

Automatic Generation Method and Implementation of Web Front-End Components Based on Generative AI

Shengping Liu

Pflugerville, Travis County, Texas 78660, United States

Keywords: Generative Artificial Intelligence; Web Front-End Components; Multimodal Requirements Analysis; Code Generation; Rendering Feedback; Automated Testing

Abstract: To solve the problems of ambiguous requirement expression, difficulty in balancing visual reproduction and interactive correctness, and lack of verifiable feedback in the generation of web front-end components by generative artificial intelligence, this paper proposes a multimodal requirement parsing and rendering feedback closed-loop method that integrates natural language, interface screenshots, and engineering context. First, map user intent, visual elements and component library constraints into structured requirement vectors; then, build a component hierarchy graph and generate code according to joint semantic, visual, functional and accessibility goals. Next, the generated components are run in an isolated browser, and then iteratively improved by altering the DOM structure, screenshot similarity, interaction tests, and engineering quality gatekeeping. The engineering realisation is a multi-tier architecture that includes a model orchestration layer, component planning layer, code synthesis layer, sandbox rendering layer and quality governance layer. According to the review statistics based on publicly available data from Design2Code and WebCode2M, while mainstream models have a good foundation for overall visual similarity, the gap between models widens significantly with an increase in page code length and structural complexity, and element recall, layout relationships, and interactive behaviour remain the main bottlenecks. Based on the research results, automated generation systems for real-world engineering projects should move beyond a one-time prompt generation model and transition to a closed-loop system of "structured planning - constrained synthesis - executable verification - difference-driven repair".

1. Introduction

Web front-end development is to provide an interactive user interface for the requirements. The programmes of the works include HTML, CSS and JavaScript, as well as visual and experiential constraints such as layout, colour, font, responsive behaviour and accessibility. The typical process has several collaborations among the product, design and development teams, and frequent adjustments are needed for the boundaries of components, state changes and exception paths across design drafts, requirements documents and code. Generative artificial intelligence can directly

create modules according to a natural-language description, screenshot or existing code to offer a new technological path for reducing the prototype cycle and alleviating redundant work in low-code development. However, the front-end components are not isolated text snippets; their correctness depends on browser rendering semantics, component library specifications, state management, event binding and cross-device adaptation. Therefore, "the ability to generate code" does not mean "the capacity to produce maintainable modules".

Recently, many large-scale multimodal models have improved their understanding of screenshots and code synthesis, but public benchmarks still show widespread problems such as missing elements, incorrect hierarchy, and layout deviations in complex pages. More importantly, many generation systems only focus on screenshot similarity in their evaluation, but real-world projects need clickable buttons, submitable forms, operable keyboards, installable dependencies, successful builds, and adherence to agreed-upon directory structures and design tokens. Without an execution environment and automated tests, the model may only achieve local visual approximation through absolute positioning, hard-coded dimensions, or redundant styles, and will be difficult to maintain.

Therefore, the research goal of this paper is not to design a single prompt template, but to establish an automated generation chain for engineering delivery: First, to unify the representation of textual requirements, screenshot information, design systems and code context; second, to decompose the entire page into reusable and composable component diagrams; third, to generate code based on semantics, visuals, functionality and accessibility as joint constraints; fourth, to automatically discover differences and trigger local fixes through browser rendering, structural comparison and interaction testing; and fifth, to provide scalable system implementation methods and quality gates. The three main contents of this paper are introduced below: the proposed multimodal structured requirement model, a component diagram-driven constrained generation strategy, and a rendering feedback loop. Verify the statistics of publicly available research data from the past three years and demonstrate that the proposed technical route is both necessary and feasible.

2. Current Status Analysis of the Research Topic

2.1 From Screenshots to Code: Benchmark and Data Evolution

Design2Code built 484 actual web page test cases and evaluated the screenshot-to-code generation effect based on element matching, text, position, colour and CLIP similarity. The main deficiencies of the current model have been identified as poor recall of visual elements and failure to recover layout relationships [1]. This work promoted evaluation based on browser rendering results rather than code string similarity, but the data mainly served as a test set and was too small to support fine-tuning across all fields.

WebSight uses language models to generate web page concepts and HTML, creating around 2 million screenshot-code pairs that provide large-scale synthetic data for training visual language models on the correspondence between interfaces and code [2]. Synthetic data has the advantage of a clean structure and low acquisition cost, but it is relatively lacking in page complexity, dependencies and real business noise, and the high scores achieved by the model in the synthetic domain cannot be directly translated into the usability of real projects.

