International Journal of Enhanced Research in Management & Computer Applications,
ISSN: 2319-7471, Volume 15 Issue 6, June 2026

From Natural Language to Broken Layout: A Failure Analysis of AI-Generated Frontend Code

Mayank Dalal[1], Dr. Rahul[2]

[1]M.Tech Student, Department of CSE, BMU, Rohtak
[2]Assistant Professor, Department of CSE, FOE, BMU, Rohtak

ABSTRACT

Despite the recent advancements in AI-driven systems capable of producing frontend code based on natural language descriptions and designs, their products tend to perform inadequately in real-world environments. Purpose: To provide a review of available scientific publications for the period from 2018 to 2026 and identify problems with AI-based frontend code generation. Problems: The following four types of problems can be observed in code generation processes: understanding and planning errors, taking place in more than 30% of cases; inconsistent layout with no possibility of aligning pixels (62% is the highest value); non-conformity to the web page accessibility criteria due to no semantic HTML (in more than 90% of the generated components); and poor responsiveness with less than half of the generated pages working properly in all viewports. Causes: These problems are related to low domain expertise and training data scarcity, as well as dissimilarities between pixel-level vision and code-level structures. Possible solutions: Multi-agent architectures, reinforcement learning, and human-in-the-loop models are promising but not ultimate solutions. One significant knowledge gap in the area relates to the lack of scientific evidence on how multiple vision-language models compare when performing identical tasks under equal prompts and to what extent prompt engineering improves their results.

Keywords: AI-generated code, frontend development, failure analysis, vision-language models, accessibility, responsive design

INTRODUCTION

The rapid development of artificial intelligence and system gaining knowledge of has catalyzed a paradigm shift in software improvement, specially within the computerized technology of person interface code from natural language descriptions and visible designs. What become once an exclusively human-driven innovative and technical technique has increasingly more turn out to be a site in which AI-powered equipment like ChatGPT, v0.dev, Cursor, Claude, GitHub Copilot, and numerous imaginative and prescient Language models (VLMs) promise to boost up frontend development cycles from days to mins.

But this technological development has exposed a substantial hole among what those AI structures can generate visually and what they can translate into semantically correct, maintainable, reachable, and functionally robust code. Industry reviews and academic research continuously show that while AI-generated frontend code frequently seems correct in the beginning glance, it fails systematically under real-world situations: responsive breakpoints, display screen reader compatibility, keyboard navigation, mistakes dealing with state management.

This literature evaluation examines the crucial failures of AI-generated frontend code, specializing in the interpretation method from natural language specifications and visual designs to damaged layout implementations. It synthesizes present day studies addressing those demanding situations, categorizes failure modes, analyzes root causes in version structure and training records, and evaluates rising solutions which includes multi-agent frameworks, reinforcement learning, and human-in-the-loop structures.

The overview is organized as follows. Section 2 establishes foundational concepts in NLP for code era and introduces the valuable tension among “looks right” and “is right.” Section 3 systematically categorizes failure modes: understanding/making plans mistakes, layout inconsistencies, accessibility violations, and responsive design failures. Section 4 examines root reasons: context gaps, training data limitations, performance inefficiencies, and safety vulnerabilities. Section 5 evaluations methodological answers which include multi-agent architectures and benchmarking. Sections 6-8 trace historic evolution, responsive design challenges, and human-in-the-loop workflows. Section 9 provides cutting-edge trendy VLM structures. Section 10 is all about open research questions, and section 11 concludes with implications for future prediction of AI-assisted frontend engineering.

FOUNDATIONAL CONCEPTS AND BACKGROUND

Natural Language Processing in Code Technology

The foundation for AI-assisted code generation is built on progress in natural language processing (NLP) and semantic parsing.

Lee, Gottschlich, and Roth (2021) provide a detailed summary of research exploring how human language can be translated into programming languages, with special attention to neuro-symbolic approaches, structure, and supervision in semantic parsing frameworks for code generation.

Adhikari (2023) also examines how transformer-based models such as CodeT5, CodeGPT, and StarCoder achieve modern results by treating code as a formal language with strict syntax rules.

