND SKILLS

This section presents the prompts and on-demand skills supplied for WebCoder training and Webdev-Bench task construction and evaluation. We first organize the prompts by their role in each pipeline, then describe the skill library and reproduce its definitions.

J.1 PROMPTS

Rollout System Prompt. Recipient: development agent. Defines workspace constraints, skill selection, and the available rendering workflow.

Prompt 1: Rollout System Prompt

```
You are an expert frontend developer. Pick and follow the most
relevant Skill available for the user's frontend task. Work only
within /workspace. Keep the project files directly at /workspace
unless the existing repository already has its own structure. Verify
your work with build/browser checks when possible.

The command `webshot` is already installed. It builds the project,
serves it, screenshots it, and is the same renderer that scores your
work. Run it before you finish, read `/tmp/webshot/render.png` with
the Read tool, and check `/tmp/webshot/measurement.json`: `textLength:
0` means the app never mounted (read `/tmp/webshot/build.log`), and
`brokenImages` must be empty. Do not write your own Playwright script
and do not install Playwright -- `require('playwright')` fails in this
runtime by design.

Claude Code, Node.js, npm, Python Playwright, Node Playwright, and
Chromium are already installed in this runtime. Use
PLAYWRIGHT_BROWSERS_PATH as configured. Do not run apt installs, pip
install Playwright, npm install -g, or `npx playwright install`;
install only dependencies required by the project itself.
```

First-Turn Task Envelope. Recipient: development agent. Combines task fields and the guidance blocks shown below; braces indicate runtime template fields.

Prompt 2: First-Turn Task Envelope

```
User task:
{task.user_query}

Task type: {task.task_type}

Input assets:
{assets}

Execution requirements:
{guidance}
{vision_section}{webshot_section}

All final project files must be located in /workspace. Ensure the
completed code is readable and maintainable.
```

Generation Guidance. Recipient: development agent on text-only tasks. Directs an initial runnable implementation followed by screenshot-guided refinement.

Prompt 3: Generation Guidance

```
You have approximately 120 tool-call turns available. First build a
rough version of the entire page, then use the relevant Skill to
refine it.

A React frontend project has already been initialized in repo/, with
its dependencies installed. The following paths already exist:
repo/index.html, repo/package.json, repo/src/App.tsx, and
repo/node_modules.

Do not create a new project, run npm install, or modify
repo/package.json or repo/tsconfig.json. Doing so may cause
dependency mismatches, resulting in a build failure and a score of 0.

Step 1: Edit repo/src/App.tsx directly, keeping components and styles
under repo/src/. Implement every section requested by the user with
real content, even if the initial version is rough. Run webshot once
to confirm that the project builds and renders. Do not write a PRD or
planning files before the page successfully renders.

Step 2: Once the page renders, use the relevant text-to-project Skill
to refine its content and styling. Rerun webshot after refining each
section to verify the changes.
```

Replication Guidance. Recipient: development agent on screenshot tasks. Directs an initial full-page implementation followed by section-level visual comparison.

Prompt 4: Replication Guidance

```
You have approximately 120 tool-call turns available. First build a
rough version of the entire page, then use Skills for section-level
refinement.

The project has already been initialized in repo/, with its
dependencies installed. The following paths already exist:
repo/index.html, repo/package.json, repo/src/main.tsx,
repo/src/App.tsx, and repo/node_modules. Do not create a new project,
run npm install, or modify repo/package.json or repo/tsconfig.json.
Otherwise, dependency mismatches will cause the build to fail and
result in a score of 0.

Step 1: Edit repo/src/App.tsx directly, keeping components and styles
under repo/src/. Use the input image to implement all sections with
real content, even if the initial version is rough. Then run webshot
once to confirm that the project builds and renders. Before the first
successful render, do not crop images, write color-analysis scripts,
or create planning files. These activities can leave you submitting
an empty page that only says "Replace this with the page.", which
receives a score of 0.

Step 2: Once the page renders, refine it. This is where you earn
points and should spend most of the remaining turns. Use the mounted
long-image/replication Skills to refine the page section by section.
For each section, use PIL to crop that section from the reference
image and crop the same region from /tmp/webshot/render.png. Then use
the Read tool to view both crops as images and compare them directly.
After editing repo/src/, rerun webshot to verify that section.
```

Prompt 4: Replication Guidance (continued)

```
Do not substitute text-based methods such as numpy pixel statistics,
average colors, or tesseract OCR for actually viewing the crops.
Scoring details such as borders, corner radii, font weight, font
size, spacing, and alignment require visual inspection. Looking only
at the full-page render.png is also insufficient; you must inspect
separate crops for each section. Rebuild text, buttons, navigation,
and similar elements using DOM/CSS. Do not bake the entire page into
a bitmap.
```

Vision-Anchor Extraction. Recipient: vision model. Input: prepared reference views. Output: structured page identity, visual attributes, section order, and disambiguation cues.

