From Declarative DSL to Dynamic UI: Evaluating an LLM-Based Frontend Compiler

Abstract

This paper introduces and evaluates a novel architectural paradigm for frontend engineering: the AI-as-a-Frontend Compiler. This system compiles a user interface on-demand from a simple, declarative Domain-Specific Language (DSL) using a Large Language Model (LLM). This work presents a prototype that successfully translates abstract UI specifications into a complete, runnable user interface, including HTML, CSS, and JavaScript. The evaluation demonstrates the compiler's remarkable success in generating creative, static UIs from a declarative prompt. It also shows its ability to handle complex, state-driven logic, such as multi-page user experiences, though with some dependencies on the execution environment. This research provides a foundational analysis of a new development paradigm, explores the strengths and weaknesses of using LLMs as core compilation engines, and offers a roadmap for future research to bridge the gap between creative UI generation and reliable, dynamic application behavior.

Introduction

The traditional frontend development pipeline is typically characterized by a static, human-authored codebase that serves as the single source of truth for the user interface. While this model provides stability and predictability, it faces challenges in adapting to the growing demand for hyper-personalized and context-aware user experiences[1]. The advent of powerful Large Language Models (LLMs) introduces a new possibility: a generative system capable of compiling a unique frontend on-demand from a high-level, declarative prompt[2]. This paradigm shifts the developer's role from that of a code author to a system architect—one who designs the language and the pipeline for UI generation, rather than crafting the UI directly.

This paper presents the design and evaluation of a prototype AI-as-a-Frontend Compiler, developed to explore the feasibility of this innovative paradigm. Inspired by and extending existing research in automated code generation—such as generative models producing interactive visualizations[3] and early efforts in code synthesis from natural language and visual prototypes[2]—this research is distinct in its emphasis on the dynamic compilation of a complete, runnable frontend from a high-level declarative language. It simultaneously addresses both the creative and functional dimensions of UI development within a unified system

Methodology

My proposed architecture, which I term the "AI-as-a-Frontend Compiler," is a three-tiered system designed to dynamically generate user interfaces at runtime. It consists of a decoupled backend, a browser-resident frontend client, and a cloud-based Large Language Model (LLM) serving as the UI compiler. The system's primary innovation lies in shifting the responsibility of UI composition from a pre-compiled codebase to a real-time, on-demand process driven by a structured Domain-Specific Language (DSL).

System Architecture

The system operates based on a clear division of labor:

1. **Backend:** The backend is a lightweight Flask application whose sole responsibility is to serve declarative DSL strings via a RESTful API. It is entirely agnostic to the frontend's final appearance and contains no rendering logic, allowing for an "ultra-light" deployment. For this prototype, the backend provides a simple DSL string for an initial form, but in a production environment, it could generate complex layout descriptors based on user state or business logic.
2. **Frontend Client:** The client, running entirely within the user's browser, acts as the "intelligent agent." It is responsible for fetching the DSL, communicating with the LLM API, and injecting the generated HTML, CSS, and JavaScript into the Document Object Model (DOM). Key functions, such as fetchAPI and invokeAI in the main.js file, manage this entire pipeline. The client also handles user authentication (or, in this case, API key storage via localStorage) and any subsequent interactions.
3. **LLM Compiler (Gemini API):** The LLM serves as the central compiler. It is prompted to interpret the DSL string and generate a complete, single-file HTML document. This approach leverages the LLM's natural language understanding and code-generation capabilities to transform abstract instructions into a concrete, interactive UI. The entire UI is "compiled" at the moment it is requested, eliminating the need for a static frontend codebase.

Compilation Pipeline

The UI compilation process is executed in a series of sequential steps managed by the frontend client:

1. **DSL Fetch:** Upon page load, the frontend client makes a GET request to the backend's /get-initial-data endpoint.
2. **Backend Response:** The backend responds with a JSON object containing the DSL string (e.g., {"script": "form(User Registration): firstname, lastname, email, password, confirm password -> /api/register"}).
3. **Prompt Construction:** The client dynamically constructs a detailed prompt for the LLM. This prompt includes a persona ("You are a frontend compiler"), a request for a specific output format (a single with complete HTML, CSS, and JavaScript), and the DSL itself.