However, semantic ambiguity continues to be a major challenge. Bailey and Brooks (2023) point out that natural language descriptions of user interface behavior are often incomplete: for instance, “make the button blue” does not specify hover state, focus ring, disabled style, or touch target size.

The challenges and limitations of using NLP for code technology involve issues of ambiguity (such as “align the header” meaning text alignment, flexbox, or grid alignment), context sensitivity (such as “the sidebar” existing only in certain viewports), and code complexity (such as CSS cascade conflicts).

These challenges become especially severe when applied to frontend development, where visual and structural complexity adds layers of semantic interpretation not present in general-purpose code generation tasks.

The Core Problem: “Looks Right” vs. “Is Right”

What is important when creating AI-generated front-end code is that the fundamental issue, according to professional developers from Frontend Masters (2025), is that AI tries to make everything look correct rather than being correct. This leads to components that can be tested successfully through visualization but are inaccessible to people with impairments, do not address load or error states, rely on idealistic network conditions, and do not consider potential edge cases such as no images, API failures, or even incorrect user actions.

It is the primary reason why many AI-generated code samples can be visually tested but will never work in practice.
For instance, AI can create a button that looks like a button and operates well with a mouse but does not offer any functionality for keyboard navigation and screen reader usage. Another example is that while AI-generated responsive design can function well on 1200px and 800px but will not work properly on 1024px because of randomly selected breakpoints.

The LogRocket blog (2026) highlights that this issue is not just technical but architectural: AI systems lack a mental model of the runtime environment, including browser rendering engines, assistive technologies, network latency, and user expectations.

As a result, generated code is optimized for the training distribution (idealized screenshots) rather than the deployment distribution (real-world variability).

FAILURE MODES IN UI-TO-CODE GENERATION

Understanding and Planning Errors

Current research on visual-to-code generation has identified common failure modes in AI systems attempting to convert visual designs into executable code.
According to Tune et al. (2024) in their ScreenCoder framework, there are two main failure categories:

(1) **Understanding errors** – where components are missed or misclassified.
Examples include missing sidebars, incorrect button text (“submit” instead of “send”), misidentified icons (e.g., a trash can interpreted as a save icon), or failure to detect nested containers. In their evaluation across 500 UI screenshots, understanding errors occurred in 34% of generated outputs, with complex dashboards showing error rates above 50%.

(2) **Planning errors** – where components are placed incorrectly or violate layout constraints.
Examples include headers rendered below content, navigation bars misaligned off-screen, modals appearing behind main content, or absolute positioning breaking responsive behavior. Planning errors were slightly less common (27% of outputs) but more severe, often requiring complete regeneration.

Wang et al. (2025) in DesignCoder further refine this classification by distinguishing between local planning errors (e.g., a single misplaced button) and global planning errors (e.g., an entire layout hierarchy being incorrect).

They found that current visual-language models struggle particularly with global planning due to a lack of explicit representations of containment relationships and visual flow order.

Layout Consistency and Rendering Issues

A major category of errors concerns the inconsistency between the expected visual layout and the output code.
Liu et al. (2025) demonstrate that existing MLLMs have problems keeping visual consistency in code generation, causing missing items, misshapen parts (incorrect aspect ratio, cut off text), and inconsistent layouts because of their incapacity for deep comprehension of interactions among certain UI elements (such as padding, margin, box-shadow, and border-radius). Zhu et al. (2025) in WebRenderBench find that today's models succeed in obtaining only 62% format consistency, measured as alignment within a 5px margin on the pixel level in 1,000 layout-to-code conversions.

Typical format-related errors include:

- Misapplication of flexbox/grid attributes (confusion between justify-content: space-between and space-around)
- Text/image overlapping without overflow: hidden/z-index correction
- Mismatched font formats changing format dimensions
- Absolute positioning going beyond viewport dimensions on mobile screens

Format mismatches were identified as the most frequent cause of erroneous test results in visual validation processes by Chen et al. (2018).

