AutoWeb: A Conversational and Evolutionary Multimodal Web-Building Agent

Shaobo Liang
Johns Hopkins University
sliang24@jhu.edu

Ruifeng Ji
Johns Hopkins University
rji10@jh.edu

Abstract

We present **AutoWeb**, a conversational, multimodal, and *evolutionary* web-building agent that constructs, edits, and iteratively refines websites through natural language interactions. Unlike traditional code-generation tools that function as static "one-shot" synthesizers, AutoWeb operates as a persistent agentic system that maintains project state, executes filesystem operations, and manages reusable UI components. The defining innovation of AutoWeb is its **self-evolutionary capability**: the agent can modify its own user interface (UI) and configuration at runtime in response to user needs. By bridging Large Language Models (LLMs) with a deterministic UI builder and a component-based architecture, AutoWeb demonstrates a new paradigm in Human-AI interaction where the toolchain is not fixed but co-evolves with the developer's workflow. This report details the system architecture, specifically the `UIBuilder` engine, the agentic core, and the component injection pipeline.

1 Introduction

Large Language Models (LLMs) such as Codex (1) and StarCoder (2) have revolutionized software engineering by automating code generation. However, current interactions remain largely static: the user provides a prompt, and the model returns a code artifact. This "prompt-and-regenerate" cycle fails to capture the iterative nature of real-world software design, where requirements evolve, and the workspace itself may need to change to accommodate new tasks.

Practical web development requires more than text generation; it requires state management, file manipulation, and visual feedback. Furthermore, distinct users have distinct workflows. One developer may prefer a chat-only interface, while another requires a visual wireframe uploader or specific form controls for database management. Traditional Integrated Development Environments (IDEs) are rigid; changing the interface requires complex plugin development.

To address these limitations, we introduce **AutoWeb**, an interactive agent that bridges the gap between program synthesis and flexible interface design. AutoWeb contributes three key engineering innovations:

1. **Self-Evolutionary UI:** The agent can rewrite its own configuration files to add input fields (e.g., file uploaders, color pickers) or change its visual theme on the fly.
2. **Stateful Component Management:** AutoWeb moves beyond single-file generation by maintaining a registry of reusable components (navbars, footers) and dynamically injecting them into pages via a lightweight client-side loader.
3. **Multimodal Layout Parsing:** A dedicated parser translates JSON-based wireframes into executable HTML/CSS, facilitating a "Sketch-to-Code" workflow.

Preprint. Under review.

2 Related Works

2.1 LLMs for Code Generation

Recent benchmarks by Chen et al. (Codex) (1) and Li et al. (StarCoder) (2) demonstrate that LLMs can generate syntactically correct code. However, these models operate primarily in isolated contexts. AutoWeb builds upon these foundations but wraps the LLM in an agentic loop, allowing it to read and write files to maintain long-term project consistency.

2.2 Agentic Reasoning and Tool Use

The ReAct framework (3) introduced the concept of interleaving reasoning and acting. AutoWeb implements this by equipping the LLM with a specific toolset—`fs.write`, `npm.init`, `playwright.run`—allowing the model to verify its own work. Unlike generic agents, AutoWeb's toolset includes meta-tools (`ui.update_config`) specifically designed to modify the agent's own runtime environment.

2.3 Structured Program Synthesis

Research by Yin and Neubig (4) highlights the importance of syntactic structure in generation. AutoWeb incorporates this via `layout_parser.py`, which deterministically maps abstract wireframe data (JSON) into concrete HTML structures, ensuring that visual intent is preserved even when the LLM's raw generation might hallucinate.

3 System Architecture

AutoWeb is architected as a local Python-based agent serving a dynamic HTML/JS interface. The system is composed of four primary modules: the Agentic Core, the Evolutionary Engine, the Component Registry, and the Layout Parser.

