FrontCoder: Scaling Visual Fidelity in Front-End Code Generation

Lean Feng[1], **Jian Yang**[1*], **Wei Zhang**[1], **Jing Wang**[1], **Keyi Chen**[1], **Xiaokun Yang**[1], **Weicheng Gu**[1], **Yihang Lou**[2], **Yan Bai**[2], **Xianglong Liu**[1]
[1]Beihang University; [2]Huawei
leanfeng1@gmail.com, jiayang@buaa.edu.cn

Abstract

Large language models (LLMs) for code generation have achieved remarkable progress in synthesizing functional code from natural language instructions. However, a critical challenge persists in generating visually accurate and structurally sound front-end code that faithfully renders user-intended layouts and interfaces. Most existing works focus primarily on functional correctness, overlooking the visual fidelity and rendering quality essential for front-end development. To address this gap, we present a comprehensive data construction and training pipeline to enhance front-end code generation capabilities in code LLMs. We use a three-stage training approach: continual pre-training on synthetic data, quality-controlled supervised fine-tuning, and reinforcement learning with checklist-based rewards to improve model performance. Our comprehensive evaluation on front-end code generation benchmarks reveals that even strong base models struggle with visual faithfulness and layout complexity. Our fully-trained model demonstrated substantial improvements over baseline approaches across all domains, achieving competitive performance with frontier models while maintaining generation efficiency, underscoring the critical importance of stage-aligned data curation and vision-grounded optimization in developing reliable front-end code generation systems. Our code and data are open-sourced at https://github.com/leanfeng1/FrontCoder.

1 Introduction

Front-end code generation has emerged as a critical capability for modern code large language models (code LLMs) (Hui et al., 2024; Guo et al., 2024; Rozière et al., 2023; Anthropic, 2025; OpenAI, 2025; Yang et al., 2025c), enabling automatic synthesis of HTML, CSS, and JavaScript from natural language descriptions of user interfaces. While

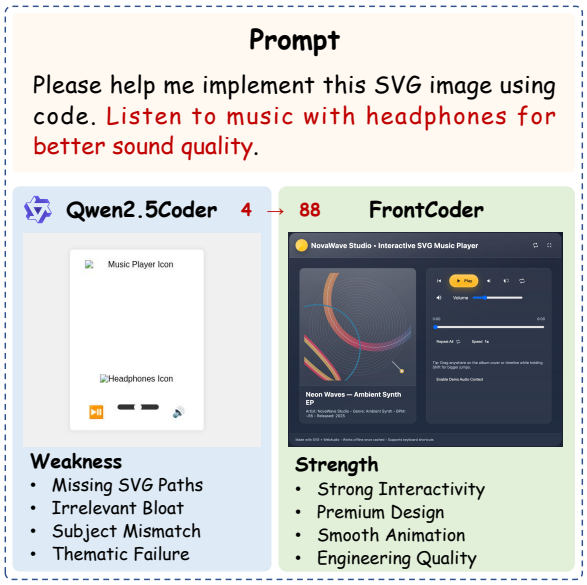

Figure 1: Comparison between Qwen2.5-Coder and our FrontCoder. While both follow the interactive SVG prompt, FrontCoder produces a more sophisticated, fully functional sandbox with precise UI controls.

recent models generate syntactically correct code, a fundamental challenge remains: ensuring front-end code produces rendered outputs that accurately match the intended layout, styling, and interactive behavior specified in user instructions.

Visual faithfulness in front-end code generation presents unique challenges, as quality depends on rendered appearance rather than logical correctness verifiable through unit tests. While benchmarks like Design2Code(Si et al., 2025) and Artifacts-Bench (Zhang et al., 2025b) employ VLM judges for visual evaluation, state-of-the-art models still struggle with complex layouts, nested DOM structures, and generating excessive boilerplate while missing critical visual requirements. Current training approaches face two key limitations: (1) pre-training corpora lack sufficient coverage of diverse compositional UI patterns (flexbox, grids, SVG motifs) and domain-specific structures (dashboards, games, visualizations), and (2) standard supervised

* Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 4485–4505
July 2-7, 2026

fine-tuning cannot effectively address behavioral failures like rendering errors and degenerative repetition that only manifest at generation time and require execution-grounded feedback to correct.

In this paper, we introduce **FrontCoder**, achieving high visual faithfulness in front-end code generation through a three-stage pipeline: continual pre-training (CPT), supervised fine-tuning (SFT), and reinforcement learning (RL). The front-end code generation requires explicit separation of concerns, where CPT instills broad *structural priors* for HTML/CSS/SVG composition, SFT establishes *instruction alignment* through high-quality executable demonstrations, and RL refines *behavioral robustness* by directly optimizing for rendered visual fidelity. We create an automated data pipeline yielding 75B tokens for CPT, 60K length-controlled high-quality instruction-code pairs for SFT, and 1524 hard prompts with fine-grained verification checklists for RL. For front-end code RL, we define a composite reward that combines hard binary gates for rendering validity and repetition avoidance with soft scores for checklist satisfaction, structural-semantic similarity to reference responses, and length regularization. We integrate a sandboxed rendering environment that executes generated HTML and captures screenshots, enabling vision-grounded reward computation that aligns optimization with actual rendered appearance rather than superficial text correctness. We further distill visual requirements into discrete 20-item verification checklists, providing interpretable and stable reward specifications that decompose complex layout constraints into verifiable criteria. We conduct extensive evaluation on ArtifactsBench (Zhang et al., 2025b), a comprehensive benchmark containing 1,825 front-end tasks across five domains (GAME, SVG, WEB, Simulation, Management Systems).

Our analysis reveals several key findings:

- **CPT Effectiveness**: Continual pre-training on our curated 75B-token corpus substantially improves structural competence, boosting AVG score on Qwen2.5-Coder-7B base model, with particularly large gains on structure-intensive domains like SVG and simulation.

- **SFT Delivers Largest Alignment Gain**: High-quality SFT on our length-controlled 60K dataset produces the largest single-stage improvement, with consistent gains across all domains demonstrating the critical importance of supervision fidelity over raw volume.

- **RL as Behavior Regularizer**: Reinforcement learning provides targeted improvements while substantially reducing output length—our SFT+RL model decreases average inference length from 35.8K to 25.8K (28% reduction), effectively eliminating rendering failures and degenerative repetition.

- **Analysis**: This research demonstrates that a data- and reward-focused training approach can achieve competitive performance even with smaller models, with their best 7B parameter model nearly matching GPT-5's performance and surpassing MiniMax-M2, while their 30B model achieves the highest score among open models. The study also reveals that reinforcement learning (RL) affects models differently depending on their initial training quality: for weaker models, RL adds essential structural improvements, whereas for already strong models from supervised fine-tuning (SFT), RL primarily serves to regulate response length and enhance robustness.

2 FrontCoder

2.1 Data Preparation and Construction

We build an automated data generation pipeline for three stages, including continual pre-training (CPT), supervised fine-tuning (SFT), and reinforcement learning (RL). We denote the overall corpus as $\mathcal{D} = \mathcal{D}_C \cup \mathcal{D}_S \cup \mathcal{D}_R$, where $\mathcal{D}_C, \mathcal{D}_S, \mathcal{D}_R$ denote the CPT, SFT, and RL corpus. Our goal aims at preserving visual faithfulness by enforcing renderable and structurally consistent HTML/CSS, and maintaining coverage and diversity to support scaling in front-end code generation and UI layout synthesis.

Teacher-model selection rationale. Our use of MiniMax-M2, GPT-5, and Gemini-3-pro-preview in data construction is based on empirical quality. In addition to their standings on ArtifactsBench, we evaluate candidate teacher models on a custom 50-sample out-of-distribution set using both automated scoring and human review over visual fidelity, structural correctness, and functional completeness. Across these checks, these three models consistently produced higher-quality front-end implementations with better layout coherence and feature completeness, which motivated their respective roles in SFT generation, reference construction,

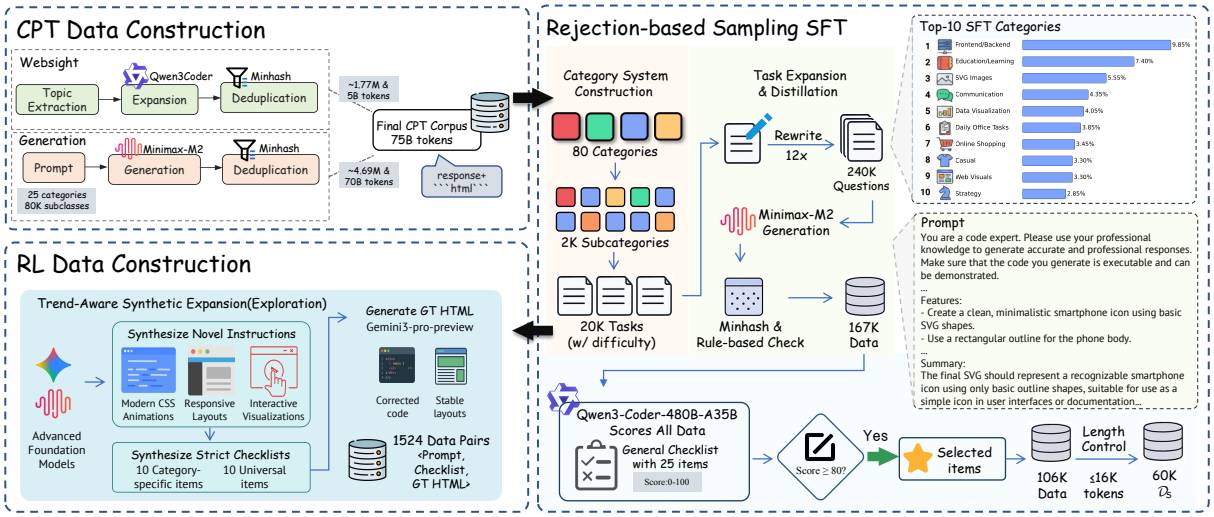

Figure 2: Overview of our three-stage data construction pipeline for front-end code generation. We construct $\mathcal{D}$ via: (i) **CPT**, combining WebSight rewriting and large-scale synthetic generation, then deduplicating and applying regex/rule-based corpus sanitation to obtain the final $\mathcal{D}_C$ corpus (75B tokens); (ii) **SFT**, using rejection-based sampling (MinHash near-duplicate removal, rule-based validation, and Qwen3-Coder-480B-A35B semantic scoring with a strict $S \geq 80$ threshold) to obtain a length-controlled set of 60K high-quality instruction-code pairs $\mathcal{D}_S$; and (iii) **RL**, using trend-aware synthetic expansion to synthesize novel instructions and strict 20-item checklists, generating 1,524 high-fidelity data triplets for structured reward specification $\mathcal{D}_R$.