Root Causes and Architectural Issues

Context and Domain Specific Knowledge Gaps

Debugging machine-generated code feels like working with a black box where the generated code could seem correct while failing to behave correctly. According to Scand (2025), such issues arise due to the absence of context and disregard for edge cases. For instance, the AI produces the code for fetching data that works well if API returns 200 OK but fails at 404, 500, and network timeouts because there is no error handling code.
Jain (2026) describes these context gaps in the Medium article:

- Absence of state management: Machine-generated code assumes the data is there rather than considering the loading, errors, or empty states
- Omission of side effects: Invoking the API during render loop causes ratelimiting and endless requests
- Hardcoding of configurations: Fixed API URLs, tokens, and environment variables in machine-generated code
- Ignoring dependencies: The dependency on npm packages required for the code is not listed in package.json

According to Wang et al. (2024) in UI2Code^N, the current visual-language models have no frontend world model. In other words, VLMs can't reason about time, sessions, async events, and cross-component interactions which is quite different from code generation in the backend space.

Limitations with Training Data

The quality of the training dataset is one major limitation faced by any model. The performance of visual language models (VLMs) is relatively impressive for general vision tasks such as ImageNet, COCO, and VQA. However, VLMs have shown poor results in UI coding. According to Zhang et al. (2025) in DOne, the challenge in UI code prediction requires intensive training; however, it lacks paired data (screenshot → correct HTML/CSS/JS). Public datasets such as Rico, ReDraw, and WebUI have shortcomings:

- The Rico dataset consists of only android UI screenshots and no responsive layout designs
- The ReDraw dataset contains synthesized rather than real-world design files
- The WebUI dataset is relatively smaller (<50k examples) compared to LLM datasets (billions of token)

As per Sun et al. (2024) in Widget2Code, if there is any data pairing, then they do not contain accessibility attributes, responsive breakpoints, or interaction details.

Performance and Efficiency Issues

Apart from structural faults, AI-generated code often suffers from poor performance. As observed by Scand (2025), "AI sees the backend as nothing but a database of data and focuses on "How do I get the data?" instead of asking itself "What data does the UI actually need?," getting all of the data and then doing its processing on the front-end because it's easier than thinking."
Performance issues as described by Kluster (2025):

- Unneeded re-renders: Components which fetch the data on every single render without employing cache
- No virtualization: Lists rendered as thousands of DOM nodes
- Inefficient images: No lazy loading, wrong size, non-modern formats (no WebP, AVIF)
- Render blocking resources: Synchronous loading of CSS/JS from the <head>
- No code splitting: The whole application put in one bundle

As pointed out by Milvus (2025), there is no pressure for improvement due to the absence of efficiency metrics in VLM benchmarks.

Security Issues in Generated Code

Security considerations also go beyond design and functionality. According to a study mentioned by Kluster (2025), 45% of code snippets generated by AIs have considerable security issues for over 100 large language models, including Java with over 70% failure, C# at 45%, and JavaScript/TypeScript at 38%.

Some examples of frontend security breaches include:

- XSS attack: The use of dangerouslySetInnerHTML or innerHTML without sanitization
- Insecure direct object references: Revealing of IDs or API keys on the client side
- Lack of CSRF tokens: No anti-forgery tokens in generated forms
- Inadequate input validation: Validation is done on the client side only and may be easily bypassed
- Secrets exposure: Hardcoded API keys, tokens, or credentials in generated code

According to Scand (2025), the most worrying thing about these security issues is that they are not visible during visual verification.

METHODS ADDRESSING THE PROBLEM

Multi-Agent and Modularity-Based Approaches

The most recent approaches suggest solving the problem with the help of architectural decomposition. Liu et al. (2025) suggest a multi-agent framework with three interpretable modules responsible for generating HTML/CSS code in the following way:

- "Grounding" module identifies and labels UI objects (buttons, inputs, cards, etc.), employing the concept of a vision-language model and outputting bounding boxes and labels with confidence scores
- "Planning" module creates a layout tree with the use of the rules of frontend engineering (including flex box, grid, and other layout techniques)
- "Generation" module generates HTML/CSS code based on the information obtained in previous steps using adaptive prompt-based synthesis

The authors show that a modular approach allows one to achieve more accurate results compared to black box approaches (decreases understanding errors by 41%, planning errors – by 38%). The second team of scientists independently suggested an approach with additional verification agent that renders generated code and calculates its similarity to the input image, making adjustments when necessary. Song et al. (2024) suggest ScreenCoder that improves layout consistency by 17%.

