

WebCompass: Evaluating Multimodal Code Agents in Web Development from Vision, Instruction, and Interaction to Implementation

Xinping Lei^{1,2,†}, Xinyu Che^{1,†}, Junqi Xiong^{1,†}, Chenchen Zhang^{1,†}, Yukai Huang^{1,†},
Chenyu Zhou^{1,†}, Haoyang Huang³, Minghao Liu⁴, Letian Zhu¹, Hongyi Ye¹,
Ken Deng³, Zizheng Zhan³, Han Li¹, Yifan Yao¹, Ming Sun³,
Zhaoxiang Zhang¹, Jiaheng Liu^{1,*}

¹Nanjing University ²Zhongguancun Academy, China

³Kuaishou Technology ⁴2077AI

[†]Equal contribution. ^{*}Corresponding author.

s-lxp26@bza.edu.cn liujiaheng@nju.edu.cn

Abstract

Large language models are increasingly deployed as interactive coding agents for front-end web development, yet existing benchmarks evaluate narrow slices of this workflow—typically text-conditioned generation with static correctness metrics—overlooking visual fidelity, interactivity, and codebase-level reasoning. We introduce **WebCompass**, a benchmark for the front-end development lifecycle spanning three input modalities (text, image, video) and three task types (generation, editing, repair). To match the diverse nature of these tasks, we pair checklist-guided LLM-as-a-Judge for patch-based evaluation with a novel Agent-as-a-Judge protocol that executes generated websites in a real browser and synthesizes targeted test cases. Experiments on ten models reveal a 25-point gap between the best closed- and open-source models, persistent weaknesses in visual quality even among frontier systems, and divergent performance profiles across task types that challenge the notion of a single "best" front-end coding agent.

1 Introduction

Large Language Models (LLMs) have rapidly evolved from passive code assistants into interactive coding agents capable of implementing substantial software changes from natural-language instructions (Yang et al., 2024a; Wang et al., 2024; Cognition AI, 2024). This progress is especially evident in **front-end web development**, where outputs can be directly executed, visually inspected, and iteratively refined. A growing body of work has proposed benchmarks that span different task types and input modalities for front-end web coding (Table 1).

Yet evaluating web coding is fundamentally different from evaluating traditional code generation. Success depends not only on functional correctness, but also on visual fidelity, interaction behavior, responsiveness, accessibility, and overall user

experience. These aspects are difficult to capture with standard code-centric metrics such as pass@k on HumanEval (Chen et al., 2021) or unit-test pass rates on SWE-Bench (Jimenez et al., 2023), which focus on algorithmic correctness or repository-level bug fixing rather than interactive front-end applications.

To address this gap, we introduce **WebCompass**, a unified multimodal benchmark and evaluation framework for front-end web coding. WebCompass spans **text, image, and video** inputs, covers **generation, editing, and repair** tasks, and adopts **task-aware evaluation** tailored to each setting. For editing and repair, we use a *checklist-guided LLM-as-a-Judge* protocol (Zheng et al., 2023), which is well suited to patch-based outputs with constrained solution spaces. For generation, we propose an **Agent-as-a-Judge** protocol (Zhuge et al., 2024), in which an autonomous agent launches the generated website in a real browser, explores it through MCP, synthesizes targeted test cases, and scores the result based on execution.

This design reflects the differing nature of web coding tasks. Editing and repair are localized and checklist-aligned, making diff-level inspection and before/after screenshots sufficient for reliable evaluation. Generation, by contrast, is open-ended and long-horizon, with correctness often depending on multi-step runtime behavior that static inspection cannot capture. By combining multimodal task coverage with execution-based evaluation, WebCompass provides a more realistic and scalable benchmark for assessing web coding agents.

Contributions. (1) **Unified front-end lifecycle coverage.** Unlike prior benchmarks that target isolated tasks or modalities (Table 1), WebCompass jointly evaluates generation, editing, and repair across text, image, and video inputs, enabling cross-task and cross-modality comparisons within a single front-end development workflow. (2) **Rig-**

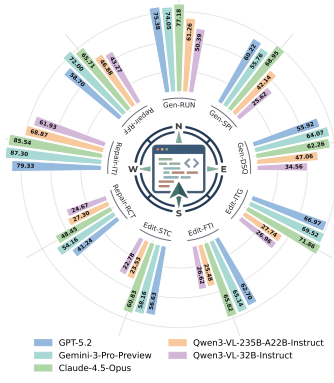


Figure 1: Radar chart of model performance across all seven task types in WebCompass.

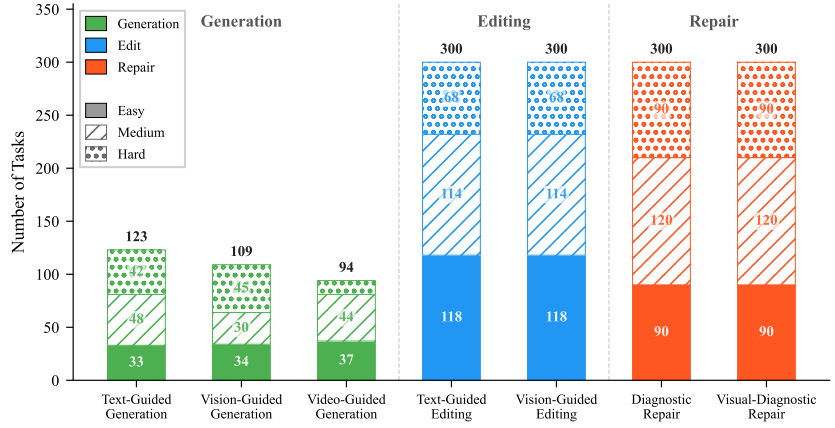


Figure 2: Difficulty distribution of WebCompass.

Benchmark	Size	Gen.	Edit	Rep.	Multi-page	Interact.	Visual	Agentic	Reverse	Input Modality
<i>Generation-Only Benchmarks</i>										
Interaction2Code (Wan et al., 2024)	504	✓	✗	✗	✗	✓	✓	✗	✗	Image
FronTalk (Wu et al., 2025)	1000	✓	✗	✗	✓	✓	✓	✗	✗	Text Image
Web-Bench (Xu et al., 2025)	1000	✓	✗	✗	✓	✓	✗	✗	✗	Text Image
FrontendBench (Zhu et al., 2025)	148	✓	✗	✗	✗	✓	✗	✗	✗	Text
WebApp1K (Cui, 2024)	1000	✓	✗	✗	✓	✗	✓	✗	✗	Text
IWR-Bench (Chen et al., 2025)	113	✓	✗	✗	✓	✓	✓	✗	✗	Video
WebGen-Bench (Lu et al., 2025)	101	✓	✗	✗	✓	✓	✗	✗	✗	Text
<i>Multi-Task Benchmarks</i>										
SWE-bench MM (Jimenez et al., 2023)	517	✗	3	4	✓	✗	✗	✗	✗	Text Image
DesignBench (Xiao et al., 2025)	900	✓	6	6	✓	✗	✓	✗	✗	Image
WebCompass (Ours)	1526	✓	16	11	✓	✓	✓	✓	✓	Text Image Video

Table 1: Comparison with prior web coding benchmarks. WebCompass is the first to support all three task types across text, image, and video modalities. **Gen.**=Generation, **Edit**=Editing, **Rep.**=Repair, **Multi-page**=Project-level multi-page testing, **Interact.**=Interactive functionality evaluation, **Visual**=Visual quality and fidelity evaluation, **Agentic**=Agent-as-a-Judge dynamic testing (using LLM agents to interact with browsers and synthesize tests), **Reverse**=Reverse-engineered deterministic repair tasks. Data sizes are reported as number of tasks or question-answer pairs.

orous and deterministic task construction. We refine underspecified queries into structured design documents for generation, synthesize context-consistent requirements without leaking implementation details for editing, and provide exact search/replace annotations mapping buggy code to clean targets for repair, ensuring reproducible evaluation. (3) **Task-aware evaluation paradigms.** We introduce an Agent-as-a-Judge protocol that combines real-browser interaction with iterative test-case synthesis for open-ended generation tasks, complementing checklist-guided LLM-as-a-Judge for constrained patch-based tasks. Both paradigms evaluate along task-adapted executability, functional, and visual axes with mandatory evidence grounding.

2 WebCompass

WebCompass covers three input modalities—text, image, and video—and three web programming tasks—generation, editing, and repair—yielding seven task categories (Figure 3). For *generation*, models produce a complete web code repository from a textual specification (**Text-Guided**), web-page screenshots (**Vision-Guided**), or a browsing video (**Video-Guided**). For *editing*, the model receives an existing repository with an editing instruction and must output a code patch; **Text-Guided Editing** provides only the instruction, while **Vision-Guided Editing** additionally supplies a screenshot of the current page. For *repair*, the model receives a repository with an issue description and must generate a fixing patch; **Diagnostic Repair** provides only the description, while **Visual-Diagnostic Repair**

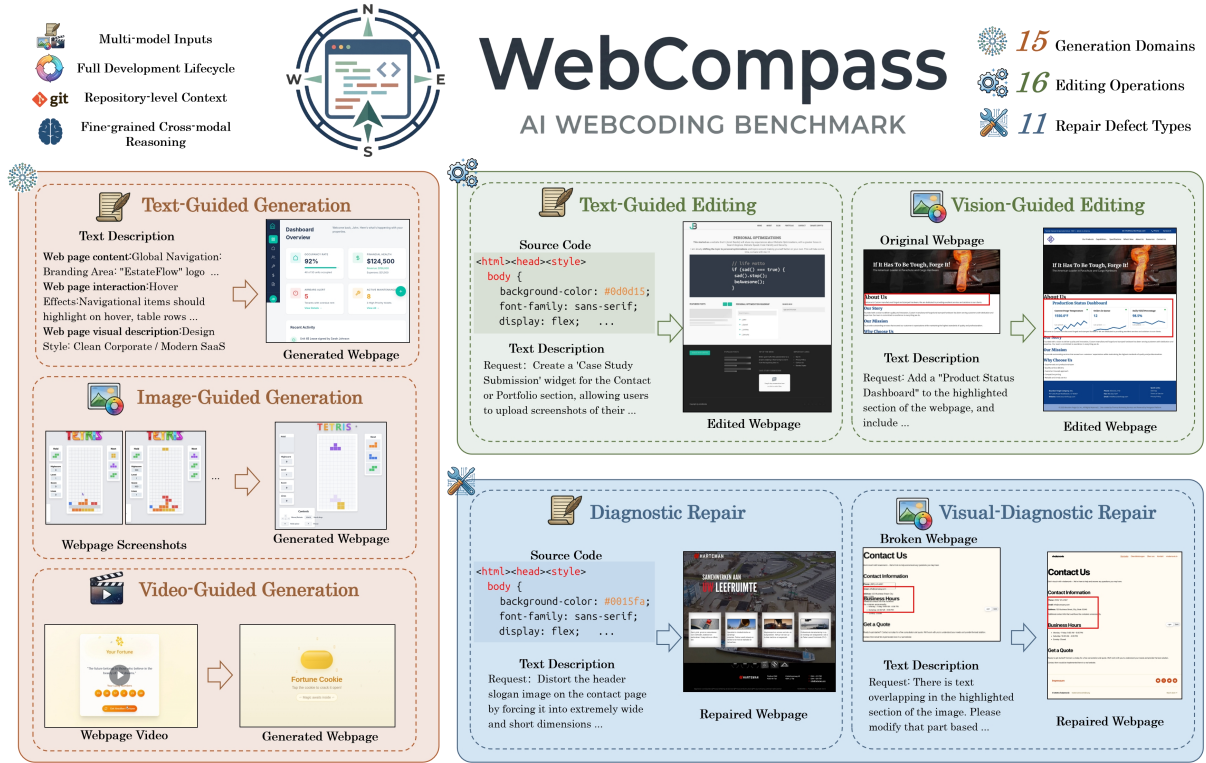


Figure 3: Overview of WebCompass. The benchmark supports three input modalities (text, image, video) and three task types (generation, editing, repair), resulting in seven complementary task categories that cover the front-end development lifecycle.

additionally includes a screenshot of the defective page.

2.1 Data Collection

We construct WebCompass through a multi-stage, human-in-the-loop pipeline (Figure 4); detailed procedures are provided in Appendix A.10.

Generation. For **Text-Guided** generation, we curate diverse web-development requests from multiple sources and expand underspecified ones into structured design specifications covering content, interaction, and appearance. For **Vision-Guided** generation, we collect visually complex webpages and augment them with multi-page screenshots and manually extracted keyframes capturing state changes. For **Video-Guided** generation, we select webpages with rich dynamic behaviors and record interaction videos following pre-planned trajectories.

Editing and Repair. Both families share a pool of high-quality *web prototypes*, obtained through length filtering, automatic quality scoring, and expert curation, then expanded into single- and multi-page variants. For **Editing**, each prototype serves as the source website, paired with new require-

ments specifying *what* to change without prescribing implementation details. For **Repair**, we work in reverse: starting from a clean prototype, we inject observable defects to create faulty source code, paired with an exact inverse edit annotation for deterministic evaluation.

Quality Control. Three layers of assurance are applied: (1) automated validators check compilability, renderability, and patch consistency; (2) LLM-based screening verifies screenshot integrity, instruction clarity, and code–description alignment; and (3) expert human review confirms task correctness, visual quality, and requirement–solution consistency.

2.2 Dataset Statistics

We define a fine-grained taxonomy covering 15 generation domains, 16 editing operation types, and 11 repair defect categories (Table 9; descriptions in Appendix A.14). The benchmark totals 1526 tasks: 123 Text-Guided Generation, 109 Vision-Guided Generation, 94 Video-Guided Generation, 300 Text-Guided Editing, 300 Vision-Guided Editing, 300 Diagnostic Repair, and 300 Visual-Diagnostic Repair. Each task is annotated with a difficulty

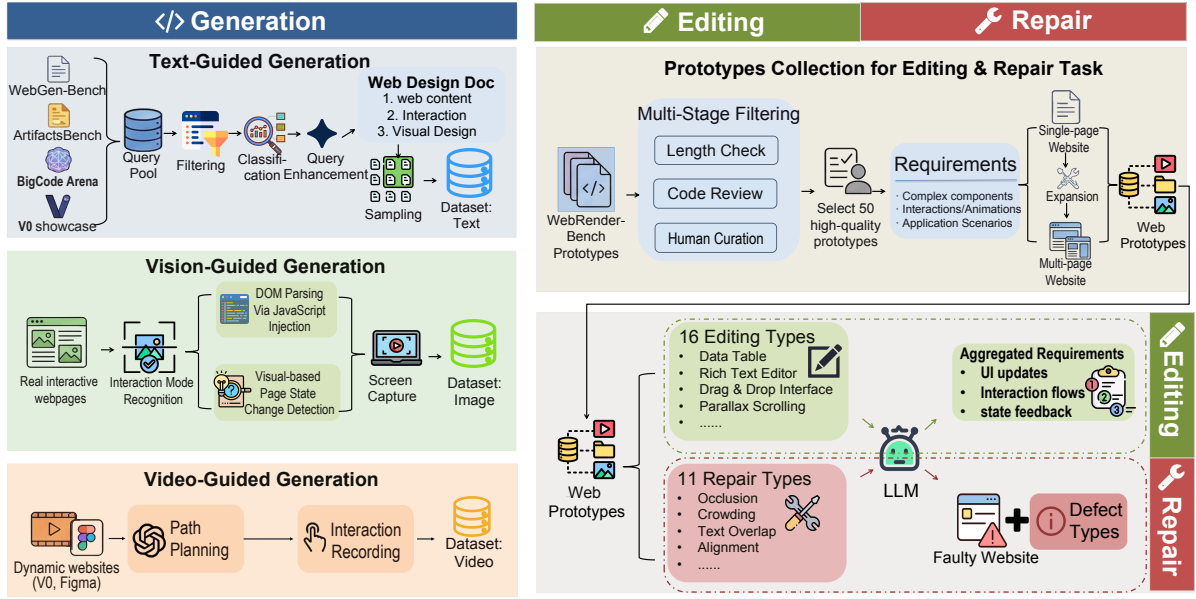


Figure 4: Data construction pipeline for WebCompass. Each task type undergoes source collection, automated filtering, and human curation.

level (Easy, Medium, or Hard); per-category breakdowns are in Table 2.

3 Evaluation Methodology

We adopt task-specific evaluation paradigms tailored to the output characteristics of each task family. For **Editing & Repair**, where models produce localized code patches, we use *LLM-as-a-Judge* (§3.1). For **Generation**, where correctness depends on end-to-end runtime behavior, we use *Agent-as-a-Judge* (§3.2). Both paradigms score along three axes—*executability*, *functional*, and *visual*—whose operationalization is task-dependent. For **Generation**, the dimensions are *Runnability* (build and launch success), *Spec Implementation* (functional behavior matching the design document), and *Design Quality* (visual polish). For **Editing**, they are *Instruction Targeting* (patch applicability and coverage of the requested locations), *Feature Integrity* (preservation of original interactions and functionality of new components), and *Style Conformance* (visual edit fidelity and consistency of unchanged regions). For **Repair**, they are *Root-Cause Targeting* (patch applicability and localization of the defect’s cause), *Interaction Integrity* (preservation of interactions and repair of interactive defects), and *Reference Fidelity* (visual agreement with the ground-truth fixed screenshot). We select the judge model based on highest agreement with human an-

notations (§4.3.1).

3.1 LLM-as-a-Judge for Editing & Repair

For each instance, we apply the predicted patches to the source repository, discard blocks that fail to apply, and launch the modified project in a headless Chromium browser to capture screenshots (Figure 5). The judge LLM receives task-dependent evidence: for **Edit** tasks, the instruction, predicted modifications, and before/after screenshots; for **Repair** tasks, additionally the defect description, ground-truth modifications, and the reference fixed screenshot. The judge scores each checklist item along the three task-specific dimensions (0–10 each), produces evidence-grounded structured JSON output, and aggregates the resulting dimension-wise scores into the final task score.

3.2 Agent-as-a-Judge for Generation

Figure 6 illustrates our *Agent-as-a-Judge* pipeline. A Code Agent, augmented with the Model Context Protocol (MCP) for real-browser control, evaluates each generated website in four stages:

(1) **Checklist generation:** an LLM produces a structured evaluation checklist defining tasks, interaction sequences, expected outcomes, and score values; this checklist remains fixed throughout to prevent circular reasoning.