WebCode2M built an additional 2.56 million web page designs, code and layout information from real network resources, and proposed TreeBLEU to measure DOM hierarchy structure recall [3]. Public statistics of this dataset show that after the page is expanded from short code to long code, the visual score and structure score of most general models decrease; that is to say, long context, deep DOM and diverse tags will amplify the generation error together.

DCGen recursively divides the entire page screenshot into semantic fragments by horizontal and vertical division, and then generates descriptions and code for the local fragments and assembles them layer by layer. It showed an improved visual similarity of as much as 14% for real-website samples [4]. This way is to reduce the scope of single visual reasoning, and although it will decrease element omissions and misalignments, if the segmentation boundary does not align with the semantics of the component, cross-regional states, shared styles and responsive relationships may still be lost.

Table 1: Comparison of representative datasets automatically generated by the front-end in the past three years with the benchmark.

Benchmark	Year	Scale	Data Type	Primary Evaluation
Design2Code	2025	484 webpages	Real-world	Element, layout, color, CLIP
WebSight	2024	2.0M pairs	Synthetic	Screenshot-to-HTML learning
WebCode2M	2025	2.56M instances	Real-world	Visual, CLIP, TreeBLEU
WebUIBench	2025	21,793 QA pairs	Real-world	Perception, HTML, cross-modal
FrontendBench	2025	148 prompt-test pairs	Curated tasks	Executable functional tests
FullFront	2025	3 workflow tasks	Standardized webpages	Design, perception, generation

Data source: Compiled from publicly available information in references [1]–[7]; all text in the table is in English.

Table 1 Analysis: Existing research has formed a hierarchical system of "real test set - large-scale training set - capability diagnostic benchmark - executable test". Design2Code focuses on rendering similarity, WebCode2M adds real large-scale training data, WebUIBench and FullFront extend to perception and end-to-end capabilities, and FrontendBench includes interaction testing in the evaluation. Therefore, it can be seen that the research direction of front-end automatic generation is moving away from static appearance reproduction towards executable, diagnosable and maintainable engineering quality.

2.2 Evaluation dimensions shift from static similarity to full-process capability

The four modules in WebUIBench are WebUI awareness, HTML programming, WebUI-HTML understanding and WebUI-to-Code. It includes 719 full web pages, 2488 slices and 21,793 question-and-answer sets, and tests 29 models [5]. Based on the above results, while the models perform relatively well in single-modal HTML knowledge, their cross-modal matching and visual localisation abilities are still weak; thus, the quality of front-end generation requires all these sub-capabilities to work together rather than relying on a single code language capability.

FrontendBench has 148 sets of prompt-test cases, five levels of tasks according to component complexity, and executes the generated code in a sandbox. The results are confirmed by the preset test scripts, and the consistency rate of automatic evaluation and expert judgment is 90.54% [6]. Therefore, the engineering system will have to carry out the steps of generation, construction, installation, event triggering and state assertion. Only "executable verification" can find interaction defects that screenshot indicators cannot show.

FullFront further expanded the evaluation to three stages: web design, page-aware question answering, and web code generation, and indicated that the current model still had a considerable gap with human experts in image processing, layout restoration and interaction implementation [7].

Therefore, the automated tools should offer support for requirement clarification, design planning, code implementation and iterative debugging, etc., rather than providing a single end-to-end generation solution for the entire development process.

2.3 Engineering Execution Environment and Software Quality Verification

WebArena uses functional correctness evaluation for extended Web tasks in a self-hosted website environment. Its best GPT-4 agent has an end-to-end success rate of only 14.41%, and human agents reach 78.24% [8]. Although this study focuses on Web agents, it also applies the same engineering idea: the usability of an interface system must be tested through real-operation paths, and local visual or single-step outputs cannot demonstrate the completion of the whole task.

VisualWebArena extends the evaluation to visually driven web page manipulation tasks and requires the model to understand images, text instructions and page states simultaneously to determine whether it has succeeded in execution [9]. Component generation refers to determining how many DOM nodes are generated during testing; click simulations, input operations, scrolling and pop-up switching, and responsive viewport adjustments should also be included.