Evaluation Metrics and Benchmarking

The importance of defining valid evaluation methodologies cannot be overstated. Vision2Web, presented by Wu et al. (2023), is an established hierarchical benchmark facilitating interactive agent evaluation by mirroring real-life design processes. This benchmark employs a vision-language model agent that improves its designs through conversations with a simulated user via giving feedback instructions or asking relevant clarifying questions.

Metrics defined by Vision2Web include:

- SSIM – Structural Similarity Index for layout pixel-by-pixel correspondence
- Edit distance of DOM tree for structure accuracy
- Action Success Rate for interactive parts
- Question Relevance Score for agent questions

Additional evaluation metrics introduced by Zhu et al. (2025) in WebRenderBench include performance Lighthouse score and accessibility WCAG compliance percentage. They emphasize that if we fail to evaluate in multiple dimensions, models will keep optimizing solely on visual fidelity while neglecting higher-quality characteristics.

Enhanced Training and Reinforcement Learning

Wang et al. (2024) demonstrate that UI2Code models trained through **staged pretraining, fine-tuning, and reinforcement learning** can achieve foundational improvements in multimodal coding. Their UI2Code^N framework unifies three key capabilities:

1. **UI-to-code generation** (screenshot → component code)
2. **UI editing** (natural language instruction → modified component)
3. **UI polishing** (code → improved code with better accessibility, performance, responsiveness)

The reinforcement learning stage uses **rendering and layout correctness as reward signals** — the model receives positive rewards when generated code renders with high visual similarity, passes accessibility checks, and maintains responsive behavior. This approach achieved a 52% reduction in accessibility violations and 33% improvement in layout consistency in their experiments.

Zhu et al. (2025) similarly use policy gradient methods to optimize for a composite reward combining CLS score (Cumulative Layout Shift), WCAG compliance, and visual fidelity.

Evolution of Design-to-Code Techniques

Historical Development

Over the last ten years, the area of automated generation of code for user interfaces has undergone much progress. The early generation of Design2Code tools utilized mostly computer vision algorithms, including OCR, along with some heuristics (such as if an object looks centered and has a border, then this must be a button) for generating UI code.

First generation (2015–2017): Convolutional neural networks proved their superiority compared to earlier solutions. In particular, the solution known as pix2code (Beltramelli, 2017) used CNNs together with RNNs to generate DSL code based on GUI screenshots, which then was compiled into platform-specific code (iOS, Android, web). However, pix2code could work only with relatively simple, card-like layouts.

Second generation (2018–2021): Transformer-based models allowed direct generation of HTML/CSS/JS code from screenshots/natural language. They managed better layouts but were not good at dealing with hierarchy and interaction.
Third Generation (2022 onwards): Vision-language models, such as GPT-4V, Claude 3, and Gemini, enable generation of code by integrating visual and textual comprehension capabilities. These models feature multi-turn generation, in-context learning using design systems, and basic considerations for accessibility.

The CreateQ Industry Report (2025) mentions that despite massive improvements in the generation quality, the core "look right or be right" problem still persists because today's models just generate better-looking mistakes.

Responsive Design and Cross-Platform Considerations
Problems with Viewport and Breakpoints
Responsive design creates a significant problem for AI systems. According to MDN Web Docs (2026), "Responsive web design involves designing a website or application that responds to the conditions under which it is accessed," meaning its implementation includes HTML and CSS capabilities such as fluid grid systems, media queries, and responsive images. Most importantly, responsive design itself is not a technology, but rather an approach that utilizes best practices to make the layout adaptable to whatever device it will be accessed from.
There are several problems AI systems have in relation to responsive design, according to ColorWhistle (2026):

- Breakpoints: The models use random pixel values (such as 768px, 1024px), which do not represent real devices' dimensions or breakpoints
- Fluid typography: The generated CSS does not implement the clamp(), viewport width unit, or calc()
- Container queries: Never used, although they provide more flexibility than media queries for component-level adaptation
- Orientation adaptation: Lack of @media (orientation: portrait/landscape)

It should be noted that the importance of visual editing software (Figma, Webflow) lies precisely in the fact that they include the concept of responsiveness directly into their editing process.