and checklist verification. Our teacher-model selection criteria and a score-normalized comparison derived from the main benchmark scores are summarized in Appendix 5.

2.1.1 CPT Data Construction

CPT is used to inject structural priors for HTML-centric layout composition (HTML/CSS/SVG and common web UI idioms), while controlling redundancy at scale. $\mathcal{D}_C = \left(\mathcal{D}_{\text{web}} \cup \mathcal{D}_{\text{syn}}\right)$, where $\mathcal{D}_{\text{web}}$ is a rewritten corpus from WebSight (Laurençon et al., 2024) and $\mathcal{D}_{\text{syn}}$ is a large synthetic corpus.

WebSight rewriting. We start from WebSight-v0.2 (capped at 1.92M entries) and use Qwen3-Coder to rewrite each sample to normalize formatting and enrich structural diversity, including layout patterns and component composition. After rewriting and normalization, this branch contributes approximately 5B tokens, denoted as $\mathcal{D}_{\text{web}}$.

Large-scale synthetic generation. We employ MiniMax-M2 to generate a synthetic corpus expanding coverage across diverse UI scenarios (e.g., dashboards, forms, charts, component libraries) and rendering styles (vanilla HTML/CSS, SVG). After normalizing to a consistent schema and filtering degenerate templates, this branch contributes approximately 70B tokens, denoted as $\mathcal{D}_{\text{syn}}$.

2.1.2 SFT Data Construction

SFT targets high-fidelity instruction following with executable and visually consistent code. We define a comprehensive task set consisting of 80 major categories and 20K concrete task definitions. For each task, we employ MiniMax-M2 to generate one canonical prompt and twelve rewritten variants, resulting in a raw corpus of 240K samples.

To ensure data quality, we implement a three-stage refinement pipeline. First, we use MinHash for near-duplicate removal, reducing the corpus to 189K samples. Second, a rule-based validation step filters out syntax and structural errors, retaining 167K samples. Finally, we perform model-based semantic scoring using Qwen3-Coder-480B-A35B-Instruct as a judge. Each sample is evaluated against a 25-criterion rubric covering five dimensions, including code quality and functional completeness. We apply a strict quality threshold and only retain samples with a weighted score of 80.0 or higher. This process yields a final high-quality set of 106K instruction-code pairs, denoted as $\mathcal{D}_S$.

To prevent over-generation and optimize training efficiency, we constrain the sequence length of the 106K set. Given that some instruction-code pairs reach 32K tokens, we use the Qwen2.5-Coder-7B-Instruct tokenizer to filter samples to a maximum total length of 16K tokens. This results in a final subset of 60K samples, denoted as $\mathcal{D}_S$. Unless oth-

erwise specified, all SFT models in our experiments are trained on this subset.

Semantic scoring rubric. The model-based filtering score is computed from a 25-criterion rubric spanning five dimensions: engineering quality and maintainability, functional completeness, reliability and security, experience and visual quality, technical depth and innovation, and requirement alignment and solution quality. The criteria cover executable correctness, code standards, redundancy, boundary handling, interaction completeness, accessibility, design professionalism, visual fidelity, performance optimization, and core function completeness. We provide the full criterion list in the appendix to make the filtering procedure explicit and reproducible.

2.1.3 RL Data Construction

RL focuses on long-tail failures and domains where the SFT model remains brittle. We curate a challenging prompt set, consisting of 1524 prompts with increased structural complexity and error prevalence, such as deeply nested GAME layouts and dense management dashboards. For each prompt in this set, we employ GPT-5 to produce a high-quality reference response and use MiniMax-M2 to generate a fine-grained 20-item checklist that specifies verifiable layout and functional requirements. This process forms the RL corpus $\mathcal{D}_R$, where each entry includes the prompt, reference response, and corresponding checklist. The checklist provides a structured reward target by decomposing visual and structural constraints into discrete, actionable criteria. To improve training stability, we utilize Gemini3-pro-preview as a verifier to audit and refine the consistency between the reference responses and their checklists, thereby reducing reward noise induced by ambiguous specifications.

2.2 Full-Process Training

We adopt a CPT-SFT-RL training approach to progressively strengthen structural competence, instruction alignment, and length-controlled, high-fidelity generation progressively. This pipeline transitions from foundational structural learning to precise instruction following and final preference optimization.

2.2.1 Continual Pre-Training

We perform CPT on the Qwen2.5-Coder-7B base model using the Llama Factory framework via autoregressive next-token prediction. This stage injects essential web-UI structural priors including HTML, SVG, and common layout items to provide a robust initialization for subsequent alignment. Our results demonstrate that applying CPT directly on the base model effectively preserves structural knowledge at scale.

2.2.2 Supervised Fine-Tuning

SFT aligns the model with explicit instructions and task demonstrations, emphasizing the generation of structured planning responses and executable code. During this stage, the model is fine-tuned on the length-controlled subset $\mathcal{D}_S$ to ensure that the internal representation captures both the logical decomposition of UI tasks and the syntax of renderable code. By balancing complexity with conciseness, this phase minimizes the risk of over-generation while significantly enhancing the model's ability to synthesize high-fidelity web components.

2.2.3 Reinforcement Learning

Reinforcement learning further refines the SFT model through behavior-level feedback to enhance robustness and efficiency. We optimize a composite reward $R(y)$ that integrates rendering validity, repetition avoidance, checklist satisfaction, similarity to reference responses, and length regularization. The reward employs two hard binary gates where $\mathbb{I}_p(y)$ indicates the absence of degenerative repetition and $\mathbb{I}_n(y)$ indicates successful sandbox rendering. The total reward is defined as:

$$R(y) = \mathbb{I}_p(y) \cdot \mathbb{I}_n(y) \cdot \big(\alpha S_{\text{chk}} + \beta S_{\text{sim}} + \gamma S_{\text{len}}\big). \quad (1)$$

where S_{chk}, S_{sim}, and S_{len} denote the checklist score, code similarity score, and length score, with weighting coefficients satisfying $\alpha + \beta + \gamma = 1$. The checklist score S_{chk} consists of 20 individual items, each valued at 5 points for a total of 100 points. The similarity score S_{sim} combines structural and semantic alignments:

$$S_{\text{sim}}(y, y^*) = \tfrac{1}{2} S_{\text{str}}(y, y^*) + \tfrac{1}{2} S_{\text{sem}}(y, y^*). \quad (2)$$

where Structural similarity S_{str} is computed by flattening the DOM tree into an inorder sequence of tag-attribute pairs Q and calculating the LCS ratio:

$$S_{\text{str}} = \frac{|\text{LCS}(Q_y, Q_{y^*})|}{\max(|Q_y|, |Q_{y^*}|)}. \quad (3)$$

For semantic similarity S_{sem}, we map HTML elements to a set of semantic roles $\mathcal{M}$ and compute the average matching score across all roles:

$$S_{\text{sem}} = \frac{1}{|\mathcal{M}|} \sum_{k \in \mathcal{M}} \text{Match}\big(E_k(y), E_k(y^*)\big). \quad (4)$$

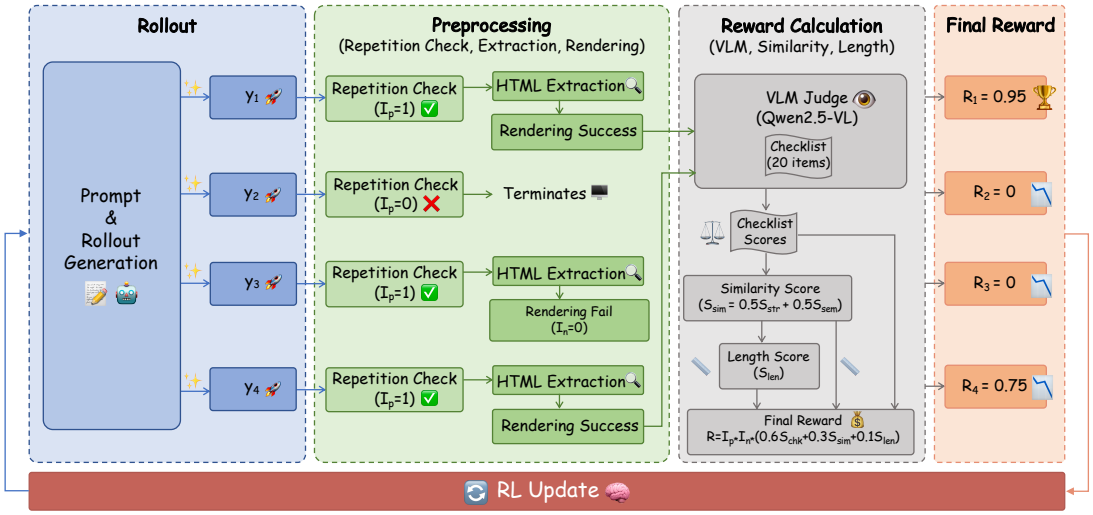

Figure 3: Overview of the RL training pipeline. The process consists of rollout generation, multi-step preprocessing (repetition and rendering checks), reward calculation using a VLM-based checklist and similarity metrics, and a final policy update based on the aggregated rewards.