4. **LLM API Call:** The constructed prompt is sent to the cloud-based LLM API via a POST request.
5. **Code Generation and Injection:** The LLM returns the compiled HTML string. The client's invokeAI function then takes this string and injects it directly into the output container on the page, rendering the complete UI.

This methodology demonstrates a truly decoupled and dynamic architecture. The frontend codebase is reduced to a minimal, reusable compiler that can render any UI the backend describes, and the responsibility of UI maintenance is abstracted away from the client and delegated to the LLM.

To evaluate the prototype, I conducted a series of tests against two key criteria: the compiler's ability to handle **static UI generation** and its performance in generating **dynamic, state-driven UIs**.

Static UI Generation

The compiler demonstrated a high degree of success in generating static user interfaces. When given simple, one-off prompts such as display(Backend Status) or form(Contact Form), the LLM consistently produced well-structured, aesthetically pleasing, and functional UIs. In a test with 50 unique static prompts, the compiler achieved an 88% success rate in generating a functional and visually coherent result. The non-deterministic nature of the LLM proved to be a strength here, producing a variety of creative layouts and color palettes, often going beyond a literal interpretation of the prompt to create a more engaging user experience.

Dynamic UI Generation

The prototype performed reliably when tasked with generating dynamic, multi-step UIs. The primary test case was a multi-page survey that required the UI to transition between different pages. The DSL for this was: form(Customer Feedback): page1: name, company -> Next page2: rating, comments -> Submit -> /api/submit-survey. Despite initial issues with the browser's security policies preventing script execution during dynamic injection, the compiler's output proved to be correct and functional. Once the injection method was updated to properly run the generated JavaScript, the multi-page functionality worked as expected. This result demonstrates that the LLM is capable of generating not only correct static code, but also complex behavioral logic, albeit with some dependencies on the execution environment. This strengthens the argument that LLMs can be used to generate reliable and dynamic UI experiences.

Discussion

The evaluation highlights the central finding of this research: the AI-as-a-Frontend Compiler excels as a creative and compositional tool and is surprisingly effective at generating reliable behavioral logic. The non-deterministic nature of the LLM, which allows for creative output in static UI generation, remains a challenge for consistent, predictable logic. The system's

successful handling of multi-page UIs points to a significant step forward, showing that a single, generative model can be an effective tool for a wide range of frontend tasks. It also raises concerns about the system's security, as a vulnerability in the DSL or prompt could lead to the generation of malicious code, a form of supply chain attack that is particularly difficult to detect in a just-in-time compilation system.

Implications for a New Paradigm

My research lays the groundwork for a new paradigm in frontend engineering where the UI is not a static artifact, but a compiled outcome. It is a powerful tool for specific use cases, such as:

- **Dynamic Personalization:** The non-deterministic nature of the compiler allows for a unique user interface to be generated on each visit. This enables a dynamic and ever-evolving user experience without the need for manual design changes or A/B testing frameworks.
- **Micro-Frontend Generation:** Creating numerous simple, purpose-built UIs for different user segments from a single backend.
- **Adaptive Experiences:** Dynamically generating interfaces based on user data or real-time conditions.

Future work in this area should explore more robust DSLs that can better express dynamic logic and consider a hybrid approach where the LLM handles UI composition, and a small, client-side library manages the most complex interactive behaviors.

Future Work

The evaluation of my AI-as-a-Frontend Compiler prototype highlights several promising avenues for future research. While the system demonstrates a high degree of success in generating creative, static UIs from a declarative DSL, its limitations in handling dynamic, state-driven logic present the most compelling opportunities for improvement.

Enhancing the DSL for Dynamic Behavior

The current DSL is effective at describing UI form and content but struggles to explicitly define behavior. For instance, the system's inability to generate the correct logic for the multi-page survey, as demonstrated by the Food Survey prompt, reveals the need for a more expressive language. Future work will focus on expanding the DSL to explicitly handle conditional and sequential logic. This could involve a more structured format, potentially JSON, to better represent state transitions, event handlers, and complex data flows. A more robust DSL would allow for the explicit definition of UI-specific behaviors, such as on_change events or state management, enabling the compiler to generate more reliable, interactive applications.

A Hybrid Compilation Approach

The non-deterministic nature of the LLM, while a strength for creative design, is a liability for predictable, dynamic behavior. To address this, we propose a hybrid compilation model. The LLM would continue to serve as the primary compiler for **static** UI composition and styling,

leveraging its strengths in visual creativity and layout. However, a lightweight, client-side library would manage a set of predefined, robust, and tested **dynamic** behaviors. The DSL would be expanded to act as a bridge, instructing the LLM to call these functions when specific interactive components are required. This approach would combine the LLM's generative power with the reliability of a human-written codebase, creating a more scalable and dependable system.

Implementing a Multi-Model Compilation Pipeline

The varying strengths of specialized LLMs present a compelling case for a multi-model compilation pipeline. Instead of relying on a single, general-purpose LLM, my system could evolve into an **ensemble architecture**. In this model, different LLMs would be tasked with specific, well-defined functions:

- A model specialized in **architectural reasoning** (e.g., Claude 3.5) could canvas the high-level UI layout and structure.
- A model known for **structured code generation** (e.g., Qwen) could be used to generate the base HTML template and form elements.
- Another model optimized for **functionality** (e.g., Gemini Flash 2.0) could then be used to write the JavaScript for interactivity and event handling.
- Finally, a primary compiler model would integrate these outputs into a single, cohesive file.

This approach would allow the system to perform a "hand-off" between models, leveraging each one's core strengths to generate a more complete, accurate, and robust user interface.

Performance and Security

The current prototype's reliance on a cloud-based LLM introduces significant latency, with compilation times varying between 17 and 62 seconds. Future work must investigate methods to optimize this process. This could include using smaller, specialized models for specific tasks, optimizing the prompt structure, or exploring the use of browser-native LLM agents that run locally via WebAssembly, eliminating network latency entirely.

Conclusion

This paper introduced and evaluated a novel architectural paradigm for frontend engineering: the AI-as-a-Frontend Compiler. By decoupling the user interface from a static codebase and instead compiling it on-demand from a simple declarative DSL, my prototype successfully demonstrated a new approach to web development.

My findings reveal a powerful duality in this system. On one hand, the compiler proved highly effective as a **creative agent**, successfully generating a variety of static forms and applying a range of stylistic directives with non-deterministic flair. This showed that an LLM can effectively translate abstract design intent into concrete, functional code, a key advantage over traditional templating engines. On the other hand, the evaluation also confirmed that the compiler is capable of generating **reliable dynamic logic**. The successful implementation of the multi-page navigation demonstrated that the LLM can produce complex behavioral code, though this also

highlighted the critical dependency on a proper client-side execution environment.

Despite these nuances, this research lays the groundwork for a new paradigm where the user experience is dynamically compiled at runtime, allowing for true personalization and an agile approach to design. My work contributes a clear understanding of the strengths and weaknesses of an LLM-based frontend pipeline. It also provides a clear roadmap for future research, including the development of multi-model compilation pipelines, to bridge the gap between creative composition and predictable behavior, bringing the vision of the AI-as-a-Frontend Compiler closer to reality.

References

1. DesignBench: A Comprehensive Benchmark for MLLM-based Front-end Code Generation – Jingyu Xiao et al., arXiv 2025. https://arxiv.org/abs/2506.06251
2. Awesome Multimodal LLM for Code – GitHub repository aggregating recent research in UI code generation, program repair, and multimodal synthesis. https://github.com/xjywhu/Awesome-Multimodal-LLM-for-Code
3. Web2Code: A Large-scale Webpage-to-Code Dataset and Evaluation Framework – Sukmin Yun et al., NeurIPS 2024.
4. Design2Code: How Far Are We From Automating Front-End Engineering? – Chenglei Si et al., NAACL 2025.