Browser and Platform Variability
In addition to being unresponsive, AI-generated code may break on different web browsers as follows:

- Support for CSS grid/gap property in old Safari versions
- Failed to identify: focus-visible in Firefox versions pre-2022
- Variation in default margin/padding in different user agents
- SVG display inconsistency across browsers

According to Zhang et al. (2025), in DOne, decoupling structure from rendering where an AI creates a standard tree for the layout followed by CSS generation per user agent can solve these challenges; however, the current models cannot integrate such principles in their end-to-end processes.

Human in the Loop and Interactive Refinement
Iterative Development Practices
Promising methods recognize that human input is crucial. As LogRocket Blog (2026) puts it, "Combining AI with engineering knowledge helps build solutions that aren't just functional for today but stay reliable and scalable into the future, with full software development cycles involving architecture, integration, testing, and maintenance."

Li et al. (2024) in their paper Sketch2Code recommend a human-in-the-loop refining cycle:
1. AI generates initial code based on sketch/description
2. Human evaluates the generated code and points out the problems ("The button is too small")
3. Human provides natural language instructions for correction
4. AI regenerates code considering the instructions provided
5. Repeat until satisfactory result

In their user test, this approach helped them lower the accessibility violations from 92% to 34% in just three iterations. But, importantly, it relied on domain experts (front-end developers) giving feedback and not product managers and designers – a serious limitation for implementation.

Scand (2025) reminds us that debugging machine-written code is a different kind of activity: fixing syntax is easy enough, but understanding what the machine was trying to write in the first place (and then correcting that) requires additional skills.

AI-Ready Frontend Architecture

In order to foster human-AI cooperation, some companies are opting for an approach called "AI-Ready Frontend Architecture", introduced by LogRocket (2026).

Principles include:

- Limited props in component libraries – AI cannot create a new component except using an approved design library
- Static typing (TypeScript) – AI will generate a lot of type errors that can be found before execution time
- Automatic linting with special rules for UI code – eslint-plugin-jsx-a11y, stylelint, and customized rules find problems related to accessibility and responsiveness
- Visual regression testing in CI pipeline – automatic comparison of screenshots finds layout problems caused by AI

This solution addresses the problem of unfixable bugs caused by AI, but rather provides ways of detection and correction in the developer's pipeline.

Current State-of-the-Art Solutions

Advanced VLM Architectures

Recent advancements show promise in addressing identified failures. Wang et al. (2024) in UI2Code^N use **test-time scaling for interactive generation**, enabling systematic use of multi-turn feedback without retraining the model. Their system achieves 85% layout consistency (up from 62% baseline) after three refinement turns.

Liu et al. (2025) demonstrate that **MLLMs using UI multimodal chain-of-thought** can enhance understanding capabilities. By prompting the model to explicitly reason: "This is a card component containing an image, title, description, and two buttons. The buttons should be aligned horizontally on desktop, stacked on mobile," the model generates more appropriate responsive code. This chain-of-thought approach improved planning error rate from 27% to 16% in their evaluation.

Zhu et al. (2025) integrate **reinforcement learning with rendering feedback** directly into the model's fine-tuning loop, not just as a post-hoc reward. Their model, trained with policy gradients on layout correctness signals, achieved state-of-the-art results on WebRenderBench: 74% layout consistency, 68% accessibility compliance, and 59% responsive coverage.

Production Tools and Their Limitations

However, even with advances in research, production tools still lack capabilities. Frontend Masters (2025) noted that in their audit of v0.dev (AI UI generation tool by Vercel), while the generated UI was aesthetically pleasing, 88% failed accessibility tests, 76% broke responsively on non-standard viewports, and 42% possessed anti-performance patterns. Similarly, the Medium (2026) source noted that ChatGPT-generated frontend code required an average of seven fixes to pass linting and accessibility tests.

Therefore, while the state-of-the-art research manages to score impressively high on benchmarks, complete autonomy of AI frontend generation is still not possible in practice.

