Dean&Francis

ISSN 2959-6157

Analysis of the Application and Prospects of Artificial Intelligence in Various Front-end Frameworks

Haochen Wang

School of Information Science and Technology, Beijing Forestry University, Beijing, China, 100083
willwalker0226@gmail.com

Abstract:

In recent years, the rapid development of artificial intelligence has influenced software engineering thoroughly. Especially in front-end development, the combination of artificial intelligence with front-end frameworks such as Vue, React and Angular is becoming a significant factor in advancing productivity and developing efficiency. This paper mainly investigates the application mechanism of AI over different front-end frameworks, and also researches the prospects of future development. This study aims to analyze the different aspects of AI integration over several frameworks, comparing their characteristics' influence on their performance in generating code and converting the layout design into code. Research adopts literature analysis and comparative study, combining existing AI development tools and academic materials. The accuracy, capacity and adaptiveness of AI within the front-end frameworks are discussed in this paper. The findings indicate that AI has established a significant efficiency advantage in front-end development by generating components automatically, optimizing the logic between code, and solving debug errors. However, due to differences in syntax features and data complexity, AI's effectiveness varies from different front-end frameworks.

Keywords: Artificial Intelligence, Front-end Frameworks, Large Language Models, Machine Learning, Multi-modal AI

1. Introduction

The potential application of Artificial Intelligence(AI) in front-end development has attracted attention from the industry. Front-end frameworks such as Vue, React, and Angular have become fundamental tools, improving development efficiency and interaction experience. Traditional front-end development, which relies on full manual coding, requires more labor, leading to higher financial and time costs in projects. However, the application of Large Language Models (LLMs) and generative AI ensures design-to-code

approaches are more approachable, offering new opportunities for assistance in front-end engineering. Previous research has made progress in code generation and automatic bug fixing. The integration of AI with front-end frameworks is worthy of systematic research. This paper aims to explore how AI can be applied in major front-end frameworks. Focusing on analyzing the implementation of these models and the differences during code generation and automatic testing. The research employs both literature review and case analysis, comparing the application of models within existing front-end frameworks, focusing on code generation accuracy, semantic consistency, and testing automation efficiency. This study not only provides insight into AI's potential for front-end developers but also contributes a theoretical foundation for practical work on AI-driven front-end automation and cross-framework integration.

2. Literature Review

2.1 AI in Software Engineering

Recent advances demonstrate that LLMs fine-tuned on code can synthesize nontrivial programs and power developer assistants. Code-specialized models such as codex are capable of functional program synthesis, they showed both strengths (high pass rates with sampling) and limitations (complex binding, long-chained operations) [1].

2.2 AI in Front-end Development

Multimodal LLMs are being evaluated specifically for translating visual design into usable code. LLMs construct a real-world benchmark and automatic metrics for converting images to front-end implementations, exposing current gaps between human design intent and generated code quality, and providing a standardized task for comparing models on visual to code generation [2].

2.3 Combining AI with Front-end Frameworks

Works on the code generation model indicates that a syntactic and semantic-aware model improves accuracy for code editing and bug correction. Research argues for integrating code structure and search to raise accuracy, a lesson that applies when targeting framework-specific idioms (React hooks, Vue reactivity, Angular decorators) [3]. Current studies on design-to-code and code generation have produced promising prototypes. However, there is still no systematic comparison across front-end frameworks. Most benchmarks focus on full Hyper Text Markup Language(HTML) or Cascading Style Sheets(CSS) generation, or on general code synthesis such as Codex. Few researchers test how framework syntax, component structures, or ecosystem tools affect generation accuracy, testing efficiency, or automatic repair. These gaps limit our understanding of how AI models perform under specific frameworks like React, Vue, or Angular. Taken together, the recent studies highlight several key points. Code-tuned LLMs show strong generative ability, and front-end research needs more multimodal benchmarks. Code-aware modeling plays an important role in improving correctness. These insights point toward clear directions for future cross-framework research.