SWE-bench evaluates a model's code modification ability based on real GitHub issues and patch tests, and requires understanding dependencies and passing test suites in a full codebase [10]. Based on this paradigm, the front-end automatic generation system should also run type checks, unit tests, end-to-end tests and build processes in the project context to avoid generating seemingly reasonable but unintegrable isolated code.

3. Raise questions

3.1 Multimodal requirements lack a unified and executable intermediate representation.

Most users state the purpose of their work in short sentences, show visual requirements through screenshots, and list the technology stack from a precompiled library. They will also offer a component library, design tokens, API documentation and coding standards. If the model outputs the above heterogeneous inputs directly, there will be problems such as inconsistent terminology, a mismatch between visual elements and data fields, and missing component attributes. Therefore, it is necessary to convert multimodal inputs into explicit sets of component types, hierarchies, style tokens, states, events, data bindings and constraints, and have subsequent generation and testing share the same requirements baseline[11-13].

3.2 There is a conflict between visual similarity, semantic correctness, and engineering maintainability.

Screenshot similarity prioritises pixel-level closeness, and engineered implementations favour semantic DOM, fluid layouts, reusable components and minimal styles. Models may use absolute positioning and many inline styles to achieve visual uniformity; at the same time, auxiliary text, error states and boundary interactions can be omitted to reduce code length. Without a common objective and a sense of urgency, the system will fail to achieve the ideal combination of "appearance", "functionality" and "maintainability"[14-15].

3.3 Long contexts and hierarchical dependencies in complex pages amplify errors

Real-world web pages have navigation, lists, forms, pop-ups, charts, and many other states. DOM depth, the number of tags, and style dependencies all increase with scale. Generating the

entire page's code at once compresses details in a small output window, and omissions in some places can change the following hierarchical structure. Simple image grid division reduces visual clutter but may conflict with the alignment and shared-state requirements of components. Therefore, the visual division, semantic component boundary and data dependency need to be planned concurrently.

3.4 Lack of execution feedback prevents error localization and closure.

Generating the result needs dependency installation, compilation, browser rendering, screenshotting, event execution and assertions. At present, many tools only return code text, and problems such as compilation errors, runtime exceptions, style overrides, unclickable elements, and keyboard inaccessibility must be identified manually. Even with "self-correction" hints, if the feedback is only a general statement such as "inconsistent effect", the model still cannot identify specific DOM nodes, CSS properties or event handlers[16-17].

4. Problem-solving strategies: Closed-loop generation method and engineering implementation

4.1 Overall Architecture

The seven links in the system introduced in this paper are: requirement access, structured parsing, component graph planning, constrained code synthesis, sandbox rendering, discrepancy diagnosis and quality governance. The system accepts text requirements, reference screenshots, design tokens, component library documents and target repository context, and outputs integrable component files, style files, test cases and generated reports. Each stage passes the information via a single component specification object, and thus any fixes can be made at the component or attribute level without needing to regenerate all pages in every iteration.

Table 2: System Functional Modules and Input/Output Design

Module	Core Input	Core Output	Engineering Responsibility
Requirement Parser	Text, screenshot, repository context	Structured UI specification	Resolve intent, states, events and constraints
Component Planner	UI specification, design system	Component graph	Define hierarchy, reuse and data dependencies
Code Synthesizer	Component graph, framework rules	Component and style files	Generate constrained, typed and reusable code
Sandbox Renderer	Generated project	DOM, screenshot, runtime logs	Build and execute in an isolated browser
Difference Diagnoser	Reference and rendered artifacts	Localized defect list	Map visual and functional gaps to nodes
Quality Gate	Code, tests, and metrics	Pass, repair or reject decision	Enforce accessibility, security and maintainability

Table 2 Analysis: The modular architecture contains the uncertain output of the large model within a deterministic engineering control chain. Requirements analysis and component planning reduce the ambiguity of the input; code synthesis only addresses confirmed local specifications;

rendering and diagnostic modules convert browser facts into node-level feedback; quality gates decide whether to proceed with fixing or reject delivery. This division of labour supports the replacement of the underlying model and helps to integrate existing component libraries, testing platforms and continuous integration processes in enterprise projects.