(2) **Browser interaction:** the agent launches the website in headless Chromium, executes check-

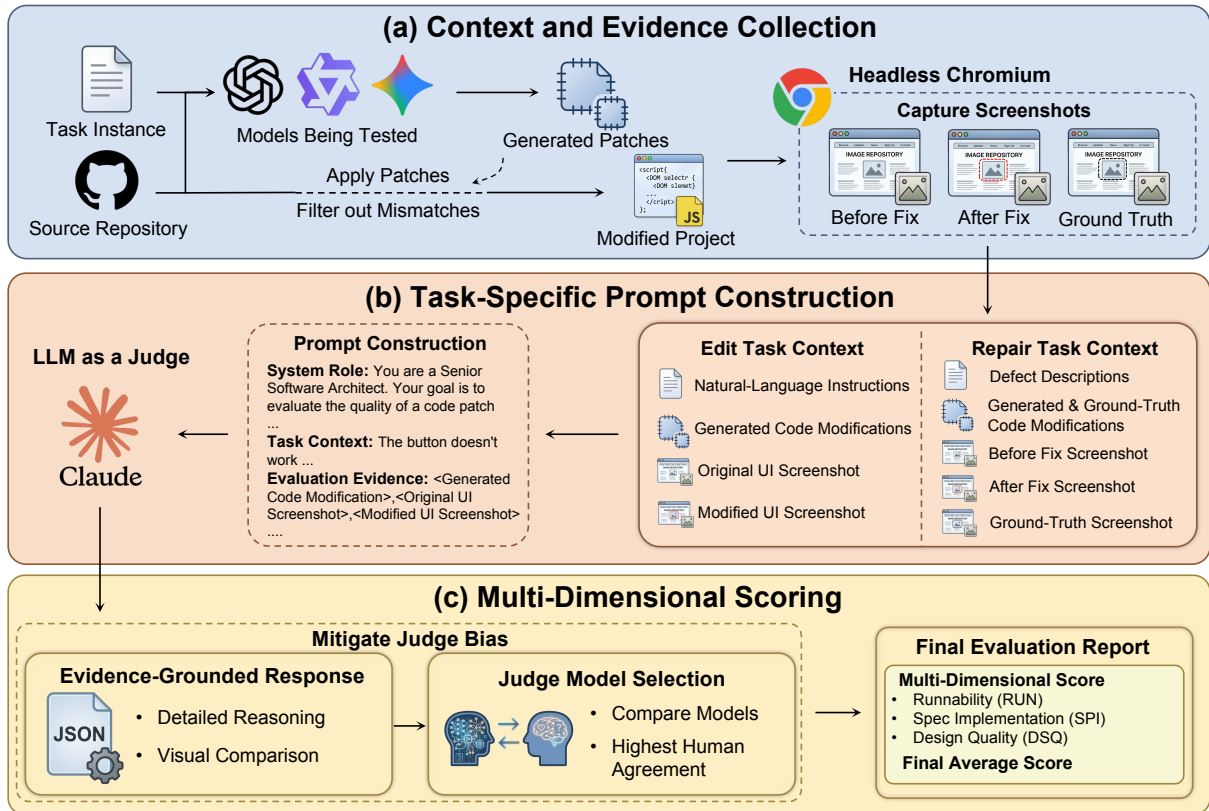


Figure 5: Illustration of the LLM-as-a-Judge evaluation pipeline.

list interactions (clicking, typing, scrolling, navigation), and records DOM snapshots, console logs, and screenshots as auditable evidence.

(3) Adaptive code verification: the agent synthesizes executable JavaScript test cases for each checklist item, programmatically verifying DOM states, CSS properties, and functional behaviors. Crucially, when implementation details differ from expectations, the agent adapts only DOM locators (e.g., element selectors and IDs) while keeping all behavioral assertions unchanged—ensuring that evaluation criteria remain anchored to the original specification rather than drifting toward the model’s output. Failed tests trigger an iterative debugging loop in which the agent inspects the actual code, diagnoses the mismatch, and re-attempts verification before assigning a score.

(4) Evidence-grounded scoring: the agent scores each item along Runnability, Spec Implementation, and Design Quality with structured justifications; scores lacking auditable evidence (screenshots, test results, or console logs) are discarded.

Three safeguards prevent evaluation bias: checklist immutability (no new criteria after Stage 1), selector-only adaptation in Stage 3, and mandatory hard-evidence grounding for every score.

Agent-as-a-Judge protocol. The Agent-as-a-Judge is designed to assess a webpage from the perspective of a normal user. The agent must first reach the relevant state through the browser UI, using ordinary operations such as clicking, typing, scrolling, and navigation. If a required function has no reachable entry point in the rendered UI, the corresponding checklist item is marked as failed (score 0). JavaScript test cases are supplementary verification after the relevant UI state has been reached: they can inspect DOM, CSS, and functional behavior, and can reduce repetitive mechanical operations (for example, repeated swipes in a 2048-style game), but they cannot alter checklist assertions or award credit for functionality that is unreachable through the UI. Code inspection may be used to identify selectors, but selector-level adaptation is the only permitted adaptation; the evaluation criteria and behavioral assertions remain immutable.

Benchmark scope. Our benchmark targets the front-end development lifecycle and front-end web development workflow: generating, editing, and repairing browser-executable interfaces built with HTML, CSS, JavaScript, and front-end frameworks. It does not evaluate back-end capabilities

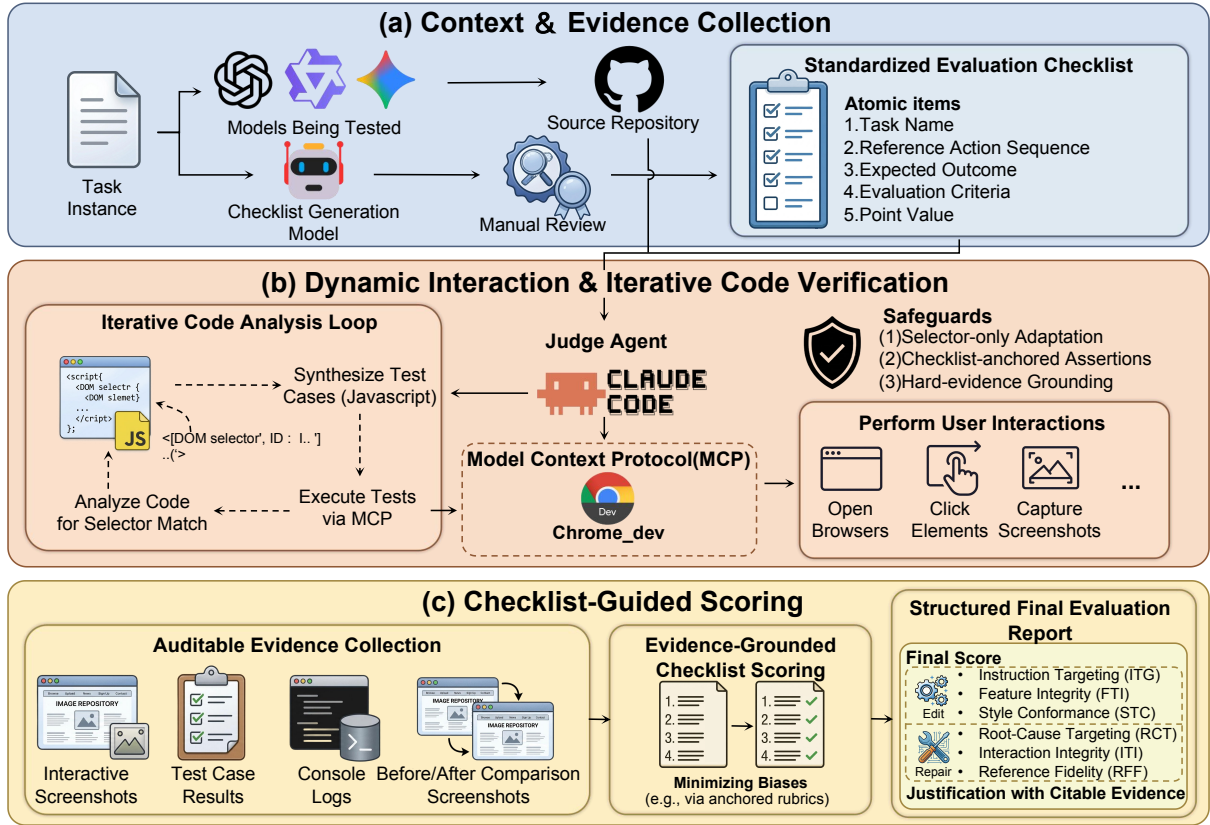


Figure 6: Agent-as-a-Judge evaluation pipeline. The MCP bridge enables bidirectional communication: the agent sends interaction commands to the browser and receives DOM snapshots, console logs, and screenshots as evidence.

such as database design, server-side logic, API development, or deployment. We therefore use precise front-end terminology throughout the paper and do not claim coverage of the full web development lifecycle or full-stack engineering.

Score aggregation. Each checklist item is scored from 0 to its stated maximum and normalized before aggregation. We compute a harmonic mean separately for each task-specific dimension, using a smoothing constant $\epsilon = 1$ for zero-scored items. Dimension scores are reported on a 0–100 scale, and *Overall* is the arithmetic mean of the nine task-specific dimension scores. Detailed aggregation and cascading-failure rules are provided in Appendix A.11.

4 Experiments

This section describes our experimental setup and provides a comprehensive overview of results. We organize experiments by **task type** (generation / editing / repair) and **input modality** (text / image / video). Beyond the main results (§4.2), we report several focused analyses: judge model selection (§4.3.1), framework-based subset evaluation

(§4.3.2), difficulty-level analysis (§A.11.2), and impact of thinking mode (§A.13.2).

4.1 Evaluated LLMs and Frameworks

We evaluate ten models from both closed-source and open-source families. All selected models natively support text, image, and video inputs, allowing us to use the same model set across all modalities. Full model details are provided in Appendix A.9.

We employ **Claude Code** (v2.0.67) as the evaluation orchestrator and the **Chrome DevTools MCP Server** (v0.19.0) for browser rendering, DOM inspection, and automated interaction verification in a headless Chromium environment. This setup enables an agent-based evaluation pipeline that programmatically assesses the functional correctness and UI consistency of generated front-end web applications.

4.2 Main Results

Table 2 presents the overall and per-task-type scores for all evaluated models.

Several key patterns emerge.

Model ranking and the closed–open gap.

Table 2: Comparison of models across different task types. **Green bold** indicates the best score in each column; **blue underline** indicates the second best. Each task has three evaluation dimensions: Generation uses Runnability (RUN), Spec Implementation (SPI), and Design Quality (DSQ); Editing uses Instruction Targeting (ITG), Feature Integrity (FTI), and Style Conformance (STC); Repair uses Root-Cause Targeting (RCT), Interaction Integrity (ITI), and Reference Fidelity (RFF). Overall is the arithmetic mean of all nine dimension scores.

Models	Generation			Editing			Repair			Overall
	RUN.	SPI.	DSQ.	ITG.	FTI.	STC.	RCT.	ITI.	RFF.	
Closed-Source Large Language Models										
Claude-Opus-4.5	77.18	68.95	62.26	71.86	65.82	60.83	48.45	85.54	65.71	67.40
Gemini-3-Pro-Preview	74.05	55.76	64.07	69.52	65.14	58.16	54.16	87.30	72.00	66.68
Gemini-3-Flash-Preview	74.87	54.32	62.42	65.95	62.35	57.21	53.18	86.84	71.65	65.42
GPT-5.2	75.38	60.22	55.92	66.97	62.70	56.63	41.24	79.33	58.70	61.90
Claude-Sonnet-4.5	65.30	50.37	56.78	60.06	53.71	45.51	40.44	80.63	61.31	57.12
Qwen3-VL Series Open-Source Large Language Models										
235B-A22B-Instruct	61.26	42.14	47.06	27.74	25.48	23.53	27.30	68.87	46.88	41.14
235B-A22B-Thinking	63.86	35.02	45.21	22.15	21.67	19.06	27.02	68.74	46.28	38.78
30B-A3B-Instruct	41.79	20.80	29.28	20.57	20.97	17.93	19.32	50.71	31.35	28.08
30B-A3B-Thinking	47.37	20.87	37.47	19.82	21.21	18.20	18.08	51.85	31.31	29.58
32B-Instruct	50.39	25.62	34.56	26.96	26.62	22.78	24.67	61.93	43.27	35.20

Claude-Opus-4.5 and Gemini-3-Pro-Preview achieve the highest Overall scores (67.40 and 66.68, respectively) with complementary strengths: Claude leads Generation RUN (77.18) and Editing ITG (71.86), while Gemini leads Repair RCT (54.16) and Repair RFF (72.00). The closed–open gap is substantial: the best open-source model (Qwen3-VL-235B-A22B-Instruct) reaches an Overall of 41.14, trailing the top closed-source model by over 26 points. Smaller open models (30B variants) fall further, reaching less than half the top closed-source scores.

Task-type patterns. For closed-source models, Generation and Editing consistently follow the ordering executability > functional > visual (e.g., Claude-Opus-4.5: RUN 77.18 > SPI 68.95 > DSQ 62.26 on Generation). Repair shows a different pattern: ITI >> RFF > RCT (e.g., Gemini-3-Pro-Preview: 87.30 > 72.00 > 54.16). This ordering is explained by the task structure: Interaction Integrity trends high because 9 of 11 defect types are visual or semantic—the interactive layer is rarely affected, so preservation is nearly automatic; the 2 interactive-class defects (Loss of Interactivity, Missing Attributes) are localized enough that models can usually repair them. Reference Fidelity is

mid-range because matching the gold fixed screenshot is nontrivial. Root-Cause Targeting is lowest because correctly locating the defect’s root cause without introducing new errors remains the hardest part of repair. Note that the functional and visual axes measure different capabilities across tasks: Editing’s Feature Integrity tests both preservation and new-component functionality, whereas Repair’s Interaction Integrity is primarily regression safety; similarly, Editing’s Style Conformance evaluates edit outcome fidelity, while Repair’s Reference Fidelity measures closeness to a gold reference. Editing is especially challenging for open-source models, where scores fall to 18–28 across dimensions, revealing a major gap in context-aware code modification.

Visual quality as the persistent bottleneck. Across all ten models, the visual dimension is the lowest-scoring axis in Generation and Editing (Design Quality and Style Conformance, respectively). Even Gemini-3-Pro-Preview, the strongest model on this axis, reaches only 64.07 on Generation DSQ. The gap is wider for weaker models and consistent across task types. Notably, Gemini-3-Pro-Preview and Gemini-3-Flash-Preview outperform GPT-5.2 on the visual axis despite comparable executability

scores, indicating that visual fidelity and functional correctness do not scale in lockstep.

4.3 Further Analysis

4.3.1 Judge Model Selection

To validate automated evaluation reliability, we compare three Claude-family judge models (Opus-4.5, Sonnet-4.5, Haiku-4.5) against human judgments on a 200-sample subset. As shown in Table 8, Claude-Opus-4.5 achieves the highest human agreement (Pearson r of 0.93–0.96 across tasks), and is adopted as the default judge. Notably, all judges show higher agreement on edit/repair tasks than on generation, consistent with the more constrained solution space of patch-based tasks. As shown in Figure 19, a comparison of full model rankings between the agent-based evaluator and human annotators further confirms strong alignment, with most rank differences being zero or at most one, validating the automatic evaluation protocol as a reliable proxy for human judgment.

4.3.2 Subset Evaluation on Different Front-End Frameworks

To assess how framework choice affects model performance, we evaluate four representative models (GPT-5.2, Gemini-3-Pro-Preview, Claude-Opus-4.5, Qwen3-VL-235B-A22B-Instruct) on a 180-task subset (60 per task category), each completed in **React**, **Vue**, and **Vanilla** (plain HTML/CSS/JS). Figure 7 presents the overall scores; per-dimension breakdowns are in Appendix A.9.

Three key findings emerge. **(1) Vanilla dominates Generation and Editing, but not Repair.** Across all four models, framework-free code consistently yields the highest scores in Generation and Editing. In Repair, however, the Vanilla advantage diminishes: for instance, GPT-5.2 achieves its best Repair score with React. We attribute this to a structural difference between task types: Generation and Editing require producing substantial new code, where Vanilla’s absence of build toolchains, framework-specific syntax (e.g., JSX, template directives), and component lifecycle conventions reduces the surface area for errors. Repair, by contrast, demands precise localization and modification of existing defects, where React’s explicit component boundaries and unidirectional data flow may help models isolate faulty code regions more effectively than unstructured Vanilla codebases. **(2) Vue consistently underperforms.** Vue yields the lowest scores in the majority of model–task combina-

tions. A plausible contributing factor lies in Vue’s single-file component (SFC) format, which interleaves three heterogeneous syntax modes—HTML-like templates with custom directives (`v-if`, `v-for`, `@click`), JavaScript/TypeScript logic, and scoped CSS—within a single file. This demands simultaneous coordination across markup, logic, and styling, increasing the likelihood of cross-block inconsistencies. By comparison, React’s JSX keeps rendering logic within standard JavaScript, and Vanilla avoids framework abstractions entirely. **(3) Open-source models share the same framework sensitivity pattern** (peaking on Vanilla for Generation/Editing) but at a uniformly lower performance ceiling, suggesting that the observed framework preferences are primarily driven by inherent task–framework interactions rather than model-specific factors.

4.3.3 Error Analysis

To understand not only how well but also *how* LLMs fail, we design a structured error analysis framework that classifies every point deduction into a two-level taxonomy spanning four domains (Code Execution, Functional, Visual/Style, and Non-Functional) with fifteen fine-grained error types, and further attributes each error to a root cause. Full taxonomy definitions, the classification prompt, and the decision flowchart are provided in Appendix A.15.1; detailed per-model and per-modality error distributions are in Appendix A.13.

Findings. Error analysis reveals that the primary bottlenecks for current models in web generation are Feature Missing and Resource Fail, which together account for 40%–55% of all errors across most models. Lower-ranked models (e.g., Qwen3-VL-30B-A3B) accumulate over 5,000 total errors dominated by fundamental issues like missing functionality and console errors, whereas top-performing models (e.g., Claude-Opus-4.5) achieve the lowest error counts (2,100) with their remaining errors shifting toward finer-grained layout and visual styling issues. Across input modalities, distinct error patterns emerge: text-conditioned generation is overwhelmingly limited by Feature Missing errors (40%–66%), indicating that models struggle most with translating natural language specifications into interactive logic; image-conditioned generation shifts markedly toward layout, color, and visual fidelity errors, exposing weaknesses in pixel-level reproduction; and video-conditioned generation exhibits a roughly balanced mix of functional

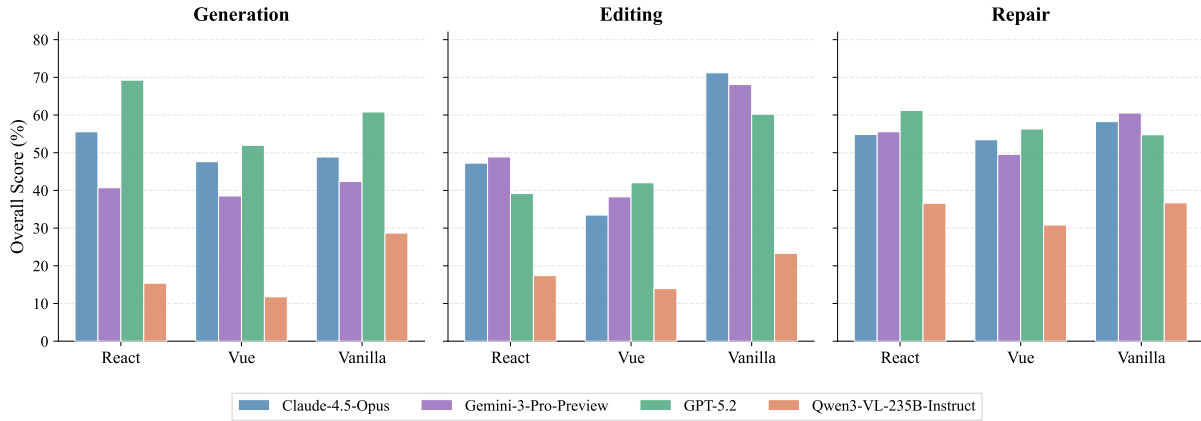


Figure 7: Overall scores across front-end frameworks for four representative models on Generation, Editing, and Repair tasks. Scores are computed as the harmonic mean of the three per-task evaluation dimensions.