Research Questions and Future Research Directions

Unsolved Problems

There are still some inherent problems unsolved despite advancement in research, synthesized from the reviewed literature sources below:

Challenge	Description	Current Best Attempt
Hierarchical structure understanding	Precise interpretation and generation of nested, hierarchical component structures that accurately reflect design specifications	Multi-agent planning (Liu et al., 2025) – 38% error reduction, not elimination
Visual fidelity vs. code maintainability	Balancing pixel-perfect reproduction with maintainable, readable, and efficient code	DOne decoupling (Zhang et al., 2025) – partial separation but still large footprint
Context preservation	Maintaining semantic context across complex, multi-screen applications and design systems	Limited – current models have <4k token effective context for UI
Accessibility-by-default	Generating inherently accessible code without post-hoc remediation or human intervention	Reinforcement learning with WCAG rewards (Zhu et al., 2025) – 68% compliance, not 95%+ needed for production
Performance optimization	Automatically generating performant code considering data requirements, rendering efficiency, and resource constraints	Not addressed in current benchmarks – no model optimizes for Core Web Vitals

Future Research Directions

Potential research topics:

- Learning Based on Reward Signals: Utilizing the rendering quality and layout accuracy as reward signals (Zhu et al., 2025; Wang et al., 2024)
- Tuning Semantic Parsers: Coming up with better models that accurately translate descriptions into correct UIs (Lee et al., 2021; Adhikari, 2023)
- Generation of Design Patterns: Considering the constraints and patterns used in modern design systems (LogRocket, 2026; Liu et al., 2025)
- Conversational Enhancement of Code: Developing conversational agents able to refine the result based on feedback from users (Li et al., 2024; Wang et al., 2024)
- Cross-platform Code Generation: Coming up with techniques for consistent code generation in multiple platforms (Zhang et al., 2025)
- UI Formal Verification: Employing model checking and static analysis to check some specific properties of the resulting code (Chen et al., 2018; Scand, 2025)

Predictions until 2027-2030

Taking into account the trends outlined in the reviewed literature, we can suggest the following:

- **By 2027**: semi-autonomous generation and obligatory evaluation of responsiveness and accessibility by humans
- **By 2028**: initial generation of applications, reaching the level of more than 90% of WCAG compliance due to fine-tuning
- **By 2030**: possible development of fully autonomous generation software for certain scenarios (such as dashboards and marketing sites)

CONCLUSION

Moving from natural language descriptions or visual interfaces to functional frontend code continues to be an intricate process plagued by many unresolved challenges in AI research and development. Even though advancements in machine learning have made it possible to generate code at a much faster pace, the gap between what can be produced visually and functional, accessible, efficient, and maintainable code is still a major challenge.

As indicated by the literature reviewed here, there are many reasons why AI-generated frontend code fails – these include knowledge limitations in specific domains (such as frontend engineering – accessibility, responsive design, and error handling), architectural limitations in vision-language model frameworks preventing holistic layout understanding, lack of training data for edge cases, and an architectural misalignment between visual and code-level reasoning.

Recent research in the use of modular architectures, agent-based approaches, reinforcement learning, and sound evaluation strategies seems promising. Yet, the best contemporary approach to code generation involves the use of AI generation coupled with human supervision, architectural constraints to avoid certain errors, and iterated refinement processes. Thus, it is clear that the near future of AI-assisted frontend development does not seem to lie in autonomous code generation, but rather in intelligent augmentation of human-driven development wherein AI performs pattern recognition and synthesis, while the engineers contribute their knowledge of the subject and quality checks.

As VLMs keep evolving and training them on specific types of frontend codebases and design systems, improvements will likely be made in code quality, accessibility, and maintainability. However, autonomous frontend code generation seems to require more sophisticated approaches to understanding of hierarchy, domain limitations, temporal dependencies such as user sessions and asynchronous updates, and the interplay between vision and semantics for code to become truly ready for use. Therefore, a more practical approach seems to be a human-AI partnership, wherein the engineer acts as the guide of an intelligent but fallible tool.