4.2 Multimodal Requirements Analysis and Specification Integration

First, extract the intent, entities, state and events from natural language in the system; perform element detection, OCR and colour and spatial relationship recognition on screenshots; and extract framework version, common components, type definitions and design tokens from repository context. The three kinds of features are not directly concatenated but aligned by component candidates; the "search box" in the text should use the same identifier as the input area in the screenshot and the SearchInput component in the code repository. The fused representation can be shown as:

$$z=W_th_t+W_vh_v+W_ch_c+b \quad (1)$$

As shown in Equation (1), h_t , h_v and h_c are the textual, visual and code context features, respectively; W_t , W_v and W_c are learnable or rule-calibrated mapping matrices; b is a bias term; and z is a unified requirement vector. In engineering implementation, z is serialised into a specification object containing fields such as component, props, state, event, token, constraint and evidence, and the input evidence corresponding to each conclusion is saved for later tracking and manual inspection.

4.3 Component Diagram Planning and Layered Generation

Based on a single specification, the system builds a directed component graph. Nodes are pages, containers or atomic components, and edges are containment, data flow, event propagation and style inheritance relationships. The planner first attempts to reuse components from the repository, then selects a standard component in the design system, and finally generates a new one. There is no fixed grid division of the complex screenshot; instead, candidate regions are generated by visual division lines, alignment relationships, repetition patterns, and business semantics. If the two regions share a state or event, they should remain in the same upper-level container of the graph to prevent local generation from affecting the whole-system logic.

The code synthesis stage has added multiple goals and will now consider requirement semantics, visual presentation, functional behaviour and accessibility. The corresponding purposes are as follows:

$$L_{gen}=\lambda_sL_{sem}+\lambda_vL_{vis}+\lambda_fL_{func}+\lambda_aL_{acc} \quad (2)$$

As shown in Equation (2), L_{sem} is the consistency between the component and the requirement specification; L_{vis} is the difference between the rendered result and the reference design; L_{func} comes from interaction testing and state assertion; L_{acc} indicates accessibility defects; and λ_s , λ_v , λ_f , and λ_a are weights set according to the risk of the project. Increase the weights of functionality and accessibility for the backend management system, and increase the weight of visuals appropriately for the marketing display page; however, no weight setting can disable the build and basic interaction access control.

4.4 Constrained Code Composition and Context Control

The compositor creates components in the order of "page skeleton—reusable components—local

styles—event logic—test cases", provides the model with only the specifications of the current node, parent and child interfaces, and necessary repository fragments each time to reduce irrelevant long contexts. The output needs to meet the syntax, type system and directory requirements for the target framework; unauthorized dependencies, remote scripts and dangerous dynamic execution are not allowed. Data-driven template generation is employed for duplicate cards, table columns and form items to avoid copy-paste code bloat, and whitelist constraints are used for media queries, theme variables and design tokens to reduce hardcoding.

4.5 Dual Feedback from DOM Structure and Rendering Visuals

The generated project installs dependencies and builds inside an isolated container, and then renders multiple viewports using a headless browser. Extract sets of DOM elements, hierarchical edges, text, bounding boxes and computed styles from the reference page and the generated page in the system. Structural Similarity is expressed in the F1 edge set format.

$$S_{\text{dom}} = \frac{2|E_g \cap E_r|}{|E_g| + |E_r|} \quad (3)$$

E_g and E_r in equation (3) are the sets of parent-child relationship edges of the generated DOM and the reference DOM, respectively. A higher S_{dom} indicates that the component hierarchy and nesting relationships are more regular. Rather than comparing HTML strings directly, this index can identify various issues such as elements in the wrong containers, a missing list hierarchy, and abnormal component boundaries after different equivalent implementations.

Visual feedback considers high-level semantics, local structure and layout relationships simultaneously, and the overall score is defined as follows:

$$S_{\text{vis}} = \alpha S_{\text{clip}} + \beta S_{\text{ssim}} + \gamma S_{\text{layout}}, \quad \alpha + \beta + \gamma = 1 \quad (4)$$

As shown in equation (4), S_{clip} is the overall visual semantic similarity; S_{ssim} focuses on pixel structure, and S_{layout} is derived from the alignment, spacing and size errors of element bounding boxes; the sum of α , β and γ equals 1. The diagnostic tool does not only return the total score, but also generates a list of "node-attribute-difference type-confidence". For example, it maps button offsets to the parent container's justify-content and maps missing text to the component's attribute, thereby triggering a local repair.