To encourage conciseness, we define a length score S_{len} with thresholds $L_{\min}$ is 12K and $L_{\max}$ is 16K:

$$S_{\text{len}}(y) = \begin{cases} 1.0 & L(y) \in [0, L_{\min}] \\ \frac{L_{\max} - L(y)}{L_{\max} - L_{\min}} & L(y) \in (L_{\min}, L_{\max}) \\ 0.0 & L(y) \in [L_{\max}, \infty) \end{cases} \quad (5)$$

Finally, we optimize the policy using a KL-regularized objective:

$$J(\theta) = \mathbb{E}_{y \sim \pi_\theta(\cdot|x)} \left[R(y) - \lambda D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{sft}}) \right]. \quad (6)$$

3 Experiments

3.1 Experimental Setup

We evaluate on ArtifactsBench (Zhang et al., 2025b), which contains 1,825 front-end code generation tasks across five domains: GAME, SVG, WEB, SI (Simulation), and MS (Management Systems). We report domain scores, the global average (AVG), and the average inference length (IFLEN). IFLEN is a practical proxy for generation efficiency, measuring whether the model can produce renderable, non-redundant code without excessive boilerplate.

3.1.1 Evaluator Validity and Robustness

We follow the ArtifactsBench evaluation protocol. ArtifactsBench reports strong agreement between MLLM judges and human experts via controlled studies. Since our evaluation set is a subset of ArtifactsBench, we inherit these validity guarantees under the same protocol. To reduce self-preference effects, we decouple training-time critics from offline evaluation, using different models for RL feedback and leaderboard-style scoring.

To further test judge robustness, we randomly sample 64 prompts and generate 4 independent rollouts per prompt from the Qwen2.5-Coder-7B-Instruct-SFT-60K model, producing 256 code-screenshot pairs. We score the same set with Qwen2.5-VL-72B and Gemini-2.5-Pro under the identical checklist-based prompt. Although the two judges operate on different absolute scales, they exhibit strong rank agreement with Kendall's Tau-b of 0.717, indicating that our checklist-based evaluation yields stable relative ordering across model families rather than reflecting idiosyncratic bias from a single VLM judge.

3.1.2 Sandboxed Rendering Environment

We execute generated code in a headless Chromium-based renderer and capture full-page screenshots with dynamic resizing to match content dimensions. These screenshots provide pixel-grounded evidence for vision-based judging, ensuring optimization targets actual rendered appearance rather than superficial text correctness.

3.1.3 RL Evaluation Protocol

During RL, we use vision-grounded rewards. For each generated response, we render the HTML/CSS in the sandbox and capture a screenshot. We then query Qwen2.5-VL-72B-Instruct with (Question, Code, Screenshot) to assign a scalar reward reflecting visual fidelity and layout correct-

ness, thereby closing the loop between code generation and rendered UI performance.

3.1.4 Backbone Models

We evaluate our methodology across multiple model families to verify scalability and architectural consistency. For CPT, we utilize Qwen2.5-Coder-7B as the primary base. For SFT, we extend the Qwen2.5-Coder-7B-CPT, Qwen2.5-Coder-7B-Instruct, Qwen3-4B-Instruct-2507, Qwen3-8B, and Qwen3-Coder-30B-A3B-Instruct to cover varying parameter scales. For RL, we prioritize the Qwen2.5-Coder-7B-Instruct and Qwen2.5-Coder-7B-Instruct-SFT, as well as the Qwen3-4B and Qwen3-8B variants. This selection ensures a rigorous assessment of structural priors across different instruction-following baselines.

3.1.5 Implementation Details

In SFT, models are fine-tuned for 2 epochs with learning rate 5×10^{-5} and global batch size 128 on $\mathcal{D}_S$. For RL, we use GRPO with learning rate 4×10^{-6} and global batch size 128, using 5% linear warmup and cosine decay. We set reward weights $\alpha = 0.6$, $\beta = 0.3$, and $\gamma = 0.1$, prioritizing checklist satisfaction while retaining similarity and length regularization. We apply a KL penalty toward the SFT reference policy with coefficient 0.02. All experiments are conducted on 16×H800 GPUs.

3.2 Main Results

Table 1 presents a comprehensive comparison of our **FrontCoder** models against other closed-source and open-source baselines.

Competitive Performance against Other Models. Our FrontCoder models achieve the best results among open-source models, trailing only behind GPT-5. FrontCoder-7B delivers an AVG score of 70.22, significantly outperforming large-scale open-source models such as GPT-OSS-120B and MiniMax-M2, while also surpassing proprietary systems like Gemini-2.5-Pro and Claude-Opus-4.1. Scaling to the MoE-based FrontCoder-30B-A3B further elevates the AVG to 71.95, narrowing the performance gap with GPT-5 (72.55) to a margin of only 0.6 points. The MoE variant exhibits particular dominance in structural-heavy domains, achieving 81.98 in SVG and 72.47 in Management Systems. These results demonstrate that high-quality alignment and MoE architectures can effectively overcome limitations in raw parameter scale for domain-specific code synthesis.

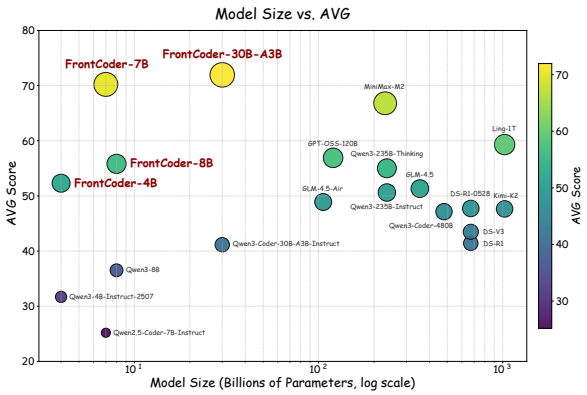

Figure 4: Performance comparison of FrontCoder against other open-source models on ArtifactsBench.

Parameter Efficiency Analysis. As shown in Figure 4, our models exhibit a distinct scaling trajectory compared to other open-source models. While typical baselines require exponential parameter growth to achieve performance gains, FrontCoder occupies a high-efficiency regime. Specifically, our models deliver results comparable to 671B+ parameters models with approximately 100× fewer active parameters, demonstrating that alignment can decouple performance from raw model scale.

3.3 Component and Stage-wise Analysis

To disentangle the sources of performance gains, we examine the training trajectory from two distinct initialization models. Specifically, we evaluate the Base model to isolate the effects of CPT and the Instruct model to isolate the impact of SFT and RL.

Effect of Continual Pre-training. Table 2 demonstrates the impact of CPT on model performance. The raw Base model achieves only 11.03 AVG, primarily limited by insufficient exposure to domain-specific HTML/SVG patterns. Incorporating CPT yields a substantial improvement of 17.70 points, reaching 28.73 AVG. This gain is most significant in structure-heavy domains, indicating that CPT enables the model to internalize hierarchical DOM structures and visual motifs. While CPT establishes foundational structural competence, subsequent SFT is essential for intent alignment, further elevating the performance to 39.56 AVG.

Representative CPT corpus cases illustrating a sidebar-heavy HTML product page and a multi-column SVG wishlist interface are provided in Appendix 5.