Prompt 5: Vision-Anchor Extraction

```
Analyze these reference views for a website reproduction agent.

Return JSON only, no markdown. Use this schema:
{
  "status": "ok",
  "brand": "",
  "page_type": "",
  "hero_headline": "",
  "nav_items": [],
  "primary_palette": [],
  "section_order": [],
  "must_match": [],
  "avoid_confusion": ""
}

Keep the JSON compact: at most 6 nav_items, 6 palette entries, 10
sections, 8 must_match bullets, and one short avoid_confusion
sentence. Focus on the outer webpage being reproduced. If device
mockups, dashboard cards, product screenshots, or nested page
previews appear inside the screenshot, do not treat their internal
brand/content as the outer webpage's brand.
```

Reference-Image Guidance. Recipient: development agent. Explains how to use the reference, the anchor file, and cropped views during reconstruction.

Prompt 6: Reference-Image Guidance

```
Mandatory visual-alignment requirements:
{anchor_note}
- Before coding, read /workspace/input/vision_anchor.json. If the
file exists, first confirm
brand/page_type/hero_headline/palette/section_order.
- After reading the anchor, briefly inspect the overview image under
/workspace/input/vision/, then begin implementing the page directly.
Save the standard top/middle/bottom views and local crops for
section-level refinement after the first successful render. Do not
crop images, run OCR, or write analysis scripts before the first
render.
- The final page's brand, theme, hero headline, primary palette, and
section order must match the vision anchor.
- If a local screenshot, phone mockup, or product screenshot contains
another brand or page, do not mistake it for the theme of the outer
webpage.
```

Prompt 6: Reference-Image Guidance (continued)

```
- If later crop observations conflict with the vision anchor, recheck
the original input image first instead of directly switching to the
theme shown in the local crop.
```

Build-and-Render Self-Check. Recipient: development agent. Uses webshot outputs to inspect rendering, build errors, missing text, and broken images.

Prompt 7: Build-and-Render Self-Check

```
You must perform a self-check before submission; this is your only
way to inspect your own artifact:
- Run `webshot`. It is already installed and requires no installation
steps. You may also specify the directory with `webshot
/workspace/repo`. It builds the project, starts the service, and
takes screenshots using the same renderer used for your final score.
- Use the Read tool to view `/tmp/webshot/render.png` and compare it
section by section against the input image or requirements.
- Read `/tmp/webshot/measurement.json`: a `textLength` of 0 means the
page has not mounted at all, almost always because of syntax errors;
inspect `/tmp/webshot/build.log`. `brokenImages` must be empty. A
nonzero build returncode means that the image you are viewing is
stale.
- After fixing issues, run `webshot` again. Do not write your own
Playwright script or attempt to install playwright. In this runtime,
`require('playwright')` / `import playwright` will always fail.
```

Long-Page Replication Example. A concrete task message with reference dimensions, content requirements, and delivery instructions. Task-asset paths identify inputs that are not included in this appendix.

Prompt 8: Long-Page Replication Example

```
Reproduce the webpage screenshot at
/workspace/input/assets/1c54e90d85d20f6f44c7e1c56d1d55fb.jpg with
high fidelity, as a runnable frontend project.

This is a SaaS / AI product landing page. Gamma's SaaS landing page
contains a complete structure with a hero slogan, CTA, feature cards,
customer logos, and testimonials. Its visual design uses soft
gradients, refined illustrations, and polished typography, with a
distinctly premium product presentation. The reference is a
3840x13606 full-page screenshot. Implement it for a desktop viewport
width of 1440 pixels, corresponding to a total page height of
approximately 5102 pixels.

Replication requirements:
- Implement **every section from top to bottom**; do not deliver only
the first viewport. Section order, grid column counts, and alignment
must match the reference.
- Reproduce the color scheme, font style, rounded corners and
shadows, and the shapes of buttons and cards as closely as possible.
- Copy the text in the reference image; do not replace it with Lorem
ipsum or placeholder words.
- Images may be replaced with placeholder blocks of the same
dimensions and similar colors, or accessible image URLs, but there
must be no blank regions or broken images.
```

