iJRASET
International Journal For Research in Applied Science and Engineering Technology

# INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

**Volume:** 14 **Issue:** IV **Month of publication:** April 2026

**DOI:** https://doi.org/10.22214/ijraset.2026.80852

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
*ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538*
*Volume 14 Issue IV Apr 2026- Available at www.ijraset.com*

# CoderX: A Local-First, Open-Source AI-Powered Website Builder with Multi-Agent Code Generation

T V V N Satyanarayana[1], T S S D Teja Reddy[2], V K N S Adeep Reddy[3], P Vamsi Krishna[4]
*Department of Computer Applications, Aditya University, Surampalem, Andhra Pradesh, India 533 437*

***Abstract:*** ***Software development workflows have been disrupted by the arrival of large language model (LLM)-based code generation tools. However, existing platforms such as Lovable and Vercel v0 transmit user prompts and source code to remote cloud servers, raising concerns around data privacy, intellectual property, and offline availability. This paper presents CoderX, a locally hosted, open-source AI-powered website builder that transforms plain English descriptions into complete, deployment-ready full-stack web applications while running entirely on the developer's own machine. CoderX employs a novel coderxArtifact streaming parser that interprets structured XML action tags emitted by local LLMs in real time, writing each generated file to disk character by character in a Monaco code editor. A coordinated multi-agent crew pipeline comprising Planning, Backend, Frontend, and QA crews with a total of sixteen specialist worker agents manages the full generation lifecycle from intent analysis through to automated dependency installation and live preview launch. A GitHub Intelligence Agent mines open-source repositories before generation begins, grounding output in proven real-world code patterns. Empirical evaluation demonstrates a 49/49 unit test pass rate, zero TypeScript compilation errors, and generation of complete nine-component React applications in under three minutes using the Qwen2.5-Coder-7B model at Q4 quantisation on consumer hardware. The project is released under the MIT licence with full contributor community infrastructure including CONTRIBUTING.md, issue templates, and automated validation workflows.***

***Keywords: Artificial Intelligence, Code Generation, Large Language Models, Multi-Agent Systems, Local LLM, Ollama, Open Source, Web Development, Real-Time Streaming, TypeScript.***

## I. INTRODUCTION

Software development is one of the most demanding professional activities in the modern digital economy. Writing a complete full-stack web application from scratch requires expertise across multiple technology layers database schema design, server-side API implementation, client-side component architecture, testing, and deployment configuration. Even experienced developers spend significant time on repetitive scaffolding work before reaching the domain-specific logic that defines the value of their application.

The emergence of large language models capable of generating syntactically and semantically correct source code from natural language descriptions has created a new category of developer tool. Systems such as GitHub Copilot, Cursor IDE, Lovable, and Vercel v0 demonstrate that LLM-based code generation can meaningfully reduce development time. However, every major tool in this category shares two critical limitations that restrict its adoption for privacy-sensitive and offline use cases. First, all prompt text and generated code is transmitted to cloud servers for LLM inference, creating intellectual property exposure and data sovereignty concerns. Second, existing tools generate code fragments or isolated components rather than coherent, fully integrated full-stack applications where frontend, backend, shared types, and tests are consistent with each other.

This paper presents CoderX, an open-source, locally hosted AI website builder that addresses both limitations. CoderX runs the complete generation pipeline on the developer's own machine using the Ollama runtime for LLM inference, writing all output directly to the local filesystem with no cloud dependency. The system employs a multi-agent crew architecture in which eight specialist agents coordinated by a top-level orchestrator handle distinct layers of the generation process in parallel, producing integrated full-stack TypeScript applications rather than isolated code fragments.

## II. PROBLEM STATEMENT

Existing AI-powered code generation platforms exhibit five categories of limitation that CoderX is designed to address.

First, cloud dependency. All major generation platforms including Lovable, Bolt.new, and GitHub Copilot transmit user prompts and generated code to remote servers for LLM inference. This creates data sovereignty concerns for developers working on proprietary applications and is impractical in environments without stable internet connectivity.