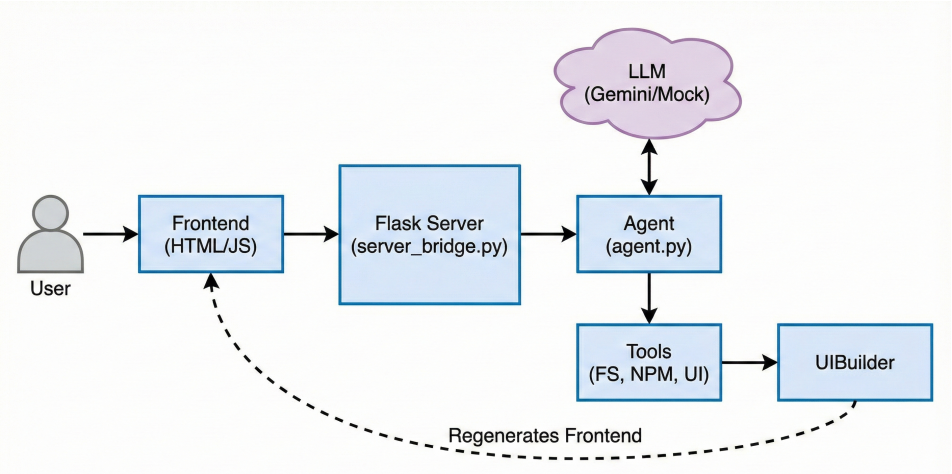

Figure 1: AutoWeb System Architecture

Figure 1: High-level architecture of AutoWeb. The User interacts with the Dynamic UI, which sends instructions to the Flask Server. The Agent processes these instructions using the LLM and executes Tools. Evolutionary tools modify the UI Config, triggering the UI Builder to regenerate the frontend.

3.1 The Agentic Core

The core logic resides in `Agent.py`. Unlike a simple chatbot, the Agent operates on a "Turn" basis with a persistent history and utilizes Google's **Gemini 2.5 Pro** model as the underlying reason-

ing engine. It manages capabilities via a `ToolRegistry`, where tools are explicitly registered and exposed to the LLM via a dynamically generated system prompt.

Context Management. To prevent context window overflow during long coding sessions, the system implements **History Compression**. The `query_llm` method monitors the history length; when the conversation exceeds 50 turns, it triggers a summarization routine. This routine iterates through older turns, preserving the user's high-level intent while collapsing verbose tool outputs (e.g., raw file content) into summary tokens (e.g., "Tool execution completed"). This preserves the narrative flow while optimizing token usage.

3.2 The Evolutionary Engine (`UIBuilder`)

The most distinct feature of AutoWeb is its ability to redesign itself. This is implemented in `ui_builder.py`, which functions as a deterministic compiler for the frontend.

Configuration-Driven Rendering. The frontend state is not hardcoded but defined in a schema file, `.waa/ui_config.json`. This file acts as the "source of truth" for the interface. The `UIBuilder` parses this JSON and maps abstract component definitions to concrete DOM elements using a template engine. For example, an input definition {`"type": "file", "id": "wireframe"`} is compiled into a specific HTML structure containing the file input and associated event listeners in `app.js` to handle data payloads.

The Evolution Loop. The self-modification process follows a strict pipeline:

1. The LLM calls `ui.update_config` to append new field definitions to the JSON schema.
2. The `UIBuilder.generate_ui()` method is triggered. It reads the updated schema and reconstructs `index.html` and `app.js` from scratch.
3. The user interface files are overwritten in place, allowing the frontend to display the new tools immediately upon a page refresh.