Prompt 8: Long-Page Replication Example (continued)

```
The technology stack is fixed to React + Vite + TypeScript, with pnpm
as the package manager. Build the project directly under
/workspace/repo, and ensure that `pnpm dev` serves it at
http://127.0.0.1:5173/ .
Do not use Playwright / Puppeteer / a Chromium installation of your
own - they cannot run in this container. To inspect the page you have
built, take a screenshot with `webshot`, then use the Read tool to
view /tmp/webshot/render.png and continue refining the page against
it. A textLength value of 0 in /tmp/webshot/measurement.json means
that the page has not mounted at all.
Before delivery, perform at least one self-check: inspect a
screenshot and confirm that there are no blank regions, horizontal
overflow, broken images, or unfinished placeholders such as
placeholder / TODO / "coming soon".
```

Short-Page Replication Example. A concrete task message from the short-page replication family, retaining the supplied reference and project requirements.

Prompt 9: Short-Page Replication Example

```
Reproduce the webpage screenshot at
/workspace/input/assets/8b01329d298dbc3863be36b8f289adb5.jpg with
high fidelity, as a runnable frontend project.

This is an education / course page. It is a sales platform for
embroidery courses and digital products. It uses soft warm colors,
refined photography, generous whitespace, and elegant serif
typography to convey the premium character of a boutique craft brand.
The reference is a 3840x9866 full-page screenshot. Implement it for a
desktop viewport width of 1440 pixels, corresponding to a total page
height of approximately 3700 pixels.

Replication requirements:
- Implement **every section from top to bottom**; do not deliver only
the first viewport. Section order, grid column counts, and alignment
must match the reference.
- Reproduce the color scheme, font style, rounded corners and
shadows, and the shapes of buttons and cards as closely as possible.
- Copy the text in the reference image; do not replace it with Lorem
ipsum or placeholder words.
- Images may be replaced with placeholder blocks of the same
dimensions and similar colors, or accessible image URLs, but there
must be no blank regions or broken images.

The technology stack is fixed to React + Vite + TypeScript, with pnpm
as the package manager. Build the project directly under
/workspace/repo, and ensure that `pnpm dev` serves it at
http://127.0.0.1:5173/ .
Do not use Playwright / Puppeteer / a Chromium installation of your
own - they cannot run in this container. To inspect the page you have
built, take a screenshot with `webshot`, then use the Read tool to
view /tmp/webshot/render.png and continue refining the page against
it. A textLength value of 0 in /tmp/webshot/measurement.json means
that the page has not mounted at all.
Before delivery, perform at least one self-check: inspect a
screenshot and confirm that there are no blank regions, horizontal
overflow, broken images, or unfinished placeholders such as
placeholder / TODO / "coming soon".
```

Text-to-Project Example. A concrete generation task with textual requirements and no reference image.

Prompt 10: Text-to-Project Example

```
The following is a product requirements document (PRD). Implement the
frontend page for "FlowTide" from scratch according to it, as a
runnable frontend project.

[Product background]
"FlowTide" is a project-collaboration SaaS product for teams of
10-200 people and is about to enter public beta. The marketing team
needs a landing page to receive paid-ad traffic and convert visitors
into free-trial registrations.

[Requirements] (5 in total; checked individually during acceptance)
1. On opening the page, the first viewport displays the product name
"FlowTide", a one-sentence value proposition, and a prominent "Start
a 14-day free trial" button.
2. The feature highlights section presents the following six
features, each as a card: task boards, Gantt charts, collaborative
documents, automated reminders, time tracking, and access management.
Each card contains an icon, a title, and a description of no more
than two lines.
3. The "Frequently Asked Questions" section contains at least four
questions and answers. Initially only the question headings are
visible. Clicking a question expands its answer; clicking it again
collapses the answer.
4. Above the footer, provide an email subscription field with an
email input and a "Subscribe" button. Submitting an invalid address
(such as abc) displays a red message below the input: "Please enter a
valid email address". Submitting a valid address displays
"Subscription successful. Please check your confirmation email" in
the same location.
5. The entire page has a bright business style with indigo as the
primary color and a white background. Primary buttons, icons, and
links share the same primary color; the hero background has a soft
gradient; cards have rounded corners and subtle shadows.

Content and delivery requirements:
- Every requirement will be checked individually. Implement all
specified visible content and interactive behavior; each missing
requirement counts as one failed requirement.
- Write real product-specific copy. Names, prices, dates, and example
data must be reasonable and credible. Lorem ipsum or filler such as
"Card One / Card Two" is unacceptable.
- Interactions mentioned in the requirements (clicking, switching,
filtering, validation, and submission) must actually work and produce
the specified feedback; do not implement them as static styling.
- Responsive behavior: present a complete desktop layout at 1440
pixels wide, with no horizontal overflow on narrow screens.

The technology stack is fixed to React + Vite + TypeScript, with pnpm
as the package manager. Build the project directly under
/workspace/repo, and ensure that `pnpm dev` serves it at
http://127.0.0.1:5173/ .
Do not use Playwright / Puppeteer / a Chromium installation of your
own - they cannot run in this container. To inspect the page you have
built, take a screenshot with `webshot`, then use the Read tool to
view /tmp/webshot/render.png and continue refining the page against
it. A textLength value of 0 in /tmp/webshot/measurement.json means
that the page has not mounted at all.
```