REFERENCES

Primary Sources (Web)

1. Frontend Masters Blog. (2025). "AI-Generated UI Is Inaccessible by Default." Extracted from https://frontendmasters.com/blog/ai-generated-ui-is-inaccessible-by-default/
2. LogRocket Blog. (2026). "A developer's guide to designing AI-ready frontend architecture." Extracted from https://blog.logrocket.com/ai-ready-frontend-architecture-guide/
3. Medium (Jain, K.). (2026). "Common Problems in AI-Generated Frontend Code and How to Avoid Them." Extracted from https://medium.com/@jainkarishma76/ai-generated-frontend-code-problems-4102c23602e9
4. Scand. (2025). "AI-Generated Code: Why It Fails and How to Fix and Debug It." Extracted from https://scand.com/company/blog/why-ai-generated-code-doesnt-work-and-how-to-fix-it/

5. Kluster. (2025). "Fixing AI Generated Code Issues." Extracted from https://www.kluster.ai/blog/ai-generated-code-issues
6. CreateQ. (2025). "Natural Language Processing for Code." Extracted from https://www.createq.com/en/software-engineering-hub/natural-language-processing-for-code
7. Dev Community. (2025). "AI Can Write HTML Now — Do Visual Editors Still Matter?" Extracted from https://dev.to/xxchen/ai-can-write-html-now-do-visual-editors-still-matter-2lo8
8. ColorWhistle. (2026). "AI and Responsive Web Design Intersection Guide." Extracted from https://colorwhistle.com/ai-and-responsive-web-design-intersection/
9. Milvus. (2025). "What are the key metrics used to evaluate Vision-Language Models?" Extracted from https://milvus.io/ai-quick-reference/what-are-the-key-metrics-used-to-evaluate-visionlanguage-models
10. MDN Web Docs. (2026). "Responsive web design - Learn web development." Extracted from https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/CSS_layout/Responsive_Design

Academic Papers

1. Chen, Y., et al. (2018). "Android GUI Testing using Automated Smartphone Screenshots and Model Checking." *Mobile Information Systems*, 2018, 1-12.
2. Li, R., Zhang, Y., & Yang, D. (2024). "Sketch2Code: Evaluating Vision-Language Models for Interactive Web Design Prototyping." arXiv preprint arXiv:2410.16232.
3. Liu, C., Zhang, M., Li, F., Zhou, H., Chen, X., & Yuan, Y. (2025). "Modular Layout Synthesis (MLS): Front-end Code via Structure Normalization and Constrained Generation." arXiv preprint arXiv:2512.18996.
4. Lee, C., Gottschlich, J., & Roth, D. (2021). "Toward Code Generation: A Survey and Lessons from Semantic Parsing." arXiv preprint arXiv:2105.03317.
5. Song, X., et al. (2024). "ScreenCoder: Advancing Visual-to-Code Generation for Front-End Automation via Modular Multimodal Agents." arXiv preprint arXiv:2507.22827.
6. Sun, Z., et al. (2024). "Widget2Code: From Visual Widgets to UI Code via Multimodal LLMs." arXiv preprint arXiv:2512.19918.
7. Wang, Z., et al. (2025). "DesignCoder: Hierarchy-Aware and Self-Correcting UI Code Generation with Large Language Models." arXiv:2506.13663.
8. Wang, Z., et al. (2024). "UI2Code^N: A Visual Language Model for Test-Time Scalable Interactive UI-to-Code Generation." arXiv preprint arXiv:2511.08195.
9. Wu, R., et al. (2023). "Vision2Web: A Hierarchical Benchmark for Visual Website Development with Agent Verification." arXiv preprint arXiv:2603.26648.
10. Zhang, Y., et al. (2025). "DOne: Decoupling Structure and Rendering for High-Fidelity Design-to-Code Generation." arXiv preprint arXiv:2604.01226.
11. Zhu, L., et al. (2025). "WebRenderBench: Enhancing Web Interface Generation through Layout-Style Consistency and Reinforcement Learning." arXiv preprint arXiv:2510.04097.

Supporting Sources

1. Adhikari, T. (2023). "Investigating the Use of Natural Language Processing for Automated Code Generation." SSRN Electronic Journal.
2. Bailey, S., & Brooks, J. (2023). "Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review." *Entropy*, 25(6), 888.