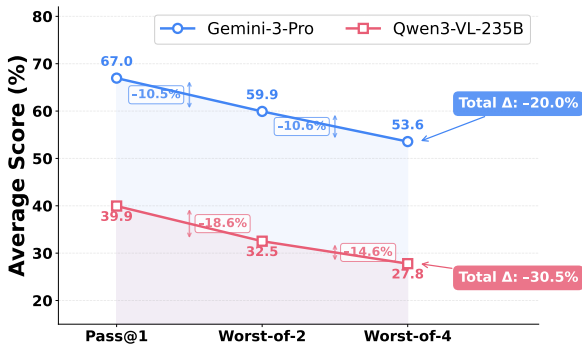


Figure 8: Consistency & Stability: Score Degradation under Worst-of-N

and visual errors, reflecting the compound challenge of simultaneously understanding temporal interaction sequences and faithfully reproducing static appearance, making it the most demanding modality.

4.3.4 Stability Analysis: Worst-of-N Evaluation

Pass@1 reflects average-case capability but may mask output inconsistency. We adopt the Worst-of- n (W@ n) protocol, sampling $n = 4$ independent generations per task and reporting the minimum score to capture the realistic lower bound of model performance.

As shown in Figure 8, both models degrade monotonically from Pass@1 to W@4, but at different rates. **Gemini-3-Pro-Preview** retains $\sim 80\%$ of its Pass@1 performance at W@4 ($66.96 \rightarrow 53.56$), with all dimensions remaining above 49%. **Qwen3-VL-235B-A22B-Instruct** degrades more sharply, retaining only $\sim 69.5\%$ ($39.95 \rightarrow 27.78$), with W@4 scores in the Edit category falling below 16% on

Instruction Targeting—indicating near-complete failure in worst-case scenarios.

These results reveal that Gemini’s advantage extends beyond higher average scores to *greater output consistency*. Since users typically rely on a single generation rather than selecting from multiple samples, output stability remains a critical open challenge for frontier models in front-end code generation. Detailed per-task and per-dimension breakdowns are provided in Appendix A.12.

5 Conclusion

We presented **WebCompass**, a multimodal benchmark for the front-end development lifecycle that unifies generation, editing, and repair across text, image, and video modalities. Its task-aware evaluation combines LLM-as-a-Judge and Agent-as-a-Judge protocols to assess executability, functional behavior, and visual quality through task-specific rubrics. Our experiments reveal that closed-source models lead by ~ 25 points over the best open-source alternatives, visual quality remains a persistent bottleneck even for frontier models, and generation, editing, and repair stress fundamentally different capabilities—no single model dominates all three. These findings suggest that advancing front-end coding agents requires not only stronger functional reasoning but also deeper visual design understanding and greater output consistency.

Limitations

While WebCompass represents a substantial step toward comprehensive evaluation of web coding agents, we acknowledge several limitations that future work should address.

Benchmark scope. WebCompass is intentionally scoped to the front-end development lifecycle and front-end web development workflow (HTML, CSS, JavaScript, and front-end frameworks). It does not evaluate back-end capabilities such as database design, server-side logic, API development, or deployment workflows, so its claims should not be interpreted as coverage of full-stack engineering. Extending the benchmark with back-end tasks would provide a broader assessment in future work.

Structured queries vs. creative intent. For generation tasks, we deliberately refine underspecified user queries into structured web design documents (specifying content, interaction, and visual appearance) to enable reproducible, automated evaluation. This design choice means that our benchmark primarily tests *instruction-following* capability rather than the ability to interpret vague, creative intent. We acknowledge this as an inherent trade-off, WebCompass prioritizes *deterministic evaluation standards over open-ended creative assessment*. Complementary benchmarks that explicitly measure creative divergence (e.g., via human preference ranking) would provide a valuable additional perspective.

Limited real-time interaction with dynamic web pages. Our evaluation protocols currently cannot perform real-time interaction with highly dynamic web pages—such as browser-based games or applications with frequent state transitions—in the way a human would. While our framework supports both natural interactions and script-based inspection, it remains challenging to keep pace with rapidly evolving page states, making it difficult to accurately assess time-sensitive behaviors such as real-time game logic, continuous animation responses, or state transitions that depend on precise timing. As a result, the evaluation quality for such highly dynamic web pages may not fully reflect their actual functionality and user experience.

Static benchmark and contamination risk. As a static benchmark, WebCompass is susceptible to data contamination if future models are trained on data that includes our tasks or similar web pages. While we mitigate this through diverse data sources and original task synthesis, maintaining a contamination-free evaluation over time may require periodic updates or dynamic task generation.

Evaluation cost. The Agent-as-a-Judge protocol, while more thorough than static evaluation, is computationally expensive. Each generation task requires launching a headless browser, executing multi-step interaction sequences, and synthesizing iterative test cases, which significantly increases evaluation time and cost compared to simpler metrics. This may limit the benchmark’s accessibility for resource-constrained research groups.

Ethical Considerations

Data licensing and source attribution. The web page data used for **generation tasks** in WebCompass is collected from publicly accessible user communities on two platforms: **Figma Community** and **v0.dev** (by Vercel).

Figma Community files are published under the **Creative Commons Attribution 4.0 International (CC BY 4.0)** license, which explicitly permits sharing and adaptation for any purpose—including research and commercial use—provided that appropriate credit is given to the original creators.¹ For v0.dev, the Vercel AI Product Terms state that content posted publicly by users—including outputs, chats, community templates, and code—“may be viewed and copied by others,” and users retain ownership of their content.² Both platforms therefore permit browsing, viewing, and reproducing publicly shared community content.

The web page data used for **editing and repair tasks** is derived from existing open-source datasets rather than directly collected from these platforms.

WebCompass is released under the **CC BY 4.0** license, which is compatible with the licensing terms of both source platforms. We provide attribution to original creators in accordance with the respective license requirements.

PII and offensive content screening. All collected web pages underwent manual review to filter out personally identifiable information (e.g., real names, email addresses, phone numbers) and offensive or discriminatory content. Flagged instances were excluded from the final dataset.

Annotator recruitment and compensation. The three human annotators were professional front-end developers recruited through internal contacts. They were paid approximately \$25 USD per

¹<https://help.figma.com/hc/en-us/articles/360042296374-Figma-Community-copyright-and-licensing>

²<https://vercel.com/legal/ai-product-terms>

hour (computed from mean annotation time of ~ 8 min/sample) and were briefed on the research purpose before starting.

Use of AI assistants. We used Claude Code (Anthropic) as the evaluation orchestrator in our Agent-as-a-Judge pipeline. During writing, AI assistants were used for grammar checking and LaTeX formatting. All scientific claims, experimental design, data analysis, and conclusions were produced by the human authors.

Acknowledgments

This work was supported by the Jiangsu Science and Technology Project (BK20251199). This work was supported in part by the National Natural Science Foundation of China (No. 62506161).

References

- Anthropic. 2025. Claude. <https://claude.ai/>. Accessed: 2026-03-15.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Rabiul Awal, Mahsa Massoud, Aarash Feizi, Zichao Li, Suyuchen Wang, Christopher Pal, Aishwarya Agrawal, David Vazquez, Siva Reddy, Juan A Rodriguez, and 1 others. 2025. Webmmu: A benchmark for multimodal multilingual website understanding and code generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 25129–25156.
- Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, and 1 others. 2025. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*.
- Tony Beltramelli. 2017. pix2code: Generating code from a graphical user interface screenshot. *arXiv preprint arXiv:1705.07962*.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Yang Chen, Minghao Liu, Yufan Shen, Yunwen Li, Tianyuan Huang, Xinyu Fang, Tianyu Zheng, Wenxuan Huang, Cheng Yang, Daocheng Fu, and 1 others. 2025. Iwr-bench: Can llms reconstruct interactive webpage from a user interaction video? *arXiv preprint arXiv:2509.24709*.
- Cognition AI. 2024. Introducing devin, the first ai software engineer. <https://www.cognition.ai/blog/introducing-devin>. Accessed: 2025-01-15.
- Yi Cui. 2024. Webapp1k: A practical code-generation benchmark for web app development. *arXiv preprint arXiv:2408.00019*.
- Wentao Ge, Shunian Chen, Guiming Hardy Chen, Junying Chen, Zhihong Chen, Nuo Chen, Wenya Xie, Shuo Yan, Chenghao Zhu, Ziyue Lin, and 1 others. 2023. Mllm-bench: evaluating multimodal llms with per-sample criteria. *arXiv preprint arXiv:2311.13951*.
- Gemini Team and Google. 2023. **Gemini: A family of highly capable multimodal models**. *Preprint*, arXiv:2312.11805.
- Google. 2025. Gemini. <https://gemini.google.com/>. Accessed: 2026-03-15.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shitong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.
- Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J Yang, JH Liu, Chenchen Zhang, Linzheng Chai, and 1 others. 2024. Opencoder: The open cookbook for top-tier code large language models. *arXiv preprint arXiv:2411.04905*.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Live-codebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.
- Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. 2024. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 881–905.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

- Zhiyu Lin, Zhengda Zhou, Zhiyuan Zhao, Tianrui Wan, Yilun Ma, Junyu Gao, and Xuelong Li. 2025. Webuibench: a comprehensive benchmark for evaluating multimodal large language models in webui-to-code. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 15780–15797.
- Chenxu Liu, Yingjie Fu, Wei Yang, Ying Zhang, and Tao Xie. 2026. Webcoderbench: Benchmarking web application generation with comprehensive and interpretable evaluation metrics. *arXiv preprint arXiv:2601.02430*.
- LMSYS Org. 2024. WebDev Arena: Benchmarking LLMs on Web Development. <https://web.lmarena.ai/leaderboard/webdev>. Accessed: 2025-01-15.
- Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. 2025. Webgenbench: Evaluating llms on generating interactive and functional websites from scratch. *arXiv preprint arXiv:2505.03733*.
- OpenAI. 2025. Openai. <https://openai.com/>. Accessed: 2026-03-15.
- Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 2024. Design2code: Benchmarking multimodal code generation for automated front-end engineering. *arXiv preprint arXiv:2403.03163*.
- Haoyu Sun, Huichen Will Wang, Jiawei Gu, Linjie Li, and Yu Cheng. 2025. Fullfront: Benchmarking mllms across the full front-end engineering workflow. *arXiv preprint arXiv:2505.17399*.
- Yuxuan Wan, Jingyu Xiao, Man Ho Lam, Junliang Liu, Yintong Huo, and Michael R. Lyu. 2024. Interaction2code: Benchmarking mllm-based interactive webpage code generation from interactive prototyping. *arXiv preprint arXiv:2411.03292*.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.
- Xueqing Wu, Zihan Xue, Da Yin, Shuyan Zhou, Kai-Wei Chang, Nanyun Peng, and Yeming Wen. 2025. Frontalk: Benchmarking front-end development as conversational code generation with multi-modal feedback. *arXiv preprint arXiv:2601.04203*.
- Jingyu Xiao, Ming Wang, Man Ho Lam, Yuxuan Wan, Junliang Liu, Yintong Huo, and Michael R. Lyu. 2025. Designbench: A comprehensive benchmark for mllm-based front-end code generation. *arXiv preprint arXiv:2506.06251*.
- Kai Xu, YiWei Mao, XinYi Guan, and ZiLong Feng. 2025. Web-bench: A llm code benchmark based on web standards and frameworks. *arXiv preprint arXiv:2505.07473*.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.
- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Liber, Karthik Narasimhan, and Ofir Press. 2024a. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*.
- John Yang, Carlos E Jimenez, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024b. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*.
- Sukmin Yun, Haokun Lin, Rusiru Thushara, Mohammad Qazim Bhat, Yongxin Wang, Zutao Jiang, Mingkai Deng, Jinhong Wang, Tianhua Tao, Junbo Li, and 1 others. 2024. Web2code: A large-scale webpage-to-code dataset and evaluation framework for multimodal llms. *arXiv preprint arXiv:2406.20098*.
- Chenchen Zhang, Yuhang Li, Can Xu, Jiaheng Liu, Ao Liu, Changzhi Zhou, Ken Deng, Dengpeng Wu, Guanhua Huang, Kejiao Li, and 1 others. 2025. Artifactsbench: Bridging the visual-interactive gap in llm code generation evaluation. *arXiv preprint arXiv:2507.04952*.
- Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and 1 others. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623.
- Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, and 1 others. 2023. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*.
- Hongda Zhu, Yiwen Zhang, Bing Zhao, Jingzhe Ding, Siyao Liu, Tong Liu, Dandan Wang, Yanan Liu, and Zhaojian Li. 2025. Frontendbench: A benchmark for evaluating llms on front-end development via automatic evaluation. *arXiv preprint arXiv:2506.13832*.
- Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, and 1 others. 2024. Agent-as-a-judge: Evaluate agents with agents. *arXiv preprint arXiv:2410.10934*.

A Appendix

A.1 Related Work

Our work is closely related to two research threads: (i) **code-capable foundation models and agentic coding systems**, and (ii) **benchmarks and evaluation frameworks for web development** that require judging both visual quality and interactive correctness.

Code LLMs and code agents. Large language models for code generation have progressed rapidly from early program synthesis benchmarks (Austin et al., 2021; Chen et al., 2021) and competition-level reasoning (Li et al., 2022) to fully interactive coding agents capable of autonomous software engineering. On the model side, both proprietary systems— Gemini-3-Pro (Gemini Team and Google, 2023), Claude-Opus-4.5 (Anthropic, 2025)—and open-source alternatives— Qwen3-Coder (Yang et al., 2025) and OpenCoder (Huang et al., 2024)—have demonstrated strong performance on standard code generation benchmarks such as HumanEval (Chen et al., 2021) and LiveCodeBench (Jain et al., 2024). On the agent side, SWE-agent (Yang et al., 2024a) and OpenHands (Wang et al., 2024) equip LLMs with tool-use interfaces for repository-level software engineering, while commercial platforms such as Devin (Cognition AI, 2024) and Cursor demonstrate the practical viability of agentic coding workflows. SWE-bench (Jimenez et al., 2023) has become the de facto evaluation framework for these agents, driving rapid progress in automated bug fixing and code editing. Nevertheless, web development introduces a distinct challenge compared to algorithmic programming or repository-level repair: success is ultimately reflected in the *user-facing artifact*—layout fidelity, design aesthetics, responsiveness, interaction logic, state transitions, and accessibility. These criteria are difficult to capture with purely code-based metrics and can be missed by evaluation suites designed primarily for functional correctness.

Benchmarks for web coding. Existing web-coding benchmarks can be categorized along two orthogonal axes: *task type* and *input modality*.

From the task perspective, benchmarks often study:

- **Generation:** producing web pages or mini-apps from requirements. This ranges from

early UI-to-code work such as pix2code (Beltramelli, 2017) and Web2Code (Yun et al., 2024), to more recent benchmarks including Design2Code (Si et al., 2024) for screenshot-to-HTML conversion, WebGen-Bench (Lu et al., 2025) for interactive website generation from scratch, DesignBench (Xiao et al., 2025) for MLLM-based front-end code generation, and Web-Bench (Xu et al., 2025) for evaluating code against web standards and frameworks. Interaction2Code (Wan et al., 2024) further extends the modality to interactive prototypes, while IWR-Bench (Chen et al., 2025) and FronTalk (Wu et al., 2025) explore video-conditioned and conversational generation settings, respectively.

- **Editing:** modifying an existing codebase to satisfy new requirements. SWE-bench Multimodal (Yang et al., 2024b) extends the original SWE-bench (Jimenez et al., 2023) to visual software domains, requiring models to interpret screenshots alongside issue descriptions.
- **Repair:** fixing defects in UI/UX or broken interactions, ranging from text-described bugs to visually grounded defect descriptions.

Several recent efforts aim at holistic, multi-dimensional coverage. WebUIBench (Lin et al., 2025) benchmarks WebUI-to-code generation with comprehensive metrics; FullFront (Sun et al., 2025) spans the full front-end engineering workflow; ArtifactsBench (Zhang et al., 2025) bridges the visual-interactive gap in LLM code generation evaluation; and WebDev Arena (LMSYS Org, 2024) provides a human-preference-based leaderboard for web development. WebCoderBench (Liu et al., 2026) proposes comprehensive and interpretable evaluation metrics, while WebMMU (Awal et al., 2025) extends coverage to multilingual website understanding.

Despite this growing body of work, most existing benchmarks focus on a single task type (typically generation) or a single input modality (typically text or static images), and their evaluations often rely on either weak proxies (e.g., single screenshot similarity or DOM heuristics) or brittle scripted tests that require strict attribute conventions. WebCompass addresses these limitations by spanning three modalities, three task types, and employing execution-grounded evaluation that tests end-to-end runtime behavior.

Evaluation paradigms for interactive visual artifacts. Evaluation methods for web-facing artifacts commonly fall into three classes:

- **Rule-/test-based evaluation.** Deterministic test suites, as employed by SWE-bench (Jimenez et al., 2023) and Web-Bench (Xu et al., 2025), provide precise and reproducible verdicts but typically require heavy instrumentation, strict naming conventions, and substantial engineering effort to achieve good coverage across diverse implementations.
- **Agent-based interaction.** Web agents—as pioneered by WebArena (Zhou et al., 2023) and extended to multimodal settings in VisualWebArena (Koh et al., 2024)—can explore an artifact by interacting with the page and checking outcomes. However, coverage remains challenging: predefined action spaces may miss complex behaviors, and long-horizon workflows are hard to validate end to end.
- **LLM/MLLM-as-a-Judge.** Language or multimodal referees (Zheng et al., 2023; Ge et al., 2023) can scale to open-ended designs and assess multiple dimensions jointly, but may be subjective without careful rubric design and evidence grounding.

Our work adopts a task-aware combination of these paradigms. For **editing** and **repair**, we use *checklist-guided* LLM-as-a-Judge (Zheng et al., 2023) to anchor evaluation in per-task, fine-grained criteria with evidence grounding. For **generation**, where acceptable solutions are diverse and interactivity is open-ended, we introduce an *Agent-as-a-Judge* protocol (Zhuge et al., 2024) that combines browser-based interaction (via the Model Context Protocol) with iterative test-case synthesis, providing stronger and more realistic validation than any single evaluation paradigm alone.

A.2 Case Study: Full Evaluation Pipeline Walkthrough

We illustrate the Agent-as-a-Judge protocol on Task #648 (*Weather Dashboard*)—a front-end application featuring dynamic backgrounds, autocomplete search, unit conversion, responsive layout, and glassmorphism styling.

Step 1: Setup

The agent receives the source repository (index.html, app.js, styles.css, weather-data.js), a fixed checklist of 12 items spanning three dimensions (100 pts total), and Chrome DevTools MCP tools. Its task is to fill in the score and reason for every item.