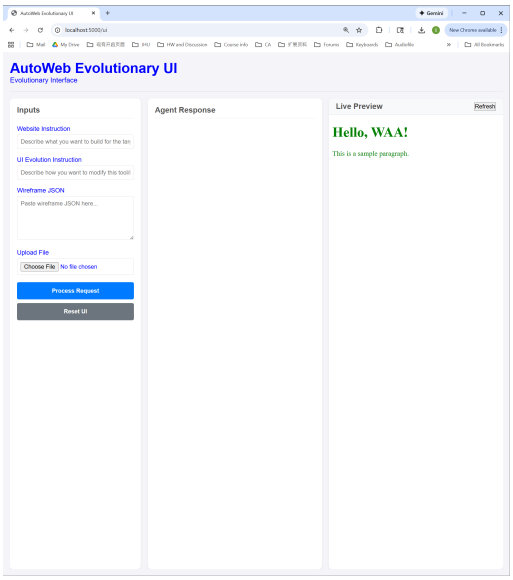

Figure 2: *
Before Evolution

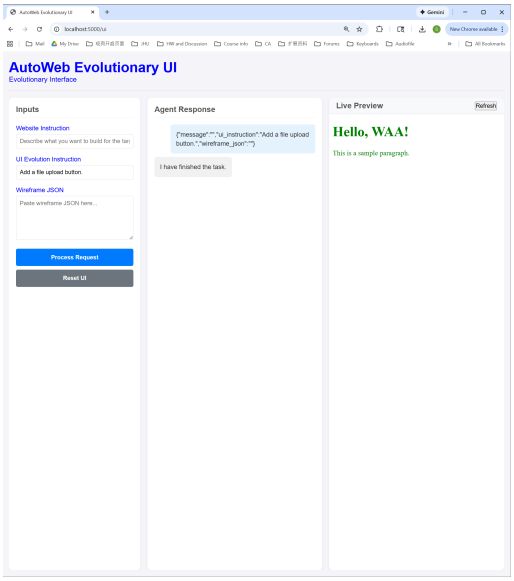

Figure 3: *
After Evolution

Figure 4: Demonstration of UI Evolution. Left: The default interface with standard inputs. Right: The interface after the user instructs the agent to "Add a file upload button," causing the system to modify its own source code and re-render.

3.3 Component and Page Registry

To support complex, multi-page web applications, AutoWeb avoids duplicating code. It implements a component-based architecture.

Dynamic Client-Side Injection. Instead of relying on heavy build steps, the Agent is instructed via its System Prompt to implement a lightweight component loader pattern.

1. Components (e.g., Navbars) are stored as HTML fragments in the `components/` directory.
2. The agent generates a custom script, `loader.js`, which is included in every page.
3. At runtime, `loader.js` scans the DOM for elements with the `data-component` attribute. It asynchronously fetches the corresponding HTML fragment and injects it into the page.

This ensures that the generated code remains human-readable and standard-compliant while allowing the agent to update a single component file and have that change propagate instantly across the entire website.

3.4 Multimodal Layout Parsing

To facilitate multimodal design, `layout_parser.py` implements a deterministic translation layer.

JSON-to-CSS Mapping. The parser accepts a hierarchical JSON representation of a webpage. It traverses this tree, mapping abstract layout types (e.g., "hero", "grid", "card") to standard HTML structures and pre-defined CSS utility classes. For instance, a "grid" node is translated into a container with the `.grid` class, which applies responsive CSS rules (e.g., `display: grid` and `auto-fit` columns). This deterministic mapping ensures that vague or hallucinated layout instructions from the LLM are grounded in valid, responsive visual styles.

4 System Capabilities and Demonstration

We tailored AutoWeb to handle distinct "Website Instructions" (target domain) and "UI Evolution Instructions" (meta domain). Below we detail the capabilities demonstrated by the final prototype.

4.1 Capability 1: Self-Modification

In our testing, we initiated the agent with a basic chat interface. We then instructed the agent: *"Change the primary color to purple and add a file upload button for logos."* The agent successfully executed `ui.update_config`. The `UIBuilder` regenerated `style.css` with the new color palette and injected a `<input type="file" id="logo_upload">` into the DOM. The system performed this without restarting the backend server, validating the "hot-swapping" architecture of the evolutionary engine.

4.2 Capability 2: Full-Stack Project Scaffolding

Using the `npm.init` and `server` tools, the agent demonstrated the ability to:

- Initialize a `package.json` with Express.js dependencies.
- Create a server entry point (`index.js`).
- Launch the server in the background using Python's `subprocess` module and monitor logs via `npm.logs`.

This proves the agent can manage long-running processes, a significant advantage over standard chat-based coding assistants which typically generate code but cannot execute servers.

4.3 Capability 3: Recursive Directory Management

The implementation of `fs.tree` (in `tools/fs.py`) allows the agent to visualize the directory structure recursively. When the user asks "What does the project look like?", the agent utilizes this tool to generate a visual tree, ensuring it remains aware of file locations and depth throughout the session.

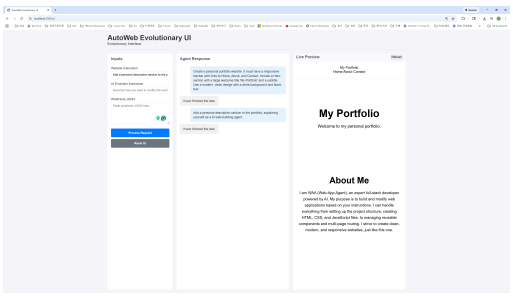

Figure 5: Agent Interface (Input)

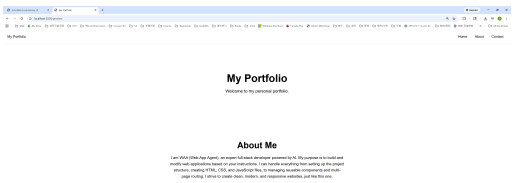

Figure 6: Generated Website (Output)

Figure 7: Multimodal Layout Generation. Left: The AutoWeb interface where the user submits instructions or wireframe data. Right: The resulting fully functional website generated by the agent, featuring a responsive grid layout derived from the instructions.

4.4 Capability 4: Verification via Testing

AutoWeb integrates `Playwright` and `Supertest` tools. After generating a webpage, the agent can optionally run `playwright.run` to take screenshots or verify that specific text appears on the rendered page. This closes the loop between generation and verification, moving closer to autonomous software engineering.

5 Conclusion and Future Work

AutoWeb demonstrates that an LLM-driven agent can act as a **meta-developer**, building not only the target software but also the tools required to build it. By shifting from a static interface to an evolutionary one (`ui_builder.py`), we empower users to customize their development environment through natural language. The system's architecture—separating the evolutionary engine from the coding agent while bridging them with structured tools—provides a robust foundation for the future of collaborative programming.

Looking forward, we propose five key directions to advance AutoWeb from a prototype to a deployable engineering assistant:

- **Backend Tool Synthesis (Logic Evolution):** Currently, AutoWeb's evolutionary capability is limited to its frontend (UI). We aim to extend this to the backend, enabling the agent to write, register, and verify new Python tools dynamically. For instance, the agent could self-author a `database.py` tool or a `git.py` wrapper, effectively expanding its own action space and logic capabilities without human intervention.
- **High-Fidelity Multimodal Input:** While the current `LayoutParser` relies on structural JSON wireframes, future iterations will integrate Vision-Language Models (VLMs) such as GPT-4V or Gemini Pro Vision. This would allow the system to parse raw inputs—such as whiteboard sketches, Figma screenshots, or hand-drawn diagrams—directly into executable DOM structures, bypassing the intermediate JSON abstraction.
- **Cross-Platform Target Generation:** To move beyond web development, we plan to abstract the synthesis layer to support mobile frameworks like React Native or Flutter. This would allow the agent to generate native mobile applications from the same conversational interface, utilizing a unified internal representation for layout logic.
- **Granular Version Control:** The current system offers only a basic reset function. We propose implementing a fine-grained state management system that tracks the project history as a traversable tree. This would allow users to undo specific actions, branch off experimental features, and navigate arbitrarily between project states, mirroring professional Git-based workflows.
- **Cost-Performance Optimization:** To address the high computational costs of continuous agentic reasoning, we will investigate model cascading techniques. By delegating routine

tasks (e.g., CSS tweaks) to smaller, cheaper models and reserving state-of-the-art LLMs for complex architectural planning, we can significantly improve the price-performance ratio for real-world deployment.

References

[1] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, and many others. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*, 2021.

[2] Raymond Li, Leandro von Werra, Thomas Wolf, and colleagues. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[3] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629*, 2022.

[4] Pengcheng Yin and Graham Neubig. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2017.

[5] Sumit Gulwani. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of POPL*, pages 317–330, 2011.