3. Integration Patterns of AI and Front-End Frameworks

3.1 Code Generation and Completion

One of the primary ways artificial intelligence integrates with front-end frameworks is by generating or completing code components automatically. LLMs specialized for code are often fed with natural language descriptions, UI component specifications, or existing code snippets, and tasked with generating components in frameworks like Vue, React, or Angular. In Vue, generated code often includes template syntax and reactive data bindings. In React, AI tends to generate JavaScript XML(JSX) code containing functional components, hooks, and state logic. In Angular, code generation accounts for decorators, TypeScript typing, and module structure. Due to differences in syntax specifications, component lifecycle models, and module dependency architectures, the accuracy of AI tools in generating ready-to-use code varies significantly. Benchmarks such as Web2Code demonstrate that while AI models can generate HTML from a design draft with fidelity, they struggle to maintain layout hierarchy precisely and handle nested components well when complexity increases[4]. Models evaluated in the WebMMU benchmark also reveal that despite visual element recognition capabilities, converting that into semantically correct code that adheres to framework specifications remains error-prone, especially when dealing with framework-specific syntax structures [5].

3.2 Design-to-Code Automation

Design-to-Code refers to systems that directly convert visual design drafts into front-end code. Such a pipeline generally involves visual understanding, element detection, layout parsing, and code synthesis. The Design2Code benchmark introduced nearly 500 real-world webpage screenshots to evaluate current multi-modal LLMs' ability to convert designs into accurate HTML/CSS and common JavaScript outputs. The results indicate that top models like GPT-5 have advanced capabilities to replicate basic

ISSN 2959-6157

layouts and visual elements. However, there's incompetence when it comes to accurately positioning smaller UI components, maintaining spacing, responsiveness, or alignment under different screen sizes [6]. Another study FullFront, examines the entire front-end engineering workflow, including design perception, comprehension Q&A, and code generation. Reveals that design-to-code is merely one component within the overall process. Performance degrades significantly when interaction dynamics or real-world complexities are involved [7].

3.3 Automated Testing and Debugging

AI also assists in testing and debugging front-end frameworks by generating test cases, detecting bugs or UI anomalies, and sometimes automatically correct code. Test requirements vary across frameworks. Vue often focuses on verifying the correctness of reactive systems, state changes, and template bindings. React projects might prioritize unit tests of functional components and hooks. Angular requires more extensive structural and dependency tests due to services, dependency injection, and lifecycle hooks. Some models infer missing test scenarios by analyzing properties and attributes, state transitions or changes in UI element trees. However, existing benchmarks indicate that AI-assisted automation often overlooks edge-cases. Such as Angular's dependency injection lifecycles or Vue's reactivity caveats. Automation struggles to preserve test robustness when UI evolves. Furthermore, although improvements like "self-healing selectors" or automatic test updates are prospective, current tools remain more useful for simple components or stable UI rather than complex and dynamic applications.

4. Comparative Analysis and Application Prospects

4.1 AI Integration in Vue

Vue's template-syntax and declarative directives (v-bind, v-model) make it especially susceptible to template-driven generation: the separation between structure (HTML-like templates) and behavior (script) reduces ambiguity in mapping visual components to code, helping models produce semantically coherent templates and wiring for two-way data flows. This clear separation of focus provides an ideal structured input for multimodal AI systems, enabling models to more accurately map interface elements to underlying data logic. Two-way binding (v-model) simplifies state synchronization patterns, which can lower downstream verification cost when models generate interactive behavior from mockups. Research indicates that integrating data flow analysis within code generation process enables AI systems to simulate reactive updates before code completion, significantly reducing runtime inconsistency caused by data flow errors [8]. Practical pipelines that target framework-specific templates (rather than raw HTML) report higher fidelity when the target framework exposes compact, declarative primitives—an observation reflected in recent design-to-code systems.

4.2 AI Integration in React

React's JSX blends markup with JavaScript, offering a single, expressive substrate that is easy for large, token-based models to learn from large corpora of open-source projects. JSX's explicit component structure and popular idioms (props, hooks) benefit models trained on abundant React examples, improving generation recall for component APIs. However, React's functional patterns—especially Hook-driven state and effect management—introduce subtle lifecycle constraints that models must respect to avoid bugs. Research indicates that missing or misordered Hook dependencies can lead to hard-to-track race conditions and memory leaks, making this one of the most common error patterns in AI-generated code [9]. Component reuse and correct Hook placement remain open challenges for current generators and often require post-generation static checks or repair loops. Throughout, emerging AI code assistants like React Agent enhance logic error detection rates by verifying compliance with Hook rules through the construction of component dependency graphs. Surveys and benchmarks emphasize that framework-specific training and self-correction mechanisms materially improve outcome quality.