Dimension	Items	Pts	Focus
Runnability	1	10	Error-free loading, no console/network fail
Spec Implementation	2–9	62	Functional correctness of specified UI behaviors
Design Quality	10–12	28	Visual quality, design system, styling

Step 2: Agent Trajectory (86 tool calls)

The full evaluation took 86 tool calls across four phases. For each checklist item, the agent follows: *record initial state* → *interact* → *capture evidence* → *verify outcome* → *write score*.

Phase I–II: Reconnaissance & Launch. The agent inspects the project structure, reads all source files in parallel, starts a local HTTP server, and navigates to the page.

Phase III: Per-Item Evaluation. Selected examples illustrating different evidence types:

- **Item 1 (Page Load).** `take_screenshot` + `list_console_messages` + `list_network_requests` reveal a 404 on `favicon.ico` → **7/10** (resource 404, −3).
- **Item 3 (7-Day Time Travel).** Hover and click the Friday card; the hero updates from “Light Rain” to “Extreme Heat”; “Return to Now” reverts correctly → **12/12**.
- **Item 5 (Unit Toggle °C↔°F).** Baseline snapshot shows 18°C/27°/12°. After toggling, the hero converts correctly to 64°F/81°F/54°F, but a second snapshot reveals hourly and weekly forecasts do not convert → **4/8** (−4).
- **Item 7 (Error Handling).** Submitting “Xyz123” yields no modal, no alert, no feedback. `evaluate_script` confirms an error modal exists in the DOM but is never shown → **0/7** (no feedback, −7).
- **Item 4 (Dynamic Background).** `evaluate_script` extracts the theme/condition state across two cities; the agent notes the app uses browser local time

rather than the location’s timezone → **8/10** (−2).

- **Items 2, 6, 8, 9** (autocomplete search, mobile responsiveness, metrics grid, hourly scroll) pass cleanly. **Items 10–12** (Design Quality) are scored against a full-page screenshot.

Phase IV: Verification. A final grep confirms no score: null remains in the checklist.

Step 3: Final Scores

Dim.	Max	Score	%	Main Deductions
Runnability	10	7	70%	favicon 404
Spec Implementation	62	54	87%	unit toggle (−4); no error feedback (−7); local-time (−2)
Design Quality	28	28	100%	—
Total	100	84	84%	

Step 4: Tool Usage Summary

The 86 calls break down as: browser interaction 13 (click/fill/hover/key), browser observation 26 (screenshot/snapshot/evaluate_script/console/network), browser control 5 (navigate/resize/wait), file I/O 27, shell 5, progress tracking 10. Observation calls outnumber interaction calls roughly 2:1, reflecting the protocol’s emphasis on *evidence grounding* over action volume.

Takeaways. This case illustrates three properties of the protocol: (i) **multi-source evidence**—a single item can require screenshots, DOM snapshots, console logs, and synthesized scripts to verify (Items 4, 5, 7); (ii) **partial-credit fidelity**—the unit toggle works for the hero but not for forecasts, and the score reflects exactly that (4/8, not 0 or full); (iii) **deduction transparency**—every point lost is anchored to an auditable reason.

A.3 Evaluation Dimension Scope

Design principle. Top-level dimensions should be *unified, stable, and cross-task comparable*; finer-grained attributes are surfaced through the error taxonomy rather than promoted to top-level dimensions, which would dilute reliability.

Task-adapted dimensions. The benchmark preserves three stable axes—executability, functional behavior, and visual quality—while naming their operational dimensions according to the task family. This design maps to the user-perceived web

outcome chain and can be measured reliably under our automated protocol (Pearson $r \in [0.93, 0.96]$, §4.3.1) without implying that the same evidence is required for every task:

- **Generation:** *Runnability*, *Spec Implementation*, and *Design Quality* measure build/launch success, conformance to the design document, and visual polish.
- **Editing:** *Instruction Targeting*, *Feature Integrity*, and *Style Conformance* measure patch applicability, preservation and addition of functionality, and visual consistency.
- **Repair:** *Root-Cause Targeting*, *Interaction Integrity*, and *Reference Fidelity* measure defect localization, regression-safe behavior, and agreement with the fixed reference.

Why not Accessibility, Responsiveness, or Maintainability? We do track these—but at a finer granularity rather than as top-level dimensions:

- **Responsiveness and Accessibility are task-dependent, not universal.** A data-visualization dashboard and a mobile landing page demand entirely different responsive behaviors; a form-heavy application and a static showcase have incomparable accessibility requirements. Elevating them to top-level dimensions would impose a uniform rubric on all 1,526 tasks and inject inapplicable deductions. Instead, we surface them through the error taxonomy: E4.2 (Accessibility) and E4.3 (Responsiveness) are first-class fine-grained error categories (see Figure 15), and Repair includes dedicated defect types—*Missing alt/aria-label attributes* and *Color & Contrast* (mapping to WCAG)—whose per-model performance can be read independently (Figure 10).
- **Maintainability is intentionally excluded.** Unlike the above two, maintainability concerns code engineering quality (decoupling, naming, duplication) and is orthogonal to user-facing runtime quality. It is more naturally studied under longitudinal repository evolution (e.g., SWE-bench settings); WebCompass focuses on artifact-level front-end quality.

A.4 Evaluation Protocol: Comparison and Ablation

Core distinction. The Agent-as-a-Judge possesses the full capabilities of a senior front-end

test engineer: it reads source code for static analysis, synthesizes JavaScript test cases on demand, and communicates bidirectionally with a real Chromium instance via MCP for screenshots, console logs, and arbitrary user interactions.

Difference from GUI-agent approaches (e.g., WebGen-Bench). Pure GUI interaction can be costly in multi-turn, deep-state scenarios. For example, verifying the victory screen of a 2048 game may require dozens of arrow-key presses and careful handling of animation timing. The protocol therefore permits supplementary JavaScript checks after the agent has reached the target state through the visible UI, reducing repetitive mechanical operations while preserving the user-facing reachability requirement. It does not permit direct mutation of hidden state or invocation of an internal function to manufacture a state that a user cannot reach; an unreachable state receives a failed checklist score.

Difference from script-based approaches (e.g., Web-Bench). Pre-written test scripts require strict naming conventions (fixed id, data-testid); when a model adopts a reasonable but different name, the script fails, and “evaluation failure” is misread as “model failure.” Our *selector-only adaptation* allows the agent to re-locate selectors after reading the candidate code, while keeping all behavioral assertions and checklist criteria unchanged. Selector adaptation cannot be used to bypass a missing UI entry point.

Ablation study. All four key designs (browser interaction, synthesized tests, selector adaptation, evidence grounding) live only in the Generation pipeline (Agent-as-a-Judge); Edit/Repair use LLM-as-Judge over patches and screenshots. The ablation is therefore conducted on Generation only.

The four designs are not fully independent: without a browser there is no runtime context for JS injection, so synthesized tests presuppose browser interaction; selector-only adaptation is a safeguard *within* the synthesized-tests capability, not an independent axis. We therefore organize the ablation as three progressively degraded configurations, each corresponding to a mainstream front-end testing paradigm:

Config.	Capabilities	Industry Analog	r vs. human
Full (ours)	Browser + supplementary JS verification + selector adaptation + evidence grounding	Senior FE QA engineer	0.93
– Synth. tests	Browser + GUI actions only (click/type/scroll/screenshot)	GUI agent (e.g., WebGen-Bench)	0.78 (−0.15)
– Browser	Static code review by LLM	Static code analyzer	0.62 (−0.31)

(i) Supplementary tests provide the largest single-step gain (−0.15 r when removed). Without JavaScript verification, the agent is restricted to MCP’s basic GUI actions—the WebGen-Bench-style paradigm—for every repetitive observation. Three failure clusters dominate the drop:

- **Repetitive interaction sequences** (for example, the swipes needed to play a 2048-style game)—GUI agents spend more turns and are more exposed to animation timing or asynchronous loading.
- **Post-interaction inspection**—after a reachable state is established, synthesized JavaScript can inspect DOM and CSS properties without replacing the preceding UI actions.
- **Cross-component consistency**—a supplementary script can query multiple visible DOM nodes in one call and cross-validate their state.

(ii) The browser environment contributes an additional −0.16 r (cumulative −0.31). Removing the browser leaves only static code review, dropping r to 0.62. This isolates failure modes that static analysis cannot reach: runtime errors (uncaught Promise rejections, undefined property access), resource-loading failures (broken CDN links, font fallbacks visible only in console/network), and rendering-level visual issues (post-CSS layout deltas, media-query triggers). Browser interaction establishes user-visible reachability; supplementary scripts improve the efficiency and precision of verification after that reachability has been demonstrated.

A.5 Full-Benchmark Cost Decomposition

Table 8 reports judge-side per-sample cost on a 200-sample subset. The full cost decomposition (per task type, all numbers based on January 2026 public API rates; hardware excluded) is shown in Table 3.

Three observations are worth flagging:

1. **Generation dominates judge cost** (\$4.66 vs. \$0.30–\$0.35 per sample, $\approx 13\times$), driven by

Task Type (Modalities)	#Samples	Judge	Judge \$/sample	Judge Total (per model)	Cand. Gen. \$/sample	Wall-clock/sample
Generation (T / I / V)	326	Agent-as-Judge (Opus-4.5)	\$4.66	\$1,519	0.12–1.85	6–18 min
Edit (T / I)	600	LLM-as-Judge (Opus-4.5)	\$0.35	\$210	0.06–0.48	1–4 min
Repair (T / I)	600	LLM-as-Judge (Opus-4.5)	\$0.30	\$180	0.04–0.41	1–3 min
Per-model total	1,526	—	—	≈\$1,909	≈120–1,140	—
All 10 models	—	—	—	≈\$19,090	≈1,200–11,400	—

Table 3: Full-benchmark cost decomposition per task type. T=Text, I=Image, V=Video.

Agent-as-a-Judge’s multi-turn browser interaction and test synthesis (avg. 38.4 MCP calls/sample) versus the single-pass scoring of LLM-as-Judge on Edit/Repair. Generation’s per-sample wall-clock (6–18 min vs. 1–4 min) reflects the same gap and is further bounded by API rate and concurrency limits. As a result, although Generation accounts for only 21% of samples (326/1,526), it consumes ~80% of the per-model judge budget.

- Judge cost is decoupled from candidate-model cost.** The judge cost is fixed by the judge model, while candidate generation cost spans nearly two orders of magnitude across the 10 evaluated models (\$0.04–\$1.85 per sample). Smaller open-source models (e.g., Qwen3-VL-30B) are cheaper to generate with, but pay the same per-sample judge cost—total benchmark cost is dominated by the protocol, not by which model is under test.
- Affordable-subset protocol.** For resource-constrained groups, we release a 180-sample mini subset (already used for the framework comparison in §4.3.2). Evaluating a single model on this subset costs ≈\$230, or ≈\$2,300 across all 10 models, completing in ~1.5 hours per model.

The full-benchmark per-model judge cost (≈\$1,909) plus generation cost (≈\$120–\$1,140 depending on the model) is comparable to or below typical frontier-model evaluation budgets, and the mini subset offers a roughly 8× cheaper entry point.

A.6 Data Availability

To support reproducibility, we release the full WebCompass dataset (1,526 tasks across all modalities and task types). The dataset is available at <https://anonymous-hf.up.railway.app/a/2mznnrm8oed/>.

A.7 Disclosure of LLM Assistance

Disclosure of LLM Assistance. The authors independently conceived and executed all scientific ideas, algorithmic implementations, and experimental data analyses. Large Language Models (LLMs) were employed exclusively as auxiliary tools for language editing and enhancing the clarity of the manuscript. No experimental data, training samples, or reported results were generated using LLMs without rigorous human verification.

A.8 Experimental Hyperparameters

Temperature and top- p are tuned per each model’s official recommendations; the maximum output token limit is set sufficiently large to prevent truncation.

A.9 Per-Dimension Framework Evaluation

Table 4 provides the per-dimension breakdown of the framework subset evaluation summarized in Section 4.3.2; abbreviations follow Table 2.

Table 5 provides a complete list of all the large language models and vision-language models evaluated in our benchmark.

A.10 Detailed Data Construction Pipeline

To ensure that WebCompass reflects realistic web-development scenarios, we adopt a multi-stage, human-in-the-loop pipeline to construct high-quality instances for all seven task types. Figure 4 summarizes the process.

Text-Guided Generation We design the Text-Guided Generation set to satisfy two goals: (i) realistic and actionable user requirements, and (ii) broad coverage of webpage categories. To this end, we collect initial queries from four complementary sources: WebGen-Bench (Lu et al., 2025), ArtifactsBench (Zhang et al., 2025), BigCode Arena, and high-quality web showcases from V0. We then embed all queries with BGE-M3 and perform k-means clustering to reduce redundancy.

To ensure diversity, we use an LLM to annotate each query with category and difficulty labels, ob-

Model	FW	Generation			Edit			Repair		
		RUN.	SPL.	DSQ.	ITG.	FTI.	STC.	RCT.	ITI.	RFE.
GPT-5.2	React	62.08	60.57	<u>47.88</u>	43.60	40.10	35.86	54.58	82.40	62.56
	Vue	<u>65.08</u>	<u>56.79</u>	45.29	<u>45.73</u>	<u>43.30</u>	<u>38.38</u>	<u>48.85</u>	76.77	59.14
	Vanilla	75.13	60.18	56.38	65.20	61.87	55.79	44.10	<u>79.82</u>	<u>60.34</u>
Gemini-3-Pro-Preview	React	61.05	47.29	46.11	<u>54.37</u>	<u>50.31</u>	<u>43.92</u>	<u>44.02</u>	<u>81.67</u>	<u>64.12</u>
	Vue	<u>71.01</u>	55.98	<u>59.61</u>	43.58	39.13	34.23	39.25	74.70	57.11
	Vanilla	75.02	<u>55.70</u>	64.49	74.84	69.48	62.06	50.95	86.62	65.78
Claude-Opus-4.5	React	<u>79.16</u>	71.91	56.78	<u>55.84</u>	<u>49.10</u>	<u>42.24</u>	<u>44.47</u>	78.37	<u>63.97</u>
	Vue	72.85	66.99	<u>58.74</u>	40.14	38.27	30.04	41.14	<u>81.94</u>	62.85
	Vanilla	79.23	<u>71.72</u>	62.59	78.31	72.47	65.37	47.21	85.56	64.38
Qwen3-VL-235B-A22B-Instruct	React	<u>45.76</u>	<u>25.80</u>	<u>36.11</u>	<u>20.57</u>	<u>18.15</u>	<u>15.84</u>	26.16	<u>62.63</u>	<u>45.33</u>
	Vue	42.18	23.17	27.46	15.12	16.02	13.31	22.22	57.00	37.56
	Vanilla	62.16	39.97	45.01	26.33	24.99	22.19	<u>25.09</u>	71.46	46.89

Table 4: Per-dimension subset evaluation across different front-end frameworks. Each model is tested on 60 randomly sampled tasks per category using React, Vue, and Vanilla HTML/JS. **Green bold**: best; blue underline: second best per framework.

Model
Claude-Opus-4.5 (Anthropic, 2025)
Claude-Sonnet-4.5 (Anthropic, 2025)
Gemini-3-Pro-Preview (Google, 2025)
Gemini-3-Flash-Preview (Google, 2025)
GPT-5.2 (OpenAI, 2025)
Qwen3-VL-32B-Instruct (Bai et al., 2025)
Qwen3-VL-235B-A22B-Instruct (Bai et al., 2025)
Qwen3-VL-30B-A3B-Instruct (Bai et al., 2025)
Qwen3-32B (Yang et al., 2025)

Table 5: List of evaluated models.

taining five independent annotations per query and taking the majority vote as the final label. We then apply stratified sampling across categories and difficulty levels, resulting in 123 text-guided generation queries.

A practical challenge is that many real-world requests are underspecified, which leads to highly variable outputs and makes automatic evaluation difficult. To reduce subjective ambiguity while preserving realism, we use an LLM acting as a product manager to rewrite such requests into structured design documents that explicitly specify (1) page content, (2) interaction behavior, and (3) visual appearance.

Vision-Guided Generation Most existing webpage screenshot datasets contain relatively simple UIs and thus provide limited challenge for modern models. We therefore build our Vision-Guided Gen-

eration set partly from WebRenderBench, which contains visually complex webpages. Since each website typically includes only one screenshot, we augment the data by parsing subpage URLs referenced in `index.html`, randomly selecting two, and capturing their screenshots with Playwright. To test whether models can reproduce multi-page websites and their dependencies, we inject a JavaScript overlay into the main-page screenshot to highlight the positions of subpage links with colored bounding boxes.

Because screenshot collection is sensitive to network instability and dynamic content loading, we perform multiple rounds of LLM-based verification to detect artifacts such as missing assets, blank regions, or layout corruption, followed by manual inspection.

Video-Guided Generation Static screenshots cannot adequately capture dynamic effects such as animation or multi-step interaction. To address this limitation, we manually browse webpages from V0 and Figma and construct dynamic visual inputs in two forms.

First, for some webpages we manually extract keyframes corresponding to important state transitions; these form the dynamic keyframe-sequence portion of the benchmark. Second, for the Video-Guided Generation task, we manually select webpages with rich interactive behaviors, explore them thoroughly, and record complete interaction videos. Annotators are instructed to first familiarize themselves with each webpage, plan a comprehensive

interaction path, and then perform the final recording to maximize coverage of dynamic functionality.

Prototype Collection for Editing and Repair

Editing and repair tasks share a common pool of high-quality web prototypes (Figure 4, top). We construct these prototypes from the WebRenderBench test set via a three-stage pipeline: *length filtering* → *automatic quality scoring* → *human curation*.

- **Stage 1: Length filtering.** We require the total character count across all code files to be between 32k and 64k, and each file to be no longer than 48k characters. These bounds preserve realistic multi-file complexity while avoiding instances that are either trivial or too large for stable evaluation.
- **Stage 2: Quality scoring.** For candidates that satisfy the length constraints, we use GPT-4o to perform code review on a 10-point scale and retain those scoring at least 9, leaving 81 candidates.
- **Stage 3: Human curation and expansion.** We manually select 50 high-quality prototypes. Each is preserved as a SINGLE-PAGE website and also extended into a MULTI-PAGE variant by adding extra pages, inter-page navigation, and shared resources. These variants together form the prototype pool for downstream editing and repair tasks.

Text-Guided and Vision-Guided Editing Starting from each prototype as the executable *source* website, we create editing instances by introducing new or enhanced requirements drawn from 16 predefined high-level task types. These task types cover complex components (e.g., data tables, rich-text editors, drag-and-drop interfaces), interaction and animation effects (e.g., parallax scrolling), and holistic application scenarios.

For each task type, we write requirements that clearly describe *what* should be changed—including UI adjustments, interaction flows, and state feedback—while intentionally omitting implementation details such as selectors, class names, and CSS values. This design improves both realism and fairness. The resulting requirements, together with the source website, form the text-guided editing instances; vision-guided variants additionally include a reference screenshot.

Diagnostic and Visual-Diagnostic Repair Repair tasks are constructed in a verifiable reverse manner. We start from a clean prototype as the *destination* website and use an LLM to inject observable, explainable frontend defects from 11 repair categories, producing a faulty *source* website. The model is then asked to repair the source back to the destination.

