WebCoderBench: Benchmarking Web Application Generation with Comprehensive and Interpretable Evaluation Metrics

Chenxu Liu[1*†], Yingjie Fu[1*], Wei Yang[3], Ying Zhang[24‡], Tao Xie[14‡]
[1]Key Lab of HCST (PKU), MOE; SCS, Peking University, China
[2]Key Lab of HCST (PKU) & NERCSE, MOE; Peking University, China
[3]University of Texas at Dallas, USA
[4]Beijing Tongming Lake Information Technology Application Innovation Center, China
Correspondence: zhang.ying@pku.edu.cn, taoxie@pku.edu.cn

Abstract

Web applications (web apps) have become a key arena for large language models (LLMs) to demonstrate their code generation capabilities and commercial potential. However, building a benchmark for LLM-generated web apps remains challenging due to the need for real-world user requirements, generalized evaluation metrics without relying on ground-truth implementations or test cases, and interpretable evaluation results. To address these challenges, we introduce WebCoderBench, the first real-world, generalized, and interpretable benchmark for web app generation. WebCoderBench comprises 1,572 real user requirements, covering diverse modalities and expression styles that reflect realistic user intentions. WebCoderBench provides 24 fine-grained evaluation metrics across 9 perspectives, combining the rule-based and LLM-as-a-judge paradigms for fully automated, objective, and general evaluation. Moreover, WebCoderBench adopts human-preference-aligned weights over metrics to yield interpretable overall scores. Experiments across 12 representative LLMs and 2 LLM-based agents show that there exists no dominant model across all evaluation metrics, offering an opportunity for LLM developers to optimize their models in a targeted manner for a more powerful version.

1 Introduction

Web applications (web apps), leveraging their standardized, lightweight, cross-platform characteristics and vast customization ecosystem, have emerged as a key battleground for large language models (LLMs) to capitalize on their commercial potential. In industrial scenarios, LLMs take user requirements as input and automatically generate the code of the corresponding web app as output. For these LLMs, a real-world, generalized, and interpretable benchmark not only facilitates objective understanding of their capabilities but also provides critical direction for subsequent optimization.

Benchmarking LLM-generated web apps demands substantial manual effort and specialized design to address three key challenges. First, the dataset of the benchmark should be collected from real-world user requirements **(authenticity)**. Due to the various backgrounds, users can describe their requirements in different styles and expect web apps of diverse complexities and specialization. It is necessary to keep the authentic user requirements to reflect real usage. Second, the evaluation metrics of the benchmark should be general to accommodate the open-ended nature of natural-language instructions **(generality)**. Real-world requirements are often vague and can be satisfied through multiple implementations and designs, making it impractical to rely on fixed ground-truth implementations or test cases. Third, the evaluation results produced by the benchmark should both align with user preferences and provide interpretable insights **(interpretability)**. User satisfaction with web apps is inherently multi-dimensional, shaped by diverse preferences that correspond to different LLM capabilities. Hence, fine-grained and interpretable evaluation outcomes are essential for comprehensive analysis and targeted improvement.

Existing benchmarks for evaluating LLMs on web app generation fall into three main categories, each facing limitations in addressing the aforementioned challenges. First, synthetic benchmarks (Lu et al., 2025; Sun et al., 2025; Zhu et al., 2025; Xu et al., 2025; Zhang et al., 2025) construct datasets using LLM-generated or expert-written natural language requirements, failing to capture the diversity and authenticity of real-world user expressions. Second, reference-based benchmarks (Beltramelli, 2018; Vu et al., 2025; Li et al., 2024; Gui et al., 2024; Yun et al., 2024; Gui et al., 2025) re-

[*]Equal contribution.
[†]The email address of the first author is: chenxuliu@stu.pku.edu.cn
[‡]Corresponding authors.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 11632–11666
July 2-7, 2026

quire LLMs to reproduce web apps from provided screenshots or sketches, thus lacking the generality needed to evaluate open-ended user requirements. Third, arena-style benchmarks (LMArena.ai, 2025; Xiao et al., 2025) collect real-world user requirements and rank LLMs through large-scale human voting in blind evaluations (Chiang et al., 2024). While these results align with human preferences, such benchmarks depend heavily on manual annotations and fail to deliver fine-grained, quantitative evaluations for a deep analysis.

To fill this gap, we propose WebCoderBench, the first real-world, generalized, and interpretable benchmark offering comprehensive and automated evaluation metrics for LLM-based web app generation. For **authenticity**, WebCoderBench contains 1,572 real-world user requirements collected from an online LLM service of our industrial partner. The collected requirements span multiple modalities and cover a wide range of expression styles, from precise to ambiguous and from technical to colloquial, corresponding to expected web app artifacts of varying complexity, faithfully capturing the expressions of real users. For **generality**, WebCoderBench provides 24 fine-grained evaluation metrics across 9 perspectives, combining rule-based metrics with the LLM-as-a-judge paradigm to achieve fully automated and objective evaluation without relying on ground-truth implementations or test cases. For **interpretability**, WebCoderBench not only reports scores of individual metrics, but also leverages user-preference-based weights across metrics to derive an overall score that aligns with real-world user priorities, providing both quantitative and human-aligned insights.

Experiments across 12 representative LLMs and 2 LLM-based agents demonstrate that WebCoderBench provides interpretable and human-aligned evaluations of web app generation. The results reveal users' varying preferences across different perspectives of web app quality and uncover the strengths and weaknesses of existing models in detail, offering actionable insights for improvement. Notably, even the most advanced LLMs fail to achieve the highest scores across all 24 metrics.

In summary, this paper makes the following main contributions:

- We present WebCoderBench, the first benchmark that enables comprehensive, interpretable, and automated evaluation for web app generation.
- We build a dataset of 1,572 real user requirements, faithfully capturing real-world usage.
- We design 24 fine-grained evaluation metrics spanning 9 perspectives, integrating the rule-based and LLM-as-a-judge paradigms, with human-preference-based weighting for interpretability and fairness.
- We evaluate 12 representative LLMs and 2 LLM-based agents on our benchmark. The results show that WebCoderBench can provide interpretable results for targeted improvements of existing LLMs. Our results are publicly available[1].

2 Related Work

This section first introduces LLM-driven systems capable of generating web apps, and then lists benchmarks for evaluating their capabilities.

2.1 LLM-Driven Systems for Web App Generation

LLM-driven systems for web app generation mainly fall into three categories. The first category is foundation models, including general-purpose models (e.g., GPT5 (OpenAI, 2025), Claude (Anthropic, 2025b), Gemini (Team et al., 2023), DeepSeek (Liu et al., 2024), Qwen (Yang et al., 2025), Doubao (ByteDance, 2025a)) and code-specialized models (e.g., DeepSeek-Coder (Guo et al., 2024), Qwen-Coder (Hui et al., 2024), StarCoder (Li et al., 2023)). These models serve as the core reasoning engines, capable of multi-language code synthesis, long-context reasoning, and supporting iterative repairs via execution feedback. The second category is IDE/CLI-centric coding agents (e.g., Cursor (Anysphere, 2025), Trae (ByteDance, 2025c), Windsurf (Windsurf, 2025), and Claude Code (Anthropic, 2025a)). These agents support manipulating local projects through project bootstrapping, dependency resolution, multi-file editing, and command-line tool control. The third category is chat-based coding agents (e.g., Manus (Manus, 2025), MiniMax agent (MiniMax, 2025a), Doubao Coding (ByteDance, 2025b), and Lovable (Lovable, 2025)). These agents provide an accessible conversational interface to translate natural language requirements directly into

[1] https://huggingface.co/spaces/CompileError/WebCoderBench

web applications, often utilizing cloud-based sandboxes for instant rendering and deployment without requiring local environment setup.

2.2 Benchmarks for Web App Generation

Web app generation has long been an active research topic. Since the pioneering work of pix2code (Beltramelli, 2018), a large body of benchmarks and datasets (Bhathal and Gupta, 2025; Vu et al., 2025; Li et al., 2024; Gui et al., 2024; Yun et al., 2024; Gui et al., 2025) has focused on generating web apps from screenshots or sketches. These approaches typically conduct evaluation by comparing the generated web app code or rendered screenshots against the corresponding ground-truth code or original screenshots.

In recent years, the emergence of LLMs has greatly lowered the barrier to web app generation, enabling users to create customized applications through natural-language or multi-modal requirements. However, as summarized in Table 1, existing benchmarks fall short of addressing the full spectrum of the three challenges in this domain. First, some benchmarks rely on LLM-generated or expert-curated requirements, failing to capture the authenticity and diversity of real user behavior. Second, some benchmarks depend on predefined test cases or ground-truth code for evaluation. Third, some benchmarks are not fully automated, heavily relying on manual labeling.

3 Dataset

WebCoderBench includes 1,572 real-world user requirements across 20 application categories. Of these requirements, 1,413 are text-only, 123 include images, and 36 include URLs as additional resources. In terms of clarity, 78 requirements are vague, 730 are intermediate, and 764 are clear. We further divide these 1,572 requirements into five complexity levels with 179, 433, 658, 259, and 43 samples from simple to complex, respectively. Overall, the dataset exhibits sufficient diversity to represent real-world user requirements.

3.1 Data Collection

The data collection pipeline of WebCoderBench is illustrated in Figure 1. We construct the original dataset by collecting one week of anonymized and filtered real-world online data from our industrial partner and randomly sampling 5,000 user requirements.

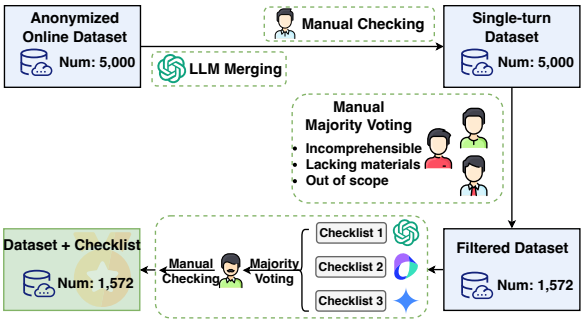

Figure 1: The dataset construction process of WebCoderBench.

Requirement: Create a Pomodoro timer application with a minimalist design that plays a sound when the timer finishes.
Visual style: Use #B52B2D (tomato red) as the primary color and #F9F9F5 (light yellow) as the background color.
Functional ground-truth: ["Set timer duration", "Start timer", "Pause timer", "Reset timer", "Play sound when timer ends"]
Visual ground-truth: ["Minimalist design", "Primary color #B52B2D (tomato red)", "Background color #F9F9F5 (light yellow)"]
Content ground-truth: ["The sound to play when timer ends"]

Figure 2: An example user requirement with its corresponding ground-truth checklists.

First, to reduce bias caused by revision turns that depend on specific model outputs in multi-turn requirements, we merge multi-turn requirements into single-turn ones by human experts assisted by an LLM (Gemini-2.5-pro). Manual revisions are made to ensure fluency and content anonymization.

Next, three human experts review each requirement to remove those that are incomprehensible, lack supplementary materials (e.g., images), or are inapplicable to native HTML scenarios. We further conduct text-level and semantic-level deduplication using MinHash (Broder, 1997) and MiniLM (Wang et al., 2020) semantic embeddings, following the practice of ArtifactsBench (Zhang et al., 2025). The filtered dataset retains 1,572 requirements.

Finally, to enable objective evaluation, we establish ground-truth checklists for each requirement across three dimensions: functionality, visual design, and content. We adopt three LLMs (GPT-5-Chat-2025-08-07, Gemini-2.5-pro, and Doubao-Seed-1.6) to infer ground-truth checklists for each dimension. After that, human experts merge and validate the outputs to produce the final ground-truth checklists. Figure 2 shows an example user requirement with its corresponding ground-truth checklists. The checklists comprise only high-level and minimal requirement points that the web app artifact should satisfy.

Table 1: Comparison with existing related benchmarks.

Benchmark	Modal	Sample	Source	Automated	Metrics	Ground Truth
WebGen-Bench (Lu et al., 2025)	Text	101	LLM + human experts	Yes	LLM as a judge + test case execution	Test cases
FullFront (Sun et al., 2025)	Text + Image	1,800	LLM	Yes	Code & visual similarity + LLM as a judge	Code + Images
FrontendBench (Zhu et al., 2025)	Text	148	LLM	Yes	Test case execution	Test cases
Web-Bench (Xu et al., 2025)	Text	100	Human experts	Yes	Test case execution	Test cases
WebDev Arena (LMArena.ai, 2025)	Text	10,501	Real user requirements	No	Pairwise manual labeling	-
Design Arena (Arena, 2025)	Text	-	Real user requirements	No	Pairwise manual labeling	-
ArtifactsBench (Zhang et al., 2025)	Text	1,825	LLM + human experts	Yes	LLM as a judge	Checklist
WebCoderBench (ours)	Text + Image + URL	1,572	Real user requirements	Yes	(Rule-based + LLM as a judge) * preference	Checklist

Table 2: Dataset statistics of WebCoderBench.

Type	Number	Type	Number
Application Category		**Clarity of Requirement**	
- AI-powered	74	- Clear	764
- BBS	6	- Intermediate	730
- Corporate Website	41	- Vague	78
- Data Visualization	59	**Style of Expression**	
- E-commerce	40	- Technical	683
- Enterprise Backend	116	- Colloquial	724
- Entertainment	435	- Role-playing	60
- Fintech	32	- Analogy	105
- Health Care	14	**Artifact Complexity**	
- IoT Interface	10	- Highly Simple	179
- Multimedia	34	- Simple	433
- News Media	5	- Medium	658
- Online Education	131	- Complex	259
- Online Office Suite	3	- Highly Complex	43
- Personal Webite	49	**Input Modality**	
- Public Service	27	- Text Only	1,413
- Scientific Demo	69	- Text with Images	123
- Social Media	13	- Text with URLs	36
- Tourism	10		
- Utility Website	404		

3.2 Dataset Statistics

We perform detailed and multi-dimensional classifications of each requirement to enable an in-depth analysis of the characteristics and distribution of the dataset. The results are shown in Table 2, with all tags manually labeled by human experts assisted by an LLM (Gemini-2.5-pro).

In terms of application category, WebCoderBench covers a wide range of web apps, including Utility Websites (404 requirements), Entertainment (435), and Online Education (131), which together account for the majority of the samples, reflecting the diversity of real-world web scenarios. Less frequent but important categories, such as AI-powered, Fintech, and Scientific Demo, are also included, showing the variety of the dataset.

Regarding clarity, most requirements are clear (764) or intermediate (730), while only a small portion (78) are vague, showing that our dataset primarily focuses on interpretable and executable tasks, but also retains challenging tasks for LLMs to explore. The style of expression dimension indicates that both technical (683) and colloquial (724) descriptions are prevalent, capturing different levels of formality in user expression, with occasional role-playing (60) and analogy-based (105) requirements adding linguistic diversity.