4.6 Interaction Testing, Accessibility, and Quality Access Control

Functional verification automatically generates test paths according to the states and events in the specification, such as initial rendering, clicks, input, validation, submission, loading, errors and empty states. Browser logs console exceptions, network failures and layout overflows; static verification conducts type checking, code style rules, dependency audits and accessibility checks. Focus checks for buttons, links, form controls and pop-ups include semantic tags, visible focus, keyboard operations, ARIA relationships and colour contrast. If the defect can be found and corrected at a low cost, apply a patch; otherwise, if the requirement evidence is missing or the repair will introduce new serious problems, manual verification is required.

Among the trade-offs of quality, latency and inference cost, a deployment utility function is selected.

$$J = w_q Q - w_l \frac{L}{L_{\text{max}}} - w_c \frac{C}{C_{\text{max}}} \quad (5)$$

Q is the overall quality of semantics, vision, functionality and accessibility in Equation (5); L

and C are the actual latency and inference cost, respectively; $L_{\max}$ and $C_{\max}$ are the upper limits of the project; and w_q , w_l , and w_c are the business weights. Therefore, the system can choose to generate a strong model in one go, perform local repair of a lightweight model, or carry out deterministic rewriting of the rule engine and avoid calling the high-cost model in all steps.

4.7 Engineering Implementation Plan

The prototype front-end is a React component model that builds a preview and manual confirmation interface, and the server-side uses a task orchestrator for specification parsing, model calls, code writing, and a test queue. The repository analyzer extracts component exports, attribute types, routes, style references and test entry points from an abstract syntax tree, and builds a vector index for reusable symbols. Generated files first enter a temporary branch, do not directly overwrite the main code, and each patch log records the version number, input evidence, model identifier, difference metrics, and test results for reproduction and accountability auditing.

The rendering layer is in a containerized build environment and uses a headless browser; network access, file-system permissions, CPU time and memory are restricted, and unauthorised operations by the generated code are prevented. Capture page snapshots at different sizes of the desktop, tablet and mobile viewports, and save DOM snapshots, accessibility trees, console logs, and network traces. The Quality Governance Layer establishes upper limits, for example, requires a successful build, a 100% pass rate for critical interactions, and a zero threshold for high-severity accessibility defects. Visual Scores have a lower bound based on the type of project.

A closed-loop repair mechanism uses a defect priority queue to address issues in this order: first, build and runtime errors; second, functional failures; third, missing elements and layout deviations; finally, details such as colours and fonts. Each time, only the minimum set of files related to the defect evidence is modified, and the relevant tests are rerun. Automatic iteration stops when two consecutive rounds of quality improvement fall below a threshold, avoiding invalid loops. This way, the large model serves as a constrained candidate solution generator, and the determination of deliverability is handled by browser facts, test scripts and engineering rules.

5. Public Benchmark Statistical Verification and Discussion

5.1 Data Sources and Statistical Scope

To ensure the accuracy and verifiability of the data in the figures and tables, no experimental values have been built or interpolated in this paper. Figure 1 and Table 3 are directly taken from the automatic evaluation table in the Design2Code paper, and Figure 2 is compiled from the general model benchmark table in the WebCode2M paper. Figures and Tables are used to determine whether the hierarchical planning, structural evaluation and closed-loop repair proposed in this paper address the bottlenecks already identified in the publicly available studies. These publicly available values are not presented as the independent experimental results of the prototype in this paper.

Analysis of Figure 1: The CLIP visual similarity of all models in the figure ranges from 81.7% to 90.4%, and the large models can reproduce the general style and general composition of the webpage. The top-level visual differences among the models do not fully reflect the quality of the elements and structures. The fine-tuned model WebSight VLM-8B reached 87.6% and was close to the commercial model; thus, the domain data was effective. However, high CLIP scores may still be accompanied by text illusions, missing nodes, or missing interactions. Therefore, the system needs to check the DOM, events and test results simultaneously.