4.3 AI Integration in Angular

Angular's strong TypeScript typing and modular architecture offer clear advantages for producing well-typed, maintainable code: type constraints can be used during decoding to reduce compilation errors, improving trustworthiness in enterprise settings. Full-Front benchmarking indicates that Angular projects achieve approximately 40% higher type safety metrics than non-TypeScript projects, providing a solid foundation for static verification of AI-generated code [10]. At the same time, Angular's verbose module, DI patterns and configuration-heavy scaffolding raise the bar for generation: models must handle richer metadata and project-level conventions. Type-constrained decoding methods and multi-stage generation are promising directions to reconcile Angular's complexity with automated generation needs.

5. Cross Frameworks Comparison

5.1 Overview

To examine which front-end framework is most adaptive for AI development, researchers illustrate the way multimodal large language models (MLLMs) integrate within different front-end frameworks, and systematically compared their performance across three major frameworks, React, Vue, and Angular on the DesignBench benchmark [11]. Each framework employs distinct LLMs to generate framework-specific syntax and component architectures by AI, thereby reproducing web content. Metrics include visual fidelity (CLIP), which measures the semantic similarity between generated web pages and their original counterparts [12]; compilation success rate (CSR), which indicates the proportion of samples that compiled successfully in the overall sample. There is also a metric called code modification similarity (CMS) [13]. For calculating code modification similarity, set A is defined as the lines changed in the ground-truth code, while set B represents the lines altered in the generated code. CMS is calculated as $CMS(A,B) = | A \cap B / A \cup B |$. Additionally, the assessment incorporates MLLM-as-Judge evaluation scores[14] to gauge output quality.

5.2 Quantitative Comparison

Research provides a systematic quantification of the framework-dependent performance variations discussed above. It details the performance scores of React, Vue, and Angular within multiple models across different tasks, Empirical results using metrics in order to reveal clear framework-dependent performance differences.

Table 1. The framework performance on designbench within different models

Metric	Framework	Claude-3.7	GPT-4o	Gemini-2.0	Llama-90B	Average
Design Generation						
CLIP	React	0.81	0.77	0.76	0.71	0.7625
	Vue	0.84	0.77	0.69	0.54	0.71
	Angular	0.61	0.59	0.61	0.54	0.5875
CSR	React	0.96	0.97	0.91	0.95	0.9475
	Vue	0.98	0.95	0.84	0.75	0.88
	Angular	0.69	0.71	0.71	0.74	0.7125
Design Editing						
MLLM Score	React	8.18	8.01	7.82	6.16	7.5425
	Vue	8.37	8.19	8.06	6.26	7.72
	Angular	8.02	8.29	9.14	5.66	7.7775
	Vanilla	9.15	9.23	9.03	7.7	8.7775
CMS	Vue	0.41	0.37	0.33	0.22	0.3325
	React	0.47	0.53	0.38	0.27	0.4125
	Angular	0.69	0.61	0.64	0.47	0.6025
CSR	React	1	0.98	1	0.92	0.975
	Vue	0.98	0.95	0.96	0.92	0.9525
	Angular	0.91	0.91	1	0.87	0.9225
Design Repair						
MLLM Score	React	6.79	6.36	6.33	4.22	5.925
	Vue	6.6	6.26	6.08	4.78	5.93
	Angular	6.86	5.93	5.29	4.65	5.6825
CMS	React	0.49	0.28	0.18	0.05	0.25
	Vue	0.31	0.26	0.18	0.05	0.2
	Angular	0.58	0.51	0.40	0.31	0.45
CSR	React	1	1	1	0.93	0.9825

