Vivek Sharma, 2026, 14:3
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

International Journal of Science, Engineering and Technology
An Open Access Journal

AI-Powered UI Component Generation: A Natural-Language-to-Code Approach Using Large Language Models

Vivek Sharma[1], Dr. Jasbir Kaur[2], Sandhya Thakkar[3]
[1]Student of MCA, Guru Nanak Institute of Management Studies, Mumbai, India
[2]Director, GNIMS B-School, Head of Information Technology and HR, Guru Nanak Institute of Management Studies, Mumbai, India
[3]Assistant Professor, Guru Nanak Institute of Management Studies, Mumbai, India

Abstract- Front-end developers spend a significant portion of their time on repetitive UI markup, hindering productivity and innovation. This paper presents an AI-powered system that converts plain-English descriptions into styled HTML components using Google's Gemini language model. The tool supports three styling paradigms: plain CSS, utility-first (Tailwind), and component-based (Bootstrap). It provides an interactive editing environment with a Monaco-based code editor and a live preview pane. In a user study with 50 front-end practitioners, the system reduced average component development time by 81.5% (from 12.4 to 2.3 minutes) and received a System Usability Scale score of 86/100. The paper details the architecture, prompt engineering strategy, implementation, and evaluation. Despite its benefits, the system exhibits limitations, including inconsistent accessibility support, sensitivity to prompt phrasing, and lack of conversational refinement. These findings underline the potential and the remaining challenges of applying generative AI to front-end development.

Keywords— Artificial Intelligence, UI Generation, Large Language Models, Google Gemini, Monaco Editor, Low-Code, Code Synthesis, Human-Computer Interaction.

I. INTRODUCTION

Modern web development is increasingly demanding rapid, responsive, and accessible interface construction. According to a 2023 survey by the Front-End Developer Community, engineers spend approximately 37% of their development time on repetitive UI coding tasks such as writing HTML/CSS for common components, adjusting responsive breakpoints, and ensuring cross-browser compatibility [1]. This repetitive work not only slows down the prototyping cycle but also contributes to developer burnout and reduces the time available for higher-level design and problem solving.

Component libraries like Material-UI, Ant Design, and Bootstrap have alleviated some of this burden by offering pre-built, styled components. However, they do not eliminate the need for boilerplate code; developers must still select appropriate components, configure props, override styles, and ensure accessibility. The underlying repetition remains, merely shifted from writing raw CSS to wiring up library components. Recent breakthroughs in large language models (LLMs) – including OpenAI's GPT-4, Google's Gemini, and Anthropic's

Claude – have demonstrated exceptional code generation capa-bilities. These models, trained on billions of lines of publicly available code, can interpret natural-language instructions and produce syntactically correct, often functionally complete code. This capability opens the door to a new paradigm: describing a UI element in plain English and receiving a fully styled, ready-to-use implementation almost instantaneously.

In this paper, we introduce the AI UI Component Generator, a system that harnesses Google's Gemini Pro to convert free-form textual prompts into self-

contained HTML/CSS components. The user can choose among three output styles: plain CSS, Tailwind CSS, or Bootstrap. The system integrates a browser-based code editor (Monaco, the engine behind Visual Studio Code) and a live preview pane, enabling rapid iteration and immediate visual feedback.

Our contributions are threefold:

- We describe the design and implementation of an open-source, extensible pipeline for UI component gen-eration, including a novel prompt engineering strategy.
- We present a comprehensive empirical evaluation with 50 front-end developers, measuring development time, code quality, and user satisfaction.
- We provide a balanced discussion of the system's benefits and limitations, highlighting both the productivity gains and the remaining challenges in accessibility, prompt sensitivity, and iterative refinement.

The rest of this paper is structured as follows. Section II reviews related work in AI-assisted code generation, low-code platforms, and prompt engineering. Section III details the system architecture, including the prompt processing, model invocation, code sanitisation, and preview rendering. Section

IV outlines the technology stack. Section V presents the operational workflow with an example. Section VI describes the experimental evaluation and discusses results and limitations. Section VII outlines future enhancements, and Section VIII concludes the paper.

II. RELATED WORK

This section reviews three areas of prior work: AI-powered code generation, low-code/no-code platforms, and prompt engineering for code synthesis.

1. AI-Powered Code Generation

The automation of programming tasks has been a long-standing goal. Early template-based generators and domain-specific languages provided limited flexibility. The introduction of transformer-based LLMs trained on code marked a paradigm shift. GitHub Copilot, built on OpenAI's Codex, popularised AI-assisted coding by offering line-level and function-level completions within an IDE [5]. While Copilot significantly improves productivity, it is not designed to generate complete, self-contained UI components. Its sug-gestions are context-dependent and typically limited to a few lines.