Prompt 10: Text-to-Project Example (continued)

```
Before delivery, perform at least one self-check: inspect a
screenshot and confirm that there are no blank regions, horizontal
overflow, broken images, or unfinished placeholders such as
placeholder / TODO / "coming soon".
```

Fidelity Judge. Input: reference image and rendered image. Output: anchored grades and reasons for reconstruction quality across the specified dimensions.

Prompt 11: Fidelity Judge

```
You are an expert evaluator of frontend webpage quality. Score
strictly from the supplied screenshots; do not speculate about
content outside them.
Task type: reproduction.screenshot

Input: two images. The first is [REFERENCE], the reference design;
the second is [RENDERED], the rendered reconstruction.

Rubrics to score:
  - visual_fidelity (visual fidelity): whether the overall appearance
matches the reference, including palette, font style, component
shapes, corner radii and shadows, and the positions and proportions
of images and illustrations.
  - content_coverage (content coverage): whether every section in the
reference has been reconstructed. If only the first viewport or hero
is implemented and entire later sections are missing, assign poor or
broken regardless of how polished the completed portion is.
  - layout (layout structure): whether section order, grid column
counts, alignment, and inter-section spacing rhythm match the
reference.
  - text_match (text consistency): whether key copy, headings,
navigation labels, and button text match the reference. Different
line breaks are allowed; replacing text with Lorem ipsum or
placeholders is not.
  - runtime_visual_health (rendering health): blank pages, large
solid-color areas without content, overlapping or misaligned
elements, horizontal overflow, broken images, visible error messages,
or unfinished markers such as 'placeholder', 'TODO', or 'coming
soon'. This dimension assesses **only whether rendering is normal**.
Do not penalize mismatches with reference content or requirements
here; assess those under the other dimensions.

Scoring criteria:
For each rubric, assess the **fidelity** of [RENDERED] relative to
[REFERENCE] and choose one of the following five levels (English
lowercase only):
  excellent = Highly faithful, almost identical to the reference.
  good      = Good fidelity, with only minor differences.
  fair      = Broadly faithful, with one noticeable deviation.
  poor      = Low fidelity, with multiple obvious deviations.
  broken    = Almost no match / blank / error / completely distorted.
The reference may be a very long full-page screenshot. Compare
sections from top to bottom rather than judging only the first
viewport. A page that stops after reconstructing the hero must
receive poor or broken for content_coverage, even if the completed
portion is polished.
```

Prompt 11: Fidelity Judge (continued)

```
Mandatory rule: if the evaluated page is essentially blank, a solid
color without content, or contains only placeholder/skeleton text
(such as 'placeholder', 'skeleton', 'sections pending', 'TODO', or
'Lorem ipsum'), assign broken to **all** rubrics. Do not award
intermediate levels merely because the colors or page height are
similar.
Provide a judgment for each rubric above. Reason internally first,
then output **only one JSON object**, with no additional text.
Output JSON format:
{
  "scores": [
    {"rubric": "<visual_fidelity|content_coverage|layout|text_match|
runtime_visual_health>", "level": "<one of
excellent/good/fair/poor/broken>", "reason": "<reason in no more than
30 characters>"}
  ],
  "overall_reason": "<one-sentence overall judgment>"
}
```

Absolute Judge. Input: rendered image and task text when available. Output: anchored grades and reasons for design, content, visually inferred functionality, and rendering health.

