Synopsis on

Vibe-It: An AI-Powered Platform for Prompt-to-Project Generation

Submitted in the partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

in

Computer Science & Engineering (IoT & Intelligent Systems)

2023-2027

By

Vranda Garg (23FE10CII00062)

Ayush Sharma (23FE10CII00062)

MANIPAL UNIVERSITY JAIPUR

Under the guidance of

Prof. Govind Chhimpa

School of Computer Science and Engineering

Department of IoT & Intelligent Systems

Manipal University Jaipur

Jaipur, Rajasthan

Jan-May 2026

Introduction

Large Language Models (LLMs) have fundamentally changed how developers interact with code. Models like GPT-5.2, Claude, DeepSeek, and LLaMA are now capable of generating syntactically correct, logically coherent code across multiple languages and frameworks when given a well-structured natural language prompt. This has given rise to a new paradigm in software development commonly referred to as "vibe coding" - where the developer describes what they want in plain English and the AI writes the implementation. Tools like GitHub Copilot, Cursor, and ChatGPT have already proven that LLMs can serve as effective coding assistants, but they still operate within the context of an existing IDE and assume the user knows how to set up a development environment, manage dependencies, and run their project locally.

Vibe It takes this a step further. Instead of assisting a developer inside their editor, Vibe It is a full-stack platform where the user provides a single prompt - for example, "Build me a todo app with authentication and dark mode" - and the platform generates the entire React.js project from scratch, streams the code to the user's screen in real time, and provides a live running preview of the application, all without the user touching a terminal, installing Node.js, or configuring a single file.

The system operates on a client-server architecture. The frontend is a web-based interface where users input their prompts, view generated files in a code editor, browse the file tree, and interact with a live preview panel. The backend runs on a VPS and handles all AI interactions. When a prompt is received, it is sent to an LLM through the OpenRouter API using a streaming SDK. The model does not just return raw text - it uses structured tool calls to perform actual file operations: writing new files, reading existing ones, updating specific sections, and deleting files when necessary. This means the AI is not simply generating text output; it is actively constructing a real project file structure on the server.

Once the files are generated, the user can run their project. At this point, the platform spins up an isolated Docker container for that specific project, installs the necessary dependencies, builds and serves the application, and exposes it on a unique URL. This URL is then rendered inside an iframe on the user's interface, giving them a live, interactive preview of their running application. Each project gets its own container, ensuring complete isolation - one user's project cannot interfere with another's, and the execution environment closely mirrors a real production setup.

The platform also includes a project dashboard. Every project a user generates is saved and accessible from this dashboard. To make the dashboard visually informative, Puppeteer is used on the backend to take automated screenshots of each running project. These screenshots serve as thumbnail previews, allowing users to quickly identify and revisit their projects. Users can also iterate on existing projects by providing follow-up prompts, enabling a conversational development workflow where the AI incrementally modifies and improves the codebase based on user feedback.

In essence, Vibe It compresses the entire development pipeline - from ideation to code generation to execution to preview - into a single platform accessible through a browser. The user's only input is a natural language prompt; everything else is handled by the system.

Motivation

The motivation behind Vibe It comes from a very specific problem that anyone who has tried to build a web project from scratch understands: the sheer amount of boilerplate and setup required before you can even start working on your actual idea. To build a simple React application, a developer needs to initialize the project with a package manager, configure the build tool, set up routing, create the component hierarchy, write CSS or configure a styling framework, set up state management, handle API calls, and then figure out how to deploy and serve the application. For an experienced developer, this process takes hours. For a beginner or non-developer, it is an insurmountable barrier.

AI code generation has made significant progress in reducing this friction. Developers can now ask ChatGPT or Copilot to generate individual components, utility functions, or API handlers. But the gap between generating a single component and generating a fully functional, properly structured, runnable project is massive. A working React application is not just a collection of code snippets - it requires a coherent file structure, proper imports and exports across files, consistent state management, correct routing configuration, and a build pipeline that ties everything together. Current AI assistants do not handle this end-to-end orchestration because they operate at the file or snippet level, not at the project level.

Platforms like Lovable and Bolt have started addressing this by offering prompt-to-project generation. However, after working with these tools, several critical limitations become apparent. First, most of these platforms do not stream the generation process in real time - the user submits a prompt and waits for the output, with no visibility into what is being generated or why. This makes it difficult to catch issues early or understand the AI's approach. Second, the execution environments are often shared or sandboxed in ways that do not accurately represent how the application would behave in production. Without proper container isolation, projects can encounter dependency conflicts, port collisions, and inconsistent runtime behavior. Third, the file management capabilities are limited - the AI can generate files, but updating specific sections of existing files, deleting unnecessary files, or restructuring the project based on follow-up prompts is either poorly supported or not supported at all.