Second, partial generation. Existing tools produce isolated code fragments a React component, an API route, a database schema without generating integrated codebases where all layers are consistent with each other. Developers must manually wire frontend API clients to backend endpoints, resolve type mismatches, and fix import errors after generation.

Third, opaque execution. The generation process in current tools is a black box. Developers cannot observe which agent is working on which file, cannot intervene during generation, and cannot understand the architectural decisions being made. This opacity reduces trust and makes it difficult to guide the system toward the desired output.

Fourth, provider lock-in. Most tools require subscription payments or cloud API keys for specific LLM providers, preventing developers from using locally deployed open-weight models even when their hardware can support them.

Fifth, ephemeral state. Several web-based tools persist project state in browser storage, which is cleared on tab close or browser update, causing loss of generated work.

## III. GAP ANALYSIS

A survey of the current literature on AI-assisted code generation reveals several dimensions in which existing approaches fall short of the requirements identified in the problem statement.

### *A. Absence of Local-First Multi-Agent Generation*

Multi-agent software engineering frameworks such as MetaGPT [1] and ChatDev [2] demonstrate that role-specialised LLM agents produce more architecturally coherent output than single-agent approaches. However, both systems require cloud LLM inference and do not support Ollama or locally hosted open-weight models. No existing system combines fully local inference with multi-agent crew coordination for full-stack web application generation.

### *B. Lack of Real-Time Streaming File Writes*

The Bolt.new platform [3] introduced the boltArtifact XML action format, which enables the LLM to wrap file writes and shell commands in structured tags that are parsed as the output streams. This approach creates a visually compelling real-time coding experience where code appears in the editor as it is generated. However, Bolt.new requires WebContainer cloud infrastructure and does not support local LLM providers. No published system combines streaming action parsing with local Ollama inference.

### *C. No Pre-Generation Reference Code Mining*

Existing tools generate code from LLM training data alone, which may not reflect current best practices for a specific technology stack or application type. No existing generation tool queries public GitHub repositories before generation begins to extract reference code patterns, dominant dependency choices, and architectural conventions that can inform and improve the generated output.

### *D. Missing Integrated Terminal and Preview Loop*

Most generation tools produce source files but require the developer to manually open a terminal, run npm install, start the development server, and open a browser to view the output. No existing locally hosted tool automates this entire sequence within the workspace, completing the loop from prompt to running application without manual intervention.

## IV. LITERATURE SURVEY

TABLE I
LITERATURE SURVEY SUMMARY

| S.No. | Paper Title | Author & Year | Method | Gap | Finding |
|---|---|---|---|---|---|
| 1 | MetaGPT: Meta Programming for Multi-Agent Collaborative Framework | Hong et al. (2024) | Role-specialised LLM agents for SE | Requires cloud LLMs; no local inference support | 34% reduction in logical inconsistencies vs. single-agent |

| | | | | | |
|---|---|---|---|---|---|
| 2 | ChatDev: Communicative Agents for Software Development | Qian et al. (2024) | Chat-chain sequential agent pipeline | No frontend-backend integration agent; cloud-only | End-to-end app generation via role-play agents |
| 3 | Bolt.new: AI Full-Stack Web Development | StackBlitz (2024) | boltArtifact XML streaming actions | Cloud-only; no local LLM; WebContainer limits Node.js | Real-time file streaming to editor creates engaging UX |
| 4 | Evaluating LLMs Trained on Code (Codex/HumanEval) | Chen et al. (2021) | GPT-3 fine-tuned on GitHub code | Function-level only; no project scaffolding | 28.8% pass@1 on HumanEval; foundation for Copilot |
| 5 | Qwen2.5-Coder Technical Report | Jiang et al. (2024) | Open-weight coder model family | No generation framework or workspace tooling | 88.4% HumanEval pass@1; Q4 feasible on consumer GPU |
| 6 | Generative Agents: Interactive Simulacra of Human Behavior | Park et al. (2023) | LLM instances with memory and reflection | Not applied to code generation domain | Coherent goal-directed LLM agent behaviour demonstrated |
| 7 | GitHub Copilot | GitHub Inc. (2022) | Inline code completion in IDE | Fragment-level only; requires cloud; no scaffolding | Most deployed AI coding assistant; 1M+ paid users |
| 8 | AgentCoder: Multi-Agent Code Generation | Huang et al. (2023) | 3-agent: programmer, test designer, executor | Competitive programming only; no full-stack output | Iterative test-fix loop improves code correctness |