Prompt 12: Absolute Judge

```
You are an expert evaluator of frontend webpage quality. Score
strictly from the supplied screenshot; do not speculate about content
outside it.
Task type: generation.text_to_project

Input: one image = [AFTER], the current completed page.

Rubrics to score:
  - design_quality (design quality): visual hierarchy, whether the
palette matches the requested tone, typographic alignment, spacing
rhythm, font pairing, and overall aesthetics.
  - content_quality (content quality): whether the copy is credible,
substantive, and relevant to the topic. If the page consists of Lorem
ipsum, repeated placeholder text, or filler such as 'Card 1 / Card 2
/ Card 3', assign poor or broken for this dimension.
  - functionality (functional completeness, visual cues): usability
inferable from the screenshot; whether state-switching controls,
tables, forms, and interactive controls are fully rendered rather
than unfinished.
  - runtime_visual_health (rendering health): blank pages, large
solid-color areas without content, overlapping or misaligned
elements, horizontal overflow, broken images, or visible error
messages. This dimension assesses **only whether rendering is
normal**. Do not penalize mismatches with reference content or
requirements here; assess those under the other dimensions.

Scoring criteria:
For each rubric, choose one of the following five levels (English
lowercase only):
  excellent = Meets the requirements and reaches production-ready
commercial quality in this dimension: you can explain specifically
what makes it good and **cannot identify any possible improvement**.
  good      = Fully meets the requirements, but you can identify 1-2
improvements in spacing, alignment, palette, information density, or
copy.
```

Prompt 12: Absolute Judge (continued)

```
  fair      = Mostly meets the requirements, with one noticeable
issue.
  poor      = Mostly fails to meet the requirements, with multiple
obvious issues.
  broken    = Severe failure / blank / error / completely
inappropriate.

**Calibration rules (important for score usability):**
The default level is good. Implementing everything requested is worth
only good: meeting requirements is the passing threshold, not a
perfect score. Assign excellent only when the quality in that
dimension clearly exceeds merely completing the task correctly.
Before assigning excellent, ask yourself whether you can suggest even
one specific improvement. If you can, assign good.
Do not assign the same level to every dimension merely because the
overall impression is positive. A page usually performs differently
across dimensions. Judge each dimension independently; different
levels are allowed and encouraged.

Mandatory rule: if the evaluated page is essentially blank, a solid
color without content, or contains only placeholder/skeleton text
(such as 'placeholder', 'skeleton', 'sections pending', 'TODO', or
'Lorem ipsum'), assign broken to **all** rubrics. Do not award
intermediate levels merely because the colors or page height are
similar.
Provide a judgment for each rubric above. Reason internally first,
then output **only one JSON object**, with no additional text.
Output JSON format:
{
  "scores": [
    {"rubric": "<design_quality|content_quality|functionality|runtim⌋
e_visual_health>", "level": "<one of
excellent/good/fair/poor/broken>", "reason": "<reason in no more than
30 characters>"}
  ],
  "overall_reason": "<one-sentence overall judgment>"
}
```

Reference-Image Screening. Input: candidate reference screenshot. Output: a suitability decision for use as a reconstruction target.

Prompt 13: Reference-Image Screening

```
This is a webpage-design screenshot. Determine whether it is suitable
as a reference target for a high-fidelity replication task: the main
page content must be clearly and fully visible. Reject it if
cookie/privacy pop-ups obscure the main content, the entire page is
darkened by an overlay, the page is blank, it is an error page, or
loading is visibly incomplete. On the final line, output only usable
or unusable.
```

Task Authoring. Input: task seed or blueprint and the available reference material. Output: development requirements and an associated acceptance checklist. Supplied revision annotations are retained.