OpenAI's Codex demonstrated the ability to translate natural-language comments into code across various program-ming languages [4]. However, its applications have focused on algorithmic and backend tasks rather than presentation-layer code. Subsequent models, including GPT-4 and Gemini, have expanded the scope, but UI generation remains relatively underexplored despite representing a large portion of everyday front-end work.

Other systems, such as Amazon CodeWhisperer and Replit's Ghostwriter, provide cloud-based AI assistance, but they lack dedicated support for UI components with live preview and framework switching. Our system addresses this gap by combining a specialised prompt pipeline with an interactive development environment tailored for UI generation.

2. Low-Code and No-Code Platforms

Low-code and no-code platforms (e.g., Webflow, Bubble, Wix) have democratised web development by enabling visual assembly of interfaces through drag-and-drop interactions [16]. These tools lower the barrier for non-programmers and acceler-ate the creation of simple websites and applications. However, they impose limitations on customisation, often lock users into proprietary ecosystems, and generate opaque code that cannot be easily exported or integrated into existing projects.

Our approach occupies a middle ground: it leverages the expressive freedom of LLM-based prompting while constrain-ing the output to well-defined, self-contained UI modules. The generated code is clean, readable, and exportable, allowing developers to

integrate it into their preferred workflow without sacrificing flexibility.

3. Prompt Engineering for Code Generation

Prompt engineering has emerged as a critical factor in obtaining high-quality LLM outputs. The way a prompt is phrased, structured, and contextualised significantly influences the model's behaviour. Reynolds and McDonell [2] showed that explicit formatting instructions and context-aware prompts reduce errors in generated code by up to 40%. Few-shot prompting – providing examples of desired outputs – further improves performance, especially for tasks with complex formatting.

In the UI domain, specific challenges arise from the need to generate visually coherent, responsive, and accessible markup.

Ebrahimi et al. [11] explored prompt design patterns for generating accessible HTML, finding that explicit instructions to include ARIA attributes increased their presence from 23% to 71%. Similarly, instructing the model to use semantic HTML elements significantly improves structural quality. We adopt these insights in our prompt processing pipeline, as detailed in Section III.

III. SYSTEM ARCHITECTURE

The system follows a client-server architecture, with generative processing performed on the server via API calls. The pipeline consists of four main stages: prompt processing, model invocation, code synthesis, and preview rendering.

1. Prompt Processing and Engineering

User-entered descriptions are enriched with contextual cues and constraints before being sent to the model. The enrichment includes:

- Framework specification: The selected styling framework (plain CSS, Tailwind, or Bootstrap) is explicitly stated with version and key conventions.
- Structure guidance: The prompt specifies expected HTML tags (e.g., <h2> for titles, <ul> for lists) and semantic roles.
- Quality constraints: Instructions for responsiveness, ARIA attributes, colour contrast, and mobile-first design are appended.
- Formatting example: A brief example of desired output is included to establish stylistic consistency.

This structured prompt generation improves adherence to the expected output format and reduces malformed code. Table I illustrates the transformation of a user prompt.

2. Model Invocation

The enriched prompt is sent to Google Gemini Pro via a secure REST API with the following configuration:

- Temperature: 0.3 (balance between creativity and deter-minism).
- Top-P: 0.95 (allow moderately diverse token selection).
- Max tokens: 2048 (sufficient for most UI components).
- System instruction: A custom system prompt biases the model toward clean, semantic HTML with appropriate styling.

Gemini was selected for its strong performance on code generation benchmarks and its ability to adhere to formatting instructions consistently.

3. Code Synthesis and Sanitisation

The raw response undergoes light post-processing:

- Markdown removal: Extraneous code fences are stripped.
- Tag balancing: Ensures well-formed HTML.
- Sanitisation: Potentially harmful elements (e.g.,
- <script>, <iframe>) are removed.
- Formatting: Consistent indentation is applied for read-ability.

The resulting HTML/CSS snippet is passed to both the editor and the preview renderer.

Table 1: Sample Prompt Transformation

Stage	Content
User Input	"A pricing card with a title, price, features list, and button."
Transformed	"Generate a pricing card component using Tailwind CSS. Include a title (h2), price display (p), unordered list of features, and a primary action button. Ensure the component is responsive, includes appropriate ARIA attributes, and follows the utility-first pattern. Use dark text on light background. Provide complete HTML with inline styles. The component should be self-contained and visually polished."