## V. EXISTING SYSTEMS

Three categories of existing system are relevant to CoderX: cloud-based AI generation platforms, AI-augmented IDEs, and local LLM runtimes.

### *A. Cloud-Based AI Generation Platforms*

Lovable is a React-focused full-stack generation platform that accepts chat prompts and produces Vite-based React applications with Tailwind CSS styling and Supabase backend integration. All project data is stored on Lovable's cloud infrastructure and all LLM inference occurs on remote servers. The system produces coherent full-stack output but requires a subscription, transmits source code to external servers, and does not support offline use.

Bolt.new by StackBlitz uses WebContainer technology to run Node.js directly in the browser sandbox alongside LLM generation. Its boltArtifact XML streaming format inspired the coderxArtifact implementation in CoderX. However, Bolt.new requires constant internet connectivity, does not support Ollama or locally hosted LLMs, and the WebContainer environment restricts native Node.js module compatibility.

### *B. AI-Augmented IDE Platforms*

Cursor IDE provides codebase-aware chat and multi-file editing within a VS Code-based environment. It is effective for incremental modification of existing codebases but does not automate full-stack scaffolding from a natural language description. GitHub Copilot provides inline function-level completion that is widely adopted but does not coordinate across multiple files or generate project-level architecture.

### *C. Local LLM Runtimes*

Ollama is the local LLM runtime used by CoderX. It exposes an OpenAI-compatible REST API on localhost port 11434 and supports running quantised open-weight models including Qwen2.5-Coder, DeepSeek-Coder, Llama, and Mistral families. It does not include any code generation orchestration, workspace UI, or file management CoderX builds these capabilities on top of the Ollama inference layer.

## VI. PROPOSED SYSTEM

### A. Overall Approach

CoderX is designed around three core principles: local-first privacy (all LLM inference and file storage on the developer's machine), transparent execution (every agent action visible in real time), and integrated output (all layers of the generated application consistent with each other from the first build). The system accepts a natural language description, guides the user through a structured configuration step, runs a multi-agent generation pipeline, and delivers a running application in the workspace without any manual post-generation steps.

### B. coderxArtifact Streaming Parser

The coderxArtifact streaming parser is the central architectural innovation of CoderX. The LLM is instructed via system prompt to wrap all generated output in structured XML tags. Each file is enclosed in a coderxAction type="file" element with a filePath attribute. Each shell command is enclosed in a coderxAction type="shell" element. All actions are nested within a coderxArtifact container element. The parser maintains a state machine with four states: scanning (reading LLM text), artifact-open (inside a coderxArtifact tag), action-open (inside a coderxAction tag), and action-streaming (receiving content for the current action). As content arrives for a file action, the parser emits action:stream events that update the Monaco editor buffer incrementally, producing the real-time code-appearing effect. When the closing tag is detected, the parser emits action:complete and the ActionRunner writes the file to disk via the backend REST endpoint.