Prompt 14: Task Authoring

```
You are an expert task author for a frontend evaluation benchmark.
Given a blueprint specification, create a task consisting of a
development request and a hidden acceptance checklist.

[Mandatory rules]
1. Write the development request in Chinese. Describe "what is
needed" in the voice of a real product manager. Never prescribe an
implementation (do not mention React, components, or CSS class
names).
2. Each requirement corresponds to an operator specified by the
blueprint. Requirements must form a realistic progression, with later
requirements building on the artifacts of earlier ones.
3. Provide 3-6 checklist items per requirement. Each item contains: m
(requirement index, starting at 1) / dimension (coverage | visual |
functionality | interaction) / w (weight; default 1.0, core items
1.5) / description.
4. Items must be behavioral assertions that can be judged solely by
observing the running application. Implementation-specific assertions
and ambiguity are prohibited.
5. [Difficulty requirements - critical] Apply the blueprint's levers,
and include at least one "exact-value assertion or combined-state
assertion" per requirement:
   - Exact numerical assertions: specify concrete numbers and
calculation relationships (e.g., "annual price = monthly price x 12 x
85%, rounded down"; the checklist verifies the exact number).
   - State coupling: cross-assertions about state shared between
requirements (e.g., "when a filter is active, hidden products cannot
be added to the cart and the badge count remains unchanged").
   - Boundary and exceptional paths: empty states, idempotence under
rapid repeated clicks, deletion down to zero items, and very long
text.
   - Data scale x combined states: at least 20 records, with diagonal
spot checks across combinations of filtering, sorting, and
pagination.
   - Nonlocal changes: theme-change or refactoring requirements must
explicitly overlap with styles or state established by earlier
requirements, with assertions that the overlapping behavior is
preserved.
5b. [Hardening rules - effective after calibration on 2026-09-13,
addressing the ceiling where advanced models score 0.98 across the
set]
   - Composite tasks must contain 4-5 requirements, at least 40 data
records, at least 10 interactive stateful elements, and at least 4
cross-requirement coupling assertions, expanding the scope that must
be fully correct.
   - [Reduce explicitness - most important] Phrase requests naturally
as a product manager would, rather than as an exam: supply data in
prose or an untidy operations list, not a neatly formatted table;
express calculations colloquially ("charge the whole year at 85% of
the monthly rate and drop the remainder", rather than a formula).
Some commitments should be implicit ("do not mess up the previous
version's style", "just make sure the numbers add up"), with their
exact meaning checked by the checklist. Understanding requirements
and routing them to the appropriate actions should itself be
evaluated.
   - Changes to existing-project tasks must touch shared state or
global styles already present in the repository; purely additive,
isolated sections are not allowed.
   - Use only these hardening methods. Pixel-level nitpicking,
brainteasers, and assertions that depend on coincidences remain
prohibited. The difficulty ceiling remains "a careful human frontend
developer can complete it correctly in two hours".
```

Prompt 14: Task Authoring (continued)

```
5c. [Second hardening round - effective after calibration of batch 1
on 2026-09-13; the batch mean of 0.963 remained too high; empirically
effective failure points were repairs, data counts, and preservation
of legacy layouts]
   - [No free-point items] Every checklist item must bind to a
precise assertion that can fail (a concrete value, order, state, or
boundary). Pure existence descriptions ("there is a navigation bar")
must not be standalone items; existence may only be a prerequisite
clause of a precise assertion.
   - [Double the emphasis on repairs] For existing-project composite
tasks, choose injected defects that are difficult to detect:
prioritize asynchronous races, conditional boundaries, and missed
state updates over detached event handlers. Each repair requirement
must have at least 4 checklist items, including at least 2 regression
assertions that the repair does not damage neighboring functionality.
   - [Multi-step interaction chains] Every composite task must
contain at least one requirement with a stateful interaction chain of
at least 3 steps (a wizard, shopping-cart checkout, or an operation
after multilevel filtering). The checklist verifies both intermediate
and final states of the complete chain.
   - [Implicit trade-offs] Every task must contain at least one
apparent tension between two requirements (e.g., "make it prominent"
versus "preserve a minimalist style"). The agent must make a
reasonable trade-off and apply it consistently throughout the site.
The checklist evaluates consistency: any reasonable solution receives
credit, but inconsistent decisions receive a lower grade.
   - From batch 2 onward, all composite tasks must be at the
challenge level.
6. Difficulty must be reasonable: assertions should be at a level
that "a careful human frontend developer can get right within two
hours". Pixel-level nitpicking, brainteasers, and tricky assertions
that depend on coincidences are prohibited.
7. For challenge-level tasks, at least 2 combined-state assertions
must span multiple requirements.
8. If the blueprint includes a repair operator, design a hidden
defect matching the injection type. In the corresponding development
request, describe only its symptoms in the user's voice (never name
files or code). Also provide a saboteur_prompt instructing an
injection agent to introduce that precise defect, with no console
errors, no changes to unrelated functionality or styling, and no
comments left behind.
9. Replication seed: requirement 1 is high-fidelity replication of
the supplied reference image, limited to its visible region; images
may use placeholders. Visual checklist items must refer to specific
visible elements, colors, or layout features in the reference. [v3.2
strict rule] Replication requirements must **preserve the reference
image's original text and content**. Do not request "replace it with
our business copy": the VLM visual judge scores text consistency, so
replacing the copy makes the visual score structurally invalid.
Business-specific adaptations must appear only as later, independent
requirement operators.
10. PRD seed: requirement 1 is to build a site from scratch according
to a PRD. Write the PRD content into request, including concrete
copy, data, and structure.
11. Existing-project seed: the task begins with "modify the existing
repository". Requirements start directly with modification operators.
You will receive the repository file tree, source excerpts, and
screenshots.

[Output] Output only one JSON object, without Markdown fences:
```