The artifact complexity dimension spans from highly simple to highly complex outputs, with most samples falling into medium (658) and simple (433) levels, showing balanced difficulty for model evaluation. In terms of input modality, our dataset mainly includes text-only requirements (1,413), complemented by text with images (123) and text with URLs (36), allowing evaluation of both unimodal and multi-modal understanding abilities. It is worth noting that, unlike the task of generating code from screenshots (Beltramelli, 2018; Gui et al., 2025), our user requirements can include images and URLs that are intended to serve as page content rather than as reference designs.

In terms of length statistics, the shortest requirement in our dataset contains 3 characters, while the longest contains 16,198 characters, with an average length of 313.74 characters. Measured using the GPT-5 (OpenAI, 2025) tokenizer, the shortest requirement contains 3 tokens, the longest 9,235 tokens, and the average length is 202.31 tokens.

In summary, these statistics show that our dataset covers a wide variety of web app types, styles, and complexities, making it promising for evaluating LLMs in web app generation.

4 Evaluation Metrics

To comprehensively assess the quality of web app artifacts generated by LLMs, we design a set of evaluation metrics from multiple perspectives. Inspired by practices in the text-to-image domain (Hartwig et al., 2025), our evaluation considers two major aspects: general quality and alignment quality. These aspects are further divided into nine perspectives encompassing 24 detailed evaluation metrics, jointly providing a comprehensive assessment. The metrics are defined based on

Table 3: The evaluation metrics used in WebCoder-Bench, with rule-based metrics in white background, while metrics using the LLM-as-a-judge paradigm are shaded in gray. The "Input" and "Render" columns indicate the input modality (S: screenshot; C: code) of each evaluation metric and whether it requires page rendering.

Aspects	Perspectives	ID + Evaluation Metrics	Input	Render
General Quality	Code Quality	1. General Functionality Correctness	C	✗
		2. Best Practices	C	✓
		3. Error Handling	C	✗
		4. Runtime Console Errors	C	✓
		5. Static Syntax Checking (Linting)	C	✗
	Visual Quality	6. General Visual Experience	S	✓
		7. Component Style Consistency	C	✗
		8. Icon Style Consistency	C	✓
		9. Layout Consistency	S	✓
		10. Layout Sparsity	S	✓
		11. Visual Harmony Degree	S	✓
	Content Quality	12. Copywriting Quality	C	✗
		13. Media Quality	C	✗
		14. Placeholder Quality	C	✗
		15. Resource Validity	C	✗
	Performance Quality	16. General Performance	C	✓
	Accessibility	17. Accessibility Core Metrics	C	✓
		18. Cross-Browser Compatibility	C	✓
		19. Mobile Device Compatibility	C	✓
	Maintainability	20. Code Redundancy Rate	C	✓
		21. Comment Rate	C	✗
Alignment Quality	Functional Alignment	22. Functional Alignment	C	✗
	Visual Alignment	23. Visual Alignment	C	✗
	Content Alignment	24. Content Alignment	C	✗

public standards (e.g., Google Lighthouse (Google, 2025), Lint (Hint, 2025; Stylelint, 2025; ESLint, 2025), W3C design principles (Consortium, 2025)) and internal guidelines from our industrial partner (e.g., visual design and copywriting standards). All the metrics are listed in Table 3, with detailed information provided in Appendix A. Each metric produces a quantitative score ranging from 0 to 100, where a higher score indicates higher quality.

4.1 General Quality

General quality measures the overall quality of the generated web app, regardless of the specific user requirement. We evaluate general quality from six distinct perspectives, using a combination of automated rule-based metrics and the LLM-as-a-judge paradigm (with Gemini-2.5-pro as the judge according to our experience).

Code Quality evaluates the functional correctness and implementation of the generated code, focusing on correctness, robustness, and adherence to engineering practices.

Visual Quality evaluates the visual presentation and design of the generated web app, emphasizing layout consistency, stylistic coherence, and overall aesthetic experience.

Content Quality evaluates the informativeness and the quality of resources loaded in the generated web app.

Performance Quality evaluates the runtime behavior of the generated web app, measuring the efficiency of page rendering and resource loading.

Accessibility evaluates the disability-friendliness and platform compatibility of the generated web app, assessing usability across different devices and browsers.

Maintainability evaluates the readability, reusability, and ease of long-term maintenance of the generated code.

4.2 Alignment Quality

Alignment quality measures how well the generated web app meets the corresponding user requirements. We evaluate alignment quality using human-labeled ground-truth checklists (Section 3.1) as references and employing the LLM-as-a-judge paradigm (with GPT-5-chat as the judge according to our experience). The three metrics, namely **Functional Alignment**, **Visual Alignment**, and **Content Alignment**, evaluate the consistency between the generated web app and the user requirement in terms of functionalities, visual appearance, and textual/multimedia content, respectively.

4.3 Weight Assignment

After defining evaluation metrics, a crucial process is to combine the outcome scores in a meaningful way to generate an overall score, reflecting human preference. Our process of generating the overall score is shown in Figure 3. Existing studies (Lin et al., 2025; Zhang et al., 2025) typically assign weights to each metric either uniformly or heuristically. However, real-world users do not treat all the metrics equally and can prioritize certain perspectives (e.g., emphasizing code quality while paying less attention to maintainability). Consequently, heuristic weighting neither aligns with user priorities nor ensures fairness.

To obtain preference-aligned weights for each metric, we conduct an internal survey with our industrial partner. Specifically, we ask each participant to rank the 24 metrics according to their perceived importance. To reduce the cognitive burden on participants, each of them is first asked to rank the nine perspectives listed in Table 3, and subsequently rank the metrics within each perspective. During a three-day survey period, our questionnaire receives 1,076 views and 899 responses, yielding

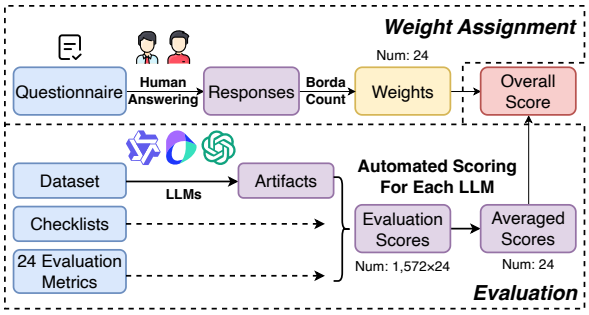

Figure 3: The weight assignment and evaluation workflow of WebCoderBench.

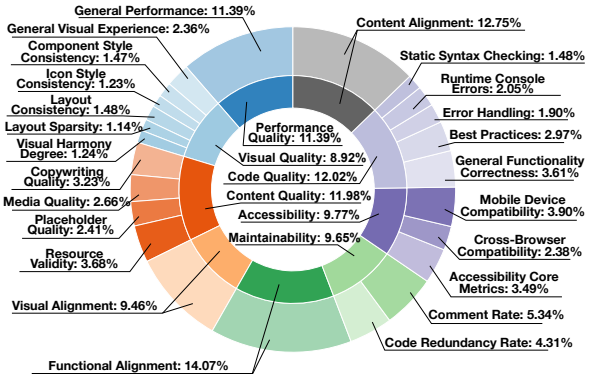

Figure 4: The weight proportion of each perspective and each evaluation metric.

a response rate of 83.55%. We further filter the responses by completion time, retaining only those that take more than two minutes to complete. As a result, we obtain 141 valid responses. According to a recent study (Memon et al., 2020), a sample size of more than 100 responses can be considered as sufficient for ranking and regression analyses. Among the valid responses, there are 21 data scientists, 19 product managers, 22 legal personnel, 23 front-end/back-end developers, 2 designers, 39 operations staff, and 15 quality assurance personnel, according to their user personas.

We further apply the Borda Count (Young, 1974) algorithm to extract weights from responses. This algorithm assigns a score to each item (perspective or metric) based on its position in each participant's ranking. The weight of each item is then obtained by summing its scores across all participants and normalizing by the total score over all items. The weight of each metric is calculated by multiplying its own weight by the weight of the perspective to which it belongs. The weights of all the metrics sum to 1. The resulting weights are shown in Figure 4.

The overall score of each model is calculated by summing the z-scores (Al Shalabi et al., 2006) of all the metrics, averaged over the 1,572 samples in the dataset, and weighted by the corresponding metric weights. We adopt z-scores to ensure that the scores of different metrics are on a comparable scale with consistent discriminability, thereby emphasizing the effect of the weights. The z-scores are calculated by $z_{i,j} = \frac{x_{i,j} - \mu_j}{\sigma_j}$, where $x_{i,j}$ denotes the raw score of the i-th sample on the j-th metric, and μ_j and σ_j denote the mean and standard deviations of this metric over all samples, respectively.

Through this process, we establish a data-driven mechanism that grounds the metric weighting scheme in authentic human preferences rather than arbitrary heuristics. The derived weights effectively capture how users implicitly trade off among different quality perspectives, enabling our overall evaluation score to reflect real-world user preferences.

5 Evaluation Results and Analysis

5.1 Settings

We evaluate 12 representative LLMs and 2 LLM-based agents on WebCoderBench. The selected models span multiple families (e.g., DeepSeek (Liu et al., 2024), Qwen (Yang et al., 2025), Gemini (Team et al., 2023), GLM (Zeng et al., 2025), GPT (OpenAI, 2025)), multiple versions (e.g., base, instruct, thinking, coder), multiple modalities (e.g., text-only, multi-modal), and multiple capability types (e.g., standard (Chen et al., 2025; MiniMax, 2025b), agentic (Manus, 2025; MiniMax, 2025a)). We invoke each LLM using its standard APIs with the recommended parameters and identical system prompts. We let each model generate a single artifact for each requirement.

For LLM-based agents, we manually enter prompts and requirements through their web interfaces to obtain the generated artifacts, and further constrain their outputs to native HTML by appending specific control prompts. Due to labor constraints, we collect artifacts for 165 requirements from these agents, corresponding to approximately one-tenth of the full dataset.

To enable uni-modal LLMs (e.g., DeepSeek) to handle multi-modal requirements, we use Gemini-2.5-pro to generate textual descriptions for each image and provide these descriptions as input to the uni-modal LLMs. The descriptions are restricted to objective visual content and do not introduce

Table 4: Weighted z-scores for each perspective, the overall score, and the ranking of each model. The percentage value indicates how many standard deviations the weighted value deviates from the average. Rows 1–8 correspond to open-source LLMs, rows 9–12 to closed-source LLMs, and rows 13–14 to LLM-based agents. Note that the scores of LLMs are averaged among the whole dataset, while those of LLM-based agents are averaged among a random subset of 165 requirements.

ID	Model	Code Quality	Visual Quality	Content Quality	Performance Quality	Accessibility	Maintainability	Functional Alignment	Visual Alignment	Content Alignment	Overall Score (Sum)	Ranking
1	DeepSeek-R1-0528	-0.31%	-1.27%	1.81%	0.28%	1.33%	1.09%	-0.98%	0.21%	0.25%	2.41%	7
2	DeepSeek-V3.1	-0.86%	-1.60%	-1.04%	0.82%	-1.18%	-0.04%	0.00%	0.35%	-0.32%	-3.88%	13
3	DeepSeek-V3.1-Terminus	0.17%	2.11%	-1.58%	1.37%	1.09%	2.58%	0.52%	-0.03%	0.03%	6.26%	4
4	DeepSeek-V3.1-Thinking	-0.98%	-1.51%	-0.44%	1.17%	-1.05%	-0.04%	-0.14%	-0.24%	-0.50%	-3.73%	12
5	GLM-4.5	0.28%	2.33%	5.03%	0.46%	0.17%	0.61%	1.58%	0.84%	0.46%	11.76%	2
6	Qwen3-Coder-Plus	-0.63%	-1.65%	-0.43%	1.70%	0.89%	1.51%	-1.55%	-1.04%	-1.10%	-2.31%	11
7	Qwen3-235B-A22B-Instruct	-1.38%	0.32%	-0.60%	0.68%	0.22%	0.53%	-0.05%	-0.01%	0.25%	-0.04%	10
8	MiniMax-M2	1.13%	-0.94%	-0.56%	0.40%	-1.06%	2.27%	3.20%	1.50%	1.96%	7.89%	3
9	Gemini-2.5-Pro	0.89%	0.99%	-0.26%	-2.25%	0.33%	-1.74%	0.91%	1.02%	0.38%	0.27%	8
10	GPT-4o-2024-11-20	-2.00%	-1.69%	-1.91%	-2.49%	-0.41%	-5.37%	-11.04%	-6.62%	-7.12%	-38.65%	14
11	GPT-5-Codex-High	0.41%	2.30%	-1.14%	-2.14%	1.78%	-7.52%	2.10%	1.80%	2.39%	-0.03%	9
12	GPT-5-High	2.15%	1.36%	-1.17%	1.27%	-2.05%	0.88%	5.51%	2.36%	3.50%	13.81%	1
13	Manus	0.25%	-0.76%	0.85%	-0.56%	-1.00%	1.58%	3.29%	0.33%	1.26%	5.24%	5
14	MiniMax Agent	-0.21%	-2.45%	6.27%	-1.39%	-1.30%	1.00%	1.71%	0.77%	0.03%	4.43%	6

any additional information beyond what is observable in the original images. Our goal is to evaluate each model based on the modalities supports by it, rather than penalizing uni-modal models for lacking visual capabilities (e.g., assigning a score of zero or excluding them from multi-modal requirements). We believe that this setup provides a fair comparison under modality constraints.

Due to the diversity of user requirements, we observe two distinct usage patterns for images and URLs. A majority of the requirements ask the LLM to use images and URLs as reference examples for implementation (95/123 for images and 29/36 for URLs). The remaining requirements ask the LLM to include images and URLs as assets in the final generated web apps (28/123 for images and 7/36 for URLs). For all requirements, we encode images in the base64 format before passing them to the model, and URLs are included as strings within the input specification. For requirements that treat images and URLs as assets, the generated web app includes images in the base64 format and incorporates URLs as hyperlinks in the final output.

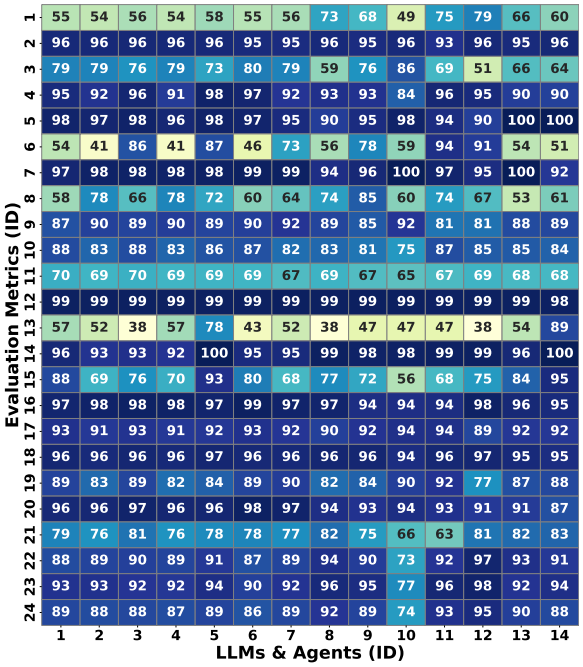

Figure 5: The detailed raw scores of 24 evaluation metrics for each LLM and LLM-based agent, with the x-axis indices denoting the IDs of evaluation metrics (corresponding to Table 3), and the y-axis indices denoting the IDs of models (corresponding to Table 4).