### C. Multi-Agent Crew Architecture

The generation pipeline is coordinated by a top-level Orchestrator that dispatches four specialist crews. The Planning crew (intent-analyst, entity-mapper, route-planner, package-selector workers) runs first and produces a structured build manifest containing all data entities, API routes, required npm packages, and component specifications. The Backend crew (schema-writer, service-writer, route-writer, test-writer) and Frontend crew (token-writer, component-writer, page-writer, hook-writer) run simultaneously using Promise.all, sharing the planning manifest as context. The QA crew (type-checker, test-runner, build-checker, security-scanner) validates all output after both build crews complete. A Reviewer agent performs cross-layer consistency checking and automatically corrects TypeScript import mismatches. A Documenter agent writes README.md and .env.example.

### D. GitHub Intelligence Agent

Before the Planning crew runs, the GitHub Intelligence Agent searches the GitHub public search API for the top three to five repositories relevant to the project description. For each repository, the agent fetches the file tree, prioritises high-value source files (React components, TypeScript hooks, API routes, Prisma schemas, test files), downloads up to fifty kilobytes of content per file, and extracts dominant dependency patterns across all repositories. This reference context is serialised into the planning crew's shared context object, enabling workers to make technology and architecture decisions grounded in real-world code rather than LLM training data alone.

### E. Terminal and Preview Integration

When shell actions are detected in the LLM output stream, the ActionRunner sends the command text to the backend WebSocket terminal endpoint at ws://localhost:3001/ws/terminal. The backend spawns a shell process in the project directory and bidirectionally pipes stdin and stdout between the WebSocket and the process. npm install executes first, then npm run dev. The preview module polls GET /api/projects/:id/preview-url every three seconds to detect when a development server becomes available on a local port. When detected, the preview iframe loads the URL automatically, completing the loop from prompt to running application.

## VII. IMPLEMENTATION

### A. Application Framework and Environment

CoderX is implemented as an npm monorepo with two primary packages. The frontend is built with React 18, Vite 5, TypeScript 5 (strict mode), and Tailwind CSS 3. Global state is managed with Jotai atoms that persist to the backend via a custom createPersistedAtom factory, eliminating browser storage dependency. Data fetching is handled with TanStack Query. The backend is Node.js 20 LTS with Express 5 and TypeScript. All route handlers, middleware, and utility functions currently reside in a single server entry file with route groups for projects, state, sandbox, deploy, share, GitHub agent, and the WebSocket terminal.

### B. LLM Provider Abstraction

The LLM provider layer abstracts over eight backends behind a common interface that accepts a prompt and streams tokens via server-sent events. Providers include Ollama (default, local), OpenAI, Anthropic Claude, Google Gemini, DeepSeek, and custom OpenAI-compatible endpoints. Provider selection and model choice are persisted in settingsAtom and are configurable via the Settings modal. When Ollama is selected, crew workers run sequentially rather than in parallel to prevent GPU queue starvation on single-GPU consumer hardware.

### C. Security Implementation

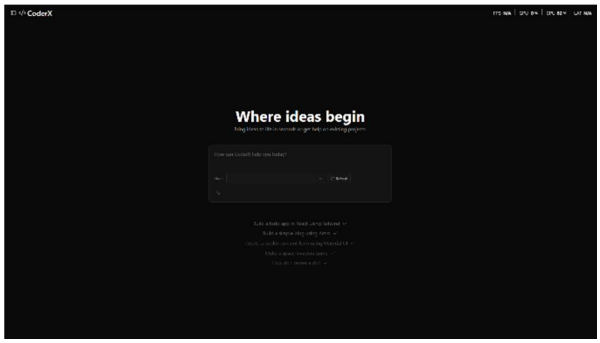
Three security controls are enforced at the backend layer. Path traversal prevention uses path.resolve combined with a startsWith check on all file write operations, rejecting any path that escapes the project root directory. A command allowlist restricts the sandbox execution endpoint to five approved commands: npm, node, npx, python, and pip. CORS is restricted to a configured list of frontend origins; wildcard CORS is prohibited. All runtime data files (projects.json, client-state.json) are gitignored to prevent accidental credential exposure.