The injected defects span three dimensions:

- **Visual layout:** occlusion, crowding, text overlap, misalignment, insufficient contrast, overflow, and distorted proportions.
- **Semantics & structure:** incorrect semantic or nesting structures, and missing attributes.
- **Interaction usability:** broken interactions and loss of interactivity.

We then generate natural-language repair instructions that only provide vague diagnostic hints instead of fully describing the bug, thereby preventing leakage of implementation details. To guarantee determinism and support automatic evaluation, each repair instance is paired with an exact text-level search/replace annotation corresponding to the inverse of the defect-injection edit. This ensures a unique runnable solution, reproducible source-to-target transformation, and precise automated verification.

Ecological Validity of Repair Defects The 11 repair categories are grounded in real-world frontend failures rather than chosen arbitrarily. We analyze more than 200 community submissions on V0 and their corresponding GitHub Issues, identifying the most frequent frontend anti-patterns and bug types. Categories such as *Occlusion*, *Overflow*, and *Loss of Interactivity* correspond directly to common real-world failure modes, improving the ecological validity of the repair benchmark.

Quality Control We apply a three-layer quality assurance process across all tasks.

Automated checks. Every instance must compile and render without fatal errors in a headless Chromium environment. For editing and repair tasks, patches must apply cleanly to the source repository. For repair tasks, search/replace annotations are further verified to be the exact inverse of the defect-injection edits.

LLM-assisted screening. We use an LLM for multi-round quality checks on instructions and visual assets. For vision-guided generation, the LLM

filters screenshots with blank regions, missing assets, or broken layouts. For editing and repair tasks, it checks that instructions are unambiguous, consistent with the underlying code changes, and free of implementation leakage.

Human curation. Finally, expert annotators review all instances for task correctness, screenshot/video quality, difficulty labeling, and alignment between requirements and the ground-truth patches. Any instance failing these checks is revised or discarded.

A.11 Evaluation Implementation Details

We provide additional implementation details supplementing the evaluation methodology described in §3.

Infrastructure. All experiments are conducted on a Linux server. Each code generation task is allocated a maximum execution timeout to prevent infinite loops or hanging processes. For the *Agent-as-a-Judge* evaluation, we employ Claude Code (version 2.0.67) as the evaluation orchestrator, paired with the Chrome DevTools MCP Server (version 0.19.0) running a headless Chromium instance for browser-based rendering, DOM inspection, and automated interaction verification.

LLM-as-a-Judge inputs. For editing and repair tasks, the judge model (Claude-Opus-4.5) receives the following inputs as context: (i) the original task requirement (text and/or images), (ii) the source code repository, (iii) the model-generated code patch, (iv) build and runtime logs from applying the patch, and (v) before-and-after screenshots captured in headless Chromium. The judge evaluates each checklist item independently, providing a score and a brief evidence-based justification. Scores are aggregated within the applicable task-specific dimensions (ITG/FTI/STC for Editing and RCT/ITI/RFF for Repair) and then into the reported overall score using equal weighting across the nine dimensions.

Agent-as-a-Judge workflow. For generation tasks, the evaluation agent operates in the following steps:

1. **Project setup:** The agent installs dependencies and launches the generated project in a headless Chromium environment via a local development server.

2. **Checklist loading:** The agent receives the structured evaluation checklist generated from the task query (as described in Stage 1 of §3.2).
3. **Browser interaction:** Using MCP tools, the agent captures full-page screenshots, clicks elements, types text, scrolls pages, and navigates between pages to verify interactive behaviors.
4. **Supplementary verification:** After reaching each relevant state through the visible UI, the agent writes and executes JavaScript test cases to inspect DOM states, CSS properties, and functional behaviors. Failed checks trigger the iterative debugging loop described in §3.2; scripts may adapt selectors but may not alter assertions or create an unreachable state.
5. **Scoring:** The agent scores each checklist item based on collected evidence and produces a structured JSON report.

Score aggregation and cascading-failure handling. For each task instance, let $\{s_1, s_2, \dots, s_n\}$ denote the checklist item scores and $\{m_1, m_2, \dots, m_n\}$ their corresponding maxima. We normalize each item as $r_i = s_i/m_i$; when $s_i = 0$, we use $r_i = \epsilon/m_i$ with $\epsilon = 1$. The dimension score is the harmonic mean

$$s_{\text{task}} = \frac{n}{\sum_{i=1}^n 1/r_i}, \quad (1)$$

computed separately for each task-specific dimension, multiplied by 100, and reported on a 0–100 scale. The *Overall* score is the arithmetic mean of the nine dimension scores.

For generation tasks, we handle cascading failures explicitly. A complete build failure sets Spec Implementation and Design Quality to 0; a partial rendering failure evaluates Design Quality and Spec Implementation only on the rendered and reachable portions, respectively; and a runtime crash scores Runnability and Design Quality on the initial render while assigning 0 to untestable Spec Implementation items. These rules ensure that failures are reflected without producing undefined or inflated scores.

A.11.1 Task-Type Breakdown

We decompose Edit and Repair performance into fine-grained subtask categories (Figures 9 and 10) for the three top-performing closed-source models to reveal where models succeed and struggle.

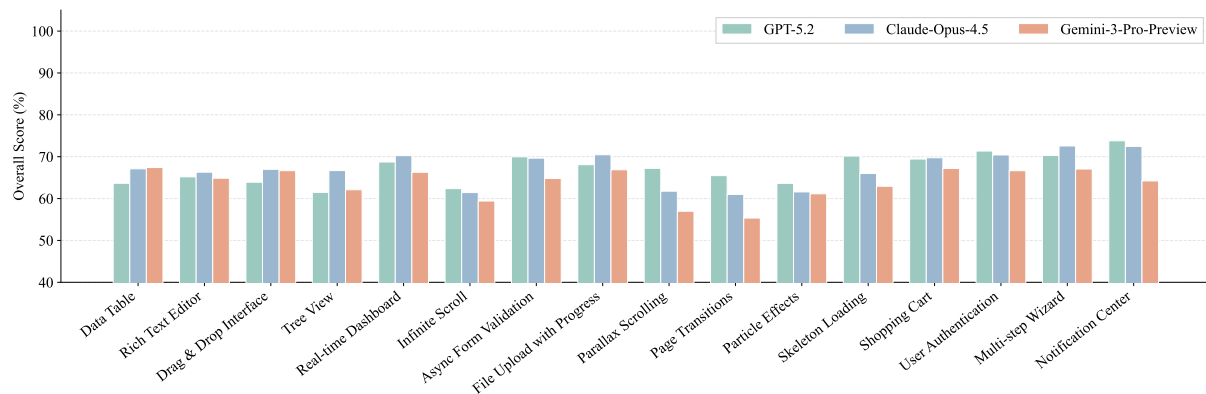


Figure 9: Overall score breakdown for editing tasks across 16 operation types. Scores are computed from the harmonic mean of the three task-specific dimensions (Instruction Targeting, Feature Integrity, and Style Conformance) per subtask, averaged over all instances containing that operation type. Three top-performing closed-source models are compared.

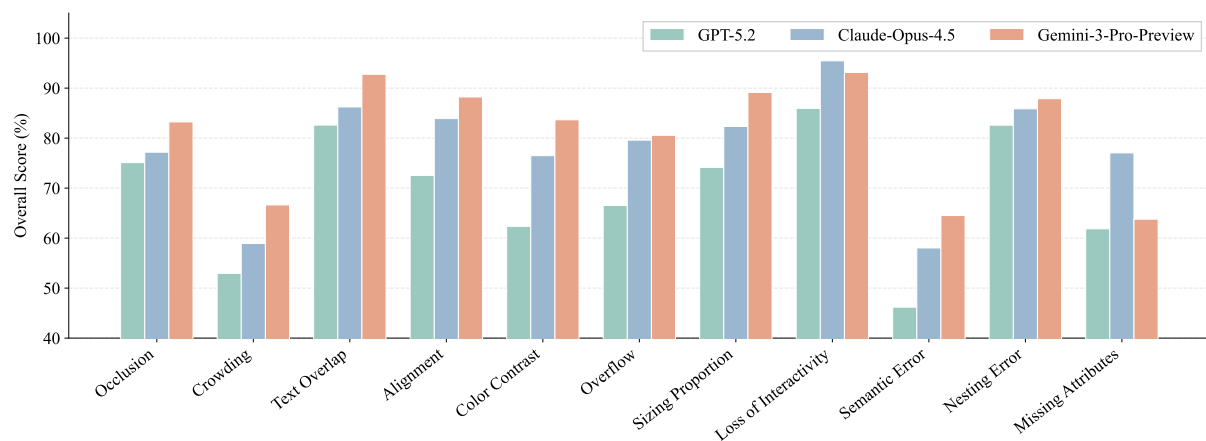


Figure 10: Overall score breakdown for repair tasks across 11 defect categories. Scores are computed from the harmonic mean of Root-Cause Targeting, Interaction Integrity, and Reference Fidelity, as in Figure 9.

Editing: animation tasks are the hardest. A clear difficulty hierarchy emerges across editing operation types (Figure 9): Business Scenario tasks (e.g., Shopping Cart, Multi-step Wizard) are consistently the easiest, followed by Real-time & Async tasks, then Interactive Components, with Advanced Animation tasks (Parallax Scrolling, Page Transitions, Particle Effects) forming the hardest category. This ordering is stable across all three models, suggesting that editing difficulty scales with the degree of visual dynamism and cross-component coordination required.

Repair: semantic-level defects resist fixing. A similar difficulty gradient appears in repair (Figure 10). Structural and interactive defects—Loss of Interactivity, Nesting Error, Text Overlap—are reliably fixed, as they manifest in localized DOM structures or event handlers. Semantic-level defects, however, prove substantially harder: Semantic

Error elicits the lowest scores across all models, followed by Crowding and Missing Attributes. These categories require reasoning about design intent and implicit visual constraints that go beyond pattern-matching on code structure.

Subtask excellence vs. instance-level consistency. An instructive discrepancy emerges for editing: GPT-5.2 outperforms Gemini-3-Pro-Preview on 13 of 16 subtask types when averaged per category, yet trails on instance-level scores in Table 2. This reversal stems from our harmonic-mean aggregation—GPT-5.2 exhibits higher cross-subtask variance, and the harmonic mean penalizes low outliers steeply. *This highlights that multi-requirement evaluation rewards consistency, not just peak subtask performance.* No such reversal occurs for repair, where Gemini-3-Pro-Preview leads on all 11 defect categories, reflecting genuinely superior repair capability.

A.11.2 Difficulty-Level Analysis

Each task in WebCompass is annotated with a difficulty label—*Easy*, *Medium*, or *Hard*—determined by the complexity of required functionality, the number of interactive components, and the sophistication of the visual design. This stratification enables us to examine how model performance degrades as task difficulty increases.

To further investigate how model capabilities scale with task complexity, we break down the evaluation results across three difficulty levels (*Easy*, *Medium*, *Hard*) for each task category. For each difficulty–task combination, we compute the average score across that task’s three evaluation dimensions and visualize the results as grouped bar charts in Figures 11, 12, and 13.

Across all task categories and evaluation dimensions, model scores consistently decrease as task difficulty increases. This degradation is particularly striking in generation tasks on the *Spec Implementation* dimension, where model scores on Hard tasks are substantially lower than those on Easy tasks (e.g., Gemini-3-Pro-Preview drops from 89.83 to 37.64), suggesting that maintaining faithful specified behavior becomes disproportionately challenging as task complexity grows.

Cross-task observations. Qwen3-VL-235B-A22B-Instruct consistently ranks last across all difficulty–task combinations, with a particularly large gap on Hard tasks. Notably, the performance gap between top-tier models and Qwen3 widens as difficulty increases, suggesting that stronger reasoning capabilities become increasingly critical for complex front-end tasks.

A.11.3 Patch Complexity Analysis

Beyond scores, we analyze the *structural complexity* of generated patches using two metrics: **Changed Lines** (added + deleted lines) and **Patch Count** (number of contiguous diff hunks). For repair, we also include the human-authored **Ground Truth** baseline. Figure 14 shows the distributions.

Edit patches are far larger than repair patches. Edit tasks yield median patch sizes of 646–1,976 changed lines across models, while repair patches are much smaller, with medians of 16–19 lines. This highlights the localized nature of repair compared with the broader restructuring often required for editing.

Models differ substantially in edit verbosity.

Claude-Opus produces the largest edit patches (median ~2,000 lines), about 3× larger than Gemini models (~650 lines), despite only modest quality differences (Table 2). This suggests that larger rewrites do not necessarily translate into better outcomes.

Repair patches are close to Ground Truth, but with heavier tails.

For repair, all models have median patch sizes close to the human-authored baseline (~18 lines), indicating generally accurate defect localization. However, their distributions show heavier right tails (up to 172 lines, versus 38 for Ground Truth), suggesting occasional over-editing.

Patch count is relatively stable across models.

Compared with changed lines, the number of diff hunks varies less (7–12 hunks), implying that tasks largely determine *where* edits occur, while models differ more in *how much* code they modify at each site.

A.12 Detailed Worst-of-*n* Stability Results

Table 6 extends the Worst-of-*n* stability analysis presented in the main text by reporting *Pass@1*, *W@2*, and *W@4* scores across all nine evaluation dimensions, grouped by task category.

A.13 Error Analysis

To gain deeper insight into the failure modes of current models on web generation tasks, we conduct a fine-grained error analysis across all evaluated models. Figure 15 presents the overall error distribution, while Figure 16 provides a breakdown by input modality (text, image, and video).

Overall Error Distribution. As shown in Figure 15, across all models, the most prevalent error categories are *Feature Missing* (E2.1) and *Resource Fail* (E1.3), which together account for approximately 40%–55% of all detected errors in most models. *Feature Missing* errors, where required interactive components or functional modules are entirely absent from the generated page, consistently dominate the error landscape, indicating that models frequently fail to fully implement the specified functionality. *Resource Fail* errors, corresponding to broken asset references or failed external resource loads, constitute the second largest share, reflecting a common tendency of models to hallucinate non-existent re-

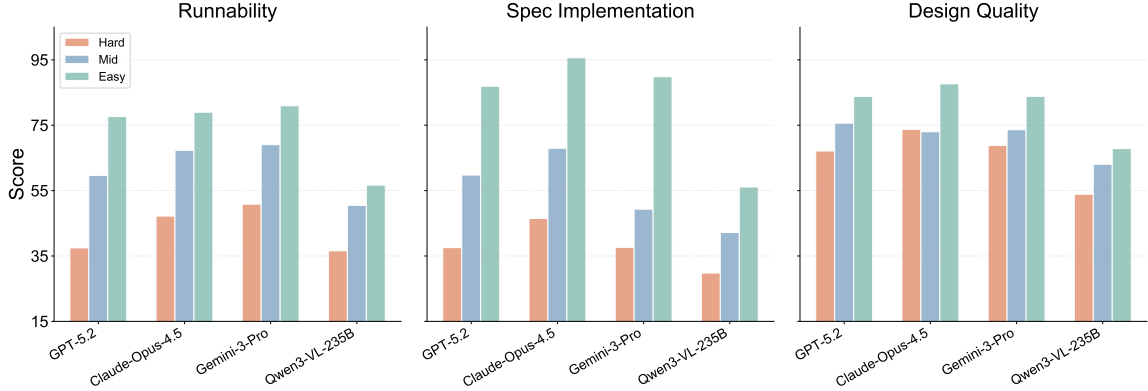


Figure 11: Generation task: per-dimension scores (Runnability, Spec Implementation, Design Quality) across three difficulty levels. Each model has three bars representing Hard (red), Medium (blue), and Easy (green).

Table 6: Consistency and Stability Results for Gemini-3-Pro-Preview and Qwen3-VL-235B-A22B-Instruct ($n = 4$). $Pass@1$, $W@2$, and $W@4$ are reported across all nine evaluation dimensions grouped by task category; $\Delta \downarrow$ denotes the relative drop from $Pass@1$ to $W@4$.

Metric	Generation			Editing			Repair		
	RUN.	SPI.	DSQ.	ITG.	FTI.	STC.	RCT.	ITI.	RFF.
<i>Gemini-3-Pro-Preview</i>									
$Pass@1$ (%)	75.31	56.90	60.68	74.84	69.48	62.06	50.95	86.62	65.78
$W@2$ (%)	73.55	47.74	56.03	67.41	62.41	53.77	39.83	80.41	58.19
$W@4$ (%)	68.92	39.80	49.27	63.45	57.89	49.51	31.31	72.07	49.80
$\Delta \downarrow$ (%)	8.48	30.05	18.80	15.22	16.68	20.22	38.55	16.80	24.29
<i>Qwen3-VL-235B-A22B-Instruct</i>									
$Pass@1$ (%)	61.22	38.59	42.77	26.33	24.99	22.19	25.09	71.46	46.89
$W@2$ (%)	52.75	28.42	34.57	19.54	17.62	17.64	21.00	62.46	38.68
$W@4$ (%)	44.42	23.56	29.09	15.91	14.50	13.76	17.66	58.23	32.86
$\Delta \downarrow$ (%)	27.44	38.95	31.99	39.57	41.98	37.99	29.61	18.51	29.92



Figure 12: Edit task: per-dimension scores across difficulty levels.

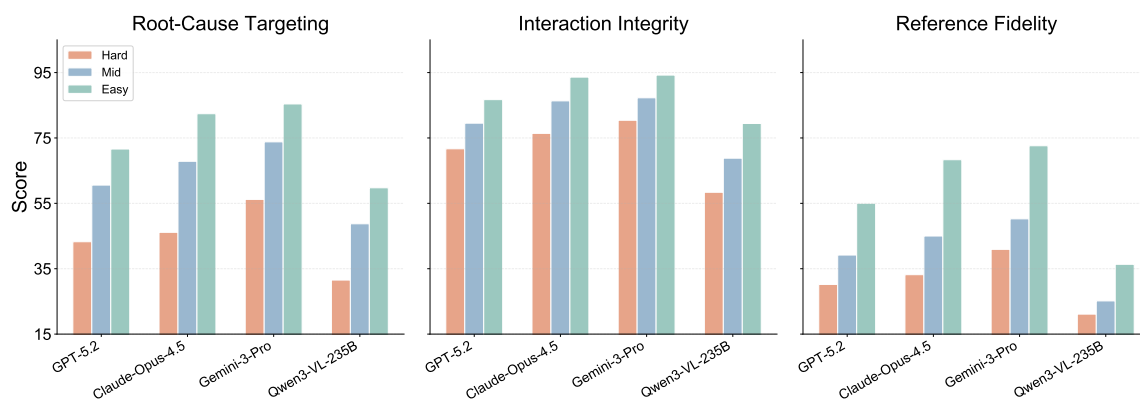


Figure 13: Repair task: per-dimension scores across difficulty levels.

source paths. Visual styling errors—including *Layout Error* (E3.1), *Color/Theme Error* (E3.2), *Typography Error* (E3.3), and *Visual Fidelity* (E3.6)—collectively represent another substantial portion of errors, particularly prominent in higher-performing models such as Claude-Opus-4.5, Gemini-3-Pro-Preview, and GPT-5.2, where basic functionality is more reliably generated but fine-grained visual fidelity remains challenging. Notably, lower-ranked models such as Qwen3-VL-30B-A3B exhibit substantially higher total error counts (exceeding 5,000), with a disproportionate share of fundamental errors like *Feature Missing* and *Console Error* (E1.2), suggesting that these models struggle with both code correctness and specification adherence. In contrast, top-performing models like Claude-Opus-4.5 achieve the lowest total error counts (2,100) with a more balanced error profile, where styling and layout issues become the relative bottleneck rather than missing functionality.

Error Analysis by Input Modality. Figure 16 reveals distinct error patterns across the three input modalities, reflecting the fundamentally different

challenges each poses to generation models.

For **text input** tasks, where models receive natural language descriptions specifying the desired webpage, the dominant error type is overwhelmingly *Feature Missing* (E2.1), accounting for 40%–66% of errors across models. This is followed by *Feature Incomplete* (E2.2) and interaction-related errors. This pattern indicates that when translating textual specifications into functional web pages, models most frequently fail at implementing the required interactive behaviors and dynamic features. Since text descriptions tend to specify complex interaction logic (e.g., “clicking the button should filter the table”), the primary bottleneck lies in the models’ ability to reason about and implement functional requirements rather than visual appearance.

For **image input** tasks, where models must reproduce a webpage from a screenshot, the error distribution shifts markedly toward visual and structural fidelity issues. While *Feature Missing* remains significant (18%–30%), visual errors such as *Layout Error* (E3.1), *Color/Theme Error* (E3.2), and *Visual Fidelity* (E3.6) become substantially more prominent compared to text input. Resource fail-

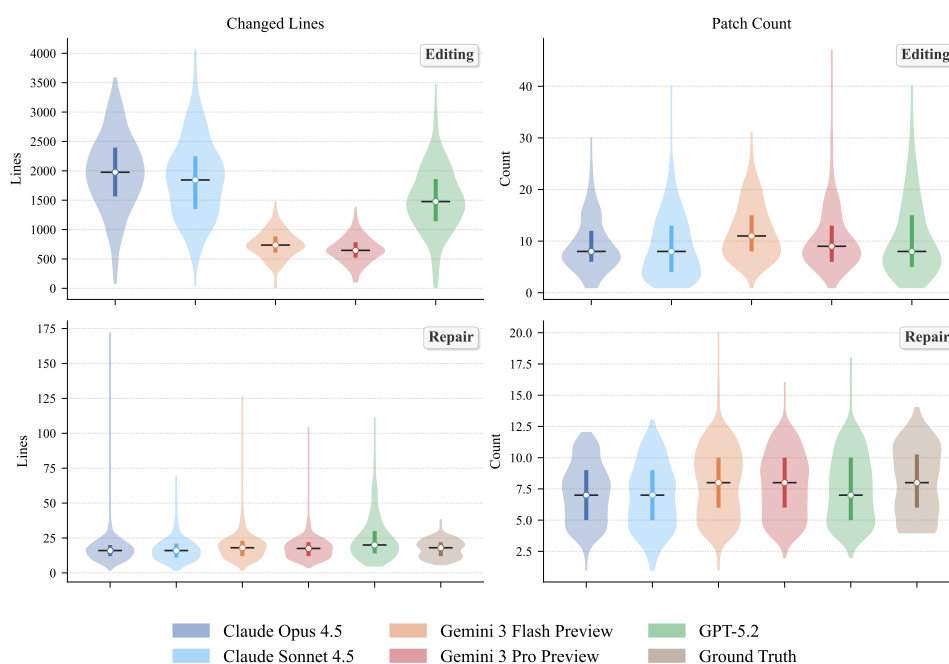


Figure 14: Distribution of patch complexity across models. Top row: Edit tasks; bottom row: Repair tasks (with Ground Truth baseline). Each violin shows the full distribution; the thick bar marks the interquartile range (Q1–Q3) and the white dot marks the median.

ures (E1.3) also feature heavily, as models attempt to recreate visual assets they observe in screenshots but cannot faithfully reproduce. This pattern confirms that image-conditioned generation primarily challenges models’ visual perception and pixel-level reproduction capabilities—models can often identify the general structure but struggle to precisely replicate spatial layouts, color schemes, and typographic details.

For **video input** tasks, which present screen recordings of interactive webpages, the error profile reflects a compound challenge that combines both visual reproduction and functional implementation. The error distribution is notably more balanced across categories: visual errors (*Layout Error*, *Color/Theme Error*) and functional errors (*Feature Missing*, *Feature Incomplete*) appear in roughly comparable proportions. This is because video inputs simultaneously convey both the static visual appearance and the dynamic interactive behaviors of a webpage, requiring models to accurately perceive temporal sequences of user interactions while also faithfully reproducing the visual design. The dual demand makes video-conditioned generation the most challenging modality, as reflected in the relatively high error counts even for strong models.

A.13.1 Text-Only Model vs. Vision-Language Model

To investigate whether the additional visual grounding capacity of a vision-language model benefits—or hinders—pure front-end code generation, we conduct a controlled comparison between QWEN3-32B (text-only) and QWEN3-VL-32B-INSTRUCT (vision-language variant) on three representative task types that reflect real-world web-development workflows: Text-Guided Generation, Text-Guided Editing, Diagnostic Repair.

Key observations. As shown in Table 7, QWEN3-VL-32B-INSTRUCT leads on eight of the nine task-specific dimensions and on Overall (36.49 vs. 31.60). Its largest advantages are in the visual dimensions (DSQ, STC, and RFF), consistent with a stronger visual grounding prior; it also scores higher on the functional dimensions in the editing and repair settings. QWEN3-32B retains an advantage only on Generation Runnability (56.28 vs. 44.48), indicating that text-only modeling can still provide a benefit for basic build and launch reliability.

Quantitative Error Distribution in Edit and Repair Tasks. To complement the qualitative observations above, we conduct a comprehensive quantitative breakdown of error frequencies across Edit

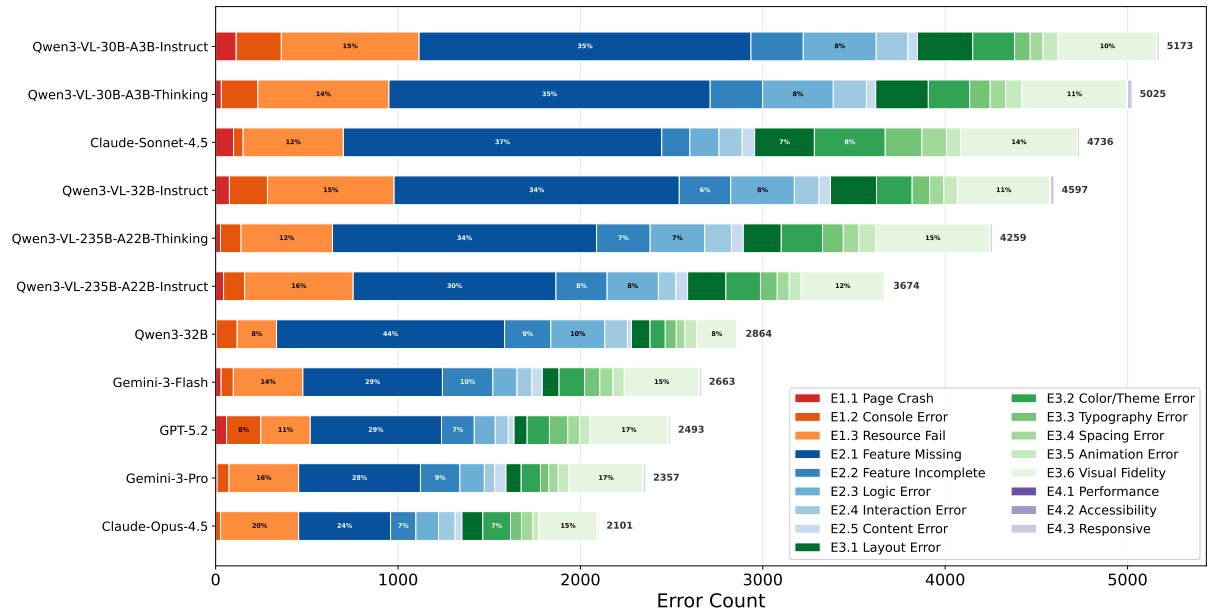


Figure 15: Overall error distribution across all evaluated models on web generation tasks. Each horizontal bar represents the total error count for a model, with colored segments indicating the proportion of each error type. *Feature Missing* (E2.1) and *Resource Fail* (E1.3) consistently dominate across models, while top-performing models exhibit a more balanced error profile with relatively higher proportions of visual styling errors.

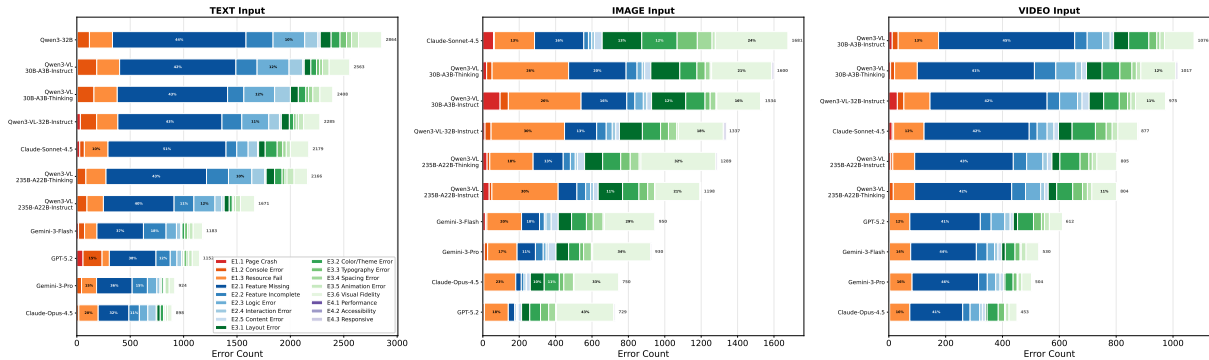


Figure 16: Error distribution breakdown by input modality: text (left), image (middle), and video (right). Text-conditioned generation is predominantly limited by functional errors such as *Feature Missing* (E2.1), image-conditioned generation is more challenged by visual fidelity and layout errors, and video-conditioned generation exhibits a balanced mix of both functional and visual errors, reflecting its compound difficulty.

Models	Generation			Editing			Repair			Overall
	RUN.	SPL.	DSQ.	ITG.	FTL.	STC.	RCT.	ITL.	RFF.	
Qwen3-32B	56.28	6.50	49.10	21.52	21.17	17.92	21.42	56.73	33.72	31.60
Qwen3-VL-32B-Instruct	44.48	14.53	56.92	28.16	27.86	24.17	26.01	64.14	42.15	36.49

Table 7: Comparison of QWEN3-32B and QWEN3-VL-32B-INSTRUCT on the **Text**-only subset across three front-end task types. Columns use the task-specific dimensions in Table 2; Overall is the arithmetic mean of the nine dimension scores. **Green bold** indicates the best score in each column.

and Repair tasks. By analyzing the automated checklist deduction logs, we categorize the failures into specific sub-types. Figures 17 and 18 illustrate

the total error counts and their proportional distributions across models for Edit and Repair tasks, respectively.

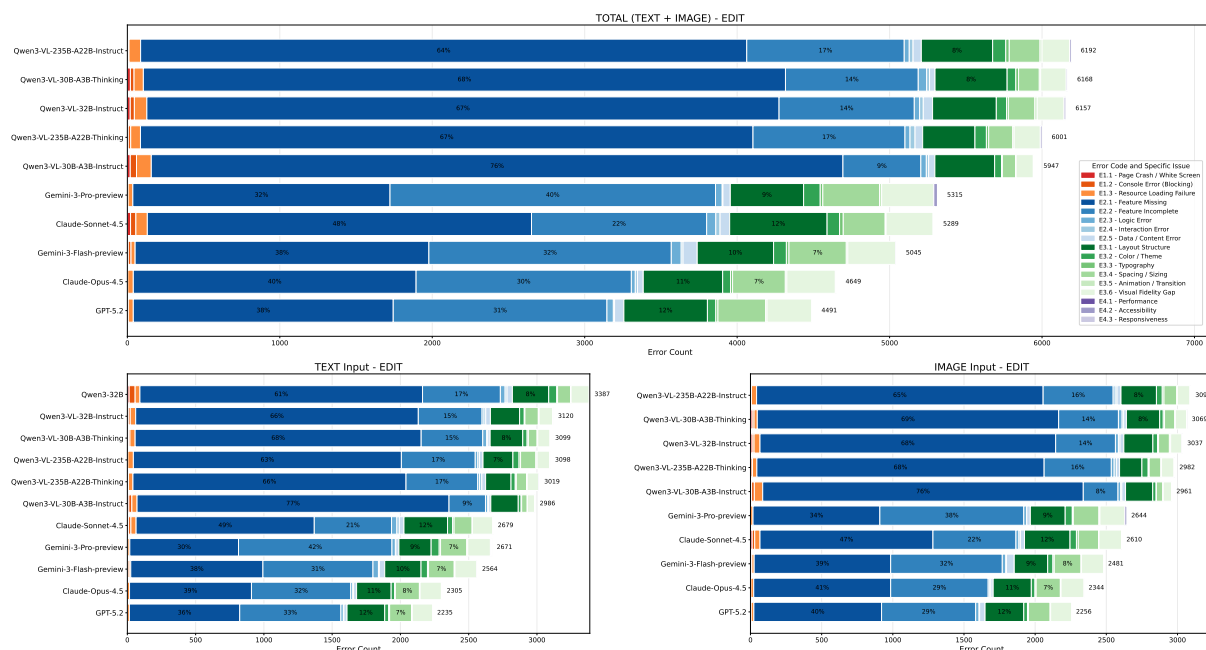


Figure 17: Quantitative distribution of error types for **Edit tasks**. The errors are categorized into Blocking/Crash (Orange/Red), Logic & Features (Blue), Visual & Layout (Green), and Accessibility/Performance (Purple). The total error count for each model is displayed on the right.

Several striking patterns emerge from this quantitative lens:

- **Edit Tasks are Bottlenecked by Feature Completeness and Logic (E2).** As shown in Figure 17, the vast majority of errors in editing tasks stem from the *Feature Missing* (E2.1) and *Feature Incomplete* (E2.2) categories (represented in dark and medium blue). For open-source models like Qwen3-VL-30B-A3B-Instruct, E2.1 alone accounts for up to 76% of all checklist failures. Even for top-tier closed-source models, these logical and feature-level omissions dominate (e.g., Claude-Opus-4.5: 40% E2.1 and 30% E2.2). This aligns with our qualitative finding that models often suffer from “partial implementation,” losing track of complex or multi-step editing instructions.
- **Visual Fidelity (E3) is the Secondary Challenge in Editing.** Visual/Layout errors (green segments) form the second-largest block in Edit tasks. Closed-source models notably struggle with *Layout Structure* (E3.1) and *Visual Fidelity Gap* (E3.6), indicating that while they can write the logical JavaScript, aligning CSS properties precisely with the user’s aesthetic intent remains difficult.
- **Repair Tasks Fail Primarily due to Unad-**

dressed Defects (E5.1). Figure 18 reveals a completely different error paradigm for Repair tasks. The newly introduced repair-specific categories (pink/magenta) overwhelmingly dominate the distribution. Specifically, *Defect Not Addressed* (E5.1) is the primary failure mode. For weaker models (e.g., Qwen3-VL-30B-A3B-Thinking), a staggering 74% of errors occur because the generated patch simply fails to fix the original bug. Even Gemini-3-Pro-Preview, the absolute best performer in Repair (§4.2), sees 49% of its errors coming from E5.1.

- **The “Over-editing” Penalty in Repair (E5.3).** We also observe a notable proportion of *New Defect Introduced* (E5.3) errors in Repair tasks (ranging from 8% to 12% for closed-source models). This quantitatively corroborates the heavy right-tail distribution observed in our Patch Complexity Analysis (§A.11.3): models that generate excessively large patches (like Claude and GPT-5.2) frequently break previously working functionality while attempting to fix a localized bug.
- **Modality Consistency.** When splitting the analysis between Text Input and Image Input (bottom panels of both figures), the proportional distribution of error categories remains

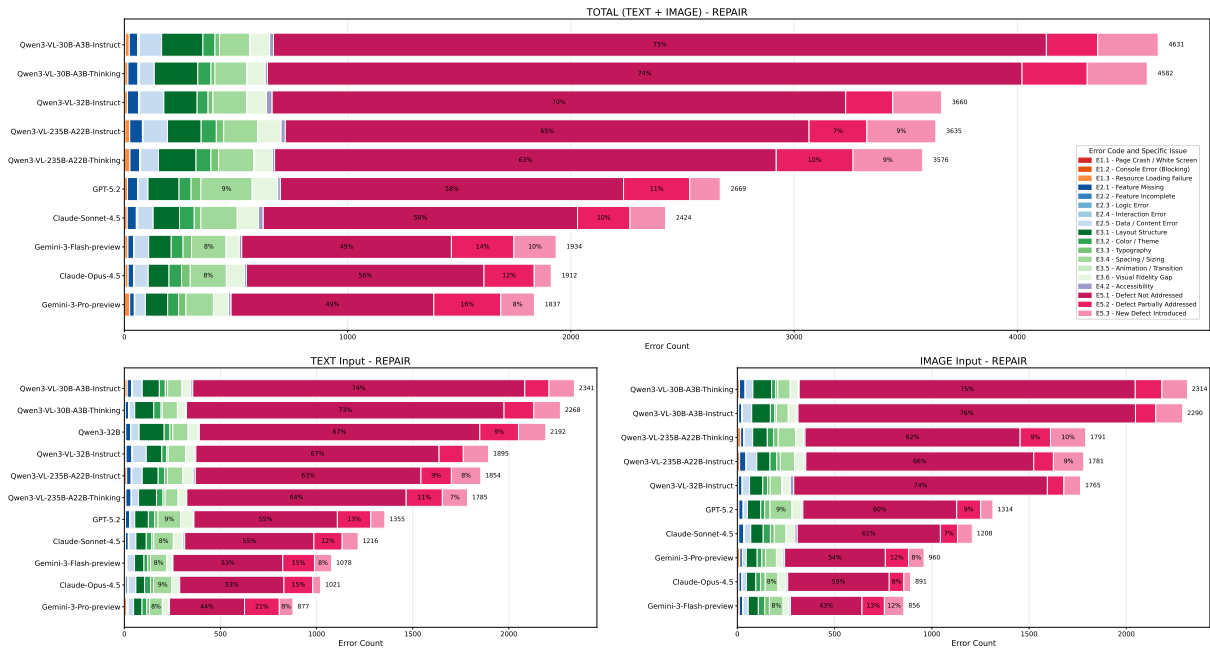


Figure 18: Quantitative distribution of error types for **Repair tasks**. In addition to standard web errors, Repair tasks introduce defect-resolution specific errors (Pink/Magenta): Defect Not Addressed, Partially Addressed, and New Defect Introduced.

remarkably stable within each model. This suggests that the core weaknesses—failing to implement complete features in Edit, and failing to diagnose the root cause in Repair—are fundamental reasoning bottlenecks rather than modality-specific perceptual failures.