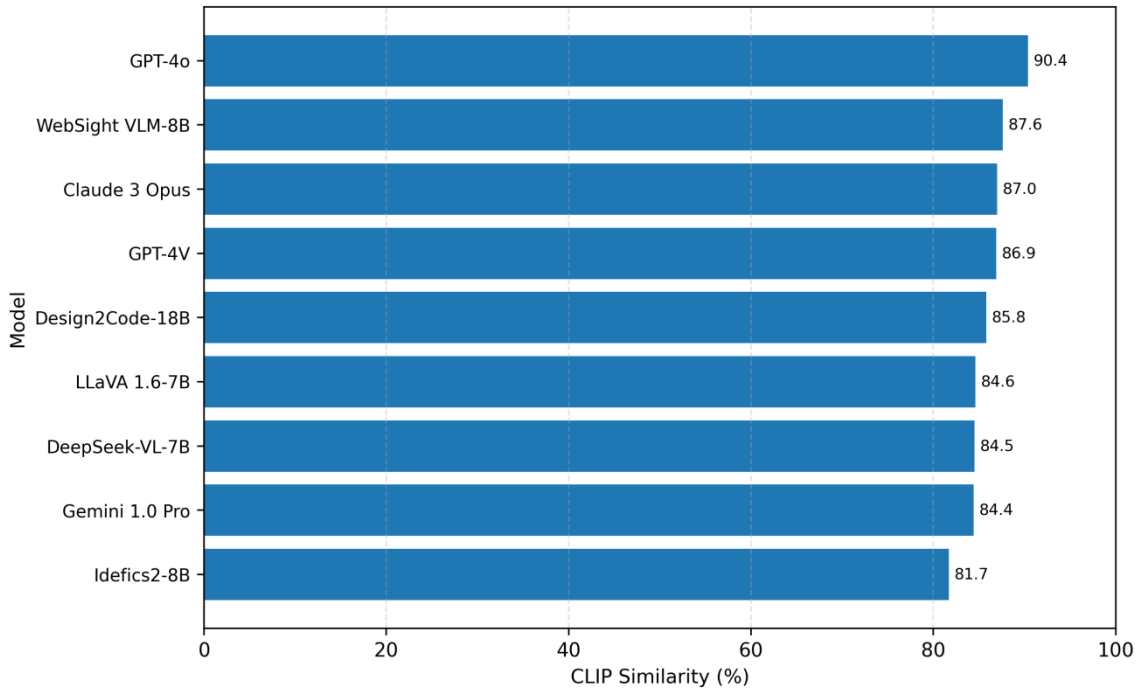


Figure 1: CLIP Visual Similarity of Different Models in the Design2Code Public Benchmark.

Data source: Si et al. [1], Table 1; except for WebSight VLM-8B and Design2Code-18B, the others are directly prompted.

Table 3: Fine-grained metrics for business models directly suggested by Design2Code.

Model	Block Match	Text	Position	Color	CLIP
GPT-4o	93.0	98.2	85.5	84.1	90.4
GPT-4V	85.8	97.4	80.5	73.3	86.9
Claude 3 Opus	90.2	97.5	77.9	71.4	87.0
Gemini 1.0 Pro Vision	80.2	94.6	72.3	66.2	84.4

Data source: Si et al. [1], Table 1; all indicators are in percentages.

As shown in Table 3, the Text metrics for all four business models exceed 94%, but the Position and Color metrics are relatively low, and for Gemini 1.0 Pro Vision, the Color metric is only 66.2. Therefore, the problem is no longer limited to OCR or text copying; the reasons for delivery failure are now more complex, including missing information, incorrect positioning of that information, and inconsistencies in style. Independently model the structure, layout and visuals of this paper, map the differences to specific nodes precisely, and avoid having a single overall score hide local failures.

Figure 2 shows that GPT-4o achieved visual scores of 0.85, 0.81 and 0.82 for short, medium and long code groups, respectively, and showed good stability. GPT-4V and Claude were at 0.62 and 0.37, respectively; Gemini was still in the range of 0.34 to 0.38. Therefore, a weaker model is more likely to accumulate errors as the length of the code and complexity increase. Component graph decomposition, local context and incremental repair can divide long pages into several verifiable subtasks and thus address this problem mechanically.