ISSN 2959-6157

	Vue	1	1	0.96	1	0.99
	Angular	0.93	1	1	0.79	0.93

As Table 1 shows, React exhibited better performance, with CLIP scores in the range of 0.7–0.8 and CSR between 0.9–0.95. Moreover, React demonstrates higher stability and maintains consistent performance across different models. In contrast, Vue exhibits slightly lower stability when switching between large language models, with significant fluctuations in CLIP and CSR scores. Its overall performance remains consistently strong, achieving scores of 0.7 and 0.87. Notably, when using Claude and GPT models, Vue achieves the highest CLIP and CSR scores among these three frameworks. However, Angular underperformed, showing CLIP scores in the 0.5–0.65 range and CSR around 0.70. This indicates that the syntactic rigidity of Angular's TypeScript and module dependencies poses challenges for current MLLMs, while the declarative nature of React's JSX and Vue's template system provides a relatively more forgiving generation space.

5.3 Compilation Error Patterns

Compilation logs show framework-specific failure modes that reveal the models' underlying comprehension gaps. React errors are predominantly characterized by "Unexpected Token" and "Expression Expected" syntax issues, alongside the increasingly common missing "use client" directive. Vue compilation failures most often involve structural problems such as "Missing End Tag," "Unexpected EOF" (End of File), and various "Attribute Error" messages. Angular's error profiles are typically defined by "incomplete block" declarations and numerous "component import/export errors" related to its module system.
These findings indicate that model-specific enhancements significantly reduce syntax-related errors, particularly for React and Vue. However, due to Angular's reliance on nested module declarations and TypeScript's strictness, it remains the most challenging to handle.

5.4 Error Correction Ability

A follow-up experiment sampled 30 pages with different compilation failures to evaluate self-repair capabilities. Average repair rates across all models and frameworks remained modest (0.53 overall), with per-framework variation shown in Table 2.

Table 2. Compile error fix ratio of different frameworks

Framework	Claude-3.7	GPT-4o	Gemini-2.0	Llama-90B	Average
React	70%	60%	60%	50%	60%
Vue	40%	50%	80%	60%	57.5%
Angular	70%	50%	80%	80%	70%

Angular achieves the highest average repair rate at 0.7, outperforming React (0.6) and Vue (0.575). This trend reveals distinct performance differences across frameworks. Angular's strong showing is driven by models like Gemini-2.0 and Llama-90B, while React and Vue also demonstrate reasonable repair capabilities. Such variation reflects how framework-specific syntax complexity, particularly Angular's TypeScript based architecture, influences MLLMs' ability to repair structural code. In the future, the correction robustness still needs further improvements in order to cope with the development of large enterprise applications.

5.5 Componentization and Reusability

Across all frameworks, AI-generated front-end code exhibited weak componentization. Quantitative analysis showed that the average componentization rate was only 0.24% in React, about 5% in Vue, and approximately 19% in Angular. Despite Angular achieved a comparatively higher ratio, closer examination indicated that most components constituted superficial replicas instead of reusable generalizations, implying that the models encountered difficulties in transforming beyond fixed code blocks. In Vue, for instance, the model often regenerated static HTML for repetitive elements rather than utilizing the v-for directive, demonstrating limited understanding of template-based iteration. Similarly, React's low ratio revealed limited learning of JSX-based composition patterns, as AI systems often generated single-purpose elements instead of functional and reusable components with props or hooks. These findings indicate that although AI can imitate framework syntax, it remains deficient structural abstraction and contextual understanding required for genuine component-level reusability, an essential characteristic for scalable, maintainable front-end projects.

5.6 UI Issue Recognition Accuracy

In the design repair task, models were prompted to detect

UI inconsistencies of DesignBench. Average UI issue recognition accuracy was below 0.40 across all frameworks.

Table 3. UI issue identification accuracy of different frameworks

Framework	Claude-3.7	GPT-4o	Gemini-2.0	Llama-90B	Average
React	41.60%	43.70%	42.50%	28.30%	39.03%
Vue	26.50%	28.70%	32.10%	37%	31.08%
Angular	39.30%	41%	26.50%	14.70%	30.38%

As Table 3 shows, the React framework achieves the highest recognition accuracy at 0.39. Vue and Angular perform relatively poorly, with similar accuracies of 0.31 and 0.3 respectively. Both React and Angular exhibit lower accuracy on the Llama-90B dataset, significantly below other models. Results indicate that even top-tier MLLMs cannot yet reliably align visual cues with functional UI expectations across frameworks.