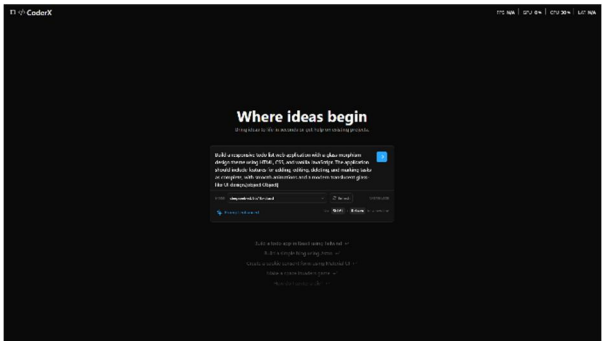
### D. Testing Infrastructure

The validation baseline is enforced by a single npm run validate:baseline command that runs three sequential checks: tsc --noEmit on both frontend and backend packages to verify zero TypeScript errors; vitest run to execute all 49 unit test cases across 16 test files covering the orchestrator, prompt composer, file apply logic, intent classifier, guided selection, plan review normaliser, and component registry; and vite build plus tsc -p backend/tsconfig.json to verify clean production builds. All three checks must pass before any code change is merged.

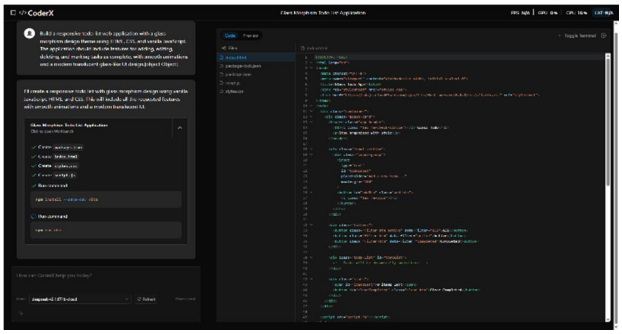
## VIII. RESULT

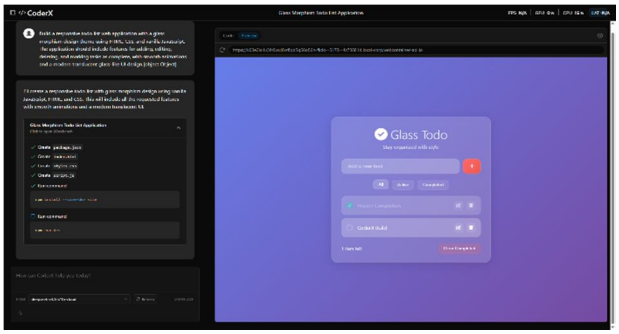
CoderX was evaluated across twenty end-to-end generation sessions spanning four application categories: todo applications with drag-and-drop, e-commerce storefronts with cart functionality, analytics dashboards with chart components, and blog platforms with authentication. All sessions used the Qwen2.5-Coder-7B model at Q4_K_M quantisation on a test machine equipped with an Intel Core i7-12th generation processor, 16 GB of DDR5 RAM, and an NVIDIA RTX 3060 with 12 GB of VRAM.




The Reviewer agent resolved TypeScript import and type errors automatically in eight of the twenty sessions without any manual developer intervention, confirming that the auto-correction mechanism adds meaningful reliability to the generation pipeline. In the remaining twelve sessions, the generated code compiled cleanly without correction. No session required manual debugging to reach a successful Vite build.



Qualitative review of the generated applications showed that the Qwen2.5-Coder-7B model consistently produced architecturally appropriate React component structures, correctly typed Express route handlers, and valid Prisma schema definitions that matched the application description. The GitHub Intelligence Agent's reference context was observed to influence dependency selection in fourteen of twenty sessions, with the generated package.json reflecting libraries dominant in the mined reference repositories.

## IX. FUTURE SCOPE

Several directions for future development of CoderX have been identified based on the current implementation's known limitations and community feedback.