Beyond solving technical limitations, the platform is motivated by a practical need: enabling rapid prototyping. Startups need to validate ideas quickly, students need to build projects for learning and submissions, freelancers need to produce MVPs for clients, and designers need functional prototypes to present to stakeholders. In all of these cases, the bottleneck is not the idea itself but the implementation effort required to turn that idea into something tangible and interactive. Vibe It removes this bottleneck by reducing the implementation effort to a single prompt, making it possible to go from concept to running application in minutes rather than days.

Statement of Problem

Building a web application - even a simple one - requires a developer to work across multiple layers of the stack. The frontend needs to be implemented in a framework like React with proper component architecture, routing, and state management. Styling needs to be applied using CSS, Tailwind, or a UI library. If the application requires data persistence, a backend or database needs to be configured. The project needs to be bundled, served, and made accessible through a URL. And throughout this process, the developer needs to manage dependencies, handle configuration files, debug runtime errors, and maintain a consistent project structure. Each of these steps introduces friction and requires specialized knowledge.

Existing tools address fragments of this problem but none of them provide a complete, integrated solution. The following table summarizes the capabilities and shortcomings of current approaches:

Existing Method	Pros	Cons
Traditional Manual Development (VS Code, terminal, npm/yarn)	Full control over every aspect of the project; no limits on what can be built; developer owns the entire codebase; works with any framework, library, or tool	Requires deep knowledge of the framework, build tools, package managers, and deployment; extremely time-consuming for simple projects; high setup overhead even for basic applications; not accessible to non-developers at all
No-Code Platforms (Wix, Bubble, Webflow)	Zero programming knowledge required; drag-and-drop interface for quick layouts; built-in hosting; suitable for landing pages and simple CRUD apps	Generated code is proprietary and inaccessible; heavy vendor lock-in; cannot build anything beyond the platform's predefined components; poor performance and scalability; not suitable for custom logic or complex interactions
AI Code Assistants (GitHub Copilot, Cursor, Codeium)	Context-aware code suggestions directly in the editor; supports multiple languages; significantly speeds up writing boilerplate; learns from project context	Requires an existing development environment and project setup; generates code at the file or function level, not at the project level; no built-in execution or preview; user must manually integrate, test, and run the generated code; useless without programming knowledge

Vibe It addresses these gaps by building a platform that operates at the project level rather than the file level. The AI does not just generate code - it constructs a complete file system using structured tool calls, giving it the ability to create, read, modify, and delete files as needed. The generation process is streamed token-by-token to the user's browser, providing full transparency. Each project runs in its own Docker container, ensuring clean isolation and accurate preview behavior. And the iterative prompt-response workflow allows users to refine and evolve their projects through natural conversation, just as they would if they were pair-programming with

another developer. The result is a platform that compresses the entire development lifecycle into a browser-based interface driven entirely by natural language.

Methodology / Planning of Work

The development of Vibe It will be carried out in six clearly defined phases, each building on the deliverables of the previous one.

Phase 1 – Research and System Design: This phase involves studying the architectures of existing AI project generation platforms to understand their design decisions and identify their technical limitations. The OpenRouter API documentation, Vercel AI SDK streaming capabilities, and Docker containerization patterns will be studied in detail. Based on this research, a complete system architecture will be designed covering the following: the client-side application structure (file editor, file tree, prompt input, live preview iframe), the server-side AI pipeline (prompt processing, model selection, streaming response handling, tool call execution), the file system layer (how generated files are stored, organized, and served to the client), and the container orchestration layer (how Docker containers are created, managed, started, and stopped for each project). Database schema design will also be completed in this phase, defining how users, projects, files, and prompt histories will be stored and related.

Phase 2 – Backend Development and AI Pipeline: The backend server will be developed and deployed on a VPS running a Linux-based environment. The core of the backend is the AI pipeline, which works as follows: the user's prompt is received via an API endpoint, enriched with system-level instructions that define the AI's behavior (e.g., "You are a React developer. Generate a complete project based on the user's description. Use tool calls to create files."), and sent to the LLM through the OpenRouter API. The response is streamed back using the Vercel AI SDK's streaming utilities, which handle chunked transfer encoding and server-sent events to deliver tokens to the client in real time. The AI model is configured with a set of tool definitions - writeFile, readFile, updateFile, deleteFile - each with a defined schema specifying the required parameters (file path, content, etc.). When the model invokes a tool call, the backend intercepts it, executes the corresponding file system operation, and returns the result to the model so it can continue generating. This tool-call-based approach is critical because it means the AI is not just outputting text - it is performing structured, validated operations on the actual project file system.