Impact of SFT and RL Alignment. Table 3 details the performance evolution starting from the

Model	Size	IFLEN	SCORE					
			GAME	SVG	WEB	SI	MS	AVG
Closed-source Models								
GPT-4o	🔒	4.8K	33.04	33.75	34.22	31.44	32.10	33.54
o3-2025-04-16	🔒	–	54.33	56.37	52.95	55.75	50.21	54.04
Gemini-2.5-Pro	🔒	–	58.38	65.33	58.12	55.54	53.18	57.74
Claude-Opus-4-1	🔒	–	61.63	57.03	60.11	58.87	57.43	59.76
Gemini-3-pro-preview	🔒	–	66.47	84.65	61.94	66.63	55.08	64.52
GPT-5	🔒	–	77.89	73.40	71.31	79.41	64.95	72.55
Open-source Models								
JanusCoder-8B	8B	–	36.39	30.47	40.07	41.92	44.75	39.60
DeepSeek-V3-0324	671B	11.4K	45.29	40.20	45.56	37.22	42.17	43.50
Qwen3-Coder-480B-A35B-Instruct	480B	17.6K	49.27	40.18	48.11	49.66	46.06	47.15
Kimi-K2-Instruct	1T	7.1K	47.08	50.61	46.81	48.88	46.15	47.65
DeepSeek-R1-0528	671B	19.2K	50.46	45.06	47.86	54.08	42.69	47.73
Qwen3-235B-A22B-Instruct-2507	235B	20.8K	50.67	40.41	52.19	50.24	50.83	50.62
GLM-4.5	355B	21.9K	54.79	51.79	51.66	52.06	47.30	51.33
GPT-OSS-120B	120B	16.0K	53.88	54.19	58.77	57.69	56.97	56.91
MiniMax-M2	230B	–	–	–	–	–	–	66.80
Comparison on Baselines								
Qwen3-4B-Instruct-2507	4B	21.0K	30.79	35.98	32.47	33.08	30.45	31.70
Qwen3-8B	8B	6.9K	34.58	36.37	38.08	36.15	35.92	36.52
FrontCoder-4B *(Ours)*	4B	33.3K	51.84	61.65	52.87	54.83	50.65	52.31
FrontCoder-8B *(Ours)*	8B	34.7K	55.72	66.89	56.03	58.82	53.02	55.82
Qwen2.5-Coder-7B-Instruct	7B	4.9K	25.21	20.58	28.56	24.49	26.27	25.19
Qwen3-Coder-30B-A3B-Instruct	30B	18.1K	–	–	–	–	–	41.16
FrontCoder-7B *(Ours)*	7B	25.8K	70.97	80.80	69.65	**72.77**	68.07	70.22
FrontCoder-30B-A3B *(Ours)*	30B	39.4K	**71.65**	**81.98**	**70.72**	70.41	**72.47**	**71.95**

Table 1: Performance of various models on the **ArtifactsBench**, evaluated across five front-end code generation task categories (GAME, SVG, WEB, SI: Simulation, MS: Management System). The table compares both closed-source and open-source models, including our proposed **FrontCoder**. **IFLEN** (Inference Length) represents the average length of the generated code response and serves as a proxy for generation efficiency; since the reasoning chain length is not accessible for certain closed-source models, this field is left empty for them. The columns under **SCORE** report results on each task category, and **AVG** denotes the overall average score on the entire benchmark. The best score in comparison on baselines group is marked in **bold**, and the next best score is underlined. Values marked with – are not publicly disclosed by the official benchmark and models.

Stage	IFLEN	AVG	Δ
Base	7125.55	11.03	-
+ CPT	7762.51	28.73	+17.70
+ CPT + SFT	5918.32	39.56	+28.53

Table 2: Impact of CPT. Base model: Qwen2.5-Coder-7B. Δ denotes improvement over the base model.

Qwen2.5-Coder-7B-Instruct baseline. Fine-tuning on $\mathcal{D}_S$ yields significant improvement of 41.91 points, raising the AVG score from 25.19 to 67.10. This performance gap underscores the insufficiency of general-purpose instruction tuning for front-end synthesis, which demands specialized, executable code demonstrations to achieve structural alignment. RL further optimizes this trajectory by acting as a behavioral regularizer. While SFT increases the average inference length to 35.8K, the application of RL improves the AVG score to 70.22 while simultaneously reducing the IFLEN by approximately 28% to 25.8K. These results confirm that vision-grounded rewards effectively eliminate redundant code and enhance generation efficiency without compromising visual fidelity.

Figure 5 illustrates the convergence dynamics across different initialization models. When starting from the Instruct baseline, output length remain relatively stable, with IFLEN consistently maintained below 2K, while the reward keeps fluctuating. Conversely, the optimization trajectory for the SFT checkpoint is characterized by significant efficiency gains; the validation reward im-

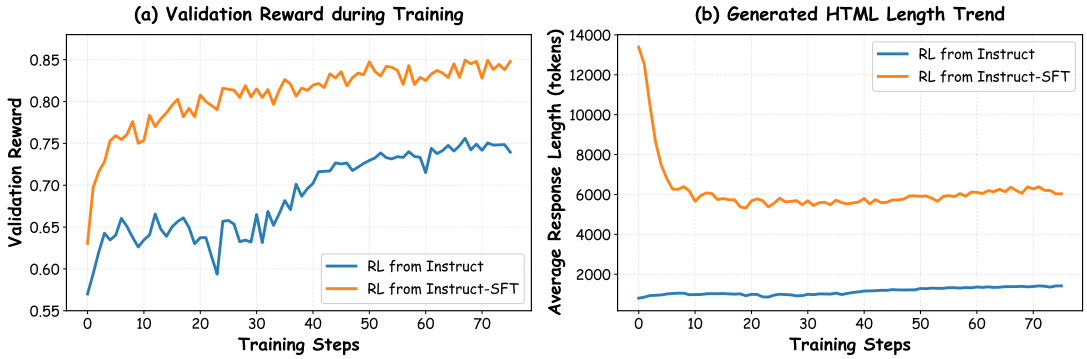

Figure 5: RL dynamics on Qwen2.5-Coder-7B-Instruct. Left: validation reward for weak (Instruct) vs. strong (SFT) initialization. Right: IFLEN evolution, showing that the length-aware reward tends to expand short outputs and compress overly long ones.

Stage	IFLEN	AVG	Δ
Instruct	4941.77	25.19	-
+ RL	6644.43	33.10	+7.91
+ SFT	35821.34	67.10	+41.91
+ SFT + RL	25816.16	**70.22**	+45.03

Table 3: Impact of SFT and RL. Base model: Qwen2.5-Coder-7B-Instruct. Δ denotes improvement over the instruct model.

proves steadily while IFLEN decreases monotonically from 35.8K to 25.8K. This divergence confirms that RL functions as a length-aware structural optimizer primarily when tasked with refining high-complexity generations, whereas it maintains stable output density for simpler initializations.

3.4 Ablation on SFT Data Strategy

Given that SFT contributes the largest alignment gain, we further investigate the trade-offs between data quantity, quality filtering, and length constraints in Table 4.

Dataset	Size	IFLEN	AVG
SFT-Rule-Filtered	167K	33778.26	59.67
SFT-Score-Filtered	106K	38769.39	**71.30**
SFT-Length-Control	60K	35821.34	67.10

Table 4: Ablation of SFT data filtering and length control strategies on Qwen2.5-Coder-7B-Instruct.

Semantic Filtering and Data Quality. Comparing the 167K rule-filtered subset with the 106K score-filtered version reveals that data quality exerts a more decisive influence on performance than raw volume. Despite a 40 percent reduction in sample size, semantic scoring yields an improvement of 11.63 points in AVG score. This validates the effectiveness of model-based filtering in identifying visual fidelity and execution integrity, proving superior to traditional heuristic rules.

Balancing Performance and Efficiency. The 106K subset achieves the highest raw performance of 71.30 but introduces significant length inflation with an IFLEN of 38.8K. By imposing stricter sequence constraints to derive the 60K subset, we observe a moderate performance decrease to 67.10 in exchange for enhanced inference efficiency. However, the subsequent RL stage effectively recovers this gap, reaching a score of 70.22 while preserving the efficiency gains. Consequently, the SFT-60K configuration is selected as the optimal initialization for RL to balance instruction adherence with practical deployment costs.

3.5 Failure Analysis of Imperfect Generations

We defer the full 1,825-sample failure taxonomy and representative hard SVG case studies to Appendix 5 and Appendix 5, where we provide the detailed failure distribution table and qualitative breakdown.

4 Related Work

Visual and Front-End Code Generation. Code generation (Yang et al., 2025c) for visual interfaces has progressed from screenshot-to-markup translation to interactive front-end engineering. Early systems such as Pix2Code (Wüst et al., 2024) and Web2Code (Yun et al., 2024) focus mainly on static UI skeletons, while recent methods improve fidelity through structural decomposition and agentic synthesis, including DCGen (Wan et al., 2025), UICopilot (Gui et al., 2025b), and ScreenCoder (Jiang et al., 2025). Benchmarks such as De-

sign2Code (Si et al., 2025), WebBench (Xu et al., 2025), and FullFront (Sun et al., 2025a) further stress multi-step workflows and interactive behaviors. To close the perception gap, many systems incorporate multimodal encoders, GUI grounding, or visual feedback: DesignCoder (Chen et al., 2025a) and SeeClick (Cheng et al., 2024) align visual elements with code regions, while ReLook (Li et al., 2025) and JanusCoder (Sun et al., 2025b) use VLM critics or unified multimodal interfaces.

LLMs for Code and Data Synthesis. High-quality data remains central to strong code LLMs. Beyond large-scale code corpora used by models such as StarCoder (Li et al., 2023), recent work emphasizes synthetic instruction, reasoning, and long-horizon supervision, as in OSS-Instruct (Wei et al., 2024), Auto-Evolve (Xu et al., 2024), and Seed-Coder (Zhang et al., 2025c). In web generation, WebCode2M (Gui et al., 2025a) and WebGen-Bench (Lu et al., 2025) pair code with rendered visuals and interaction traces, while ArtifactsBench (Zhang et al., 2025b) and CodeCriticBench (Zhang et al., 2025a) introduce VLM-based judging for rendered artifacts. Related efforts improve multimodal consistency, robustness, factuality, and task coverage through cross-level alignment (Meng et al., 2026), adversarial test generation (Shi et al., 2026), code QA benchmarking (Yang et al., 2025b), and industrial code-model training (Yang et al., 2026a). Other works extend code-oriented reasoning to text-to-SQL via pivot languages (Chi et al., 2025) and study knowledge removal in LLMs as asymmetric two-task learning (Xiao et al., 2026).

Reinforcement Learning for LLM and Code. RL is widely used to align models beyond next-token prediction. For conventional code generation, CodeRL (Le et al., 2022) and PPOCoder (Shojaee et al., 2023) use unit tests and execution traces as rewards, but such signals are often unavailable in front-end tasks where correctness is primarily visual and layout-based. Feedback-driven refinement methods, including Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), and CRITIC (Gou et al., 2024), explore iterative self-correction, while ChipSeek (Chen et al., 2025b) leverages EDA tool feedback with execution- and performance-based rewards. Recent RLVR work further decouples exploration and exploitation in semantic space (Huang et al., 2026). For code LLM alignment and evaluation, CodeArena (Yang et al., 2025a) uses human preference signals, Qwen2.5-xCoder (Yang et al., 2025d) studies multi-agent collaboration for multilingual instruction tuning, and InCoder-32B (Yang et al., 2026b) together with InCoder-32B-Thinking (Yang et al., 2026c) unify industrial software domains with execution-grounded training and error-driven reasoning. BiasScope (Lai et al., 2026) and Wu et al. (Wu et al., 2026) further expose biases and fairness issues in LLM-as-a-judge and router evaluation pipelines.

5 Conclusion

In this work, we address the challenge of generating visually accurate and structurally sound front-end code by presenting a three-stage pipeline prioritizing functional correctness and visual fidelity. Through continual pre-training on synthetic data, quality-controlled supervised fine-tuning, and reinforcement learning with checklist-based rewards, our approach demonstrates systematic data curation and vision-grounded optimization are essential for developing reliable front-end code generation systems. Evaluations reveal that while strong base models struggle with visual faithfulness and layout complexity, our model achieves substantial improvements over baselines, attaining competitive performance with frontier models while preserving efficiency. These findings underscore the importance of incorporating visual rendering quality into training objectives, moving beyond functional correctness to meet practical demands. We hope to inspire future research in vision-grounded code generation, bridging the gap between natural language instructions and pixel-perfect implementations.

Limitations

While our method improves front-end code generation through a three-stage pipeline, several limitations remain. First, evaluations are limited to ArtifactsBench, and generalization to other frameworks (e.g., React, Vue) is not fully validated. Second, VLM-based reward signals may introduce biases and may not align with human preferences or accessibility standards. Finally, our sandboxed environment focuses on visual fidelity but does not assess interaction, dynamic behavior, or responsiveness.

Ethics Statement

We consider the ethical implications of data collection and usage. Our training data consist of publicly available resources and synthetic samples

generated by existing models, with adherence to licensing requirements. We apply deduplication and filtering to remove potentially sensitive content, including PII, credentials, and security-critical code. These measures aim to support responsible research while mitigating privacy and misuse risks.

Acknowledgments

This work is supported by the Fundamental Research Funds for the Central University (Grant No. GW2025-19) and supported by State Key Laboratory of Complex & Critical Software Environment (Grant No. SKLCCSE-2025ZX-26).

References

Anthropic. 2025. Introducing claude 4.

Yunnong Chen, Shixian Ding, YingYing Zhang, Wenkai Chen, Jinzhou Du, Lingyun Sun, and Liuqing Chen. 2025a. Designcoder: Hierarchy-aware and self-correcting UI code generation with large language models. *CoRR*, abs/2506.13663.

Zhirong Chen, Kaiyan Chang, Zhuolin Li, Xinyang He, Chujie Chen, Cangyuan Li, Mengdi Wang, Haobo Xu, Yinhe Han, and Ying Wang. 2025b. Chipseek-r1: Generating human-surpassing rtl with llm via hierarchical reward-driven reinforcement learning. *arXiv preprint arXiv:2507.04736*.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. 2024. Seeclick: Harnessing GUI grounding for advanced visual GUI agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 9313–9332. Association for Computational Linguistics.

Yongdong Chi, Hanqing Wang, Yun Chen, Yan Yang, Jian Yang, Zonghan Yang, Xiao Yan, and Guanhua Chen. 2025. Pi-sql: Enhancing text-to-sql with fine-grained guidance from pivot programming languages. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 25120–25144.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2024. CRITIC: large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Yi Gui, Zhen Li, Yao Wan, Yemin Shi, Hongyu Zhang, Bohua Chen, Yi Su, Dongping Chen, Siyuan Wu, Xing Zhou, Wenbin Jiang, Hai Jin, and Xiangliang Zhang. 2025a. Webcode2m: A real-world dataset for code generation from webpage designs. In *Proceedings of the ACM on Web Conference 2025, WWW 2025, Sydney, NSW, Australia, 28 April 2025- 2 May 2025*, pages 1834–1845. ACM.

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

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196.

Fanding Huang, Guanbo Huang, Xiao Fan, Yi He, Xiao Liang, Xiao Chen, Qinting Jiang, Faisal Nadeem Khan, Jingyan Jiang, and Zhi Wang. 2026. Semantic-space exploration and exploitation in rlvr for llm reasoning. *Preprint*, arXiv:2509.23808.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

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

Peng Lai, Zhihao Ou, Yong Wang, Longyue Wang, Jian Yang, Yun Chen, and Guanhua Chen. 2026. Biasscope: Towards automated detection of bias in LLM-as-a-judge evaluation. In *The Fourteenth International Conference on Learning Representations*.

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

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li and Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, and 46 others. 2023. Starcoder: may the source be with you! *Trans. Mach. Learn. Res.*, 2023.

Yuhang Li, Chenchen Zhang, Ruilin Lv, Ao Liu, Ken Deng, Yuanxing Zhang, Jiaheng Liu, Wiggin Zhou, and Bo Zhou. 2025. Relook: Vision-grounded RL with a multimodal LLM critic for agentic web coding. *CoRR*, abs/2510.11498.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. 2025. Webgen-bench: Evaluating llms on generating interactive and functional websites from scratch. *CoRR*, abs/2505.03733.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Chunlei Meng, Guanhong Huang, Rong Fu, Runmin Jian, Zhongxue Gan, and Chun Ouyang. 2026. Clcr: Cross-level semantic collaborative representation for multimodal learning. *arXiv preprint arXiv:2602.19605*.

OpenAI. 2025. Introducing gpt-5.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, and 6 others. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Jingwei Shi, Xinxiang Yin, Jing Huang, Jinman Zhao, and Shengyu Tao. 2026. Codehacker: Automated test case generation for detecting vulnerabilities in competitive programming solutions. *arXiv preprint arXiv:2602.20213*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based code generation using deep reinforcement learning. volume 2023.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 2025. 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, NAACL 2025 - Volume 1: Long Papers, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, pages 3956–3974. Association for Computational Linguistics.

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

Qiushi Sun, Jingyang Gong, Yang Liu, Qiaosheng Chen, Lei Li, Kai Chen, Qipeng Guo, Ben Kao, and Fei Yuan. 2025b. Januscoder: Towards a foundational visual-programmatic interface for code intelligence. *CoRR*, abs/2510.23538.

Yuxuan Wan, Chaozheng Wang, Yi Dong, Wenxuan Wang, Shuqing Li, Yintong Huo, and Michael R. Lyu. 2025. Divide-and-conquer: Generating UI code from screenshots. *Proc. ACM Softw. Eng.*, 2(FSE):2099–2122.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Wanxing Wu, He Zhu, Yixia Li, Lei Yang, Jiehui Zhao, Hongru Wang, Jian Yang, Benyou Wang, Bingyi Jing, and Guanhua Chen. 2026. Towards fair and comprehensive evaluation of routers in collaborative llm systems. *arXiv preprint arXiv:2602.11877*.

Antonia Wüst, Wolfgang Stammer, Quentin Delfosse, Devendra Singh Dhami, and Kristian Kersting. 2024. Pix2code: Learning to compose neural visual concepts as programs. In *Uncertainty in Artificial Intelligence, 15-19 July 2024, Universitat Pompeu Fabra, Barcelona, Spain*, volume 244 of *Proceedings of Machine Learning Research*, pages 3829–3852. PMLR.

Zeguan Xiao, Siqing Li, Yong Wang, Xuetao Wei, Jian Yang, Yun Chen, and Guanhua Chen. 2026. Modeling llm unlearning as an asymmetric two-task learning problem. *arXiv preprint arXiv:2604.14808*.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Kai Xu, YiWei Mao, XinYi Guan, and ZiLong Feng. 2025. Web-bench: A LLM code benchmark based on web standards and frameworks. *CoRR*, abs/2505.07473.

Jian Yang, Jiaxi Yang, Wei Zhang, Ke Jin, Yibo Miao, Lei Zhang, Liqun Yang, Zeyu Cui, Yichang Zhang, Zhoujun Li, Binyuan Hui, and Junyang Lin. 2025a.

Codearena: Evaluating and aligning codellms on human preference. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, EMNLP 2025, Suzhou, China, November 4-9, 2025*, pages 9672–9683. Association for Computational Linguistics.

Jian Yang, Wei Zhang, Shawn Guo, Zhengmao Ye, Lin Jing, Shark Liu, Yizhi Li, Jiajun Wu, Cening Liu, X Ma, and 1 others. 2026a. Iquest-coder-v1 technical report. *arXiv preprint arXiv:2603.16733*.

Jian Yang, Wei Zhang, Yizhi Li, Shawn Guo, Haowen Wang, Aishan Liu, Ge Zhang, Zili Wang, Zhoujun Li, Xianglong Liu, and 1 others. 2025b. Codesimpleqa: Scaling factuality in code large language models. *arXiv preprint arXiv:2512.19424*.

Jian Yang, Wei Zhang, Shark Liu, Jiajun Wu, Shawn Guo, and Yizhi Li. 2025c. From code foundation models to agents and applications: A practical guide to code intelligence. *arXiv preprint arXiv:2511.18538*.