A.13.2 Impact of Thinking Mode on Performance

Recent work on reasoning-enhanced LLMs has introduced “thinking” or “chain-of-thought” modes that encourage models to reason step-by-step before producing a final answer (Guo et al., 2025). To investigate whether this paradigm benefits web development tasks, we compare the Instruct and Thinking variants of two Qwen3-VL models that appear in our evaluation: Qwen3-VL-235B-A22B and Qwen3-VL-30B-A3B.

As shown in Table 2, the impact of thinking mode varies across task types and evaluation dimensions. On **Generation** tasks, both Thinking variants achieve higher *Runnability* scores than their Instruct counterparts (63.86 vs. 61.26 for 235B; 47.37 vs. 41.79 for 30B), indicating that chain-of-thought reasoning helps with code structural correctness. However, the 235B Thinking model suffers a notable *Spec Implementation* drop (35.02 vs. 42.14), while the 30B model shows negligible change (20.87 vs. 20.80). On **Editing** and **Repair**

tasks, differences between Thinking and Instruct variants are relatively minor for both scales.

The limited impact on Edit and Repair tasks likely reflects that these tasks exceed the current capability boundary of Qwen3-VL models. As shown in Table 2, both models score substantially lower on Editing than on Generation (e.g., 27.74 vs. 61.26 on *Instruction Targeting* for the 235B Instruct), suggesting that accurately comprehending existing code, locating modification points, and producing precise changes poses a fundamental challenge. When the task difficulty surpasses the model’s base competence, thinking mode cannot compensate for the lacking skills—there is insufficient domain knowledge for the reasoning chain to meaningfully build upon.

For Generation tasks, where models perform relatively better, the *Spec Implementation* degradation of the 235B Thinking variant is notable. Our error analysis reveals that this model produces significantly more *Feature Missing* errors—cases where required interactive features are absent from the output. We attribute this to **attention dilution** caused by lengthy reasoning chains. Web development prompts often specify multiple requirements simultaneously—layout, styling, interactive behaviors, and responsiveness. The 235B model, with its greater capacity, generates substantially longer thinking chains than its 30B counterpart, pushing

the original feature specifications far from the code generation point in the context window. This makes it easier for the model to overlook specific requirements, producing structurally sound but incomplete implementations. The 30B model’s shorter reasoning chains preserve proximity to the original prompt, explaining its stable *Spec Implementation* scores.

A.13.3 Judge Model Selection

Score Agreement with Human Judgments. To validate the reliability of automated evaluation, we compare three Claude-family judge models—Opus-4.5, Sonnet-4.5, and Haiku-4.5—on a balanced 200-sample subset (50 per task type), using Gemini-3-Pro-Preview as the candidate model. Human judgments on the same samples serve as the ground-truth reference. Table 8 reports human agreement (Pearson r) and estimated cost.

Judge Model	Generation	Edit	Repair	Cost Analysis
	r	r	r	Cost
Claude-Opus-4.5	0.93	0.94	0.96	\$4.66
Claude-Sonnet-4.5	<u>0.88</u>	<u>0.90</u>	<u>0.89</u>	\$2.34
Claude-Haiku-4.5	0.76	0.79	0.81	\$1.02

Table 8: Judge model comparison. We report human agreement (Pearson r) and estimated cost per sample. **Green bold**: best; blue underline: second best.

Human annotation protocol. Three independent annotators, each with ≥ 3 years of front-end development experience and familiarity with React, Vue, and Vanilla stacks (none overlap with the paper authors), independently scored the same 200 instances (50 per task type plus 50 cross-modality controls). Annotators received the exact same rubric as the LLM judge—the prompt in Appendix A.15.1—including task-specific 0–10 dimensions (Runnability/Spec Implementation/Design Quality for Generation, Instruction Targeting/Feature Integrity/Style Conformance for Editing, and Root-Cause Targeting/Interaction Integrity/Reference Fidelity for Repair), with explicit deduction anchors and reference screenshots. Mean annotation time: ~ 8 min/sample. Per-sample ground truth = mean of three annotators’ scores.

Inter-annotator and judge–human agreement. We report both quadratic-weighted Cohen’s κ and Pearson r (leave-one-out: each annotator vs. the

mean of the other two; each judge vs. the aggregated human scores).

Judge	Generation		Editing		Repair		Average	
	κ	r	κ	r	κ	r	κ	r
Claude-Opus-4.5	0.71	0.93	0.80	0.94	0.85	0.96	0.79	0.94
Claude-Sonnet-4.5	0.65	0.88	0.72	0.90	0.78	0.89	0.72	0.89
Claude-Haiku-4.5	0.53	0.76	0.62	0.79	0.70	0.81	0.62	0.79
Human–Human	0.75	0.95	0.82	0.95	0.83	0.96	0.80	0.95

A clear quality–cost trade-off emerges. Claude-Opus-4.5 achieves the highest human agreement ($\kappa = 0.79$, $r = 0.94$), approaching the human–human ceiling ($\kappa = 0.80$); Sonnet-4.5 offers a competitive balance at roughly one-third the cost with a modest agreement drop; Haiku-4.5 is substantially cheaper but less reliable ($\kappa = 0.62$). Two per-task patterns are noteworthy: (1) all judges agree more on edit/repair than on generation, consistent with the constrained solution space of patch-based tasks; and (2) image-conditioned tasks yield the lowest agreement, indicating that fine-grained visual assessment remains challenging for automated judges. We therefore adopt **Claude-Opus-4.5** as the default judge, prioritizing evaluation reliability.

Comparison of model rankings between agent-based automatic evaluation and human evaluation As shown in Figure 19, the agent-based evaluator demonstrates strong consistency with human rankings across all three tasks and evaluation dimensions. In most cases the rank difference is zero or at most one, suggesting that the automatic evaluation protocol can serve as a reliable proxy for human judgment.

A.14 Task Type Descriptions

To comprehensively evaluate models across a wide spectrum of real-world web development challenges, WebCompass defines 15 generation application domains, 16 diverse editing task types, and 11 repair defect types. Table 9 provides an overview, and the following subsections detail each category.

A.14.1 Editing Task Types

The 16 editing task types span from low-level UI components to full business workflows, ensuring broad coverage of frontend engineering skills. They are organized into four categories:

Complex Components. This category includes *Data Table* (sortable, paginated, filterable table with row selection and inline editing), *Rich Text Editor* (WYSIWYG editor with formatting toolbar, link/image insertion, and form-synced output),

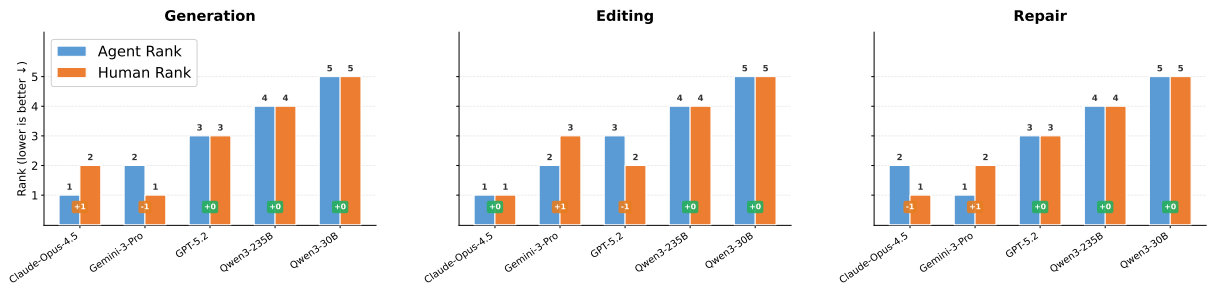


Figure 19: Comparison of model rankings between agent-based automatic evaluation and human evaluation across three tasks (Generation, Edit, Repair) and three task-adapted quality dimensions (executability, functional, and visual; see Table 2). Blue and orange bars represent the rankings assigned by the agent evaluator and human annotators, respectively (lower rank indicates better performance). The colored badges below each model name denote the rank difference (Human – Agent): **green** indicates perfect agreement ($\Delta = 0$), **orange** indicates a minor discrepancy ($|\Delta| = 1$), and **red** indicates a larger discrepancy ($|\Delta| \geq 2$). Overall, the agent evaluator shows strong alignment with human judgments, with most rank differences being zero or one.

Generation (15 Types)		Editing (16 Types)		Repair (11 Types)	
1	E-commerce & Fintech	1	Data Table	1	Occlusion
2	Enterprise & Productivity	2	Rich Text Editor	2	Crowding
3	Social & Communication	3	Drag & Drop Interface	3	Text Overlap
4	Data Science & Analytics	4	Tree View	4	Alignment
5	Content Creation & Multimedia	5	Real-time Dashboard	5	Color & Contrast
6	Entertainment & Streaming	6	Infinite Scroll	6	Overflow
7	Game Development & Gaming	7	Async Form Validation	7	Sizing/Proportion
8	Education & Learning	8	File Upload with Progress	8	Loss of Interactivity
9	Simulation & Scientific Modeling	9	Parallax Scrolling	9	Semantic Error
10	Infrastructure & System Mgmt.	10	Page Transitions	10	Nesting Error
11	DevTools & Engineering	11	Particle Effects	11	Missing Attributes
12	Logic & Workflow Visualization	12	Skeleton Loading		
13	Location Services & Transit	13	Shopping Cart		
14	Information & Personal Branding	14	User Authentication		
15	Lifestyle & Niche Utilities	15	Multi-step Wizard		
		16	Notification Center		

Table 9: Detailed taxonomy of Generation, Editing, and Repair tasks in WebCompass. Generation covers 15 application domains; Editing defines 16 modification operations; Repair addresses 11 front-end defect types spanning visual, semantic, and interactive dimensions.

Drag & Drop Interface (draggable items with drop-zone feedback, cross-container reordering, and state persistence), and *Tree View* (nested expand/collapse tree with cascading selection and search filtering).

Frontend–Backend Integration. This category covers *Real-time Dashboard* (live-updating metric cards with animated counters and sparkline

charts), *Infinite Scroll* (scroll-triggered lazy loading with skeleton placeholders and end-of-content handling), *Async Form Validation* (debounced server-side validation with inline status indicators and submit gating), and *File Upload with Progress* (drag-and-drop upload with per-file progress bars, queue management, and cancel support).

Advanced Animations. This category encompasses *Parallax Scrolling* (multi-layer differential scroll speeds with viewport-triggered fade/scale effects), *Page Transitions* (coordinated enter/exit animations such as fade, slide, and zoom between SPA content views), *Particle Effects* (canvas-based particle system with physics, cursor interaction, and connection lines), and *Skeleton Loading* (shimmer-animated placeholders matching content structure with smooth reveal).

Business Scenarios. This category includes *Shopping Cart* (full cart flow with quantity controls, real-time totals, and localStorage persistence), *User Authentication* (login, registration, and password-recovery forms with validation and auth state management), *Multi-step Wizard* (step indicator with per-step validation, cross-step data persistence, and review summary), and *Notification Center* (notification dropdown with unread badge, categorized alerts, and mark-as-read actions).

A.14.2 Repair Defect Types

The 11 repair defect types cover visual, semantic, and interactive failure modes commonly encountered in frontend development, organized into three dimensions:

Visual Layout. This dimension includes seven defect types: *Occlusion* (one element covers another due to incorrect z-index stacking), *Crowding* (spacing between elements is removed or collapsed, causing visual clutter), *Text Overlap* (text overflows its container and overlaps with adjacent content), *Alignment* (elements are offset from their expected grid or sibling alignment), *Color & Contrast* (text color is too close to the background, reducing readability), *Overflow* (content exceeds a fixed-size container without proper overflow handling), and *Sizing/Proportion* (elements are given extreme or distorted dimensions).

Semantic Correctness. This dimension includes *Semantic Error* (semantic HTML tags replaced with non-semantic equivalents, e.g., `<h1>` replaced by `<div>`) and *Nesting Error* (invalid HTML nesting, e.g., `<a>` inside `<a>`, or `<div>` inside `<p>`).

Interactive Usability. This dimension includes *Loss of Interactivity* (interactive elements disabled or blocked via pointer-events: none) and *Missing Attributes* (accessibility or functional attributes removed, e.g., alt, aria-label).

A.15 Prompt Templates

This section presents the prompt templates used in our pipeline, covering task prompts for code generation, editing, and repair (§A.15.3–A.15.5), evaluation prompts for LLM-as-a-Judge and Agent-as-a-Judge (§A.15.6–A.15.7), and auxiliary prompts for checklist generation and error analysis (§A.15.2–A.15.1).

A.15.1 Error Analysis Prompt

Below is the prompt used to classify point deductions into standardized error types and root causes. The full prompt includes the complete taxonomy definitions, a decision flowchart, point allocation rules, and worked examples.

Error Analysis Prompt (Part 1: Taxonomy & Rules)

You are a web development QA analyst. Classify each point deduction from LLM-generated web pages into standardized error types.

== Error Type Taxonomy (3-level) ==

E1 - Code Execution Errors

- E1.1 | Page Crash / White Screen | Page fails to render entirely
- E1.2 | Console Error (Blocking) | JS error that prevents feature from working
- E1.3 | Resource Loading Failure | Missing images, fonts, external dependencies

E2 - Functional Errors

- E2.1 | Feature Missing | A required feature/component is entirely absent
- E2.2 | Feature Incomplete | Feature exists but is only partially implemented
- E2.3 | Logic Error | Feature exists but produces wrong behavior/output
- E2.4 | Interaction Error | Wrong event type, trigger condition, or user flow
- E2.5 | Data/Content Error | Wrong text, numbers, items, or dynamic content

E3 - Visual/Style Errors

- E3.1 | Layout Structure Error | Wrong arrangement, grid, flex direction, positioning
- E3.2 | Color/Theme Error | Wrong colors, gradients, or visual theme
- E3.3 | Typography Error | Wrong font, size, weight, line-height
- E3.4 | Spacing/Sizing Error | Wrong margin, padding, width, height
- E3.5 | Animation/Transition Error | Missing, wrong, or poorly executed animations
- E3.6 | Visual Fidelity Gap | General mismatch with reference image/video

E4 - Non-Functional Errors

- E4.1 | Performance Issue | Lag, high CPU, memory leak, slow rendering
- E4.2 | Accessibility Issue | Missing alt text, poor contrast, no keyboard nav
- E4.3 | Responsiveness Issue | Breaks at different viewport sizes

== Root Cause Analysis (select the MOST specific one) ==
Requirement Misunderstanding | Prompt clearly stated X, LLM built Y

- Requirement Omission | LLM skipped/forgot an explicit requirement
- Insufficient Reproduction | Visual/behavioral gap vs reference image/video
- Capability Limitation | LLM lacks skill for this technique

```
Hallucination          | LLM used non-existent
                        | APIs or fabricated behavior
Oversimplification     | LLM took shortcuts, used
                        | simpler approach
null                   | Cannot determine or none
                        | of the above
```

== Decision Flowchart ==

For each deducted point:

1. Does the page crash or show blocking console errors?
-> E1.x
2. Does a required feature not work as specified?
-> E2.x
3. Does it work but look wrong?
-> E3.x
4. Does it look right but have non-functional issues?
-> E4.x

== Points Allocation Rule ==

When a single checklist item has multiple issues mentioned in reason:

- Allocate points proportionally to issue severity
- If unclear, split evenly among identified issues
- Critical failures get more points than minor issues

== Output Format ==

Return a JSON array:

```
[{"checklist_id": <id>, "task": "<task>",
  "score": <score>, "max_score": <max_score>,
  "errors": [{"type": "<E1.1|...|E4.3>",
    "description": "<concise description>",
    "points_deducted": <number>,
    "root_cause": "<label|null>"}]]
```

== Rules ==

1. sum(points_deducted) must equal max_score - score for each item
2. Each error gets exactly ONE type code
3. Full-score items -> empty errors array
4. Multiple distinct issues -> separate error objects

Error Analysis Prompt (Part 2: Few-Shot Examples)

== Example 1: Runtime error ==

Input: {"id":1, "task":"Page loads correctly",
"max_score":5, "score":0,
"reason":"Uncaught ReferenceError: initApp is not
defined.
Page shows white screen."}

Output:

```
[{"checklist_id":1, "task":"Page loads correctly", "score":0,
  "max_score":5, "errors":[{"type":"E1.2",
    "description":"Uncaught ReferenceError prevents page init",
    "points_deducted":5, "root_cause":"Hallucination"}]]
```

== Example 2: Multiple issues in one item ==

Input: {"id":3, "task":"Test mouse wake effect",
"max_score":12, "score":6,
"reason":"Wake works on mousedown+drag but NOT on
hover-only movement. Dissipation too slow (visible
after 3s)."}
Output:

```
[{"checklist_id":3, "task":"Test mouse wake effect", "score":6,
  "max_score":12, "errors":[{"type":"E2.4",
    "description":"Wake requires mousedown instead of hover",
    "points_deducted":4,
    "root_cause":"Requirement Misunderstanding"},
  {"type":"E2.3",
    "description":"Wake dissipation too slow (>3s)",
    "points_deducted":2, "root_cause":null}]]
```

== Example 3: Style issue ==

Input: {"id":5, "task":"Check header layout",
"max_score":6, "score":4,
"reason":"Logo and nav are present but laid out
vertically
instead of horizontal row."}

Output:

```
[{"checklist_id":5, "task":"Check header layout", "score":4,
```

```
"max_score":6, "errors":[{"type":"E3.1",
  "description":"Header elements stacked vertically
  instead of required horizontal layout",
  "points_deducted":2,
  "root_cause":"Insufficient Reproduction"}]]
```

Now analyze the following checklist:
[CHECKLIST]

A.15.2 Checklist Generation Prompt

Below is the prompt used to generate evaluation checklists. The prompt instructs an LLM to produce structured checklist items spanning the Generation dimensions Runnability, Spec Implementation, and Design Quality. We show the core instructions and an abridged example; the full prompt includes a complete 13-item few-shot example.

Checklist Generation Prompt (Part 1: Role & Format)

You are a senior and extremely detail-oriented code review expert.
You are proficient in multiple programming languages, frontend technologies, interaction design, and UI aesthetics. Your task is to generate a checklist for evaluating responses to the given [query].

Role Definition

- Responsibility: Act as a member of an authoritative technical review committee--objective, comprehensive, and impartial.
- Attitude: Meticulous, professional, and uncompromising.
- Aesthetic Standards: Excellent design taste and high standards for user experience.

Output Format Requirements (Each checklist item must include)

- task: A clear, single-purpose task for the UI agent to verify.
- category: One of: Runnability | Spec Implementation | Design Quality
- operation_sequence: Steps the UI agent should perform (2-4 steps, verifiable through screenshots/screen recordings).
- expected_result: Specific, observable success criteria.
- criteria: Strict scoring rules (clear pass/fail thresholds).
- max_score: Maximum score for this item (must be a number).

Checklist Generation Prompt (Part 2: Dimensions)

Evaluation Dimensions and Checklist Structure

1. Runnability (Fixed 1 item, worth 10 points)
Fixed universal checklist item:

```
{"task": "Does the page load correctly and run without errors?",
  "category": "Runnability",
  "operation_sequence": "1. Open Console panel 2. Load the page
  3. Check for red error messages 4. Check Network for 404/500",
  "expected_result": "Page loads completely, no Console errors,
  no failed requests, all static resources load successfully",
  "criteria": "Full 10 pts; JS errors -5; Resource 404 -3;
  White screen = 0; Warnings do not deduct",
  "max_score": 10}
```

```

### 2. Spec Implementation (6-10 items, worth 60-70 points)
Core part -- generated entirely based on the Query:
- First 4-5 items: core functionality, main interactions, key user journeys (most stringent, hardest to pass)
- Remaining items: secondary features, edge cases, robustness
- Score per item: 8-15 points, allocated by importance

### 3. Design Quality (1 universal + 1-2 Query-specific, 20-25 pts)
Universal item evaluates: color harmony, layout spacing (8px multiple principle), typography (body font >=14px, line-height >1.5x). Query-specific items check domain visuals (e.g., game board design, chart readability, animation smoothness).

## Important Notes
- Hard requirement: sum of all max_score must equal 100
- Hard requirement: 10-16 checklist items total
- Each item must be verifiable through screenshots/console/network panel/visual alignment checks

```

Checklist Generation Prompt (Part 3: Abridged Example)

```

## Complete Example (abridged -- showing 3 of 13 items)
Query: [Multiplayer online chess game with lobby, game board, result modal, and profile pages...]

[{"task": "Does the page load correctly and run without errors?",
  "category": "Runnability",
  "operation_sequence": "1. Open Console 2. Load page ...",
  "expected_result": "Page loads, no errors, all resources OK",
  "criteria": "Full 10; JS errors -5; 404 -3; crash = 0",
  "max_score": 10},
{"task": "Test chessboard piece selection and valid move indicators functionality",
  "category": "Spec Implementation",
  "operation_sequence": "1. Start game 2. Click white pawn (e2) 3. Verify highlights and move indicators 4. Click valid square to confirm move with animation",
  "expected_result": "Piece highlighted, dots on empty squares, rings on captures, smooth ~200ms animation",
  "criteria": "Full 12; no highlight -4; no indicators -4; no animation -2; wrong move -2",
  "max_score": 12},
{"task": "Verify dark theme with correct primary/accent colors",
  "category": "Design Quality",
  "operation_sequence": "1. Screenshot Lobby and Game pages 2. Verify dark gradient background 3. Check primary/accent colors 4. Verify semi-transparent cards",
  "expected_result": "Dark gradient bg, wood-brown primary, royal blue interactive, gold accents, cohesive dark theme",
  "criteria": "Full 5; light theme -5; inconsistent colors -2",
  "max_score": 5},
... (10 more items omitted for brevity)]

Query:
---
[QUERY]
---
```

A.15.3 Generation Prompts

We use three generation prompts corresponding to the three input modalities: text, image, and video.

All share a common output contract requiring pure Markdown with fenced code blocks. Below we present each variant.

Text-Guided Generation Prompt

```

You are a highly skilled professional front-end engineer.

Your task: Based on the web design document below, generate a complete runnable web project repository.

Hard output contract (MUST follow):
1) Your entire response MUST be pure Markdown text.
2) ABSOLUTELY NO explanations, no extra commentary.
3) Every file MUST be emitted using the following format:
  # path/to/file.ext
  ```ext
 <full file content>
  ```
4) The heading line MUST start with '# ' followed by the file path (relative path).
5) The code fence language MUST match the file type.
6) Include all necessary files so the project can run.
7) Do NOT nest triple backticks inside code blocks.

```

Few-shot examples:

```

# index.html
```html
<!doctype html>
<html>
 <head>
 <meta charset="utf-8" />
 <title>Demo</title>
 <link rel="stylesheet" href="styles.css" />
 </head>
 <body>
 Hello
 <script type="module" src="main.js"></script>
 </body>
</html>
```

# styles.css
```css
body { font-family: system-ui; }
```

# main.js
```js
console.log('ok')
```

```

Web design document:

```

---
[DOCUMENT]
---
```

Vision-Guided Generation Prompt

```

You are a highly skilled professional front-end engineer.

Your task: Based on the web design document and the reference screenshots provided, generate a complete runnable web project repository. The screenshots represent the target UI and function. The generated site should match the screenshots as closely as possible.

```

```

Hard output contract (MUST follow):
1-7) [Same output contract as Text-Guided Generation]

```

```

Startup requirements:
- MUST include a README.md with the simplest way to run locally.
- MUST be runnable via a static server (no backend). Prefer Vite or plain static files.

```

Web design document:

```
---
[DOCUMENT]
---
```

Video-Guided Generation Prompt (Part 1: Analysis Protocol)

You are a world-class front-end engineer with expertise in creating pixel-perfect web reproductions. Your task is to analyze these video frames and create a complete, production-ready web project repository that exactly replicates the demonstrated interface with professional-grade quality.

MISSION: Create a flawless HTML reproduction that matches the video demonstration in every detail, achieving 95%+ quality score.

COMPREHENSIVE ANALYSIS PROTOCOL:

- 1. TEMPORAL SEQUENCE ANALYSIS:**
 - Study frame progression to understand user interactions
 - Identify animation sequences, timing, and easing patterns
 - Map state transitions and user feedback mechanisms
 - Recognize loading states, hover effects, micro-interactions
 - Document exact timing and duration of animations
- 2. VISUAL DESIGN EXTRACTION:**
 - Extract precise color values (prefer hex codes: #RRGGBB)
 - Identify typography: families, sizes, weights, line heights
 - Measure spacing: margins, padding, gaps (use rem/em units)
 - Analyze shadows: box-shadow values, blur, spread, inset
 - Document border radius, opacity, and gradient effects
 - Note z-index layering and stacking contexts
- 3. LAYOUT & STRUCTURE ANALYSIS:**
 - Identify layout systems: Flexbox, CSS Grid, or positioning
 - Map responsive breakpoints and mobile adaptations
 - Document component hierarchy and nesting structure
 - Analyze alignment, distribution, and spacing patterns
- 4. INTERACTION PATTERN RECOGNITION:**
 - Button states: normal, hover, active, focus, disabled
 - Animation triggers: click, hover, scroll, load events
 - State management: data flow and component updates
 - User feedback: visual confirmations and error states

Video-Guided Generation Prompt (Part 2: Implementation)

TECHNICAL IMPLEMENTATION REQUIREMENTS:

HTML5 STRUCTURE (MANDATORY):

- Use semantic HTML5 elements (main, section, article, nav)
- Include proper meta tags: charset, viewport, description
- Implement accessibility: ARIA labels, alt text, roles
- Ensure valid HTML5 markup

CSS IMPLEMENTATION (MANDATORY):

- Use CSS custom properties (--variables) for theming
- Implement modern layout: Flexbox and CSS Grid
- Create smooth animations with cubic-bezier easing
- Use transform3d for hardware acceleration
- Implement responsive design with mobile-first approach
- Include CSS reset/normalize for consistency

JAVASCRIPT FUNCTIONALITY (MANDATORY):

- Write modern ES6+ with proper error handling (try/catch)
- Use requestAnimationFrame for smooth animations
- Add performance optimizations: debouncing, throttling
- Implement proper event delegation and cleanup

ANIMATION & INTERACTION STANDARDS (MANDATORY):

- Use CSS transforms for performance (translate3d, scale)
- Implement 60fps smooth animations
- Add appropriate transition durations (200-500ms)
- Create natural easing: ease-out entrance, ease-in exit

OUTPUT SPECIFICATIONS (CRITICAL):

- Your entire response MUST be pure Markdown text.
- ABSOLUTELY NO explanations or extra commentary.
- Every file MUST use: # path/to/file.ext ```text ...

- Keep assets inline; avoid external dependencies.

Begin your comprehensive analysis and create the ultimate reproduction that achieves 95%+ quality score.

A.15.4 Editing Prompts

The text-guided and vision-guided editing tasks share the same system prompt. The only difference is that the vision-guided variant additionally includes current-state screenshots in the user message. We present the shared system prompt once, followed by the two user-message variants.

Editing System Prompt (shared by Text & Vision variants)

You are an expert frontend developer. Your task is to edit the provided web code based on the given instructions. You will receive the current code and a set of editing instructions.

****Output Format Requirements:****

- Use search/replace blocks to indicate modifications
- Each block must be wrapped in <search_replace path="...">
- </search_replace> tags
- The `path` attribute must specify the relative file path (e.g., "index.html", "resources/style.css")
- Each block must contain one <search> and one <replace>

Return XML format with the following structure:

```
<search_replace path="path/to/file">
<search>
exact text to find in the original file
</search>
<replace>
replacement text with the modification applied
</replace>
</search_replace>
```

If you want to create additional files:

```
<search_replace path="path/to/new_file">
<search></search>
<replace>
complete code for the new file
</replace>
</search_replace>
```

The search block for new files should be empty.

Important:

- The <search> block must contain the EXACT text from the original file (including whitespace and indentation).
- The <replace> block contains the modified code.
- One <search_replace> block can only contain one pair of <search> and <replace>.
- You can include multiple <search_replace> blocks if you need to modify multiple locations or files.

- You must complete the task in single response.

Text-Guided Editing — User Message

```
== User Message ==
## Task Description
Task 0 - {task_type}: {description}
Task 1 - {task_type}: {description}
...

## Source Code
The following is the current code that needs to be
modified:
<code_context>
<file path="index.html"> ... </file>
<file path="style.css"> ... </file>
</code_context>
```

Vision-Guided Editing — User Message (extends Text variant)

```
== User Message ==
## Task Description
Task 0 - {task_type}: {description}
...

## Source Code
<code_context>
<file path="index.html"> ... </file>
</code_context>

## Current State Screenshots
The following screenshots show the current state:
[Current Screenshot: page_1.png]
[image_1]
[Current Screenshot: page_2.png]
[image_2]
...
```

A.15.5 Repair Prompts

Similarly, the diagnostic and visual-diagnostic repair tasks share the same system prompt. The visual-diagnostic variant additionally includes before-fix and target-state screenshots.

Repair System Prompt (shared by Diagnostic & Visual variants)

You are an expert frontend developer. Your task is to repair the provided web code based on the given defect types. You will receive the current code with a set of defect types to fix.

Here are the issue types and explanations (the code may not contain all of these issues):

- Occlusion: Elements incorrectly layered causing important content to be covered due to improper z-index or positioning.
- Crowding: Elements too close together due to missing or insufficient spacing (margins/padding).
- Text Overlap: Text overlaps with other elements due to insufficient container size or improper positioning.
- Alignment: Elements not properly aligned with the grid system or sibling elements.
- Color Contrast: Insufficient contrast between text and background colors affecting readability.
- Overflow: Content exceeds container boundaries without proper overflow handling.
- Sizing Proportion: Elements have incorrect dimensions or aspect ratios that distort their appearance.
- Loss of Interactivity: Interactive elements disabled or blocked

```
via disabled attributes or pointer-events: none.
- Semantic Error: Semantic HTML tags replaced with
  generic divs
  or spans reducing accessibility.
- Nesting Error: HTML elements nested in invalid ways
  that
  violate HTML specifications.
- Missing Attributes: Required attributes missing from
  elements
  (e.g., alt, aria-label).
```

```
**Output Format Requirements:**
[Same search/replace format as Editing prompts]
```

Important:
[Same constraints as Editing prompts]

Diagnostic Repair — User Message

```
== User Message ==
You have only {N} issues to fix, and you cannot fix more
than
{N} issues.

## Source Code
The following is the current code that needs to be
modified:
<code_context>
<file path="index.html"> ... </file>
</code_context>
```

Visual-Diagnostic Repair — User Message (extends Diagnostic)

```
== User Message ==
You have only {N} issues to fix, and you cannot fix more
than
{N} issues.

## Source Code
<code_context>
<file path="index.html"> ... </file>
</code_context>

## Current State Screenshots
The following screenshots show the current (defective)
state:
[Current Screenshot: page_1.png]
[image_1]
...

## Target State Screenshots
The following screenshots show the expected result:
[Target Screenshot: page_1.png]
[image_1]
...
```

A.15.6 LLM-as-a-Judge Prompts

We use separate judge prompts for editing and repair tasks, each with task-specific scoring dimensions on a 0–10 scale. The repair judge additionally receives ground-truth code modifications and fixed screenshots as reference.

Edit Task Judge Prompt

```
## Task Description
You are evaluating whether code modifications properly
implement
user's instructions for UI changes. You will receive:
1. Task Instructions: multi-line text, each line follows:
  Task <idx> - <task_type>: <description>
2. Generated Code Modifications: the search/replace
  blocks
3. Original UI Screenshot: the before-modification state
4. Modified UI Screenshot: the after-modification visual
  result

## Evaluation Framework
```

Score each task independently across three dimensions (0-10):

- Instruction Targeting: Patch applicability and task-attempt coverage
- Feature Integrity: Whether original and new functionality is correct
- Style Conformance: Visual quality and consistency with original style

Evaluation Criteria

1. Instruction Targeting (0-10 points)

- Are the search/replace blocks syntactically correct?
- Does the modified code run without errors?
- Are changes applied in the correct files/locations?

2. Feature Integrity (0-10 points)

- Are original UI interactions preserved?
- Are interactive elements still usable?
- Are there regressions in functionality?

3. Style Conformance (0-10 points)

- Does the modified UI match expected visual outcome?
- Is the visual style consistent where not changed?
- Are colors, fonts, spacing, and layout harmonious?

Important Notes

- CRITICAL: You MUST evaluate ALL tasks listed
- Visual results are the PRIMARY indicator of success
- Output ONLY valid JSON, no additional text

Output Format

```
{"task_scores": [{"task_idx": 0,
  "task_type": "<from Task 0 - <task_type>: ...>",
  "reasoning": "<detailed evaluation>",
  "instruction_targeting": <0-10>,
  "feature_integrity": <0-10>,
  "style_conformance": <0-10>}, ...]}
```

- CRITICAL: You MUST evaluate ALL defects listed
- Visual comparison with ground-truth is the PRIMARY indicator
- Compare the generated fix with both the defective state and the ideal ground-truth fix
- Output ONLY valid JSON, no additional text

Output Format

```
{"task_scores": [{"task_idx": 0,
  "task_type": "<from Defect 0 - <task_type>: ...>",
  "reasoning": "<detailed evaluation>",
  "root_cause_targeting": <0-10>,
  "interaction_integrity": <0-10>,
  "reference_fidelity": <0-10>}, ...]}
```

A.15.7 Agent-as-a-Judge Prompt

The Agent-as-a-Judge system uses two prompt templates: one for generation (producing code from a task description) and one for verification (scoring the generated webpage against a checklist via browser interaction). The verification prompt has two variants—with and without reference images—that share the same execution flow. We present the shared verification prompt below; the image variant additionally instructs the agent to review reference screenshots under the screenshots/ directory.

Repair Task Judge Prompt

Task Description

You are evaluating the effectiveness of UI defect repair tasks.

You will receive:

1. Defect Description: multi-line text, each line follows:
Defect <idx> - <task_type>: <description>
2. Ground-Truth Code Modifications: the ideal fix (reference)
3. Generated Code Modifications: the produced fix
4. Before-Fix UI Screenshot: defective state (red box markers)
5. After-Fix UI Screenshot: the actual repair result
6. Ground-Truth Fixed UI Screenshot: the ideal fix result

Evaluation Framework

Score each defect repair independently (0-10 per dimension):

- Root-Cause Targeting: Patch applicability and root-cause localization
- Interaction Integrity: Whether original and repaired functionality is correct
- Reference Fidelity: Visual quality vs. ground-truth reference

Evaluation Criteria

1. Root-Cause Targeting (0-10 points)

- Are the search/replace blocks syntactically correct?
- Does the modified code run without errors?
- Do changes target the root cause without introducing errors?

2. Interaction Integrity (0-10 points)

- Are original UI interactions preserved after the fix?
- Are interactive elements still usable?
- Are there regressions in functionality?

3. Reference Fidelity (0-10 points)

- How closely does the fix match the ground-truth reference?
- Are layout, colors, typography aligned with the reference?
- Does the fix look natural and harmonious?

Important Notes

Agent-as-a-Judge: Generation Prompt

```
<task>
{problem_statement}
</task>
```

Based on the design requirements in <task>, generate a complete front-end web project repository.

Agent-as-a-Judge: Verification Prompt (Part 1: Objective & Rules)

You are a strict QA website tester who must rigorously evaluate the Runnability, Spec Implementation, and Design Quality of a webpage based on a checklist (verification and scoring only -- no fixes).

Below is the user's requirements document for the webpage:

```
<docs>{instruction}</docs>
```

Your objective: Based on a normal user's browsing journey, use mcp_tools: mcp__chrome-devtools to verify whether each item in the checklist is met, update every entry in checklist.json where score is null to a definitive score, provide reproducible evidence (reason), and save screenshots to image/.

=====
Mandatory Rules (Must Be Followed)
=====

- 1) Absolutely do not modify/fix the original website project code.
- 2) The only content you are allowed to create/modify is:
 - checklist.json
 - Screenshot files in the image/ directory
- 3) Complete all tasks in a single run. Before all tasks are completed, you must call tools for verification in every round.

- 4) Reach every required state through a visible UI entry point using ordinary browser operations (click, type, scroll, and navigate). If the state or function is not reachable through the UI, mark the checklist item as failed with score 0.
- 5) JavaScript is supplementary verification only after the relevant UI state has been reached. It may inspect DOM, CSS, and functional behavior or reduce repetitive mechanical operations, but it must not mutate hidden state, invoke an internal function to bypass the UI, change an assertion, or award credit for an unreachable function.
- 6) Selector adaptation may relocate DOM selectors only. The checklist criteria and behavioral assertions are immutable.

You may only end when no entries with score=null remain in checklist.json.

Agent-as-a-Judge: Verification Prompt (Part 2: Execution Flow)

```
=====
Mandatory Execution Flow (No Steps May Be Skipped)
=====

Step 0: Prepare Output Directory
1) Ensure image/ folder exists in the project directory.
2) Take screenshots for every key state verification.

Step 1: Conduct Code Review First (Read-Only)
1) Read repository code related to page entry points,
   routing,
   interactions, requests, and error handling.
2) Compile verifiable points: entry URLs, key buttons/
   forms,
   potential error points, data sources and loading logic
3) Code review only guides test paths; final scores must
   be
   based on actual webpage behavior.

Step 2: Read checklist.json
1) Locate all entries where score is null.
2) Extract task / operation_sequence / expected_result.

Step 3: Open and Actually Test the Webpage
1) Use mcp_chrome-devtools for interactive verification
   (clicking, typing, navigating, scrolling, etc.).
2) For each item, first perform the prescribed UI
   operations and
   confirm that the target state is reachable. If no UI
   entry
   point exists, assign score 0 and record the evidence.
3) Only after the state is reached, use JavaScript as a
   supplementary check of DOM, CSS, or functional
   behavior.
4) Take screenshots as evidence. For Design Quality tasks
   ,
   combine UI screenshots for scoring.

Step 4: Immediately Write Back to Checklist
1) After each verification, write back to checklist.json:
   - score: Change from null to a definitive score.
   - reason: Single-line string with reproducible
     evidence.

=====
Key Rule: Entry Point Failure => Cascading Failure
=====
If the website entry point is unavailable (blank screen/
crash/
infinite loading): take screenshot as evidence, assign
failure
scores to all dependent items.

=====
Termination Condition
=====
```