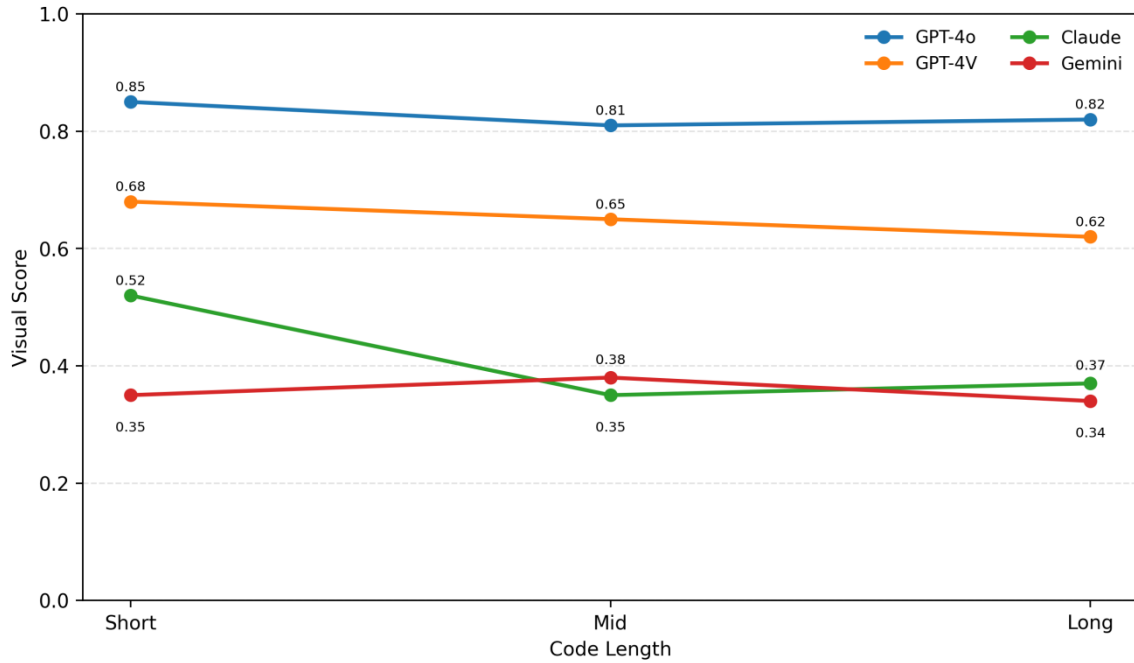


Figure 2: Variation of Visual Scores for the General Model at Different Code Lengths in WebCode2M

Data source: Gui et al. [3], Table 4; Short, Mid and Long correspond to the three code length groups in WebCode2M.

Applicability and Limitations of the Method

Together, the public data support the three conclusions. First, although the overall visual similarity is high, substantial deficiencies persist in the recall, location and colour of individual components; therefore, many metrics must be employed. Second, real-world data and large codebases vary significantly among models; therefore, the system should be divided into smaller modules for individual tasks and reuse existing components. Third, the high consistency between executable tests and human judgment indicates that sandbox validation can provide the basic quality support for an automated generation system and should not be regarded as an extra step after generation.

The method in this paper is suitable for web applications with a clear component structure, an operable build process and a reference design. It is very convenient for building the back-end interface quickly, as well as forms, marketing pages and design system components. For highly dynamic data visualisation, complex animations, WebGL, products with strong real-time collaboration, and products that inherently have uncertain requirements, visual screenshots cannot fully demonstrate the behaviour, and interactive videos, state machines, or manual design specifications are still required.

The three deficiencies of this method are: First, the publicly available statistics are based on different benchmarks and cannot directly demonstrate the absolute benefits of a specific prototype in a unified environment; Second, the DOM of the reference page may not be the best engineering implementation, and structural similarity should be used in conjunction with semantics and maintainability; Third, large model generation may introduce copyright, privacy and supply chain risks, and enterprise deployment must retain data anonymisation, dependency whitelists, manual

approval and audit logs. Future research will carry out ablation experiments in the same component library and browser environment to quantify the contributions of multimodal specifications, component graph planning, rendering feedback and test gating.

6. Conclusion

This paper studies generative AI for the automatic generation of web front-end components. It analyzes the development path of existing research on screenshot-to-code, real-world datasets, capability benchmarks and executable testing, identifies four core problems in one-time free generation: ambiguity in multimodal requirements, conflict between visual and engineering goals, accumulation of errors in complex pages, and lack of execution feedback. Based on the above, a closed-loop approach has been proposed, which includes multimodal requirement parsing, component graph planning, constrained code synthesis, dual DOM and visual evaluation, interaction testing, and quality gates. A Three-Tier Engineering Implementation Plan is also available. According to the public benchmark statistics, although the overall visual similarity of the current model is high, it still lacks precision in position, colour, element recall and long-page stability; thus, structured decomposition and difference-driven repair are required. For actual application, the front-end automatic generation system should embed the functions of large models in a reproducible build, browser and test environments, reduce the generation of entire pages through minimal patch iterations, and control delivery risks with accessibility, security and maintainability rules. In the future, a unified component-level benchmark for real-world projects will be established to address cross-framework migration, responsive behaviour, multi-user collaboration and long-term code evolution assessment, and thus raise this technology from prototype support to reliable software engineering productivity.