5.7 Key Observations of Frameworks

React and Vue outperform Angular in visual fidelity and compile stability due to simpler syntax and more lenient structural rules.

Angular's strong typing and complex dependency graph reduce generation success but yield slightly higher modular reuse when correctly produced.

MLLMs struggle across all frameworks to capture reusable, maintainable component hierarchies, which is an essential frontier for design-to-code research.

6. Conclusion

This paper explores how AI integrates within major frontend frameworks, Vue, React, and Angular. Evaluate their respective advantages, limitations and prospects. Researches reveal that AI assisted development is changing the industry by generating code and debugging automatically. However, the performance is still highly dependent on the framework's syntax and architecture. React and Vue demonstrate greater adaptability for AI-driven development due to their declarative design and concise structural patterns, leading to higher visual fidelity and compilation stability. React shows better stability and more adaptive across different LLMs, which might be benefits from sufficient open source data and the flexibility of JSX syntax. Meanwhile, Vue's template-driven model supports clearer data binding and structural mapping. Although Angular's structure is relatively more complex, its capability that excels at generating stable type-safe code, offering potential for large-scale enterprise applications prioritizing reliability. However, challenges exist. Code generated by LLMs within frameworks is commonly insufficient in component reusability, abstraction levels. These shortcomings indicate the necessity for deepened semantic modeling, framework specific training, and dynamic self repair mechanisms. Looking forward, integrating multimodal reasoning with type constrained decoding technology shows prospect for simultaneously enhancing accuracy and maintainability. The integration of AI and frontend frameworks thus unfolds along a dual path: while expanding automation and innovation opportunities, it demands models with refined interpretability and structural comprehension to achieve sustainable, scalable development.

References

[1] Chen, Mark et al. "Evaluating Large Language Models Trained on Code." ArXiv abs/2107.03374 (2021): n. pag.

[2] Si, Chenglei et al. "Design2Code: Benchmarking Multimodal Code Generation for Automated Front-End Engineering." North American Chapter of the Association for Computational Linguistics (2024).

[3] Jiang, Nan et al. "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair." 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (2021): 1161-1173.

[4] Yun, Sukmin et al. "Web2Code: A Large-scale Webpage-to-Code Dataset and Evaluation Framework for Multimodal LLMs." ArXiv abs/2406.20098 (2024): n. pag.

[5] Awal, Rabiul et al. "WebMMU: A Benchmark for Multimodal Multilingual Website Understanding and Code Generation." ArXiv abs/2508.16763 (2025): n. pag.

[6] Si, Chenglei et al. "Design2Code: Benchmarking Multimodal Code Generation for Automated Front-End Engineering." North American Chapter of the Association for Computational Linguistics (2024).

[7] Sun, Haoyu et al. "FullFront: Benchmarking MLLMs Across the Full Front-End Engineering Workflow." ArXiv abs/2505.17399 (2025): n. pag.

[8] Prototype2Code: End-to-end Front-end Code Generation (arXiv:2405.04975). arXiv https://arxiv.org/abs/2405.04975

[9] A Survey on Large Language Models for Code Generation (arXiv:2406.00515). arXiv https://arxiv.org/abs/2406.00515

[10] Type-Constrained Code Generation with Language Models (arXiv:2504.09246). arXiv https://arxiv.org/abs/2504.09246

[11] Xiao, Jingyu et al. "DesignBench: A Comprehensive

ISSN 2959-6157

Benchmark for MLLM-based Front-end Code Generation." ArXiv abs/2506.06251 (2025): n. pag.

[12] Radford, Alec et al. "Learning Transferable Visual Models From Natural Language Supervision." International Conference on Machine Learning (2021).

[13] V. Thada and V. Jaglan, "Comparison of jaccard, dice, cosine similarity coefficient to find best fitness value for web retrieved documents using genetic algorithm," International Journal of Innovations in Engineering and Technology, vol. 2, no. 4, pp. 202–205, 2013.

[14] Wang, Ruiqi et al. "Can LLMs Replace Human Evaluators? An Empirical Study of LLM-as-a-Judge in Software Engineering." Proc. ACM Softw. Eng. 2 (2025): 1955-1977.