4. Preview Rendering and Isolation

The sanitised code is embedded in an iframe with a separate document context, providing security isolation, style encapsu-lation, and safe execution. The preview updates automatically on every editor change, enabling rapid iteration.

IV. TECHNOLOGY STACK

The following technologies were selected for their maturity and suitability:

- React: For the front-end UI, with state management for responsive interactions.
- Tailwind CSS: Used for the tool's own layout and offered as an output style.
- Google Gemini (Pro): Core generative model.
- Monaco Editor: Embedded VS-Code engine with syntax highlighting and auto-completion.
- Framer Motion: Subtle animations for loading feedback.

The application runs client-side after the initial load, except for API calls, simplifying deployment.

V. OPERATIONAL WORKFLOW

A typical session follows these steps:

- The user enters a natural-language description (e.g., "a product card with image, title, price, and button").
- The user selects a styling framework: plain CSS, Tailwind, or Bootstrap.
- Upon clicking "Generate", the prompt is assembled and sent to Gemini. A loading indicator appears.
- The generated HTML/CSS appears in the Monaco editor, and the preview updates to show the visual outcome.
- The user can edit the code directly; the preview refreshes automatically.
- The final code can be copied or downloaded as an HTML file.

Table 2: Generation Time by Component Complexity

Complexity	Tokens generated	Time (s)
Simple (button, badge)	150–300	1.2–1.8
Medium (card, form)	400–700	2.1–2.7
Complex (modal, navbar)	800–1200	3.0–4.2

Example Generated Component

For the prompt: "a product card with image, name 'Premium Headphones', price $199.99, rating 4.5 stars, and an 'Add to Cart' button" and Tailwind CSS, the system produces:

```
<div class="max-w-sm rounded overflow-hidden
    shadow-lg">
  <img class="w-full h-48 object-cover"
    src="..." alt="Product image">
  <div class="px-6 py-4">
    <div class="font-bold text-xl
    mb-2">Premium Headphones</div>
    <p class="text-gray-700
    text-base">$199.99</p>
    <div class="flex items-center mt-2">
      <span
    class="text-yellow-500">****</span>
      <span class="ml-2
    text-gray-600">4.5</span>
    </div>
  </div>
  <div class="px-6 pt-4 pb-2">
    <button class="bg-blue-500
    hover:bg-blue-700 text-white font-bold
    py-2 px-4 rounded">
      Add to Cart
    </button>
  </div>
</div>
```

VI. EVALUATION AND DISCUSSION

1. Experimental Setup

We recruited 50 professional front-end developers (28 male, 22 female; average experience 4.7 years) to evaluate the system. Participants came from three companies and two academic institutions to ensure diversity. Each participant built four components (pricing card, navigation bar, form, and hero section) twice: once manually and once using the AI tool. The order was counterbalanced to control for learning effects. We measured development time, code correctness, and user satisfaction via the System Usability Scale (SUS) and open-ended feedback.

2. Quantitative Results

Table 3: summarises the time savings across component types.

Additional metrics: mean generation time 2.8 s, syntactic correctness 96.7%, style compliance 93.4%, SUS score 86/100, and accessibility attribute coverage 67.3%.

TABLE 3: TIME SAVINGS ACROSS COMPONENTS (N=50)

Component	Manual (min)	AI (min)	Reduction
Pricing Card	11.2	2.1	81.3%
Navbar	14.6	2.8	80.8%
Form	13.8	2.5	81.9%
Hero	10.1	1.8	82.2%
Footer	12.3	2.3	81.3%
Average	12.4	2.3	81.5%

3. System Usability Scale Analysis

The SUS score of 86/100 places the system in the "excellent" category, outperforming the industry average of 68. The SUS questionnaire comprises 10 items rated on a 5-point Likert scale. Our participants gave particularly high scores to items such as "I thought the system was easy to use" (mean 4.6/5) and "I felt very confident using the system" (mean 4.5/5). The lowest scores were for "I needed to learn a lot before I could get going with this system" (mean 2.1/5, indicating a low learning curve). These results confirm that the tool is intuitive and approachable, which is critical for adoption in educational and fast-paced development environments.

4. Qualitative Feedback

Users praised the speed ("it handles tedious parts") but noted the lack of iterative refinement and missing accessibility attributes. Common improvement requests included: support for multiple-turn conversations, direct editing of the generated code with live re-generation, and better handling of complex layouts like grids and flexbox. The majority of participants (82%) said they would use the tool in their daily work, especially for prototyping and initial component scaffolding.

5. Limitations

Despite its advantages, the system has several limitations:

- Accessibility: ARIA labels and semantic tags are often missing (coverage only 67.3%). This is a known weakness of current LLMs, which prioritise visual correctness over inclusive design.
- Prompt sensitivity: Small wording changes can yield different outputs, sometimes failing to match the user's intent.
- No conversational refinement: Each generation is state-less; users cannot make incremental adjustments without re-entering the full description.
- External dependency: Relies on Gemini API, introducing latency, potential downtime, and privacy concerns.
- Design fidelity: Generated components may not match exact mock-ups, requiring manual adjustments.

These limitations highlight the continued need for human oversight, especially for accessibility and design alignment.

6. Comparison with Existing Tools

Table IV compares our system with other popular tools.

Our system distinguishes itself by combining natural-language input, multi-framework output,

and live preview, making it particularly suitable for educational contexts and rapid prototyping.

Table 4: Comparison with Existing Development Tools

Feature	Copilot	Codex	Webflow	Proposed
Natural language input	Partial	Yes	No	Yes
UI component focus	No	No	Yes	Yes
Multiple framework output	No	No	No	Yes
Live preview	No	No	Yes	Yes
Exportable code	Yes	Yes	Limited	Yes
Educational usability	Low	Low	Medium	High

7. Threats to Validity

We acknowledge several threats to the validity of our study. First, the participants were self-selected and may not represent the broader developer population. Second, the components built were relatively simple; performance on more complex UI tasks may differ. Third, the study was conducted in a controlled environment, which may not reflect real-world development with its interruptions and context switching. Future work will address these by replicating the study with a larger, more diverse sample and in a live development setting.

Future Work

We plan to address the identified limitations through several enhancements:

- Conversational refinement: Add multi-turn dialogue for incremental adjustments.
- Automated accessibility checking: Integrate axe-core to detect and fix accessibility issues.
- React component generation: Extend to JSX and hooks for direct use in React projects.
- Interactive preview: Enable click-through interactions (hover, animations).
- Usage analytics: Track modifications to refine prompt templates.
- Multi-modal input: Accept wireframes or screenshots as additional context.
- VS Code/Figma plugins: Embed the generator directly into design and development tools.
- Performance optimization: Implement caching and par-allel generation to reduce latency.
- Support for more frameworks: Add support for Angular, Vue, and Svelte.

VII. CONCLUSION

We presented an LLM-based system that converts natural-language UI descriptions into fully styled HTML components. The integration of a live editor and preview creates an interactive environment that substantially accelerates prototyping. Our empirical evaluation with 50 developers confirmed an 81.5% reduction in development time and high user satisfaction (SUS 86/100). However, accessibility gaps and prompt sensitivity remain open challenges. The tool demonstrates the practical value of generative AI for front-end tasks, and we have open-sourced the code to encourage further research and community contributions.

Acknowledgment

We thank the participants of the usability study and the GNIMS faculty for their support.

REFERENCES

1. Front-End Developer Community, “State of Front-End 2023,” 2023.
2. L. Reynolds and K. McDonell, “Prompt programming for large language models,” CHI Extended Abstracts, 2021.
3. Google AI, “Gemini API Documentation,” 2024.
4. M. Chen et al., “Evaluating Large Language Models Trained on Code,” arXiv:2107.03374, 2021.
5. GitHub, “GitHub Copilot,” 2023.
6. Meta, “React Documentation,” 2024.
7. Tailwind Labs, “Tailwind CSS,” 2024.
8. Microsoft, “Monaco Editor,” 2024.
9. Bootstrap Team, “Bootstrap,” 2024.
10. J. Brooke, “SUS: A quick and dirty usability scale,” Usability Evaluation in Industry, 1996.
11. A. Ebrahimi et al., “Prompt Engineering for Accessible HTML,” ASSETS, 2023.

12. J. Meyer et al., "Cognitive Load in Front-End Development," J. Softw. Eng. Res. Dev., vol. 10, no. 2, 2022.
13. T. Brown et al., "Language Models are Few-Shot Learners," NeurIPS, 2020.
14. Cursor AI, "Cursor – AI Code Editor," 2024.
15. Replit, "Ghostwriter," 2024.
16. M. Beheshti et al., "A systematic review of low-code and no-code development platforms," J. Syst. Softw., vol. 198, 2023.
17. OpenAI, "GPT-4 Technical Report," arXiv:2303.08774, 2023.