Prompt 14: Task Authoring (continued)

```
{"title": "...", "milestones": [{"operator": "...", "request":
"..."}...],
 "checklist": [{"id":"<short English dot-separated
id>","m":1,"dimension":"coverage","w":1.0,"description":"..."}...],
 "inject": {"type":"...","saboteur_prompt":"..."} or null}
```

Independent Task Review. Input: a proposed task and checklist. Output: a review decision and actionable issues, including checks on observability and requirement traceability.

Prompt 15: Independent Task Review

```
You are an independent reviewer of an evaluation benchmark. Review
the supplied task (development request + checklist + blueprint
specification) item by item and output JSON.

[Requirement-level rubric]
1. Reasonableness: compatible with the seed, internally consistent,
and free of implementation prescriptions.
2. Verifiability: user-observable acceptance criteria are provided.
3. Nontrivial difficulty: supported by at least 2 interaction or
state assertions; purely textual or purely static requirements are
unacceptable.
4. Reasonable difficulty: assertions are not tricky (no pixel-level
nitpicking or brainteasers), and a careful human frontend developer
can satisfy them.
[Item-level rubric]
1. Observability; 2. No implementation dependence; 3. No ambiguity;
4. Four-grade distinguishability (granularity sufficient to
distinguish fully / mostly / partially / not implemented); 5.
Traceability to the original requirement.
[Coverage checks]
Every observable commitment in each requirement has a checklist item.
All interaction points are covered. Every requirement contains at
least one exact-value or combined-state assertion. Challenge-level
tasks contain at least two cross-requirement combined-state
assertions.
[Repair tasks]
The symptom description does not name files. The saboteur_prompt is
executable; the defect can be detected and repaired, and produces no
errors.

[Output] Output only JSON, without fences:
{"verdict":"pass" or "revise","issues":[{"target":"requirement N or
item id","problem":"...","fix":"..."}]}
Prefer requesting revision to being lenient. Allow pass only when
issues is empty.
```

Difficulty Revision. Input: an existing task. Output: a revised task with stronger state coupling or boundary cases, subject to the stated consistency constraints.

Prompt 16: Difficulty Revision

```
You are an expert in increasing the difficulty of benchmark tasks.
Given a task on which an advanced model scored too highly, make it
harder without changing its seed, operator, or domain.

[Only the following hardening methods are allowed]
- State coupling: make cross-requirement assertions about shared
state concrete, including consistency and mutual exclusion.
```

Prompt 16: Difficulty Revision (continued)

```
- Exact numerical assertions: provide specific numbers and
calculation relationships, and verify exact values in the checklist.
- Combined-state spot checks: with at least 20 data records, perform
diagonal spot checks across combinations of filtering, sorting, and
pagination.
- Boundary and exceptional paths: empty states, idempotence under
rapid repeated clicks, deletion down to zero, and very long content.
[Strictly prohibited] Pixel-level nitpicking, brainteasers,
assertions based on coincidences or unstated conventions, and making
requirements vague to penalize interpretation.
The revised task must remain something that "a careful human frontend
developer can complete correctly in two hours". The goal is to
differentiate performance, not to make everyone fail.

[Output] Return the complete revised task JSON, with the same schema
as the input: title/milestones/checklist/inject.
```

Controlled Defect Injection Example. A task-specific example for introducing a constrained defect. It is an example instance, not a universal injection template.

Prompt 17: Controlled Defect Injection Example

```
Introduce a stale-closure asynchronous race into the
carousel-switching logic in src/components/Hero.jsx, making it look
like a delay added for a slide transition. Defer the state update
after an arrow click into a setTimeout of approximately 180ms. At
click time, compute the target slide using the index variable
captured by the current render closure, in the form const next =
(index + dir + SLIDES.length) % SLIDES.length; setTimeout(() =>
setIndex(next), 180). Do not use a functional state update, and do
not clear or cancel the previous timer. The required effect is that a
single click behaves normally, with only a slight perceived delay,
while 2-3 rapid clicks within 180ms calculate their targets from the
same old index, so the carousel advances or retreats by only one
slide, or appears to move in the wrong direction. Mandatory
constraints: do not introduce any console errors or React warnings;
do not change any CSS, class names, DOM structure, copy, or SLIDES
data; do not touch other components such as GalleryStrip,
NewsCarousel, Header, or Footer; leave no comments, TODOs, or console
output; match the existing naming and indentation style.
```

Builder Prompt. Recipient: model under evaluation. Input: requirements and starting-project context. Output: a runnable implementation following the startup and delivery contract.

Prompt 18: Builder Prompt

```
You are a frontend engineer assigned a development request containing
multiple requirements. Complete all requirements in the current
repository.

[Development request]
{requests}
{seed_hint}
[Delivery requirements]
1. The application must run: maintain ./start.sh to launch the
service in the foreground at http://localhost:3000 . Prefer a
static/development server solution and avoid installing system-level
dependencies.
```

Prompt 18: Builder Prompt (continued)

```
2. Verify the application yourself before delivery: start it, use
playwright-cli screenshots to confirm correct rendering and the
absence of console errors, and actually exercise requirements
involving interactions. After verification, stop the service (fuser
-k 3000/tcp).
3. When everything is complete, output one line: ALL_DONE:
<one-sentence summary>

[Rules] Acceptance evaluates the running application. Only
user-observable behavior counts. Do not write test files to game
acceptance. Do not read or write the .qa directory.
```

Acceptance Prompt. Recipient: acceptance agent. Input: the running application and acceptance checklist. Output: item-wise grades and evidence from browser observations and operations.

Prompt 19: Acceptance Prompt

```
You are a strict QA acceptance engineer. The application under
evaluation is running at http://localhost:3000 .

[Part 1 - Graded acceptance of the checklist for each requirement]
The following acceptance checklist is grouped by major requirement
(JSON):
{checklist}
Check every item using actual playwright-cli interactions and
observations (open/click/fill/screenshot). Each item must have at
least one screenshot or page observation as evidence. Interaction
items must be exercised in practice. For exact-value or
combined-state items, construct the relevant state and verify the
value.

Assign each item a discrete grade from 0 to 3 (anchored criteria;
apply strictly):
- 3 = Fully satisfied: observable behavior exactly matches the item,
with no visible defects.
- 2 = Mostly satisfied: the core behavior is correct, but minor
defects remain in copy, styling, or edge-state details.
- 1 = Partially satisfied: only part of the item is implemented, or
behavior is unstable and succeeds inconsistently.
- 0 = Not implemented or unusable: the required behavior is absent,
ineffective, or produces an error.
A grade of 3 does not require exceeding expectations; satisfying the
item is enough. Every downgrade must include concrete evidence. A
button that exists but does nothing when clicked cannot receive a
high grade.

[Part 2 - Archive a full-page screenshot]
After acceptance, return to the homepage's initial state and save a
full-page screenshot to /workspace/.qa/screens/full_rep{rep}.png .
The directory already exists; use the full-page mode of
playwright-cli screenshot.

[Rules] Reading the evaluated repository's source code is strictly
prohibited. Modifying any files outside .qa is strictly prohibited.
Judge everything from the running application.
[Output] Output only one JSON object, without Markdown fences:
{{"items": [{{"id": "...", "grade": 0-3, "evidence": "..."}}, ...]}}
```

Table 10: Overview of the skills available to the training agent. Names are shortened for readability.

Skill	Role
Screenshot replication	
Pipeline coordination	Coordinates planning, implementation, and validation.
Input analysis	Interprets reference inputs and identifies the task mode.
Long-image planning	Divides tall screenshots into semantic sections.
Flow planning	Models multiple reference views as application states and transitions.
Asset processing	Prepares visual assets and selects their representations.
Code generation	Implements and integrates the planned frontend.
Validation and repair	Checks build and rendering results and repairs defects.
Text-only generation	
Text-to-project	Guides requirement analysis, implementation, and validation.

J.2 On-Demand Skills

The training agent has access to task-specific skills. Screenshot replication is supported by seven skills covering input analysis, planning, asset processing, implementation, and validation, while text-only generation uses a dedicated text-to-project skill. Table 10 summarizes their roles.