Phase 3 – Frontend Development and Streaming Interface: The frontend will be built as a modern web application providing four primary panels: a prompt input area where users type their project descriptions and follow-up instructions, a file explorer that displays the project's directory structure and updates in real time as new files are created, a code editor that shows the contents of the currently selected file with syntax highlighting, and a preview panel containing an iframe that renders the running project. The streaming integration is a key technical challenge - as tokens arrive from the backend via server-sent events, the frontend must parse them, determine whether they represent file content, tool call invocations, or conversational text, and update the appropriate UI panel accordingly. The interface will also support follow-up prompts, where the user can ask the AI to modify existing files, add new features, fix bugs, or restructure

the project, and the changes will be reflected in real time across the file explorer, code editor, and preview.

Phase 4 – Docker Container Orchestration: This phase focuses on the execution layer. Each project will be mapped to its own Docker container. When a user clicks "Run" or "Preview," the backend will programmatically create a Docker container using the Docker Engine API, mount the project's file directory into the container, install the required dependencies (npm install), start the development server, and expose the application on a dynamically assigned port. A reverse proxy will map this port to a unique subdomain or URL path, which is then sent to the frontend and loaded into the preview iframe. Container lifecycle management is critical here - containers must be started when needed, stopped after a period of inactivity to conserve resources, and cleaned up when projects are deleted. Resource limits (CPU, memory) will be applied to each container to prevent any single project from consuming excessive server resources.

Phase 5 – Project Dashboard and Thumbnail Generation: A project management dashboard will be developed where users can view all their generated projects in a grid or list layout. To generate visual thumbnails, a headless browser instance using Puppeteer will be run on the backend. After a project's container is started and the application is live, Puppeteer will navigate to the project's URL, wait for the page to render, capture a screenshot, and store it as the project's thumbnail image. This process will be triggered automatically after each successful project generation or significant update. The dashboard will display these thumbnails alongside project metadata such as the project name, creation date, last modified date, and the original prompt used to generate it.

Phase 6 – Database Integration, Testing, and Deployment: All persistent data - user accounts, project metadata, file records, prompt histories, and thumbnail references - will be stored in a database (PostgreSQL or Appwrite, to be finalized based on performance benchmarking). The complete system will undergo thorough testing at multiple levels: unit tests for individual backend functions (tool call handlers, file operations), integration tests for the AI pipeline (prompt-to-file generation flow), end-to-end tests for the complete user workflow (prompt input → code generation → file creation → container execution → live preview), and load testing to evaluate how the system performs under concurrent user sessions with multiple active containers. After testing and optimization, the final platform will be deployed on the VPS with proper logging, monitoring, and error handling in place.

Facilities Required for Proposed Work

Software Requirements:

- Operating System: Ubuntu 22.04 LTS (server environment)
- Frontend Framework: Next.js (React-based) with TypeScript
- Backend Runtime: Node.js with Express or Hono
- AI Integration: OpenRouter API for LLM access, Vercel AI SDK for streaming and tool call handling
- Containerization: Docker Engine and Docker API for per-project container management

- Database: PostgreSQL or Appwrite for persistent storage of users, projects, files, and prompt histories
- Screenshot Automation: Puppeteer (headless Chromium) for generating project thumbnails
- Reverse Proxy: Nginx or Traefik for routing container URLs
- Version Control: Git and GitHub for source code management
- Development Tools: VS Code, Postman (API testing), Docker Desktop (local development)

Hardware Requirements:

- Virtual Private Server (VPS): Minimum 8 GB RAM, 4 vCPUs, 100 GB SSD storage (to accommodate multiple concurrent Docker containers, the AI processing pipeline, and the database)
- Client Machine: Any modern computer or laptop with a web browser (Chrome, Firefox, Edge) and a stable internet connection
- Network: Stable server-side internet connectivity with sufficient bandwidth for real-time streaming to multiple concurrent users

Bibliography / References

[1] Vercel Inc., "Next.js Documentation," [Online]. Available: https://nextjs.org/docs.

[2] Vercel Inc., "AI SDK Documentation - Streaming, Tool Calls, and Agents," [Online]. Available: https://ai-sdk.dev/docs/introduction.

[3] OpenRouter, "OpenRouter API Documentation - Quickstart, Authentication, and Streaming," [Online]. Available: https://openrouter.ai/docs/quickstart.

[4] PostgreSQL Global Development Group, "PostgreSQL 18 Documentation," [Online]. Available: https://www.postgresql.org/docs/current/index.html.

[5] Docker Inc., "Docker Engine Documentation," [Online]. Available: https://docs.docker.com/engine/.

[6] DigitalOcean, "How to Install and Use Docker on Ubuntu 22.04," [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-22-04.

[7] DigitalOcean, "How To Set Up a Node.js Application for Production on Ubuntu 22.04," [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-set-up-a-node-js-application-for-production-on-ubuntu-22-04