5.2 RQ1: Main Results

We present the weighted z-scores for each perspective, the overall scores, and the model rankings in Table 4, and the raw scores of the 24 evaluation metrics in Figure 5. Detailed z-scores for each metric are provided in Appendix B due to space limits.

Across all the evaluated LLMs and agents, GPT-5-High attains the highest overall score, greatly outperforming other models through its strong ability to align with requirements and its consistently positive results across most perspectives. Among the open-source LLMs, GLM-4.5 achieves the best effectiveness and ranks second overall, while MiniMax-M2 also performs competitively. In contrast, GPT-4o exhibits the weakest effectiveness, with all scores falling below the average and six out of nine perspectives ranking last among all LLMs.

There exists no single model that dominates all

perspectives. GLM-4.5 is the only model achieving above-average effectiveness across all nine perspectives, indicating balanced and reliable capabilities. However, even GLM-4.5 cannot outperform the average in all the 24 fine-grained evaluation metrics, suggesting that current LLMs remain specialized rather than universally strong. The competition among the LLMs remains tight, with no single model emerging as decisively superior. Given our comprehensive and interpretable evaluation metrics, LLM developers are able to optimize their models in a targeted manner for a more powerful future version.

The comparison between open-source and closed-source models reveals a rapidly narrowing effectiveness gap. Although GPT-5-High remains the most effective model overall, the strong effectiveness of GLM-4.5 and MiniMax-M2 shows that recent open-source LLMs are increasingly competitive, with less than a 6% gap from GPT-5-High.

The results also illustrate the accelerated pace of model evolution and development. The models released within the past year consistently obtain high scores across multiple evaluation perspectives, while earlier models such as GPT-4o exhibit substantial gaps, performing below average in all perspectives and falling behind newer models by a considerable margin. This divergence highlights the rapid turnover in LLM effectiveness.

The LLM-based agents generally score above average but fall short of expectations. Their ability to access online resources and generate complex pages results in low performance and accessibility due to the increased complexity of external resources. The complexity of the generated UI further degreades visual quality. However, these agents typically align well with user requirements due to their planning and task-oriented reasoning capabilities. This trend can also cause an LLM-based agent to be less effective than its base model. For example, the overall score of MiniMax Agent is lower than that of MiniMax-M2, indicating that while agents introduce enhanced capabilities, these strengths come at the cost of performance and visual consistency.

Compared to coder models (e.g., Qwen3-Coder-Plus and GPT-5-Codex), generalist models (e.g., Qwen3-Instruct and GPT-5-High) present better effectiveness, showing that the task of web app generation requires not only coding ability, but also general understanding of requirements and real-world knowledge. This finding also aligns to ArtifactsBench (Zhang et al., 2025).

Table 5: The overall scores averaged over the models and the questions for each question type.

Type	Score	Type	Score
Clarity of Requirement		**Input Modality**	
- Clear	-0.02%	- Text Only	0.85%
- Intermediate	-0.55%	- Text with Images	-8.68%
- Vague	5.98%	- Text with URLs	1.54%
Artifact Complexity		**Style of Expression**	
- Highly Simple	0.94%	- Technical	-0.72%
- Simple	-1.11%	- Colloquial	1.14%
- Medium	0.18%	- Role-playing	-0.89%
- Complex	0.53%	- Analogy	-1.75%
- Highly Complex	4.82%		

Although the weighted z-scores appear numerically close (most within 3%), they represent differences in standard deviations from the mean for each metric, and each of them is weighted by a small percentage (shown in Figure 4). Therefore, even small value differences can be meaningful.

5.3 RQ2: Effect of Question Types

The overall scores averaged over the models and the questions for each question type are shown in Table 5. We aggregate the scores of all the models to show a general and overall trend, instead of analyzing each model separately.

We find that the results do not entirely conform to the straightforward intuition that the models tend to perform worse on vague and highly complex requirements. We manually inspect the outcomes for reasoning. For requirement clarity, the models generally perform better under vague requirements, because such descriptions provide greater freedom and reduce penalties from fine-grained mismatches with the requirements, whereas clear requirements impose strict constraints that are easy to violate. Regarding artifact complexity, requirements of complex artifacts offer rich functional and contextual cues that help the models infer page structure (with only 3 out of 43 highly complex requirements are classified as vague), while requirements of simple artifacts lack sufficient information and thus lead to deviations. In terms of input modality, inputs with images degrade effectiveness, indicating that the models remain unstable in mapping visual content to page structures. For expression style, the models are most effective when processing colloquial requirements, indicating that the models are more suitable for the common expressions used by the general public.

Table 6: Weights derived from the different user personas, where each column corresponds to front-end/back-end developers (Dev), quality assurance personnel (QA), data scientists (DS), product managers (PM), operations staff (Ops), legal personnel (Le), designers (Des), and all participants (ALL), respectively, and each row represents one of the nine perspectives.

Per.	Dev	QA	DS	PM	Ops	Le	Des	ALL
Code	13.5%	13.3%	12.7%	12.7%	10.3%	11.9%	5.6%	12.0%
Vis.	8.6%	7.4%	9.5%	7.5%	9.9%	8.2%	20.8%	9.0%
Con.	10.6%	13.1%	12.4%	9.2%	13.0%	13.3%	6.9%	12.1%
Per.	8.6%	11.7%	11.1%	9.5%	13.1%	13.5%	5.6%	11.5%
Acc.	8.7%	10.9%	10.6%	9.2%	9.3%	10.5%	11.1%	9.9%
Mai.	11.2%	10.0%	10.3%	11.7%	9.0%	6.8%	5.6%	9.6%
FA	14.7%	13.0%	11.5%	17.3%	13.8%	14.0%	16.7%	13.8%
VA	9.2%	8.1%	9.1%	10.4%	9.8%	8.7%	18.1%	9.7%
CA	14.9%	12.4%	12.7%	12.6%	11.7%	13.1%	9.7%	12.5%

5.4 RQ3: Effect of User Personas

The scores provided in Table 4 are calculated using weights derived from all the valid questionnaire responses. However, users who belong to different user personas can prefer different perspectives. Table 6 shows the weights derived from the different user personas in our collected responses.

The results indicate that the different personas exhibit preferences that align well with the intuitive understanding over perspectives. For instance, the designers show a pronounced preference for visual quality. The code-related personas (the first four columns) tend to place greater emphasis on code quality and maintainability, whereas the operations staff and the legal personnel prioritize content quality and performance quality. Across the three alignment-related perspectives, all the personas assign high preference consistently, with the product managers and the designers exhibiting particularly strong preferences. These patterns highlight the domain-specific expectations.

6 Conclusion

In this paper, we have introduced WebCoderBench, a comprehensive benchmark consisting of 1,572 authentic user requirements and 24 evaluation metrics, providing an automated, comprehensive, and interpretable evaluation framework for the task of web app generation. Our evaluation results have revealed a narrowing effectiveness gap between open-source and closed-source LLMs, as well as the rapid evolution of the capabilities of LLMs, with no single model achieving dominant effectiveness across all the metrics. By incorporating the weights according to human preferences, WebCoderBench enables developers to optimize their models in a targeted manner based on interpretable evaluation results.

Acknowledgements

We thank our industrial partner. We thank the anonymous reviewers for their constructive feedback.

This work was partially supported by Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118), National Natural Science Foundation of China (Grants No. U25A6023, 92464301), and the China National Petroleum Corporation Science and Technology Project under grant No. 2024ZZ46-06.

Tao Xie is also affiliated with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education China; Institute of Systems for Advanced Computing at Fudan University, China; Shanghai Research Institute of Systems of Open Computing, China.

Limitations

Our benchmark has six limitations.

First, it currently evaluates only front-end web applications and restricts implementations to native HTML. We consider that this design is reasonable because front-end functionalities cover most real user needs, and the existing LLMs are still struggling to handle complex full-stack development tasks. Native HTML also enables consistent source-code analysis, whereas framework-specific formats (e.g., React) complicate automated evaluation. Nonetheless, we plan to incorporate backend tasks and support common frameworks in future versions.

Second, our dataset and our implementation of the evaluation metrics cannot be released publicly due to internal legal constraints, because our dataset contains real user requirements. Closed-source data and evaluation also prevent data leakage and evaluation hacking. We plan to maintain and update our leaderboard actively. We also plan to prepare another batch of user requirements for open-sourcing and cross-validation, so as to verify the generalizability and robustness of our findings across different requirement sets.

Third, the dataset distribution can influence evaluation results. To reduce this limitation, we follow

a standardized data-collection pipeline to obtain sufficient and realistic user requirements.

Fourth, results can be affected by the reliability of the manual annotations. We mitigate this limitation by leveraging our industry partner's mature crowd-sourcing workflow, where annotators have at least three years of development experience. We adopt a two-stage labeling strategy in which LLMs generate labels and humans verify them. Critical steps, such as dataset filtering, are triple-annotated, and majority voting is used to determine final labels.

Fifth, the design of evaluation metrics can impact the results. We aim to build a comprehensive, interpretable, and quantitative metric suite. Guided by industrial practices, public standards, and academic insights, we develop 24 metrics across 9 perspectives. We plan to further enrich the metric set as our future work.

Finally, metric correctness and stability can influence the evaluation. For rule-based metrics, we manually inspect 50 scoring instances per metric and conduct additional code reviews to make sure that the implementations are as expected. For LLM-as-a-judge metrics, we run each metric three times on 50 requirements and use the Mann–Whitney U test (Mann and Whitney, 1947) to ensure that the score variance across runs is significantly lower than the variance across models. We follow the prior work (Sun et al., 2025) for prompt design and use different LLMs for different metrics to mitigate preference leakage and bias (Chehbouni et al., 2025; Li et al., 2025; Sheng et al., 2025).

References

Luai Al Shalabi, Zyad Shaaban, and Basel Kasasbeh. 2006. Data mining: A preprocessing engine. *Journal of Computer Science*.

Anthropic. 2025a. Claude Code.

Anthropic. 2025b. Overview | Claude.

Anysphere. 2025. Cursor.

Design Arena. 2025. Design Arena: World's largest crowdsourced benchmark for AI-generated design.

Tony Beltramelli. 2018. Pix2Code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*.

Tanvir Bhathal and Asanshay Gupta. 2025. Websight: A vision-first architecture for robust web agents. *arXiv preprint arXiv:2508.16987*.

Andrei Z Broder. 1997. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences*.

ByteDance. 2025a. Doubao.

ByteDance. 2025b. Doubao Coding.

ByteDance. 2025c. TRAE - collaborate with intelligence.

Khaoula Chehbouni, Mohammed Haddou, Jackie Chi Kit Cheung, and Golnoosh Farnadi. 2025. Neither valid nor reliable? Investigating the use of LLMs as judges. *arXiv preprint arXiv:2508.18076*.

Aili Chen, Aonian Li, Bangwei Gong, Binyang Jiang, Bo Fei, Bo Yang, Boji Shan, Changqing Yu, Chao Wang, Cheng Zhu, and 1 others. 2025. MiniMax-M1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*.

Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Banghua Zhu, Hao Zhang, Michael Jordan, Joseph E Gonzalez, and 1 others. 2024. Chatbot arena: An open platform for evaluating LLMs by human preference. In *Proceedings of the 41st International Conference on Machine Learning*.

World Wide Web Consortium. 2025. W3C standards and drafts.

ESLint. 2025. ESLint: Find and fix problems in your JavaScript code.

Google. 2025. Introduction to Lighthouse.

Yi Gui, Zhen Li, Yao Wan, Yemin Shi, Hongyu Zhang, Bohua Chen, Yi Su, Dongping Chen, Siyuan Wu, Xing Zhou, and 1 others. 2025. WebCode2M: A real-world dataset for code generation from webpage designs. In *Proceedings of the ACM on Web Conference*.

Yi Gui, Zhen Li, Yao Wan, Yemin Shi, Hongyu Zhang, Yi Su, Shaoling Dong, Xing Zhou, and Wenbin Jiang. 2024. Vision2UI: A real-world dataset with layout for code generation from UI designs. *arXiv preprint arXiv:2404.06369*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Sebastian Hartwig, Dominik Engel, Leon Sick, Hannah Kniesel, Tristan Payer, Poonam Poonam, Michael Glockler, Alex Bauerle, and Timo Ropinski. 2025. A survey on quality metrics for text-to-image generation. *IEEE Transactions on Visualization and Computer Graphics*.

HTML Hint. 2025. HTML Hint: The static code analysis tool you need for your HTML.

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

Dawei Li, Renliang Sun, Yue Huang, Ming Zhong, Bohan Jiang, Jiawei Han, Xiangliang Zhang, Wei Wang, and Huan Liu. 2025. Preference leakage: A contamination problem in LLM-as-a-judge. *arXiv preprint arXiv:2502.01534*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161*.

Ryan Li, Yanzhe Zhang, and Diyi Yang. 2024. Sketch2Code: Evaluating vision-language models for interactive web design prototyping. *arXiv preprint arXiv:2410.16232*.

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. *arXiv preprint arXiv:2506.07818*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*.

LMArena.ai. 2025. WebDev Arena.

Lovable. 2025. Lovable.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. 2025. WebGen-Bench: Evaluating LLMs on generating interactive and functional websites from scratch. *arXiv preprint arXiv:2505.03733*.

Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*.

Manus. 2025. Manus: Hands on AI.

Mumtaz Ali Memon, Hiram Ting, Jun-Hwa Cheah, Ramayah Thurasamy, Francis Chuah, and Tat Huei Cham. 2020. Sample size for survey research: Review and recommendations. *Journal of Applied Structural Equation Modeling*.

MiniMax. 2025a. MiniMax agent.

MiniMax. 2025b. MiniMax M2 & agent: Ingenious in simplicity.

OpenAI. 2025. Introducing GPT-5 for developers.

Huanxin Sheng, Xinyi Liu, Hangfeng He, Jieyu Zhao, and Jian Kang. 2025. Analyzing uncertainty of LLM-as-a-judge: Interval evaluations with conformal prediction. *arXiv preprint arXiv:2509.18658*.

Stylelint. 2025. Stylelint: A mighty CSS linter that helps you avoid errors and enforce conventions.

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*.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, and 1 others. 2023. Gemini: A family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Tung D Vu, Chung Hoang, and Truong-Son Hy. 2025. Multimodal graph representation learning for website generation based on visual sketch. *arXiv preprint arXiv:2504.18729*.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. 2020. MiniLM: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *Advances in Neural Information Processing Systems*.