Jian Yang, Wei Zhang, Yibo Miao, Shanghaoran Quan, Zhenhe Wu, Qiyao Peng, Liqun Yang, Tianyu Liu, Zeyu Cui, Binyuan Hui, and Junyang Lin. 2025d. Qwen2.5-xcoder: Multi-agent collaboration for multilingual code instruction tuning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 13121–13131. Association for Computational Linguistics.

Jian Yang, Wei Zhang, Jiajun Wu, Junhang Cheng, Shawn Guo, Haowen Wang, Weicheng Gu, Yaxin Du, Joseph Li, Fanglin Xu, and 1 others. 2026b. Incoder-32b: Code foundation model for industrial scenarios. *arXiv preprint arXiv:2603.16790*.

Jian Yang, Wei Zhang, Jiajun Wu, Junhang Cheng, Tuney Zheng, Fanglin Xu, Weicheng Gu, Lin Jing, Yaxin Du, Joseph Li, and 1 others. 2026c. Incoder-32b-thinking: Industrial code world model for thinking. *arXiv preprint arXiv:2604.03144*.

Sukmin Yun and 1 others. 2024. Web2code: A large-scale webpage-to-code dataset and evaluation framework. *arXiv preprint arXiv:2406.20098*.

Alexander Zhang, Marcus Dong, Jiaheng Liu, Wei Zhang, Yejie Wang, Jian Yang, Ge Zhang, Tianyu Liu, Zhongyuan Peng, Yingshui Tan, Yuanxing Zhang, Zhexu Wang, Weixun Wang, Yancheng He, Ken Deng, Wangchunshu Zhou, Wenhao Huang, and Zhaoxiang Zhang. 2025a. Codecriticbench: A holistic code critique benchmark for large language models. *CoRR*, abs/2502.16614.

Chenchen Zhang, Yuhang Li, Can Xu, Jiaheng Liu, Ao Liu, Shihui Hu, Dengpeng Wu, Guanhua Huang, Kejiao Li, Qi Yi, Ruibin Xiong, Haotian Zhu, Yuanxing Zhang, Yuhao Jiang, Yue Zhang, Zenan Xu, Bohui Zhai, Guoxiang He, Hebin Li, and 13 others. 2025b. Artifactsbench: Bridging the visual-interactive gap in LLM code generation evaluation. *CoRR*, abs/2507.04952.

Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, Tao Sun, Jinhua Zhu, Shulin Xin, Dong Huang, Yetao Bai, Lixin Dong, Chao Li, Jianchong Chen, Hanzhi Zhou, Yifan Huang, and 7 others. 2025c. Seed-coder: Let the code model curate data for itself. *CoRR*, abs/2506.03524.

Appendix

Detailed Ablation Study Results

This section presents a detailed ablation analysis over five task categories: Game, SVG, Web, Simulation, and Management. The reported statistics constitute the basis of the averaged results in the main paper and provide fine-grained insights into the contribution of each training stage.

Effect of CPT. Table 5 illustrates the impact of CPT on overall performance. The base Qwen2.5-Coder-7B model attains only 11.03 AVG, reflecting its limited capacity to generalize to artifact-centric generation tasks without domain exposure. After applying CPT, the resulting Base-75B-CPT model exhibits a substantial gain of 19.22 points, reaching 28.73 AVG. This improvement is consistent across all task categories, indicating that CPT effectively equips the model with domain-specific structural and semantic priors. Building upon this foundation, subsequent Supervised Fine-Tuning further amplifies performance to 39.56 AVG, demonstrating that strong domain-aware representations significantly enhance the effectiveness of downstream instruction alignment.

Effect of SFT and RL. Table 6 analyzes the relative contributions of SFT and RL, using Qwen2.5-Coder-7B-Instruct as the baseline. Directly applying RL to the instruct model yields a modest improvement from 25.19 to 33.10 AVG, suggesting that reward optimization alone is insufficient without adequate supervised grounding. In contrast, SFT serves as the dominant factor, producing a large performance jump to 67.10 AVG by establishing robust instruction-following behavior. When RL is further applied on top of the SFT-initialized policy, the full pipeline achieves the best overall result of 70.22 AVG, with particularly strong gains on SVG and Simulation tasks. These findings indicate that RL is most effective when refining an

already well-aligned and competent policy rather than compensating for missing supervision.

Effect of SFT Data Filtering Strategies. Table 7 compares different SFT data filtering strategies with varying data scales and selection criteria. The rule-filtered dataset (167K samples) achieves 59.67 AVG, providing a reasonable baseline but leaving room for improvement. Enforcing length control (60K samples) yields a notable gain to 67.10 AVG, highlighting the importance of concise and focused supervision for code generation tasks. The score-filtered strategy (106K samples) further improves performance to 71.30 AVG, outperforming both heuristic-based approaches. This result suggests that model-based quality estimation better captures semantic correctness and execution fidelity than surface-level constraints, leading to more effective supervision despite a moderate data size.

Semantic Scoring Rubric

The 25-criterion semantic scoring rubric used for SFT filtering is organized into five dimensions in Table 8.

Cross-Model Consistency of VLM Judges

To address the concern that our RL and evaluation pipeline could inherit bias from a single VLM judge, we conduct a cross-model consistency study. We randomly select 64 prompts and generate 4 independent rollouts per prompt using Qwen2.5-Coder-7B-Instruct-SFT-60K, resulting in 256 code-screenshot pairs. We then score the same pairs with Qwen2.5-VL-72B and Gemini-2.5-Pro using the identical checklist-based scoring protocol and measure agreement through Kendall's Tau-b.

Although the two judges operate on different absolute score scales, the Kendall's Tau-b of 0.717 indicates strong agreement in relative ranking. This suggests that our checklist-based visual evaluation is not dominated by idiosyncratic bias from a single judge family.

Teacher Model Selection Analysis

To guide teacher-model selection, we perform a score-normalized analysis based on a custom 50-sample out-of-distribution set using both automated scoring and human review. We align our three selection criteria with benchmark dimensions as follows: visual fidelity, structural correctness, and functional completeness. The overall score is the mean of the three criterion scores.

This view matches the qualitative teacher allocation used in our pipeline. GPT-5 is strongest on the structural correctness and functional completeness therefore used to produce high-quality reference responses; Gemini-3-pro-preview is strongest overall and on the visual-fidelity and is therefore used for checklist verification and consistency auditing; MiniMax-M2 remains a strong and balanced generator for large-scale synthesis.

Representative CPT Corpus Cases

To concretize the kinds of structural priors injected during continual pre-training, we present two representative cases from the CPT corpus.

Failure Analysis of Imperfect Generations

To move beyond anecdotal error inspection, we analyze all 1,825 imperfect generations from the Qwen2.5-Coder-7B-Instruct-SFT model on ArtifactsBench, i.e., all samples that do not achieve a perfect score. We inspect the generated code together with their checklist penalty signals and group the primary failure cause of each sample into one of eight categories.

The distribution shows that the dominant bottlenecks are functional completeness and visual or interaction fidelity rather than a single catastrophic mode. Incomplete functional implementation accounts for 33.48% of failures, followed by insufficient interaction, visual, or theme fidelity at 20.05% and deviation from core requirements at 15.34%. This broader profile directly reinforces the role of RL in our pipeline: the checklist rewards target core requirement satisfaction and fidelity, while rendering-validity and repetition penalties suppress brittle long-form failures that only appear at generation time.

Qualitative Analysis of Hard SVG Tasks

The SVG subset provides a concentrated view of the model's structural and geometric reasoning. We examine three representative hard tasks where our SFT model substantially outperforms Qwen3-4B-Instruct-2507.

Across these cases, the gains go well beyond surface styling. They reflect improved handling of complex cubic Bézier and arc path geometry, correct polar-to-Cartesian computations for dynamic pie charts, multi-stage SVG filter pipelines such as `feGaussianBlur` and `feComposite`, careful `defs` organization with gradients and clip paths, and JavaScript-driven attribute updates for interactive

Stage	Model	IFLEN	SCORE					
			GAME	SVG	WEB	SI	MS	AVG
Base	Qwen2.5-Coder-7B	7.1K	10.56	11.87	11.38	10.39	11.66	11.03
+ CPT	Base-CPT	7.8K	30.53	36.10	28.53	32.27	24.54	28.73
+ CPT + SFT	Base-CPT-SFT	5.9K	40.20	59.67	39.89	45.86	33.00	39.56

Table 5: Detailed ablation results for the CPT stage. We report the breakdown of scores on ArtifactsBench tasks. The Base model is Qwen2.5-Coder-7B.

Stage	Method	IFLEN	SCORE					
			GAME	SVG	WEB	SI	MS	AVG
Instruct	Qwen2.5-Coder-7B-Instruct	4.9K	25.21	20.58	28.56	24.49	26.27	25.19
+ RL	Direct RL	6.6K	30.50	36.15	33.81	31.38	32.38	33.10
+ SFT	SFT-60K	35.8K	66.39	77.49	65.48	68.32	67.73	67.10
+ SFT + RL	SFT-60K + GRPO	25.8K	**70.97**	**80.80**	**69.65**	**72.77**	**68.07**	**70.22**

Table 6: Detailed ablation results for SFT and RL stages. Starting from the Qwen2.5-Coder-7B-Instruct baseline, we explore the impact of applying RL directly versus the full SFT+RL pipeline. The SFT-60K+GRPO configuration yields the best performance across all sub-tasks.

Strategy	Data Size	IFLEN	SCORE					
			GAME	SVG	WEB	SI	MS	AVG
Rule-Filtered	167K	33.8K	60.17	74.75	56.50	65.08	55.86	59.67
Score-Filtered	106K	38.8K	**71.85**	**80.46**	**68.60**	**71.71**	**71.10**	**71.30**
Length-Control	60K	35.8K	66.39	77.49	65.48	68.32	67.73	67.10

Table 7: Detailed ablation on SFT data filtering strategies. We compare rule-based filtering (167K), score-based quality filtering (106K), and length-controlled filtering (60K). The score-filtered subset achieves the highest overall effectiveness.

rendering. For example, in the pie-chart task the baseline struggles to compute valid arc geometry, whereas our SFT model correctly implements polar conversions and large-arc logic. In the binoculars and system-diagram tasks, the improved outputs require layered gradients and filter stacks that depend on non-trivial hierarchical SVG structuring.

Prompts of Data Generation

In this section, we present the detailed prompts used throughout our data generation and refinement pipeline. These prompts cover the entire pipeline of the dataset construction, including initial taxonomy classification, diverse task generation, code synthesis, and the multi-dimensional quality evaluation for both SFT and Reinforcement Learning (RL) stages. Figure 6 to Figure 14 illustrate the specific instructions and constraints provided to the Large Language Models (LLMs) to ensure the diversity, technical accuracy, and visual quality of the generated front-end code.

Criterion	Description
Dimension 1: Engineering Quality and Maintainability	
Code Executability	Evaluate if the code can execute directly without syntax or logic errors.
Code Standards	Evaluate naming conventions, indentation, and code structure quality.
Engineering Quality	Assess modular design, design patterns, and code reuse.
Comment Quality	Evaluate whether comments explain structure and logic clearly.
Documentation Completeness	Check for implementation notes and usage guidance.
Code Redundancy	Check for duplicate or redundant code or unrelated functions.
Code Readability	Assess clarity and maintainability of code structure.
Dimension 2: Functional Completeness, Reliability and Security	
Boundary Handling	Evaluate edge case, exception, and error handling coverage.
Data Validation	Check input and data validation and basic security protections such as XSS and CSRF.
Interaction Completeness	Evaluate completeness of user interactions such as click, drag, and input.
Exception Handling	Evaluate handling of anomalies, user mistakes, and runtime errors.
Cross Platform	Check browser and device compatibility and responsive behavior.
Accessibility Support	Evaluate semantic HTML, ARIA, keyboard navigation, and screen reader support.
Dimension 3: Experience and Visual Quality	
Design Professionalism	Assess layout, typography, color system, and overall UI/UX quality.
Interaction Smoothness	Check animation quality, transitions, and response smoothness.
User Feedback	Evaluate loading states, error prompts, and feedback mechanisms.
Experience Enhancement	Evaluate extra user experience value beyond basics such as personalization and animations.
Visual Fidelity and Completeness	Comprehensive evaluation of visual accuracy and completeness.
Dimension 4: Technical Depth and Innovation	
Tech Selection	Evaluate whether chosen frameworks and libraries are appropriate.
Performance Optimization	Check complexity, rendering efficiency, and memory optimization.
Modern Features	Evaluate use of modern language and platform features such as ES6, CSS3, and HTML5.
Innovative Features	Check for meaningful innovation in functionality or implementation.
Dimension 5: Requirement Alignment and Solution Quality	
Core Function Completeness	Check whether core requirement functions are fully implemented.
Requirement Understanding	Assess whether the response correctly understands the prompt.
Solution Rationality	Evaluate whether the solution is reasonable and follows best practices.

Table 8: Semantic scoring rubric used for SFT filtering, with criterion names in the left column and descriptions in the right column.

Gemini Mean	Gemini Std	Qwen Mean	Qwen Std	Kendall's Tau-b
65.81	15.84	84.23	11.72	0.717

Table 9: Cross-model consistency between Gemini-2.5-Pro and Qwen2.5-VL-72B on 256 code-screenshot pairs.

Model	Visual	Structural	Functional	Overall
Gemini-3-pro-preview	72.58	56.36	60.24	63.06
GPT-5	61.20	58.54	62.92	60.89
MiniMax-M2	52.38	51.84	53.08	52.43
Qwen3-Coder-480B-A35B-Instruct	38.68	41.94	40.36	40.33

Table 10: Score-normalized analysis for teacher-model selection, derived from the ArtifactsBench scores reported in the main paper.

Case	Prompt Summary	Final Structure	Analysis
Case 1	Beauty-and-cosmetics product detail page with a Bootstrap-style glassmorphism aesthetic, responsive sidebar layout, semantic HTML, accessible navigation, scalable typography, and interactive purchase controls.	A conventional HTML document with sticky top navigation, a Bootstrap grid main region, a persistent `<aside>` filter panel, dedicated product-media and pricing sections, repeated review and recommendation cards, and script-backed interaction widgets.	This case exposes the model to a deeply nested sidebar-plus-main-content composition rather than a flat landing page. The structural prior it provides is a reusable hierarchy for coordinating navigation, filters, media, purchase actions, and repeated form controls inside a large but organized DOM tree.
Case 2	Plus-size fashion wishlist page implemented as an SVG interface with a minimalist warm-tone palette, responsive multi-column layout, progressive loading, semantic SVG structure, and modern interactive controls.	A root `<svg>` document combining `<defs>`, reusable `<symbol>` icons, navigation and filter controls, a multi-column wishlist grid, and a scripted update layer for progressive loading and interaction.	This case exposes the model to SVG-native reuse patterns based on `defs`, `symbol`, `use`, and scripted state updates rather than conventional HTML templating. The resulting structural prior is useful for learning hierarchical vector scenes where layout, icon reuse, and interaction logic must remain synchronized inside a single SVG document.

Table 11: Representative CPT corpus cases illustrating the kinds of structural priors injected during continual pre-training. For each case, we summarize the task prompt, the final output structure, and the structural prior that the sample can teach the model.

Failure Type	Count	Pct.
Incomplete Functional Implementation	611	33.48
Insufficient Interaction, Visual, or Theme Fidelity	366	20.05
Deviation from Core Requirements or Missing Core Features	280	15.34
Poor Engineering Architecture and Maintainability	196	10.74
Logical Errors or Usability Defects	189	10.36
Lack of Data Persistence or Backend Capabilities	86	4.71
Insufficient Performance and Robustness	77	4.22
Lack of Security, Accessibility, or Standard Compliance	20	1.10

Table 12: Primary failure causes across 1,825 imperfect generations from the SFT model on ArtifactsBench.

Task	Qwen3-4B	SFT	Gain
Binoculars Scene (Hard)	30	96	+66
Interactive System Diagram (Hard)	22	87	+65
Interactive Pie Chart (Hard)	18	73	+55

Table 13: Representative hard SVG tasks comparing Qwen3-4B-Instruct-2507 and our SFT model.

CPT: Prompt Template Generation

Purpose: Generate HTML prompt templates using LLM for creating beautiful, practical, and fully functional HTML front-end code.
Prompt:
You are an expert front-end designer and prompt engineer. Please generate exactly ONE English prompt template for generating beautiful, practical, and fully functional HTML front-end code.
Requirements:

1. The template MUST include {web_category}, which represents a hierarchical web category ending with a specific page function or purpose (e.g., "Gaming - Adventure Games - Action-Adventure Games - Tutorial/Help Page" or "Shopping & E-Commerce - Sports Nutrition - Product Listing Page").
2. In addition to {web_category}, the template MUST include **2 or 3 or 4** of the following variables (use curly braces {} to wrap them):
   - {design_style} - Design style (e.g., modern, minimalist, retro)
   - {features} - Feature specifications (e.g., responsive, interactive)
   - {color_scheme} - Color scheme (e.g., dark mode, pastel colors)
   - {layout_type} - Layout type (e.g., grid, flexbox, single column)

   Do NOT include all variables. Choose only the most relevant ones for a realistic and coherent web page scenario.
3. The template should be detailed and specific, describing the type of web page, its purpose, and how it should look or behave.
4. The output of this request must be **the prompt template itself** (no explanations, no numbering, no examples).
5. The template should include **positive and detailed quality requirements** about the HTML code, such as:
   - visually appealing design
   - responsive and accessible layout
   - clean and semantic structure
   - modern components or good UX practices
   - readable indentation and organized structure
6. The template must instruct the model to automatically determine appropriate interaction modes, component libraries and choose suitable front-end technologies or libraries that best fit the specific webpage context.
7. The template must clearly instruct that the model should **output only the HTML code** and nothing else.
8. The template should be in natural, clear, and professional English.

Now generate ONE unique prompt template following these requirements.

Figure 6: The prompt of generating HTML prompt templates for front-end code generation.

CPT: Category Tree Generation

Purpose: Generate fine-grained subcategories for website taxonomy classification.
Data Scale: 25 top-level categories → 2K leaf categories
Prompt:
You are an expert in website taxonomy and classification. Generate a fine-grained list of subcategories for the current category.
Current category path: {path_str}
Requirements:

- Output **only** a valid JSON object: {"subcategories": ["sub1", "sub2", ...]}
- Subcategories must be real, specific website content topics under the current category
- Do not repeat the parent category or output overly broad terms
- Subcategories must be in English, without explanations or numbering
- Generate between 5 and {max_children} subcategories; do not exceed {max_children}
- If the category cannot be subdivided further, output {"subcategories": []}
- Do not include any other text outside the JSON object

Figure 7: The prompt of generating fine-grained subcategories for website taxonomy classification.

CPT: WebSight Data Expansion

Purpose: Expand and improve WebSight dataset HTML code with enhanced features and styling.
Prompt:
You are a senior front-end development expert. Given a webpage category/theme description and original HTML code, please professionally expand and improve this HTML code.
Webpage Category/Theme: {theme}
Original HTML Code: {original_html}
Task Requirements:

1. **Maintain Category/Theme Consistency**: The expanded content must be highly relevant to the given category/theme description. Do not deviate from the theme.
2. **Code Expansion** (Key Focus):
   - Expand upon the original code, adding more practical features
   - Enrich page content and interactive effects
   - Add more components and page elements
   - Optimize CSS styling for better aesthetics
   - Enhance JavaScript interactive functionality
   - Use modern front-end libraries and best practices
   - Goal: The expanded HTML should be more complete and professional than the original
3. **Technical Requirements**:
   - Maintain clear code structure with detailed comments
   - Use modern HTML5/CSS3/JavaScript features
   - Ensure code runs directly
   - Focus on responsive design and user experience

Please begin your expansion work:

Figure 8: The prompt of expanding and improving WebSight dataset HTML code with enhanced features.

SFT: Task Generation

Purpose: Generate 10 diverse specific tasks per subcategory (2K subcategories $\rightarrow$ 20K tasks).
Prompt:
You are a senior software development and requirement analysis expert. I will give you a category and sub-category. Please generate 10 DIFFERENT and DIVERSE specific tasks under this sub-category.
Category (Major Classification): {category}
Sub-Category: {sub_category}
Requirements:

1. Generate exactly 10 specific tasks that are:
   - Under the category "{category}" and sub-category "{sub_category}"
   - DIFFERENT from each other (no duplicates)
   - Clear, actionable, and specific (5-15 words each in English)
   - Practical and implementable as web applications or tools
2. The 10 tasks should cover diverse aspects:
   - Different features or functionalities
   - Different complexity levels (simple to advanced)
   - Different use cases or scenarios
   - Different technical approaches or implementations

Please provide your 10 specific tasks in English:

Figure 9: The prompt of generating 10 diverse specific tasks per subcategory for SFT data expansion.

SFT: Variant Generation

Purpose: Generate 12 variants per task (20K tasks → 240K variants).
12 Variant Types:

1. Color Scheme - Change to dark/light mode, different color palette
2. Layout Style - Grid/List/Card layout variations
3. Interaction Mode - Click/Hover/Drag interaction changes
4. Responsive Design - Mobile-first or Desktop-first variations
5. Animation Effects - Add smooth transitions and animations
6. Accessibility Features - WCAG compliant, keyboard navigation
7. Advanced Features - Add premium features like filters, search, export
8. Minimalist Design - Simplified UI with essential features only
9. Data Visualization - Add charts, graphs, or visual representations
10. Real-time Updates - Add live data updates and notifications
11. Gamification - Add points, badges, leaderboards
12. Internationalization - Multi-language support and locale-specific features

Prompt:
You are a senior software development expert. I will give you an original task and a variant specification. Please generate a NEW task description that incorporates the variant requirements.
Original Task:

- **Category**: {category}
- **Sub-Category**: {sub_category}
- **Specific Task**: {specific_task}

Variant Specification:

- **Variant #{variant_id}**: {variant_name}
- **Description**: {variant_description}

Requirements:

1. Generate a NEW specific task (5-20 words) that:
   - Builds upon the original task: "{specific_task}"
   - Incorporates the variant requirements: {variant_name}
   - Remains under the category "{category}" and sub-category "{sub_category}"
   - Is practical and implementable as a web application
2. Make it specific and actionable:
   - Include technical details (e.g., "dark mode with #1a1a1a background")
   - Specify implementation methods (e.g., "using CSS Grid", "with React hooks")
   - Describe user interactions (e.g., "drag-and-drop interface", "hover effects")
3. Ensure distinctness:
   - The task should feel different from the original
   - It should still solve the same problem but with the variant's approach
   - Keep it concise but specific

Please generate the variant task:

Figure 10: The prompt of generating 12 variants per task for SFT data diversity.

SFT: Code Generation

Purpose: Generate complete HTML/CSS/JavaScript code for 240K variant tasks.
Prompt:
You are a code expert. Please generate a complete, production-ready web application based on the following requirements.
Category: {class}
Sub-Category: {sub_category}
Task: {variant_task}
Variant Type: {variant_type}
Requirements:

1. Generate a complete HTML file with embedded CSS and JavaScript
2. Include all necessary styling and functionality
3. Make it visually appealing and user-friendly
4. Ensure the code is clean, well-commented, and follows best practices
5. The application should be fully functional and ready to use

Output Format:
Please provide ONLY the complete HTML code (including CSS and JavaScript), without any markdown code fences or explanations.
Generate the code:

Output: Complete standalone HTML file with embedded CSS and JavaScript

Figure 11: The prompt of generating complete HTML/CSS/JavaScript code for variant tasks.

SFT: Quality Scoring System (25 Dimensions)

Purpose: 25-dimensional quality scoring system for filtering SFT data.
Scoring Categories:

- **Code Quality (4):** Executability (1.2), Core Completeness (1.5), Standards (1.0), Engineering (1.0)
- **Functionality (3):** Boundary Handling (1.1), Data Validation (1.1), Interaction (1.2)
- **User Experience (3):** Design (1.0), Smoothness (1.0), Feedback (1.0)
- **Response Quality (4):** Understanding (1.3), Rationality (1.2), Comments (0.8), Docs (0.7)
- **Technical (3):** Tech Selection (1.0), Performance (1.1), Modern Features (0.9)
- **Innovation (2):** Innovative Features (0.8), UX Enhancement (0.7)
- **Robustness (2):** Code Redundancy (0.8), Exception Handling (1.1)
- **Other (4):** Cross-Platform (1.0), A11y (0.7), Readability (1.0), Visual Fidelity (1.2)

Note: Numbers in parentheses are weights. Each criterion scored 0/2.5/5/7.5/10.
Prompt:
You are an expert code reviewer. Score the following question-response pair using 25 criteria.
Scoring Criteria (25 items, total {total_max_score} points)
{criteria_text}
Content to Score
Question: {question}
Response: {response}
Requirements:

1. **Strict scoring**: Don't give perfect scores easily
2. **Comprehensive**: Evaluate all dimensions
3. **Find issues**: Actively look for defects and improvement areas
4. **Compare to standards**: Compare against industry best practices
5. **Objective**: Base scores on facts, not code length

Figure 12: The prompt of 25-dimensional quality scoring system for filtering SFT training data.

RL: Question Generation with Checklist

Purpose: Generate 2000 RL training prompts across 16 categories with 6 tech stacks.
16 Task Categories: Classic Games (12%), Action Games (6%), Puzzle Games (4%), Strategy & Card (4%), CRUD & Management (12%), Business Apps (10%), Data Charts (6%), Dashboards (3%), Maps (2%), SVG & Graphics (8%), Calculators (6%), Media Processing (6%), Physics Sims (5%), Animation (4%), Communication (3%), E-commerce (2%)
6 Tech Stacks: HTML5 Canvas + JS, Vanilla JS, SVG + CSS Animation, Three.js, React + TypeScript, ECharts
Prompt Template:
ROLE SETTING: You are a {role}. Your skills include: {skills}.
TASK BACKGROUND:
You need to use {tech_stack_name} technology stack to complete the following front-end development task. Please implement using {tech_stack_desc} and integrate all code into a single HTML file.
PROJECT REQUIREMENTS:
Make sure the code you generate is executable for demonstration purposes. {task}
TECHNICAL REQUIREMENTS:

1. Use {tech_stack_name} to implement all functionality
2. The code must be complete and runnable, with no parts omitted
3. Add clear comments explaining key logic
4. Implement responsive design to adapt to different screen sizes
5. Include necessary error handling and user feedback

OUTPUT FORMAT: Please first briefly explain your implementation approach in 2-3 sentences, then output the complete code.
Please ensure the code can be directly copied and run to demonstrate the complete functionality.

Figure 13: The prompt of generating RL training questions across 16 categories with checklist-based evaluation.

RL: VLM Scoring with Screenshot

Purpose: Vision-Language Model (VLM) scoring for RL reward calculation when HTML rendering succeeds.
Reward Formula:

$$R(y) = I_{rep}(y) \times I_{render}(y) \times (\alpha \cdot S_{chk} + \beta \cdot S_{sim} + \gamma \cdot S_{len})$$

Where:

- I_{rep}: Repetition detection (0 or 1)
- I_{render}: Rendering success (0 or 1)
- S_{chk}: Checklist score (0-5, normalized from 20 items)
- S_{sim}: Similarity = $0.5 \times S_{str} + 0.5 \times S_{sem}$
- S_{len}: Length score (optimal: 12K-16K tokens)
- $\alpha = 0.6$, $\beta = 0.3$, $\gamma = 0.1$

Prompt (with rendered screenshot):
You are an HTML/UI expert. Score the following HTML render based on the requirements and checklist.
User Requirement: {question}
HTML Code: {extracted_html}
Scoring Checklist (20 items, 0-5 points each): {checklist_text}
Based on the rendered image and HTML code, score each item 0-5.
Output ONLY a JSON object with a "scores" array of 20 integers.

Figure 14: The prompt of VLM scoring with rendered screenshots for RL reward calculation.