References

- [1] Si C, Zhang Y, Li R, Yang Z, Liu R, Yang D. *Design2Code: Benchmarking Multimodal Code Generation for Automated Front-End Engineering*. *Proceedings of NAACL-HLT 2025*, 2025: 3956-3974. DOI: 10.18653/v1/2025.naacl-long.199.
- [2] Laurençon H, Tronchon L, Sanh V. *Unlocking the Conversion of Web Screenshots into HTML Code with the WebSight Dataset*. *arXiv preprint arXiv:2403.09029*, 2024.
- [3] Gui Y, Li Z, Wan Y, Shi Y, Zhang H, Su Y, Chen B, Chen D, Wu S, Zhou X, Jiang W, Jin H, Zhang
- [4] Wan Y, Wang C, Dong Y, Wang W, Li S, Huo Y, Lyu M R. *Automatically Generating UI Code from Screenshot: A Divide-and-Conquer-Based Approach*. *arXiv preprint arXiv:2406.16386*, 2024.
- [5] Lin Z, Zhou Z, Zhao Z, Wan T, Ma Y, Gao J, Li X. *WebUIBench: A Comprehensive Benchmark for Evaluating Multimodal Large Language Models in WebUI-to-Code*. *arXiv preprint arXiv:2506.07818*, 2025.
- [6] Wang, B. (2025). *Methods of Load Optimization for Computer Systems Based on Physical Principles*.
- [7] Guo, X. (2025, March). *Research on Blockchain-Based Financial AI Algorithm Integration Methods and Systems*. In *2025 IEEE International Conference on Electronics, Energy Systems and Power Engineering (EESPE)* pp. 816 – 821. IEEE.
- [8] Guo, X. (2025, April). *Research on Financial Trading Algorithms Based on Deep Reinforcement Learning*. In *2025 IEEE 3rd International Conference on Control, Electronics and Computer Technology (ICCECT)* pp. 1711 – 1715. IEEE.

- [9] Yu, X. (2026). *Exploration of Multi-Channel Conversion Path Optimization Methods Based on A/B Testing*.
- [10] Jing, X. (2025, November). *In-Depth Analysis of the Optimization Decision Model of Financial Investment Portfolio Based on Machine Learning Algorithms in Python Scikit-Learn Library*. In *International Conference on Computational Technologies for Research in Data Analytics* pp. 130 – 140. Cham: Springer Nature Switzerland.
- [11] Liu, X. (2026). *Research on the Application of Artificial Intelligence in Consumer Privacy Data Protection*. *Procedia Computer Science*, 279, 956-965.
- [12] Liu, X. (2026). *Construction of Consumer Data Privacy Protection System Based On Blockchain Technology*. *Procedia Computer Science*, 282, 2082-2091.
- [13] Huijie Pan. *Discussion on Low-Latency Computing Strategies in Real-Time Hardware Generation*. *International Journal of Neural Network* (2025), Vol. 4, Issue 1: 57-64.
- [14] Yuan, Y. (2026). *Research on Memory Management and Dynamic Reasoning Methods for Intelligent Agents Based on Large Language Models*.
- [15] Yang, Y. (2026). *Causal Inference-based Identification of Incremental Effects in Digital Advertising and Optimization Pathways for Resource Allocation*. *Advances in Computer and Communication*, 7(2).
- [16] Sun, L. (2026). *Consistency Optimization Strategies for Distributed Advertising Systems Under Cross-regional Deployment*. *Advances in Computer and Communication*, 7(2).
- [17] Han, W. (2025, November). *An Empirical Study on Building a High-precision Forecasting System for Financial Market Trends Based on the PyTorch Deep Learning Framework*. In *International Conference on Computational Technologies for Research in Data Analytics* (pp. 518-528). Cham: Springer Nature Switzerland.