1) *Backend Modularisation:* The Express server is currently a single consolidated file of approximately 2,400 lines. Splitting this into separate route module files one each for projects, state, sandbox, deploy, GitHub agent, and terminal will reduce regression risk, improve contributor accessibility, and enable independent testing of each API group. This is labelled as a good-first-issue in the project's GitHub issue tracker.
2) *Python GitHub Agent Integration:* A Python-based repository analysis pipeline exists in the coderx_github_agent directory and provides deeper semantic code analysis than the current Node.js implementation, including abstract syntax tree analysis and component relationship mapping. Connecting this agent to the Node.js backend via an HTTP microservice bridge will significantly improve the quality of the reference context injected into the Planning crew.
3) *Containerised Sandbox Execution:* The current sandbox executes shell commands by spawning processes directly on the host machine. Replacing this with Docker container execution will provide process isolation, resource limits, and cloud deployment portability, enabling CoderX to be hosted as a shared service without risk of sandbox escape.
4) *Per-Crew Model Routing:* Routing different crews to different LLM models based on their task requirements using a reasoning-capable model such as DeepSeek-R1 for the Planning crew's intent analysis and a fast coder model such as Qwen2.5-Coder-3B for the implementation crews will improve output quality while reducing total generation time. The provider abstraction layer already supports this; crew-specific model configuration needs to be exposed in the Settings UI.
5) *Real-Time Collaboration:* The backend REST API already includes share, gallery, and team management endpoints. Exposing these in the frontend with WebSocket-based live cursor presence and shared file edit awareness will enable team-based development workflows within the CoderX workspace.