Windsurf. 2025. Windsurf - the best AI for coding.

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*.

H Peyton Young. 1974. An axiomatization of Borda's rule. *Journal of Economic Theory*.

Sukmin Yun, Rusiru Thushara, Mohammad Bhat, Yongxin Wang, Mingkai Deng, Jinhong Wang, Tianhua Tao, Junbo Li, Haonan Li, Preslav Nakov, and 1 others. 2024. Web2Code: A large-scale webpage-to-code dataset and evaluation framework for multimodal LLMs. *Advances in Neural Information Processing Systems*.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, and 1 others. 2025. GLM-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*.

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*.

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*.

Appendices

Table of Contents:

A Evaluation Metrics

We present the detailed information of each evaluation metrics in Table 7, Table 8, and Table 9. In each of the tables, the "Purpose" column describes the rationale of each metric and the aspect that each metric aims to evaluate. The "Implementation Detail" column describes the implementation detail of each metric in natural language. The "Score Calculation" column describes how the final scores (ranging from 0 to 100) are computed, with scaling applied to ensure a uniform distribution and sufficient discriminability.

Table 7: A detailed introduction to the purpose, implementation details, and score calculation formulas of our evaluation metrics (Part I).

Metric	Purpose	Implementation Detail	Score Calculation
General Functionality Correctness	This metric is used to evaluate the overall quality, correctness, and compliance of JavaScript, CSS, and HTML code on a webpage.	This metric parses code files using a large language model and evaluates them according to ten predefined code-quality criteria, with each criterion scored on a 0–10 scale.	score = sum(scores_of_ten_check_rules)
Best Practices	This metric aims to assess the webpage's adherence to web-development best practices, including but not limited to the use of HTTPS, avoiding vulnerable JavaScript libraries, avoiding deprecated APIs, providing correct document declarations, properly setting image aspect ratios, and making reasonable permission requests.	This metric integrates the Lighthouse auditing framework to construct a best-practices evaluation system consisting of nine core indicators. It parses the Lighthouse-generated JSON report and extracts the composite score of the "best-practices" category as the overall result.	score = Lighthouse_best_practices_score * 100
Error Handling	This metric is designed to systematically evaluate the coverage of exception-handling mechanisms in JavaScript code by identifying functions requiring error handling and comparing them with the actual implementation of exception-protection structures, thereby providing a quantitative basis for code-quality analysis.	This metric implements a multi-level analysis pipeline to detect exception-handling mechanisms. It first extracts embedded JavaScript code from the HTML document, including content inside script tags and event-handler code. It then identifies functions using regex patterns that match multiple function-definition forms (function declarations, function expressions, arrow functions, etc.). A predefined heuristic rule set filters functions requiring error handling based on name prefixes and risk-operation keywords (such as asynchronous operations or data-storage APIs). For each identified target function, syntax analysis is applied to detect whether try-catch blocks, Promise.catch, or similar exception-handling structures are present.	score = (1 - log2(1 + (no_error_handling_num / (require_error_handling_num + 1)))) * 100
Runtime Console Errors	This metric evaluates HTML code quality by detecting error-level messages (such as ERROR, SEVERE) in the browser console to quantify code robustness.	This metric uses Selenium WebDriver to construct a controlled browser-testing environment. The HTML is loaded in headless mode to simulate runtime behavior. The core logic includes: counting lines of code via the file system as a baseline parameter; capturing runtime console output via browser log APIs; filtering logs using predefined error-level categories (e.g., ERROR, SEVERE). A state-isolation mechanism clears cookies and storage before each test to ensure independence.	errors_per_1k = (total_errors / total_lines) * 1000 raw_score = 100 - (errors_per_1k * 20) score = max(0, final_score)
Static Syntax Checking (Linting)	This metric aims to establish a multi-language code-quality evaluation benchmark by performing static analysis to detect syntax and compliance errors in HTML, CSS, and JavaScript files, thereby quantifying code quality and providing standardized scoring.	This metric initializes a multilingual detection class by configuring supported file-extension sets and ignorable rule sets. Based on extension-based routing, it invokes htmlhint, eslint, and stylelint for syntax and compliance checks. It parses each tool's JSON output to extract error-level descriptions, rule identifiers, and location information, applies rule-filtering to exclude specified rules, and aggregates results into a structured issue list and directory-level statistics.	errors_per_1k = (total_errors / total_lines) * 1000 raw_score = 100 - (errors_per_1k * 20) score=max(0, raw_score)
General Visual Experience	This metric reflects the user's overall perception when using the target web application, emphasizing evaluations of aesthetics, design quality, and stylistic consistency between visual presentation and content.	This metric uses a multimodal large model combined with specially designed prompts to output n scores across n evaluation dimensions, each expressed as an x–y interval.	score = min(sqrt(score_generated_by_LLM) * 10 + 20, 100)
Component Style Consistency	This metric evaluates the use of card components in the target webpage, focusing on whether card elements (such as titles, icons, and body text) exhibit consistent and standardized structure and styling.	This metric extracts all components in the webpage that contain "card"-related identifiers and filters out nodes that fail structural requirements (such as fewer than two children, inconsistent child-node counts within a group, or mismatched node hierarchy or tags). It then identifies parallel card components and evaluates consistency and completeness of titles, icons, and body-text elements to detect inconsistent components.	score = (1 - log2(1 + (inconsistent_num / (total_num + 1)))) * 100
Icon Style Consistency	This metric evaluates whether icon assets used in the web application belong to a consistent visual style, and whether inconsistencies exist in size, line weight, or underlying shape.	This metric extracts all SVG elements in the HTML and groups them according to shared container relationships. It evaluates icon-library consistency across six dimensions: icon-set consistency, size uniformity, stroke-width uniformity, background-shape consistency, background-color consistency, and background-padding uniformity, and reports the number n of failed dimensions.	score = max(0,100-25*failed_dimention_num)

Table 8: A detailed introduction to the purpose, implementation details, and score calculation formulas of our evaluation metrics (Part II).

Metric	Purpose	Implementation Detail	Score Calculation
Layout Consistency	This metric assesses the visual alignment quality of webpage layouts by detecting multi-dimensional alignment of elements (e.g., edges, center lines) and bottom alignment in multi-column layouts, thereby quantifying layout regularity.	This metric performs alignment evaluation through the following steps: applying Canny edge detection and morphological operations to extract element contours from the webpage screenshot; filtering based on size, nesting, and banner-region masks to retain valid layout elements; quantifying positional deviations of left/right edges, top/bottom boundaries, and center points; and for multi-column layouts, grouping elements into rows and comparing upper/lower boundaries to check inter-column alignment. The final score combines multi-dimensional alignment rates and bottom-alignment issues, accompanied by detailed diagnostics.	score = (1 - log2(1 + (total_errors / (total_elements + 1)))) * 100
Layout Sparsity	This metric evaluates spatial utilization and content density of webpage screenshots by detecting the largest continuous homogeneous grayscale region (blank rectangle) to quantify wasted whitespace. By calculating blank-space ratios and converting them into scores, it assesses whether the layout is overly sparse or contains extensive ineffective whitespace, supporting design-quality evaluation and user-experience optimization.	This metric identifies blank regions using a combination of grayscale-tolerance bucketing and maximal-rectangle detection. The image is converted to grayscale, and pixels are grouped into tolerance buckets (default threshold: 80), treating similar grayscale values as homogeneous and avoiding noise from minor variations. For each bucket's binary mask, a histogram-based maximal-rectangle algorithm with a monotonic stack finds the largest rectangular area. The largest rectangle across all buckets is recorded, and its area ratio to the full image is computed as the blank-space rate. A nonlinear mapping converts the rate into a 0–100 score, penalizing layouts with large blank areas. A visualization highlights the detected region with a red bounding box.	score = min(sqrt(100-sparsity_rate)*10, 100)
Visual Harmony Degree	This metric evaluates the color harmony of webpage interfaces by analyzing the spatial distribution and perceptual characteristics of dominant colors in the image, providing objective metrics for assessing color diversity, balance, and aesthetic coherence.	This metric evaluates color harmony using computer-vision and color-space conversion techniques. It extracts dominant colors via K-means clustering and performs multi-dimensional analysis in HSV space: color diversity via Euclidean distance, saturation balance via mean and variance, brightness contrast via value range and distribution concentration, hue harmony via geometric relations on the hue circle (complementary, triadic, etc.), and temperature balance via warm–cool ratios. A weighted blend (hue 0.30, brightness 0.25, saturation 0.20, diversity 0.15, temperature 0.10) produces the overall harmony score.	score = (diversity * 0.15 + saturation * 0.20 + brightness * 0.25 + hue * 0.30 + temperature * 0.10) * 100
Copywriting Quality	This metric systematically evaluates the overall quality of textual content in HTML interfaces. Through multi-dimensional quantitative analysis (such as accuracy, clarity, conciseness, consistency, and user-friendliness), it ensures conformity with UX design norms and industry compliance standards.	This metric extracts pure text from the HTML structure and evaluates it using rules built from predefined domain-specific vocabularies (such as system operations, compliance terms, and technical terminology). Using a dimensional scoring framework, it computes deviations from baseline across 15 quality indicators (such as terminology consistency, sentence-structure complexity, and information hierarchy). Quantitative features include lexical matching, statistical patterns (e.g., sentence length, punctuation mixing), and structural conformance (e.g., heading hierarchy). Scores are aggregated into an overall quality index.	score = Average(sub_score_1, sub_score_2, ..., sub_score_14)
Media Quality	This metric evaluates the quality of media resources (such as images and videos) to ensure compliance with accessibility standards (e.g., WCAG) and industry best practices, specifically by checking image clarity and video playability.	This metric parses the DOM and embedded scripts to extract all image and video resource URLs. Image clarity is assessed via a Laplacian-variance method; video availability is assessed by verifying metadata readability and first-frame decodability.	score = (clarity_score * 0.7) + (media_accessbility_score * 0.3)
Placeholder Quality	This metric systematically evaluates the appropriateness of image placement in webpage design by detecting four categories of issues—placeholder image misuse, image distortion, repetitive use, and improper SVG placeholders—and generating quantitative scores to measure visual coherence.	This metric uses BeautifulSoup to extract image elements and contextual metadata. It identifies placeholder images based on a predefined domain list, detects distortion by comparing natural and rendered aspect ratios via remote size probing and local attributes, identifies reuse by aggregating base URLs within card containers, and assesses the appropriateness of SVG placeholders in large containers using container-size analysis and semantic context (with special exemptions such as QR-code cases).	score = (1 - log2(1 + (bad_image_num / (total_image_num + 1)))) * 100

Table 9: A detailed introduction to the purpose, implementation details, and score calculation formulas of our evaluation metrics (Part III).

Metric	Purpose	Implementation Detail	Score Calculation
Resource Validity	This metric evaluates the availability of embedded resources by checking whether images, videos, audio files, and stylesheets referenced in HTML documents are accessible, ensuring content completeness and user-experience quality. As part of the web-quality benchmark, it quantifies resource load success rates.	This metric parses the HTML DOM to extract two classes of resource paths: static resources referenced via predefined tags (img, video, script, etc.) and URLs embedded in JavaScript code (matched with regex patterns such as imageUrl and src). It filters out template placeholders and special protocols (e.g., javascript:), then validates URLs by protocol: remote resources via layered HTTP requests (HEAD first, fallback to GET, with redirect and SSL handling), and local resources via file-system checks. The system outputs structured validation states and error attributions.	score = (1 - log2(1 + (invalid_resource_num / (total_resource_num + 1)))) * 100
General Performance	This metric uses the Lighthouse tool to automate performance assessments of HTML pages, quantifying performance metrics and generating detailed performance reports.	This metric integrates Lighthouse's performance-analysis modules. It launches an isolated local HTTP server to ensure a consistent environment, runs Lighthouse with performance-only collection and strict timeouts, and extracts the Performance category score and five key metrics (FCP, LCP, TBT, CLS, TTI), converting them into percentage-scale values.	score = Lighthouse_performance_score * 100
Accessibility Core Metrics	This metric systematically evaluates accessibility compliance of webpage content by detecting key accessibility features (such as alternative text, keyboard navigation support, and semantic structure) for alignment with WCAG standards, providing quantitative benchmarks for inclusive design.	This metric invokes the Lighthouse accessibility module and parses the resulting JSON report. It evaluates 11 core indicators, including alternative text, button names, contrast ratios, heading structure, form labels, link descriptions, ARIA compliance, document title, language declaration, and viewport configuration.	score = Lighthouse_accessibility_score * 100
Cross-Browser Compatibility	This metric evaluates compatibility across different browser versions, specifically examining the use of CSS properties and JavaScript APIs. It identifies features that may fail in unsupported target browsers to quantify compatibility levels.	This metric integrates the MDN Browser Compatibility Database (BCD) as a reference source to automate compatibility checks. It uses Playwright to load the webpage and inject scripts to extract actual CSS properties (from stylesheets and inline styles) and JavaScript APIs in use. These features are compared against BCD data to determine support status for the target browser versions.	score = (compatible_features / all_features) * 100
Mobile Device Compatibility	This metric evaluates layout adaptability in mobile viewports by detecting whether unexpected horizontal scrolling occurs at the document root, thereby quantifying compliance with responsive-design standards and providing a baseline metric for layout viewport width.	This metric hosts the HTML files on a local HTTP-server cluster and uses Playwright to emulate an iPhone 12 Pro environment. Injected JavaScript computes the difference between scrollWidth and clientWidth of documentElement to quantify horizontal overflow. Measurements are triggered on DOMContentLoaded and stabilized with rendering-frame synchronization and microsecond-level delays. Overflow pixels are mapped to an audit score.	score = max(0, 100 - (horizontal_overflow_pixels))
Code Redundancy Rate	This metric quantifies unused JavaScript and CSS resources to assess code redundancy, providing measurable indicators for performance optimization and code-quality maintenance.	This metric integrates relevant Lighthouse audits to extract redundancy metrics for unused JavaScript and unused CSS rules. It parses the Lighthouse JSON report to obtain numeric parameters such as potential byte savings and resource details. If no redundant resources are detected, a full score is returned; otherwise, the mean of relevant audit scores is used as the composite rating.	score = (Lighthouse_unused_javascript + Lighthouse_unused_css_rules) / 2 * 100
Comment Rate	This metric evaluates the readability and maintainability of HTML code by analyzing comment coverage and converting the comment ratio into standardized scoring.	This metric parses the HTML file to build a multilingual comment-detection framework. Using a line-based statistical method and state-machine algorithm, it tracks multi-line-comment boundaries (HTML , CSS /* /, JavaScript // and / */) and distinguishes comment lines from code lines. The comment-line ratio is used to compute a normalized evaluation score.	score = min(sqrt(comment_rate) * 10 + 60, 100)
Functional Alignment	This metric evaluates whether the HTML webpage satisfies user-specified functional modules and interactions.	This metric uses a large language model to parse HTML code and validate it against predefined functional criteria.	score = (passed_check_point_num / all_check_point_num) * 100
Visual Alignment	This metric evaluates whether the HTML webpage meets user-specified visual-design standards.	This metric uses a large language model to parse HTML code and validate it against predefined visual inspection criteria.	score = (passed_check_point_num / all_check_point_num) * 100
Content Alignment	This metric evaluates whether the HTML webpage satisfies user-specified content semantics and designated data.	This metric uses a large language model to parse HTML code and validate it against predefined content-semantic criteria.	score = (passed_check_point_num / all_check_point_num) * 100

B Unweighted Z-score Results

We present the detailed and unweighted z-scores in Table 10. Z-scores represent differences in standard deviations from the mean for each metric. We adopt z-scores to resize the raw scores of different metrics to a comparable scale, and then weight them according to our derived weights.

In this table, the values are greatly larger than those in Table 4, since these values are unweighted. The z-scores faithfully reflect the differences between models.

We can also rank the models according to their scores on each individual metric, yielding 24 sub-leaderboards that reflect the models' capabilities on specific metrics. Model developers can then target optimization efforts toward the metrics on which their models rank lower.

Table 10: Detailed unweighted z-scores for each metric and each model.

ID	Model	Code Quality					Visual Quality					Content Quality					Performance Quality	Accessibility			Maintainability		Functional Alignment	Visual Alignment	Content Alignment
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	DeepSeek-R1-0528	-0.3699	0.1256	0.1622	0.0938	0.1041	-0.3956	-0.0470	-0.5862	-0.0533	0.2614	0.1952	-0.0512	0.1059	0.0128	0.4508	0.0248	0.1433	0.1725	0.1076	0.0422	0.1701	-0.0697	0.0219	0.0195
2	DeepSeek-V3.1	-0.4223	0.1158	0.1518	-0.0284	0.0602	-0.8833	0.0801	0.2612	0.0505	-0.0731	0.0427	-0.0097	-0.0654	-0.1864	-0.1057	0.0718	-0.0806	-0.2108	-0.1006	0.0593	-0.0553	-0.0001	0.0368	-0.0252
3	DeepSeek-V3.1-Terminus	-0.2846	0.1971	0.0791	0.1495	0.1060	0.7438	0.0655	-0.2315	0.0350	0.2399	0.1721	-0.0305	-0.5671	-0.1470	0.1044	0.1203	0.1017	0.1550	0.0930	0.1273	0.3809	0.0369	-0.0029	0.0024
4	DeepSeek-V3.1-Thinking	-0.4038	0.1054	0.1582	-0.0948	0.0385	-0.8668	0.0488	0.2901	0.0781	-0.0606	0.0490	-0.0133	0.1260	-0.1942	-0.0728	0.1027	-0.0838	-0.1078	-0.1274	0.0760	-0.0680	-0.0098	-0.0252	-0.0395
5	GLM-4.5	-0.2080	0.1642	-0.0181	0.2018	0.1080	0.7800	0.0864	0.0392	0.0327	0.1503	0.0780	-0.0276	0.8704	0.2424	0.6038	0.0404	-0.0311	0.2128	-0.0587	0.0725	0.0558	0.1122	0.0885	0.0361
6	Qwen3-Coder-Plus	-0.3550	-0.0543	0.1874	0.1608	0.0849	-0.6834	0.1364	-0.4843	0.0737	0.1757	0.0451	0.0008	-0.3977	-0.0472	0.2005	0.1492	0.1679	-0.0163	0.0883	0.1934	0.1257	-0.1105	-0.1105	-0.0861
7	Qwen3-235B-A22B-Instruct	-0.2647	-0.1941	0.1561	-0.0488	-0.0281	0.2797	0.1230	-0.3037	0.1430	-0.1480	-0.1542	0.0517	-0.0613	-0.0344	-0.1425	0.0599	0.0006	-0.1347	0.1377	0.1295	-0.0049	-0.0037	-0.0006	0.0196
8	MiniMax-M2	0.5860	0.0799	-0.4363	0.0143	-0.2846	-0.3207	-0.2503	0.0883	0.0294	-0.0894	0.1052	0.0234	-0.5651	0.1595	0.1304	0.0355	-0.2779	0.1584	-0.1204	-0.0715	0.4824	0.2272	0.1584	0.1541
9	Gemini-2.5-Pro	0.3541	-0.1512	0.0824	-0.0155	-0.0445	0.4479	-0.0888	0.5616	-0.1118	-0.2170	-0.1729	0.0612	-0.2556	0.1294	-0.0244	-0.1978	0.0667	0.1441	-0.0633	-0.1404	-0.2124	0.0645	0.1077	0.0302
10	GPT-4o-2024-11-20	-0.6459	0.0849	0.3867	-0.3998	0.1112	-0.2261	0.2475	-0.4843	0.1350	-0.6292	-0.3277	0.1031	-0.2531	0.1347	-0.5137	-0.2183	0.2383	-0.725	0.1246	-0.0470	-0.9680	-0.7851	-0.7003	-0.5588
11	GPT-5-Codex-High	0.6954	-0.6779	-0.1503	0.1408	-0.0671	1.0080	0.0008	0.1264	-0.2603	0.2346	-0.0971	-0.0988	-0.2587	0.1573	-0.1401	-0.1876	0.2241	0.0950	0.1973	-0.1243	-1.3085	0.1495	0.1899	0.1874
12	GPT-5-High	0.9259	0.1093	-0.6730	0.0981	-0.2977	0.8959	-0.2097	-0.2034	-0.2432	0.0554	0.0790	-0.0662	-0.5889	0.1692	0.0545	0.1114	-0.4192	0.2234	-0.2856	-0.2415	0.3605	0.3915	0.2497	0.2745
13	Manus	0.2278	-0.0221	-0.1865	-0.2492	0.2409	-0.4364	0.2511	-0.1441	0.0068	0.0733	-0.0144	-0.0222	-0.0770	0.0752	0.2568	-0.0490	-0.0310	-0.4325	0.0352	-0.2326	0.4846	0.2342	0.0348	0.0985
14	MiniMax Agent	-0.0796	0.1917	-0.2998	-0.1372	0.2409	-0.5171	-0.4619	-0.4263	0.0195	-0.0248	-0.0166	-0.1069	1.3456	0.2398	0.6680	-0.1217	-0.0239	-0.6377	0.0761	-0.4752	0.5712	0.1215	0.0812	0.0020

C Analysis of Unscorable Cases

In this section, we analyze the cases in which each evaluation metric fails to produce a valid score. When computing averaged scores and overall scores, we exclude unscorable cases rather than assigning them a score of zero, since extreme values would substantially distort the z-score distribution. Given that the number of unscorable cases for each model does not differ greatly across metrics, we believe that computing the average z-scores over only scorable cases is reasonable.

The number of unscorable cases for each evaluation metric and each model is shown in Table 11. We conduct a manual analysis, and classify the unscorable cases into four categories.

First, no scorable content is found in the artifact (i.e., the denominator of the metric is zero). This category takes a major proportion among unscorable cases. Metrics such as Media Quality and Icon Style Consistency are computed using invalid media or inconsistent icons divided by the total number of media items or icons. However, many artifacts contain no media or icons at all, resulting in a zero denominator.

Second, external tool execution faces failures. Metrics such as Best Practices and Performance rely on external tools (e.g., Lighthouse) for scoring. For some artifacts, Lighthouse fails to produce a score due to issues such as timeouts and rendering errors.

Third, the LLM-as-a-judge paradigm is disturbed by artifact content. In some cases, the LLM repeatedly generates the same character (e.g., ">") within the artifact, even repeating for hundreds of thousands of tokens. When scoring such artifacts, the large number of meaningless tokens prevents the LLM judge from reliably following the scoring prompt, resulting in failure.

Fourth, business logic causes the page to stay in a loading state or to exit immediately. In some cases, the artifact's business logic keeps the page always loading, or terminates when the required local data files are missing. When these artifacts are evaluated using browser-based automation, the process times out or fails to render, making them unscorable.

Table 11: Number of unscorable cases of each evaluation metric for each model.

ID	Model	Code Quality					Visual Quality					Content Quality			Performance Quality	Accessbility			Maintainability	Functional Alignment	Visual Alignment	Content Alignment
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	DeepSeek-R1-0528	0	15	99	8	0	3	1160	1455	646	3	3	2	1543	1406	1268	15	15	4	4	11	0	0	485	91
2	DeepSeek-V3.1	0	22	106	7	0	0	1307	1541	670	0	0	3	1506	1313	1204	22	22	2	4	20	0	0	485	91
3	DeepSeek-V3.1-Terminus	1	38	28	27	0	0	1263	1542	621	0	0	0	1549	1444	1364	38	38	27	27	11	0	0	485	91
4	DeepSeek-V3.1-Thinking	0	18	101	1	0	0	1309	1535	665	0	0	2	1501	1307	1196	18	18	1	1	17	0	0	485	91
5	GLM-4.5	0	37	32	28	0	0	1211	1506	505	0	0	0	1425	1398	1272	37	37	27	27	10	0	0	485	91
6	Qwen3-Coder-Plus	0	13	122	3	0	0	1302	1558	665	0	0	0	1552	1441	1408	13	13	3	3	10	0	0	485	91
7	Qwen3-235B-A22B-Instruct	7	45	161	33	0	0	1424	1558	874	0	0	0	1483	1320	1175	45	45	24	24	21	0	1	486	97
8	MiniMax-M2	5	18	61	3	0	0	1335	1441	735	0	0	2	1533	1429	1287	18	18	2	2	16	0	2	487	94
9	Gemini-2.5-Pro	2	54	175	21	0	0	1387	1460	865	0	0	1	1518	1345	1299	54	54	21	21	33	0	3	487	92
10	GPT-4o-2024-11-20	1	77	442	31	0	0	1516	1568	1236	0	0	0	1504	1368	1232	77	77	31	31	46	0	0	485	91
11	GPT-5-Codex-High	0	27	195	12	0	0	1130	1503	588	0	0	1	1533	1389	1235	27	27	12	12	15	0	0	485	91
12	GPT-5-High	0	17	32	9	0	1	1291	1377	556	0	0	1	1529	1411	1382	17	17	7	7	10	0	0	485	91
13	Manus	1	10	2	3	0	0	120	144	60	0	0	1	145	137	119	10	10	0	0	10	0	1	44	7
14	MiniMax Agent	1	0	1	1	0	0	112	134	46	0	0	0	140	130	95	0	0	0	0	0	0	1	44	7

D Questionnaire for Collecting Human Preferences

The questionnaire that we use for collecting human-preference-aligned weights is presented as follows. We randomly reorder the options in each question to avoid bias.

Questionnaire Contents:

We are studying the effectiveness of LLMs in the web application generation task. In this task, users provide the model with web-development requirements (for example: "Help me build an XXX web application"), and the model generates the corresponding code of the web application.

Please take the perspective of a real end user and, based on your personal preferences, rank the importance of the evaluation dimensions listed below. The ranking results will help us better understand users' priorities in actual use.

1. When using a model to generate web applications, which perspectives do you care about the most? Please rank the following dimensions according to your level of importance.
- Visual Quality: whether the webpage appears aesthetically pleasing and professional, and whether it adopts appropriate color combinations.
- Code Quality: whether the webpage code is correctly implemented, free of bugs or errors, and compliant with engineering best practices.
- Content Quality: whether the copywriting is clear to understand, the images and videos are high-resolution, and the information is rich.
- Performance Quality: whether the webpage loads smoothly and runs efficiently in practice.
- Accessibility: whether the webpage benefits disabled people and displays correctly across different devices and browsers.
- Maintainability: whether the webpage code is easy to read and modify.
- Functional Alignment: whether the webpage functions and business logic are implemented according to your specifications.
- Visual Alignment: whether the layout, colors, and appearance are implemented according to your specifications.
- Content Alignment: whether the text, images, videos, and other resources in the webpage follow your specifications.

2. When using a model to generate web applications, which aspects of visual aesthetics do you care about the most? Please rank the following dimensions by importance.
- General Visual Experience: assessing the overall visual experience when using the webpage, emphasizing aesthetics, design quality, and consistency between style and content.
- Component Style Consistency: assessing the usage of paratactic components and whether their internal elements (titles, icons, text, etc.) are stylistically consistent.
- Icon Style Consistency: evaluating whether the icons used follow a unified style and whether inconsistencies exist in size, line weight, or background shape.
- Layout Consistency: assessing how orderly the layout is, and whether components in rows or columns are properly aligned.
- Layout Sparsity: evaluating the structural rationality of the layout and checking for excessively long or wide empty areas.
- Visual Harmony Degree: evaluating the harmony of the webpage's color scheme, including diversity, balance, and aesthetic coherence.

3. When using a model to generate web applications, which aspects of code quality do you care about the most? Please rank the following dimensions by importance.
- General Functionality Correctness: evaluating whether the implemented functions and business logic are correct.
- Best Practices: evaluating adherence to web development best practices, including avoiding vulnerable JavaScript libraries, deprecated APIs, improper document declarations, or unreasonable permission requests.
- Error Handling: evaluating the code's ability to handle exceptional inputs (null values, errors, etc.).
- Runtime Console Errors: assessing dynamic correctness by rendering the code in a browser and checking console logs for severe errors.
- Static Syntax Checking: evaluating whether the code's syntax and style are correct.

4. When using a model to generate web applications, which aspects of content and media resources do you care about the most? Please rank the following dimensions by importance.
- Copywriting Quality: assessing the overall quality of the text accuracy, clarity, brevity, contextual consistency, user-friendliness, and compliance with UX and industry standards.
- Media Quality: assessing the quality of media such as images and videos, including clarity and playability.
- Placeholder Quality: evaluating whether placeholder images are used appropriately, without misuse, distortion, excessive repetition, or improper placement.
- Resource Validity: assessing whether embedded resources (images, videos, audio, stylesheets) are accessible, previewable, and successfully loaded (e.g., no broken links or 404 errors).

5. When using a model to generate web applications, which aspects of accessibility do you care about the most? Please rank the following dimensions by importance.
- Accessibility Core Metrics: evaluating support for users with visual impairments and other groups, checking whether the readable description text is included, the keyboard navigation is supported, and the compliance with WCAG standards.
- Cross-Browser Compatibility: evaluating whether the webpage renders and functions correctly across different browsers.
- Mobile Device Compatibility: assessing layout adaptability in mobile viewports and checking for overflow or abnormal rendering on mobile devices.

6. When using a model to generate web applications, which aspects of code maintainability do you care about the most? Please rank the following dimensions by importance.
- Code Redundancy Rate: assessing how much unnecessary or unused code is present, leading to tedious code.
- Comment Rate: evaluating whether natural-language comments are sufficient to aid developers in understanding the code.

E Manual Labeling Guidelines

This section presents our guidelines for human annotators to merge multi-turn requirements, filter the dataset using majority voting, build ground-truth checklists, and assign labels to each requirement.

E.1 Merging Multi-Turn Requirements

Guidelines for merging multi-turn requirements:

Input: Multi-turn user requirements, the merged requirements produced by an LLM.
Output: The merged requirements that are checked and processed by human annotators.
Workflow:

- Only turn-level additions or deletions are allowed; modification-only turns or turns without substantive requirement semantics should be removed.
- In cases of contradictory requirements, the later turn takes precedence; earlier conflicting turns should be discarded.
- Model-generated merged requirements are displayed in the "summary" column. Human annotators must validate outputs and correct issues according to Table 12.

E.2 Dataset Filtering

Guidelines for dataset filtering:

Input: User requirements.
Output: A True / False label for each requirement, indicating whether the requirement is usable.
Workflow: Requirements that belong to the following cases should be marked as "not usable".

1. **Ambiguous or Logically Incoherent Requirements:** The requirement is unclear or logically inconsistent. Examples include:
   - The user only uploads an HTML file without specifying any requirements.
   - Requesting an "infinite block map" in HTML without further explanation.
2. **Missing Supplementary Data:** The user's requirement lacks essential supplementary materials, such as images or links required to fulfill the request. Example:
   - "Please generate a military training commemoration website with sections: 1. Title 2. Photo Wall (use only uploaded reference images, no external resources)." In this case, the required reference images are not provided.
3. **Non-native Web Scenarios or Non-Web Implementation Languages / Frameworks:** The requirement specifies a context that is not native to standard web apps, or requires implementation in languages / frameworks outside of native HTML / CSS / JS. Example:
   - Using ESP32-S3 with a ST7789V display and FT6236U touch panel to create an interactive demo.
4. **Difficult-to-understand Requirements:** The requirement cannot be quickly interpreted to identify the main functional requirements by an expert with front-end development experience. Examples include:
   - Not suitable: "Toilet Man" (ambiguous).
   - Suitable: "Implement a Tetris game" or "Generate a Bomberman-style mini-game."

E.3 Building Ground-Truth Checklists

Guidelines for building ground-truth checklists:

Input: User requirements, the functional, visual, and content ground-truth checklists generated by three distinct LLMs.
Output: The functional, visual, and content ground-truth checklists validated and modified by human annotators.
Workflow: You are required to validate and modify the ground-truth checklists generated by LLMs based on the user requirements. The possible operations include:

- Addition: Add requirements that are explicitly mentioned by the user, but missing in the LLM-generated checklists.
- Deletion: Remove requirements that are presented in the LLM-generated checklists but are not mentioned by the user and are unreasonable or should not appear in the final checklists.
- Retain: Retain requirements that at least two out of three models consider essential. Note: Different models can express the same requirement differently, and you should merge semantically equivalent items (e.g., "bomb timing and explosion" and "bomb detonates on timer").

The definitions and examples of each ground-truth checklist are shown in Table 13.

E.4 Assigning Labels

This section presents a detailed classification scheme and the underlying rationale for the requirements in the dataset.

E.4.1 Input Modality

The human annotators are asked to label each requirement according to Table 14.

E.4.2 Clarity of Requirement

The human annotators are asked to label each requirement according to Table 15.

E.4.3 Style of Expression

The human annotators are asked to label each requirement according to Table 16.

E.4.4 Artifact Complexity

The human annotators are asked to label each requirement according to Table 17.

Table 12: Possible issues of requirements merged by the LLM.

Issue	Description
Loss of Key Information	Relevant content present in the original context is lost due to compression.
Redundant Content	After compression, repeated content still appears, or multi-turn requirements are merely concatenated without proper merging.
Hallucinated Content	The model invents or incorrectly recalls information not present in the original dialogue.
Content Rewriting / Paraphrasing	User requirements are not fully preserved; the model rephrases or simplifies user expressions.
Fail to Handle Conflicts	Contradictory instructions across turns are not properly resolved.

Table 13: Illustration of ground-truth checklists.

Dimension	Definition	Example
Visual ground-truth checklist	Visual design requirements mentioned by users	"I need a **red, industrial-style themed** webpage with a **blue** button below the form for submission."
Functional ground-truth checklist	Widget constraints mentioned by users	"I need **a table** displaying xxx information, with **a search box** above the table to input and filter results, and **a menu bar**."
	Functional points mentioned by users	"The website can **search for types of marine sharks**."
	Interaction actions mentioned by users	"I need the mouse to **hover over** the enter button for 3 seconds to enter the application; I need to **drag** the card to the schedule list to modify the schedule."
Content ground-truth checklist	Content to be displayed mentioned by users	"I need to display **today's news** on the webpage."

Table 14: Classification of input modalities in user requirements.

Input Modality	Definition	Examples
Text Only	The user specifies requirements exclusively via textual description, without visual or structural references. This mode relies entirely on natural language to convey intended functionality, style, or content.	"I want a black website header with three navigation links."
Text with Images	The user provides one or more images, design sketches, or screenshots, accompanied by a textual explanation specifying desired modifications or adaptations. The images can be used as either references or content. This approach leverages multimodal inputs to enhance specification clarity.	"[Upload one or more website screenshots] Replicate the layout and color scheme of this page, but replace the text content with mine."
Text with URLs	The user provides a specific web address (URL) as a reference point, requesting replication, adaptation, or stylistic emulation based on the referenced online resource.	"Study the page style of https://www.apple.com/mac/ and create a similar product introduction page for my offering."

Table 15: Classification of requirement clarity levels.

Level	Designation	Description	Required Capability	Examples
C1	**Clear**	The user provides exceptionally concrete and detailed requirements, effectively resembling a concise specification document. This may include explicit functional elements, user interface components, and even prescribed interaction sequences.	*Precise Execution*: The LLM is required to implement all specified details with exact fidelity, allowing minimal scope for autonomous interpretation.	"Create a contact form with three required fields: 'Name' (text input), 'Email' (email input), and 'Message' (text area). Below these fields, place a 'Submit' button. Upon successful submission, clear the form and display: 'Thank you for your message!'."
C2	**Intermediate**	The user articulates a clear objective or core functionality while omitting most implementation details. The emphasis is on the desired outcome (*what* is needed) rather than the process (*how* to achieve it).	*Interpretation and Completion*: The LLM must infer the essential goal and proactively complement missing specifications based on industry best practices or commonly observed design patterns (e.g., UI layout, interaction flows, error handling).	"Develop a to-do list application." "I would like a weather forecast app."
C3	**Vague**	The user presents only a loosely defined concept, intuition, or open-ended query, without a specific functional target. The requirement is exploratory in nature and encourages divergent thinking.	*Creative Generation*: The LLM must engage in extensive association, reasoning, and ideation, and may proactively propose potential directions or features. In such cases, the notion of "correctness" is inherently non-deterministic.	"Create a tool to improve my work efficiency." "Design an engaging homepage for my personal blog." "Suggest ways to make my photos look more stylish."

Table 16: Classification of expression styles.

Level	Designation	Description	Required Capability	Examples
S1	**Technical**	The user issues requirements in a precise, objective, and often technical manner, akin to a developer or product manager providing implementation directives.	*Technical Terminology Comprehension*: The LLM must be capable of interpreting domain-specific jargon (e.g., "SPA", "API", "responsive layout", "hook") and accurately mapping such terminology to concrete implementation strategies.	"Implement a SPA, containing a 'Header' component and a reusable 'Button' component." "Generate a RESTful API backend framework for the 'user' entity, including CRUD endpoints."
S2	**Colloquial**	The user communicates in everyday, informal language, similar to conversing with a friend.	*Natural Language Understanding (NLU)*: The LLM must possess strong NLU capabilities to accurately extract core requirements and key entities from casual, idiomatic descriptions.	"Hey, could you make me a small website to showcase photos of my cat so friends can view them?" "I just want a simple expense tracker to record daily spending, with a monthly total view."
S3	**Role-playing**	The user defines a context or adopts a role, thereby embedding the request within a rich narrative framework. This approach provides extensive situational information.	*Contextual Comprehension and Empathy*: The LLM must adopt the specified role, understand the authentic pain points and latent needs apparent in the given scenario, and produce outputs aligned with the contextual demands.	"As a fitness coach, I need an application to manage my clients' profiles and their training schedules." "Assume I am organizing a conference; I require a simple check-in page."
S4	**Analogy**	The user describes requirements by drawing analogies to familiar applications or concepts.	*Knowledge Transfer and Abstraction*: The LLM must identify the core functionalities and interaction patterns of the referenced analogy, abstract them, and adapt these elements to a new application domain.	"Create a kanban board similar to Trello, but simpler." "I would like a photo filter feature similar to Instagram." "Develop a voting tool akin to WeChat group voting."

Table 17: Classification of complexity levels of expected artifacts.

Level	Designation	Description	Function	Business Logic	UI/UX	Examples
L1	**Highly Simple**	A single, isolated functionality without data persistence, typically serving as a tool or static display.	Stateless single-function implementation, such as calculation, conversion, or plain text rendering. No data storage or backend involvement.	Direct linear logic: input → process → output, without conditional branching, multi-user roles, or state changes.	Minimalist interface: single page containing only essential I/O components (e.g., text field, button, label) and no navigation.	"Generate a Celsius-to-Fahrenheit temperature converter." "Create a page displaying 'Hello, World!'."
L2	**Simple**	A basic CRUD application centered on a single core entity.	Single-entity CRUD: manages one main object (e.g., to-do item, note) with fundamental data persistence (local storage or simple database).	Simple state management: create, read, update, and delete operations for a single entity without complex relations or access control mechanisms.	Single-page dynamic application (SPA): operations are handled entirely within one page by component showing/hiding/updating, including lists and simple forms.	"Build a to-do list application that allows adding, deleting, and marking tasks as completed." "Create a simple note-taking application with list and view functionality."
L3	**Medium**	Involves multiple interrelated functional modules, or includes simple workflows and role distinctions.	Multi-module / multi-entity relationships: at least two associated entities (e.g., users and articles, products and categories). May involve basic third-party API calls (e.g., weather data).	Conditional and role-based logic: distinguishes between simple user roles (e.g., administrator vs. standard user). Supports basic workflows (e.g., article publication requiring review), with well-defined data validation rules.	Multi-page / multi-view navigation: incorporates multiple pages or views (e.g., homepage, detail page, admin panel) with structured routing and navigation (menus, tabs).	"Develop a simple blogging system with user registration/login, allowing users to publish articles and administrators to review them." "Create a book management system that enables adding book information (title, author) and viewing by author categories."
L4	**Complex**	Comprises complex business processes, multi-system integration, and rich interactive interfaces.	Integration with multiple systems/services closely tied to core business (e.g., payment gateways, mapping services, SMS verification). Requires handling of asynchronous operations and data aggregation.	Complex multi-step workflows: business logic involves sequential processes and state changes (e.g., e-commerce order: cart → address → payment → confirmation). Includes advanced permission handling and validation rules.	Dynamic, feature-rich interfaces: advanced forms, data filtering, sorting, visualizations. Real-time or near-real-time updates based on user actions or backend data. Requires responsive design.	"Develop an online food ordering application where users browse menus, add items to a cart, pay via Alipay, and view order status." "Build a project task board enabling creation of task cards and drag-and-drop movement between 'To Do', 'In Progress', and 'Done' lists."
L5	**Highly Complex**	Enterprise-level or platform-scale application requiring high concurrency, real-time collaboration, and advanced algorithmic or data processing capabilities.	Platform/system-level functionality: scalable architecture supporting multi-tenancy, real-time communication (WebSocket), or complex background operations (e.g., data analytics, ML model invocation).	Highly complex business rules and finite-state machines: fine-grained access control, risk management, financial computation, or multi-party synchronization logic.	Highly dynamic and collaborative UI/UX: supports real-time multi-user collaborative actions (e.g., collaborative document editing, design tools). Includes advanced data visualization and customizable layouts, requiring high performance and UX quality.	"Build a Trello-like team collaboration platform with boards, lists, and cards, supporting drag-and-drop and real-time synchronization of team member actions." "Create a basic online code editor with syntax highlighting and real-time collaborative editing among multiple users in the same session."

F Prompts for LLMs

This section presents the prompts that we use to leverage LLMs in our dataset construction process, data analysis, generating artifacts, and conducting evaluations.

F.1 Prompts for Merging Multi-Turn Requirements

During the dataset construction process, we first merge multi-turn requirements to single-turn ones using the LLM, and then let human annotators conduct validation and modification. The prompts that we use are as follows.

Prompts for merging multi-turn requirements:

You are a Requirement Analyst. Your task is to process a multi-turn conversation record regarding "application generation" requirements, and merge the contents of multiple turns.
Task Objective:
1. Determine — except for the first turn — whether each subsequent turn is: Functionality Addition (e.g., "Add XX functionality"), Functionality Fix (e.g., "Fix XX issue"), Non-functional Description (e.g., "Confirm requirement", "Start generation", "Continue"). Keep only the turns of the Functionality Addition type, and exclude all others.
2. Merge the first turn with the subsequent turns that are Functionality Addition, keeping the original description intact as much as possible, including any typos, without altering the original content — only performing a simple merge.
3. Between merged sentences, you may add or slightly modify a few words or sentences to make the text coherent and free of obvious merge traces.
4. If the original content contains JSON, only modify the value of the "text" field.
Only output the merged content. Do not provide any explanations or additional text.
Now, please merge based on the following multi-turn conversation content:
{User Requirements}

F.2 Prompts for Generating Classification Labels

To analyze the statistics of our dataset, we first employ LLMs to generate detailed labels for each requirement, and then ask human annotators to check and revise these labels. The prompts that we use are shown in Figure 6.

F.3 Prompts for Generating Ground-Truth Checklists

In order to check the alignment of the generated web apps with the corresponding requirements, we let three LLMs generate ground-truth checklists separately, and then ask human annotators to decide the final checklists for each requirement in our dataset. The prompts that we use are shown in Figure 7 and Figure 8.

F.4 Prompts for Generating Artifacts

Prompts for generating artifacts using LLMs:

You are a professional web front-end application engineer and designer. You will receive user requirements for front-end web pages and write web page code to fulfill those requirements.
Note:
1. Your output should only include the code itself, with no additional explanations.
2. You may only use native front-end languages (HTML, JS, CSS) to build the page.

Prompts for generating artifacts using LLM-based agents:

You are a professional web front-end application engineer and designer. You will receive user requirements for front-end web pages and write web page code to fulfill those requirements.
##Delivery Requirements
1. You must implement the requirements using only native front-end languages (HTML, JS, CSS).
2. If the implementation can be done in a single file, then you may deliver only one HTML file.
3. If the implementation requires multiple files, then you may deliver only three types of files: HTML files, CSS files, and JS files.
4. The use of frameworks such as React is strictly prohibited.
5. Only front-end functionality needs to be implemented; no database or backend connections are required. If backend-related functionality is involved, use mock data to simulate it.

F.5 Prompts for Evaluation Metrics

Among all of our 24 evaluation metrics, 5 of them follow the LLM-as-a-judge paradigm. We show their prompts in Figures 9 and 10 (General Functionality Correctness), Figures 11 and 12 (General Visual Experience), Figure 13 (Functional Alignment), Figure 14 (Visual Alignment), and Figure 15 (Content Alignment), respectively.

Prompts for generating classification labels:

Role

You are an experienced AI evaluation specialist, possessing an integrated perspective that combines the expertise of a software architect, senior product manager, and user researcher. You excel at accurately and objectively analyzing user requirements, and classifying them according to a rigorous set of standardized rules.

Task

Your task is to examine the content of [USER_PROMPT_TO_ANALYZE] and evaluate it strictly according to the three dimensions defined in the "Dimensions & Rubrics" section: Artifact Complexity, Prompt, and Artifact Type. For each dimension, you must provide a concise rationale for the assigned label and produce a complete analysis output in the specified JSON format.

Dimensions & Rubrics

Dimension 1: Artifact Complexity

Rules:

1. Assess the following three sub-dimensions separately: Functional Complexity, Business Logic, User Interaction. Assign a level from L1 to L5 for each.

2. Determine the final overall complexity level according to the "Highest Level Principle" — the sub-dimension with the highest level determines the final rating.

Level	Functional Complexity	Business Logic	User Interaction
L1	Single, stateless functionality; no data storage.	Linear direct logic (input → output).	Minimal interface (single page; basic components).
L2	CRUD operations on a single entity; basic data persistence.	Simple state management for a single entity.	Single-page dynamic app (lists, forms).
L3	Multiple modules/entities; simple third-party API integration.	Conditional and role-based logic (e.g., admin vs user).	Multi-page/view navigation with routing.
L4	Deep integration with multiple systems/services (e.g., payments, maps).	Complex multi-step workflows (e.g., e-commerce ordering).	Rich dynamic interactions (filtering, sorting, charts).
L5	Platform-level functionality (e.g., real-time communication, multi-tenancy).	Highly complex business rules and state machines (e.g., risk control).	Highly dynamic and collaborative UI (e.g., co-editing).

Dimension 2: Prompt Style

Rules:

1. Clarity: Choose from C1, C2, C3.

2. Expression Style: Choose from S1, S2, S3, S4.

3. The final result must include both labels.

2.1 Clarity

C1: Clear & Specific — Explicit, detailed requirements akin to a small specifications document.

C2: Goal-Oriented — Defines clear objectives but omits implementation details.

C3: Vague & Exploratory — Expresses a broad idea or open-ended question only.

2.2 Expression Style

S1: Instructional/Technical — Precise, objective, and possibly technical language.

S2: Colloquial/Natural Language — Everyday, informal wording.

S3: Scenario/Role-playing — Describes requirements via set scenarios or role assumptions.

S4: Analogy/Heuristic — Expresses ideas through analogy with well-known applications or concepts.

Dimension 3: Application Type

Rules: Select the most appropriate type from the following list:

E-commerce, Online Education Platform, Healthcare Platform, Travel Services, Financial Services, News Media Platform, Entertainment/Gaming, Multimedia Platform, Corporate Website, Online Office Platform, Enterprise Back-office Management, AI Application, Smart Device Interaction, Social Media Platform, Forum Website, Personal Website, Public Service Platform, Utility Website, Data Visualization, Science Popularization Demonstration

Output Format

Your output must be a JSON object with the following structure. No explanations or text should appear outside the JSON block.

{{ "application_complexity": {{ "final_level": "L_x_", "final_justification": "Core rationale for determining final level.", "dimensional_analysis": [{{ "dimension": "Functional Complexity", "level": "L_x_", "justification": "Concise rationale for this dimension." }}, {{ "dimension": "Business Logic", "level": "L_x_", "justification": "Concise rationale for this dimension." }}, {{ "dimension": "User Interaction", "level": "L_x_", "justification": "Concise rationale for this dimension." }}] }}, "prompt_style": {{ "clarity": {{ "level": "C_x_", "justification": "Concise rationale for clarity rating." }}, "expression": {{ "level": "S_x_", "justification": "Concise rationale for expression style rating." }} }}, "application_type": {{ "type": "Selected type from list", "justification": "Concise rationale for type selection." }} }}

[USER_PROMPT_TO_ANALYZE]

—

{single-turn requirements}

—

Figure 6: Prompts for generating classification labels.

Prompts for generating ground-truth checklists:

[System Role] You are a senior requirements analyst and evaluation standards expert, responsible for understanding user needs. We currently have a real-world user-provided requirement for a **web frontend application**. This requirement may be vague or detailed.
In order to design and deliver a **fully functional, visually appealing, and content-rich product** that satisfies the user, we must break down the requirement into three structured dimensions: **Functional**, **Visual**, and **Content**.
You must generate **Ground-Truth requirement points** in these three dimensions that correspond to the user's stated needs. Structured dimensions are as follows:
- **Functional**
- **Visual**
- **Content**
Please strictly follow the rules below and **base your analysis solely on the user requirement text**. Do not make subjective assumptions or expand beyond what is explicitly stated.
—
[Task Objective] Output the following types of **GroundTruth requirement points**:
1. **Functional**
The operations, workflows, or system functions mentioned or implied in the requirement (*"What should the system do and how should it interact with the user?"*).
2. **Visual**
Experience-related requirements concerning theme colors, responsive layout, animation effects (*"What should the system look like, and what mandatory components must be included?"*).
3. **Content**
Page language type, videos, images, music, text copy, data sources, and other materials related to display or experience (*"What content should the system present?"*).
—
[Decomposition Rules]
Functional Dimension (Functional)
Goal: Extract the **Minimal Functional Set (MFS)** required to fulfill the user's need.
Criteria:
1. **Explicit mention first**: If requirement includes operational verbs (e.g., *upload, play, share*), directly split into requirement points.
2. **Implicit completion**: If requirement is an abstract objective (*e.g., "create a file sharing platform"*), extract the minimal functional set to achieve it:
- Upload files
- Generate sharing link
- Access link to download
3. **No divergence**: Do not infer features not mentioned (*e.g., "points system", "admin dashboard"*).
4. **Consistent granularity**: Requirement point should be independently developable and testable (includes input, processing, output).
5. Do not include programming language or framework requirements (*e.g., "must use Vue framework"*).
Example:
> "The system should be able to display product videos online." → [Upload video], [Play video]
> "Users can upload and share files." → [Upload file], [Generate sharing link], [Download via link]
—
Visual Dimension (Visual)
Goal: Identify specific user demands for visual experience.
Criteria:
1. **Explicit mention first**: If requirement mentions colors or theme description (*e.g., "blue and white", "industrial style"*), extract as requirement point.
2. Focus on **theme colors, responsive layout, animation effects**.
3. Do not mention basic UI elements (buttons, input boxes, tables, etc.) unless explicitly stated.
4. If mentions *style*, *brand colors*, *animation effects*, *adaptation for mobile/PC*, it is considered a visual element.
Example:
> "Overall theme should be blue and white" → [Theme color: blue & white]
> "Interface must adapt for both mobile and desktop" → [Responsive layout]
> "Page transitions must have fade-in/out effects" → [Animation: fade-in/out]
—

Figure 7: Prompts for generating ground-truth checklists.

Prompts for generating ground-truth checklists cont.:

Content Dimension (Content)
Goal: Extract requirement points about content to be displayed.
Criteria:
1. **Explicit mention first**: If requirement lists specific media or content (*e.g., “today’s news”, “images”*), directly extract. If content (images, text, etc.) is provided by user, emphasize “provided by user”.
2. Extract page language type, images, videos, audio, music, text copy, data sources, etc.
3. Exclude logical text (*e.g., prompts, error messages, guiding instructions*).
Example:
> “Display company promotional video and background music.” → [Video: Company promo], [Music: Background track]
—
[Output Format]
The output must be in JSON format (not Markdown JSON) with the following structure:

```
{
    "functionals": [
        {
        "type": "functional",
        "name": "Functional requirement point name",
        "description": "Brief description of the function and its application scenario"
        }
    ],
    "visuals": [
        {
        "type": "visual",
        "name": "Visual requirement point name",
        "description": "Describe the purpose and presentation of the visual element (only color/responsive/animation)"
        }
    ],
    "contents": [
        {
        "type": "content",
        "name": "Content requirement point name",
        "description": "Describe the purpose and details of the content (video/image/music/copy/data source, etc.)"
        }
    ],
    "summary": {
        "functional": [
        "Function A",
        "Function B"
        ],
        "visual": [],
        "content": []
    }
}
```

Figure 8: Prompts for generating ground-truth checklists continue.

Prompts for the evaluation metric of General Functionality Correctness:

[Role]
You are a senior front-end architect and testing lead, proficient in code review, white-box testing, front-end security, performance optimization, accessibility, and industry business standards.
[Current Time]
The current system time is {date}.
[Objective]
Your task is to evaluate the quality of the web page code and assign a score from 0 to 10 for each of the following 10 criteria to reflect its quality.
A score of 10 indicates the code is perfect, with no issues found during the code review.
A score of 0 indicates the code has severe syntax errors or major vulnerabilities, preventing it from running correctly.
A score from 1 to 9 indicates the code runs correctly, with higher scores representing better performance on the respective criterion.
Please output a comma-separated list of 10 numbers enclosed in square brackets, for example: [9,8,6,4,2,0,0,0,0,0].
Each scoring point is independent; please consider and score each one separately.
[Ten Evaluation Criteria and Scoring Guidance Examples]
1. **Functional Completeness & Business Logic**: Based on business requirements, ensure all functions are implemented without omission, the logic aligns with business specifications, and check that static data conforms to scientific and business common sense.
Scoring Guide:
    10 points: All business functions are fully implemented, logic aligns with business requirements, and static data conforms to scientific common sense.
    7-9 points: Most functions are implemented, with minor flaws in the handling of a few features.
    4-6 points: Some business functions are incomplete, or there are errors in logic or issues with static data.
    0-3 points: Severe omissions in business functions, or logic is incorrect or does not meet business requirements.
2. **Output Validation**: Following the code execution flow, evaluate if the output is correct. This includes checking value outputs and system calls, verifying for logical errors, missing, or duplicate output content. It's especially important to verify that UI updates and state changes reflect business logic changes.
Scoring Guide:
    10 points: All outputs are as expected, data is correct, and UI and state updates are timely and complete.
    7-9 points: Most outputs are consistent with expectations, with minor inconsistencies in a few edge cases.
    4-6 points: There are inaccurate outputs or updates that do not occur as expected, potentially affecting the user experience.
    0-3 points: Outputs do not match expectations, system calls are not executed as required, affecting normal functionality.
3. **Forms & Critical Path Flows**: Includes pre-validation, disabled states, protection against duplicate submissions, success/failure notifications, and redirects. Ensures important flows like payments and bookings are idempotent, have state rollback mechanisms, and provide clear error messages, covering industry constraints (e.g., time windows, quantity limits).
Scoring Guide:
    10 points: Form validation, disabled states, duplicate submission protection, and success/failure notifications are all complete. Critical paths like payments and bookings have robust idempotency and exception handling.
    7-9 points: Most form and critical path flows are handled well, but some minor details are imperfect.
    4-6 points: There are obvious flaws in form and critical path flows, leading to potential duplicate submissions or state management issues.
    0-3 points: Form and critical path flow handling is missing, severely impacting the normal progression of business processes.
4. **Data Science Logic Validation**: Verify that static data and business logic within the code are reasonable, ensuring data conforms to scientific principles and industry standards. Check the accuracy of data processing methods, avoiding hard-coded values or illogical data assumptions.
Scoring Guide:
    10 points: All static data is reasonable, and data processing methods align with industry standards and scientific common sense.
    7-9 points: Most data processing logic is reasonable, but some cases may have boundary issues or do not fully adhere to best practices.
    4-6 points: Data processing methods have errors or are unreasonable, potentially leading to business logic errors.
    0-3 points: Data processing methods are clearly unreasonable or conflict with industry common sense, affecting normal functionality.
5. **List/Card Display**: Check the state management and interactive behavior of list and card components, ensuring that empty data placeholders and loading skeletons are implemented correctly, and error states are handled effectively with user notifications.
Scoring Guide:
    10 points: List/card display is perfect. Empty data placeholders, loading skeletons, and error state retries all work correctly, providing an excellent user experience.
    7-9 points: Most display effects are good, but there are minor flaws in the display for certain states.
    4-6 points: Some display features are missing, or error states do not effectively notify the user.

Figure 9: Prompts for the evaluation metric of General Functionality Correctness.

Prompts for the evaluation metric of General Functionality Correctness cont.:

0-3 points: Display is severely inadequate. Empty data placeholders, loading skeletons, and error state retries do not work, severely impacting the user experience.
6. **Correctness & Boundary Conditions**: Covers all boundary conditions, null/type checks, ensures resources are released correctly, avoids concurrency/race condition issues, and ensures functionality remains reliable under various extreme circumstances.
Scoring Guide:
10 points: The function performs perfectly under all boundary conditions, correctly handles null values and type checks, gracefully releases resources, and avoids concurrency/race conditions.
7-9 points: Most boundary conditions are handled, but a few edge cases are not fully covered or have minor errors.
4-6 points: The function fails to work correctly in some extreme cases, there are issues with resource release, or there are race conditions or omissions in null checks.
0-3 points: Boundary conditions are not considered, there are multiple null or type errors, resources are not released properly, and race conditions are severe.
7. **Security**: Includes input validation, output encoding, prevention of injection attacks (XSS/CSRF), and dependency risk control to ensure the code is free from common security vulnerabilities.
Scoring Guide:
10 points: Input validation is complete, output encoding is strict, prevention against injection/XSS/CSRF vulnerabilities is comprehensive, and dependency risks are fully controlled.
7-9 points: Most security issues are addressed, but some input validation or dependencies have potential risks.
4-6 points: Some security checks are missing, leaving potential vulnerabilities that could be exploited.
0-3 points: Severe security vulnerabilities, such as XSS or CSRF attacks, are not prevented.
8. **Branch & State Coverage**: Ensures ‘if/else/switch/ternary‘ structures comprehensively cover critical paths and boundary cases, and handle early returns/exception branches; ensures proper management of variables, loading states, disabled states, error states, and empty states.
Scoring Guide:
10 points: All branch paths (including ‘if/else/switch‘, etc.) are covered, early returns and exception handling are complete, and all states (loading, disabled, etc.) are managed reasonably.
7-9 points: The vast majority of branches and states are covered, but a few paths or states are not fully handled.
4-6 points: Some branches or states are not covered, which may lead to logical errors or unhandled exceptions.
0-3 points: Critical branches are not covered, and state management is chaotic.
9. **Data Consistency & Flow Management**: Ensures DOM updates are consistent with the state, avoids race conditions or dirty data issues caused by global variables and closures, and reduces data flow conflicts.
Scoring Guide:
10 points: Data flow management is perfect, DOM updates are always consistent with the state, and there are no race conditions or dirty data issues from global variables/closures.
7-9 points: Data flow is largely consistent, but there are a few minor inconsistencies or race condition issues.
4-6 points: Data flow is poorly managed, with obvious race condition problems or dirty data risks.
0-3 points: Data flow is severely chaotic, DOM updates are inconsistent with the state, and there are numerous race conditions and dirty data issues.
10. **Asynchronous Operations & Error Handling**: ‘fetch/Promise/async‘ and other asynchronous operations have complete error handling, timeout control, and are designed with fallback mechanisms and user-friendly error messages.
Scoring Guide:
10 points: All asynchronous operations (‘fetch/Promise/async‘) handle exceptions and timeouts, and have complete fallback mechanisms and user-friendly error messages.
7-9 points: Most asynchronous operations are handled well, but some exception or timeout handling is incomplete.
4-6 points: Exception or timeout handling for asynchronous operations is inadequate, and error messages are unclear.
0-3 points: Asynchronous operations do not handle exceptions or timeouts and lack error messages.
[Review Rule Requirements]
A score is required for each item.
[Output Rules]
Note: The final output should only contain the JSON content format. Do not wrap it in a Markdown JSON block.

```
{{
    "score": [1,2,3,4,5,6,7,8,9,10],
    "summary": [
    {{
        "evaluation1":"Evaluation Content",
        "score": "0-10",
        "reason": "Brief Reason"
    }}
    ]
}}
```

user_requirements: {user_requirements}
Web Page Code: {html_content}

Figure 10: Prompts for the evaluation metric of General Functionality Correctness continue.

Prompts for the evaluation metric of General Visual Experience:

Role Setting
You are a senior product design reviewer with a keen aesthetic intuition and extensive user experience judgment. Please evaluate the first-screen interface of a front-end application from a real user's perspective, based on your subjective feelings.
Review Mindset
- Use first impressions as an important reference
- Trust your intuition and feelings
- View the application from the perspective of an ordinary user
- Don't get too caught up in technical details; focus on the "feel"
Subjective Evaluation Criteria
0-1 Points - Design Lacking or Extremely Chaotic
- Almost no design awareness; the page appears extremely chaotic or incomplete
- Visual elements are piled up without order, lacking basic layout logic
- Color scheme is jarring or extremely disharmonious, causing strong discomfort
- Information is completely inaccessible; user experience is extremely poor
- Gives the impression of "this is a work in progress" or "something went wrong"
1-2 Points - Basic Functionality Available, but Design is Rough
- Has basic information presentation capabilities, but severely lacks a sense of design
- Visual presentation is merely "viewable," lacking aesthetic appeal and refinement
- Uses standard templates or default styles with no signs of custom design
- Color scheme is mediocre or has obvious aesthetic issues (e.g., "tacky," "outdated")
- Layout is rigid, lacking visual hierarchy and breathing room
- Gives the feeling of "it's usable, but I don't want to use it"
2-2.5 Points - Design is Acceptable, Conventional
- Design meets basic standards; visual presentation is relatively clean
- Color scheme is safe but lacks highlights, falling into the "not bad, but not great" category
- Layout is reasonable and information hierarchy is mostly clear, but lacks memorable features
- Uses common design patterns, giving a "deja vu" feeling
- Overall look and feel is ordinary; no obvious flaws, but fails to spark interest
- At a "passing grade" level; users won't dislike it, but won't be impressed either
2.5-3 Points - Design is Good, with Clear Design Intent
- Has a clear design concept and visual style; overall cohesive and unified
- Color scheme is harmonious with a certain aesthetic pursuit, showing careful thought
- Layout is well-considered, information hierarchy is clear, and visual guidance is smooth
- Has highlights in certain details (e.g., animations, icons, typography)
- Style is relatively mature, matching the product's positioning and target users
- Overall quality is good, but innovation and distinctiveness are limited
- At a "good" level; users will find it professional and comfortable
3-4 Points - Design is Excellent, Trend-setting
- Design style breaks through traditional paradigms, being both distinctive and forward-thinking, capable of leading trends in similar web design
- Theme is highly original, potentially incorporating unique cultures, niche areas, or innovative concepts to avoid homogenization
- Design philosophy is distinct, conveying clear brand values or core content through visual language, allowing users to perceive the unique design intent
- Content is deeply integrated with the design theme, supporting the visual presentation while being amplified by it, creating a synergistic "content-design" effect
- Visual presentation is refined and captivating, achieving a high degree of balance between aesthetics and functionality
- Evokes strong emotional resonance and is memorable, making users feel "wowed" or even compelled to "want to share"
Subjective Evaluation Dimensions
First Impression (Very Important!)
- The moment you open it, what is your gut reaction?
- Does it make you want to explore further, or close it?
- What is the overall "vibe"? Professional? Rough? Interesting? Boring?
Visual Experience (Use Your Aesthetics)
- Is it good-looking? Does it have lasting appeal?
- Is the color scheme harmonious? Does it feel "tacky"?
- Are there any "wow" factors?
- Is the overall feeling refined or rough?
- Is there a sense of design and quality?

Figure 11: Prompts for the evaluation metric of General Visual Experience.

Prompts for the evaluation metric of General Visual Experience cont.:

Emotional Resonance
- Does this application have a "personality"?
- Does it feel warm? Cold? Professional? Friendly?
- Does it inspire a sense of trust?

Style Fit
- Does the visual style fit the product's industry? (e.g., finance should be stable, education friendly, e-commerce energetic)
- Does the design tone match the target user group? (e.g., trendy for young people, clear for the elderly, professional for business clients)
- Is there a sense of dissonance from a "style mismatch"? (e.g., using an overly playful design in a serious context)

Information Hierarchy
- Is the primary/secondary relationship on the page clear? Can you identify the main focus at a glance?
- Is the visual weight of titles, buttons, and supplementary information reasonable?
- Are important functions prominent enough? Is secondary information appropriately downplayed?

Design Consistency
- Is the visual expression of similar elements consistent? (e.g., button styles, icon styles)
- Is the color semantics consistent? (i.e., does the same color have a consistent meaning in different places)
- Are there any confusing design contradictions? (e.g., green indicating success in one place and in-progress in another)

Output Format
Strictly output in JSON format:

```
{
    "first_impression": "[Describe your feeling the moment you opened the app]",
    "overall_summary": "[Summarize your overall impression of this app in one sentence]",
    "visual_aesthetic_evaluation": "[Subjective feelings on color scheme, refinement, sense of design, etc.]",
    "style_fit": "[Whether the style fits the industry/user group, and if there's any dissonance]",
    "information_hierarchy": "[Evaluation of primary/secondary relationships and prominence of key elements]",
    "design_consistency": "[Evaluation of color semantics and element uniformity]",
    "if_your_friend_made_this": "[Provide feedback in more authentic and direct language]",
    "subjective_score": [0-4 points, up to two decimal places],
    "scoring_rationale": "[Explain the basis for your score, e.g., why it's X points and not X±0.3]"
}
```

Review Philosophy
1. Trust your first intuition
2. Don't try to rationalize why you like or dislike it
3. React authentically like a regular user
4. There are no right answers in aesthetic judgment; trust your own feelings
5. Provide both a sentimental evaluation and a rational analysis of key elements like style, hierarchy, and consistency
6. Remain objective and friendly; when pointing out issues, offer direction rather than criticism

Figure 12: Prompts for the evaluation metric of General Visual Experience continue.

Prompts for the evaluation metric of Functional Alignment:

Role-play: You are a senior requirements evaluator, project manager, and user of web front-end applications. You have just received a web front-end application developed based on the given requirements. It is still in the early stages of development, and your task is to determine if the developer has correctly understood the user's needs and to judge at a high level whether the given web page code meets the user-specified requirements, without worrying about the correctness of the specific implementation.
Task Goal: Analyze the web front-end application code and determine if it meets the given requirements.
Input:
Web page code and **requirements** information.
Evaluation Criteria:
Please analyze whether the code implements the specified requirements from the user's perspective, based on the following dimensions:
Functional Module: Does the code contain the functionality specified in the requirement description? (Even if it's just a function to be implemented?)
Interaction: If the functionality involves user interaction (e.g., clicking a form submission button, an input box), is there a corresponding, user-visible control in the code, along with listener support?
Output:
Please output your judgment and analysis in JSON format, with the following structure:

```
[{
    "functional_requirement": "<The requirement>",
    "code_snippet": "“‘<Web Front-end Application Code>“‘(The maximum output length is 1000 characters.)",
    "is_implemented": <true/false>,
    "implementation_analysis": "<A concise analysis of how the code implements the requirement>",
    "confidence_score": <A confidence score from 0 to 1, indicating your certainty in the judgment, where 1 is the highest>
}]
```

Example:
Input:
1. **Web page code:**

```
<button id="myButton">Click me</button>
<div id="message"></div>
<script>
    document.getElementById("myButton").addEventListener("click", function() {
    document.getElementById("message").textContent = "The button was clicked!";
    });
</script>
```

2. **Requirement(s):** [“After clicking a button, the message ‘The button was clicked!’ will be displayed on the page.”]
Expected Output (JSON format):
Note: The final output should only contain the JSON content format, do not wrap it in a Markdown JSON block.

```
[{
    "functional_requirement": "After clicking a button, the message 'The button was clicked!' is displayed on the page.",
    "code_snippet":
        "<button id="myButton">Click me</button>
        <div id="message"></div>
        <script>
            document.getElementById("myButton").addEventListener("click", function()
            document.getElementById("message").textContent = "The button was clicked!";
            );
        </script>",
    "is_implemented": true,
    "implementation_analysis": "The code includes a button and a div element for displaying messages. JavaScript uses an event listener to bind a click event to the button. When the event is triggered, it modifies the textContent of the #message element to 'The button was clicked!'. The functionality perfectly matches the description; clicking the button displays the specified message on the page in real-time.",
    "confidence_score": 1
}]
```

Figure 13: Prompts for the evaluation metric of Functional Alignment.

Prompts for the evaluation metric of Visual Alignment:

Role-play: You are a senior requirements reviewer, project manager, and a user of web front-end applications. You have just received a web front-end application developed according to given requirements. It is still in the early stages of development. Your task is to determine whether the developer has correctly understood the user requirements by evaluating at a high level whether the given web page code meets the specified requirement points, without needing to verify the correctness of the specific implementation.
Task Goal: Analyze the web front-end application code to determine if it meets the given requirement points.
Input:
Web page code and **requirement points** information
Evaluation Criteria:
Please analyze from the user's perspective whether the code implements the specified requirement points based on the following dimensions:
Visual Attributes: Does the code conform to the user-specified visual design?
Page Components: Does the code include the page components (must be user-visible) specified by the user?
Output:
Please output your evaluation results and analysis in JSON format, with the following structure:

```
[{
    "visual_requirement": "<Requirement Point>",
    "code_snippet": "<Web Front-end Application Code> (The maximum output length is 1000 characters.)",
    "is_implemented": <true/false>,
    "implementation_analysis": "<A concise analysis of how the code implements the requirement point>",
    "confidence_score": <A confidence score from 0-1, indicating your certainty in the judgment, where 1 is the highest>
}]
```

Example:
Input:
1. **Web page code:**

```
.primary-button {
background-color: #007bff;
color: white;
font-size: 16px;
padding: 10px 20px;
border-radius: 5px;
}
```

2. **Requirement Points:** ["A button with a blue background, white text, a font size of 16 pixels, and a 5-pixel border-radius."]
Expected Output (JSON format):
Note: The final output should only contain the JSON content, without being wrapped in Markdown's JSON format.

```
[{
    "visual_requirement": "A button with a blue background, white text, a font size of 16 pixels, and a 5-pixel border-radius.",
    "code_snippet":
        ".primary-button {
        background-color: #007bff;
        color: white;
        font-size: 16px;
        padding: 10px 20px;
        border-radius: 5px;
        }",
    "is_implemented": true,
    "implementation_analysis": "The '.primary-button' class in the code sets the background color to #007bff, which is a shade of blue, the text color to white, the font size to 16px, and the border-radius to 5px, fully matching the requirement's description. Additionally, the padding is set to 10px 20px, which is a common padding for buttons; although not mentioned in the requirement, it does not affect the consistency of the implementation.",
    "confidence_score": 1
}]
```

Figure 14: Prompts for the evaluation metric of Visual Alignment.

Prompts for the evaluation metric of Content Alignment:

Role-play: You are a senior requirements analyst, project manager, and a user of web front-end applications. You have just received a web front-end application developed based on given requirements. It is still in the early stages of development. Your task is to determine whether the developer has correctly understood the user's requirements and to judge at a high level whether the given web page code meets the user-specified requirement points, without worrying about the correctness of the specific implementation.
Task Goal: Analyze the web front-end application code and determine if it meets the given requirement points.
Input:
Web page code and **requirement points** information
Evaluation Criteria:
Please analyze from the user's perspective whether the code implements the specified requirement points based on the following dimensions:
Content Semantics: Does the content on the page conform to the semantics specified in the user requirements?
Specified Data: Does the content on the page include the specific data that the user specified to include? (e.g., image links, data source links, text copy, video/audio links, etc.)
Output:
Please output your judgment and analysis in JSON format, with the following structure:

```
[{
    "content_requirement": "<Requirement Point>",
    "code_snippet": "<Web Front-end Application Code> (The maximum output length is 1000 characters.)",
    "is_implemented": <true/false>,
    "implementation_analysis": "<A concise analysis of how the code implements the requirement point>",
    "confidence_score": <A confidence score from 0-1, indicating your level of certainty, with 1 being the highest>
}]
```

Example:
Input:
1. **Code Snippet:**

```
<div>Welcome</div>
<img src="logo.png">
<a href="/home"></a>
```

2. **Requirement Points:** ["Display a welcome text.", "Display a company logo, using logo.png.", "Provide a link to return to the homepage."]
Expected Output (JSON format): Note: The final output should only contain the JSON content, do not wrap it in Markdown's JSON format.

```
[{
    "content_requirement": "Display a welcome text.",
    "code_snippet": "<div>Welcome</div>",
    "is_implemented": true,
    "implementation_analysis": "The code uses a '<div>'tag to contain the text Ẅelcome, thus implementing the requirement to display a welcome text.",
    "confidence_score": 0.8
},
{
    "content_requirement": "Display a company logo.",
    "code_snippet": "<img src=l̈ogo.pngs̈>",
    "is_implemented": true,
    "implementation_analysis": "The code uses an '<img>' tag to attempt to display the user-specified image named l̈ogo.png.", 
    "confidence_score": 1.0
},
{
    "content_requirement": "Provide a link to return to the homepage.",
    "code_snippet": "<a href=/̈homes̈></a>",
    "is_implemented": false,
    "implementation_analysis": "The code uses an '<a>' tag to create a link pointing to /̈home, but there is no text on the link, which fails to indicate to the user that it is a link to return to the homepage.",
    "confidence_score": 0.9
}]
```

Figure 15: Prompts for the evaluation metric of Content Alignment.