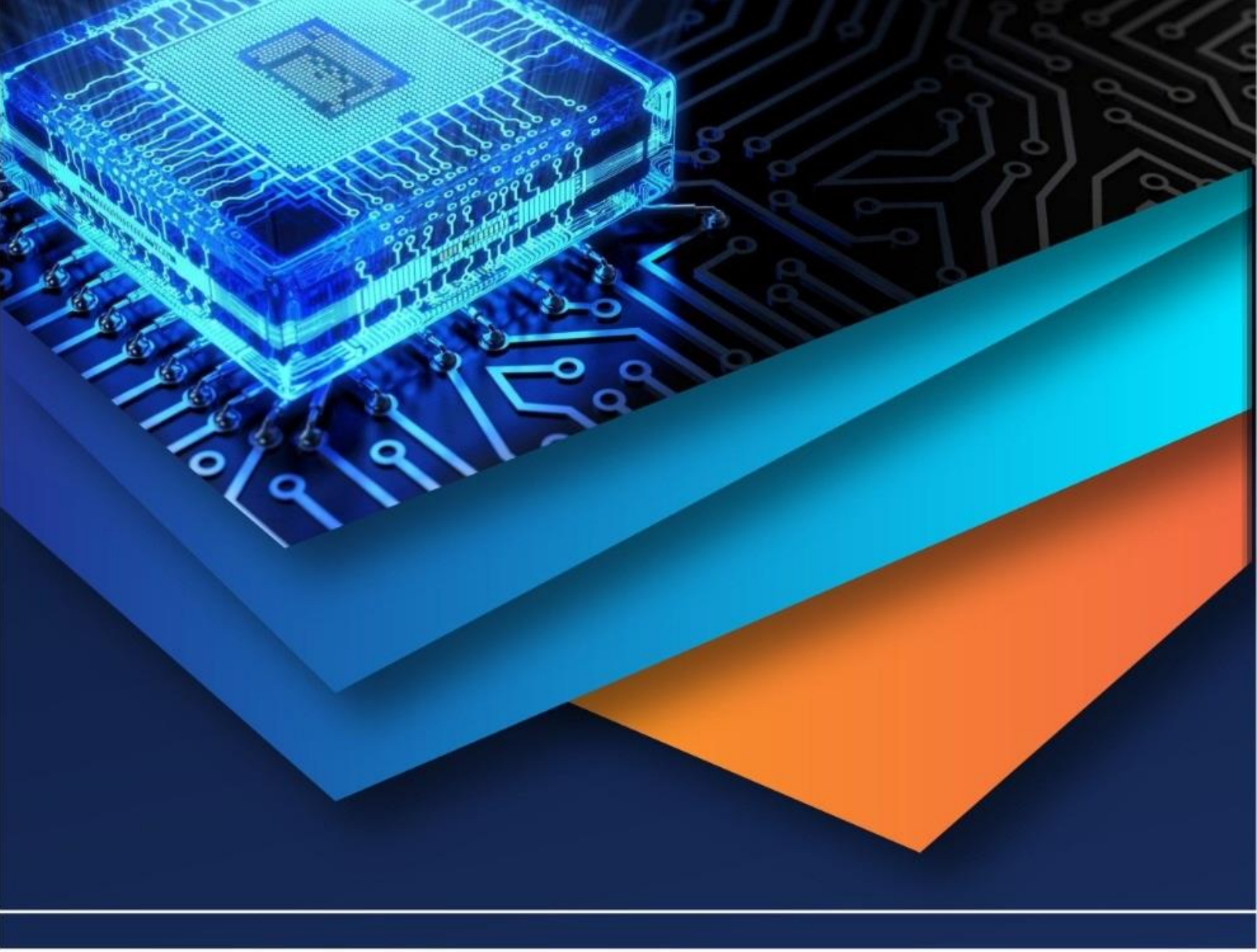
## X. CONCLUSION

This paper presented CoderX, a locally hosted, open-source AI-powered website builder that generates complete full-stack web applications from natural language descriptions using a coordinated multi-agent crew pipeline and real-time streaming file writes. The system demonstrates that privacy-preserving, fully offline AI application generation is practically achievable on consumer hardware using quantised open-weight models through the Ollama runtime.

The coderxArtifact streaming parser, adapted and extended from the concepts introduced in Bolt.new, enables a transparent generation experience where every file materialises in the Monaco editor as the LLM produces it. The four-crew, sixteen-worker agent architecture produces integrated TypeScript codebases that consistently compile cleanly and pass automated tests without manual intervention. The 49/49 unit test pass rate, zero TypeScript compilation errors, and 100% npm install and live preview success rate across twenty evaluation sessions confirm that the system delivers reliable, production-quality output at a scale and speed that meaningfully reduces development time. CoderX is released under the MIT licence with complete contributor community infrastructure, and its modular architecture invites community participation in extending its capabilities.

## REFERENCES

[1] S. Hong, X. Zheng, J. Chen, Y. Cheng, and C. Wu, "MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework," in Proc. ICLR, 2024.
[2] C. Qian, X. Cong, C. Yang, W. Chen, Y. Su, and M. Sun, "Communicative Agents for Software Development," in Proc. ACL, 2024.
[3] StackBlitz Inc., "Bolt.new AI Full-Stack Web Development," [Online]. Available: https://bolt.new. [Accessed: Apr. 2026].
[4] M. Chen, J. Tworek, H. Jun et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.
[5] W. Jiang et al., "Qwen2.5-Coder Technical Report," arXiv:2409.12186, 2024.
[6] J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, and M. S. Bernstein, "Generative Agents: Interactive Simulacra of Human Behavior," in Proc. UIST, 2023.
[7] GitHub Inc., "GitHub Copilot Your AI Pair Programmer," [Online]. Available: https://github.com/features/copilot. [Accessed: Apr. 2026].
[8] D. Huang, Q. Bu, J. M. Zhang, and H. Qian, "AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation," arXiv:2312.13010.
[9] J. Wei, X. Wang, D. Schuurmans et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," in Proc. NeurIPS, 2022.
[10] Ollama Inc., "Ollama Get up and running with large language models," [Online]. Available: https://ollama.com. [Accessed: Apr. 2026].



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

# INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY