FluxEngine: A Graph-Driven Runtime Interpreter for Visual Backend Development

Mr. M. S. Vadagave[1], Rajesh Khandagale[2], Vishwajeet Kamat[3], Aditya Powar[4], Farhan Shaikh[5]
[1]*Faculty, Department of Computer Science and Engineering (Data Science), D Y Patil College of Engineering and Technology, Kolhapur, India*
[2,3,4,5]*Student, Department of Computer Science and Engineering (Data Science), D Y Patil College of Engineering and Technology, Kolhapur, India*

***Abstract*—The growing adoption of low-code development platforms (LCDPs) has revealed a persistent gap between workflow automation tools—which model service integrations but not API semantics—and general-purpose backend frameworks, which demand conventional programming expertise [1, 2]. Existing LCDPs either generate static source code, coupling the visual model to a specific framework and introducing a synchronisation problem on every graph edit, or restrict expressiveness to fixed integration patterns unsuitable for custom business logic [3, 4]. We present FluxEngine, an open-source, self-hostable visual backend development environment that eliminates code generation in favour of a stateful, graph-driven runtime interpreter. Users compose directed acyclic graphs (DAGs) of typed functional nodes on an interactive canvas; the interpreter traverses these graphs at request-time, dynamically registers Express.js HTTP routes, and propagates a structured execution context between nodes via a dot-path expression resolver. A dual-mode telemetry system streams per-node observability data to the canvas over WebSocket in development mode and suppresses all instrumentation in production. Empirical evaluation on a commodity four-core server demonstrates a median per-node interpreter overhead of 3.1 ms and a sustained throughput of 820 req/s for a six-node authentication workflow, establishing runtime interpretation as a viable, low overhead alternative to code generation for visual backend tooling.**

***Index Terms*—low-code development; visual programming; graph execution; runtime interpreter; backend automation; node-based workflow; WebSocket telemetry**

I. INTRODUCTION

The proliferation of API-first product architectures has intensified demand for tooling that enables developers— and increasingly non-developers—to author backend logic without traversing a full software engineering workflow. Low-code development platforms have emerged as a widely studied response to this demand. Rokis and Kirikova [1] identify over forty distinct LCDP categories in the literature, ranging from UI builders to process automation tools, and note that rapid development speed and reduced dependency on specialist programmers are consistently reported benefits. Luo et al. [2] corroborate these findings through practitioner interviews, observing that LCDPs reduce time-to-prototype by an order of magnitude for CRUD-heavy applications while introducing challenges around testability, version control, and extensibility.

A critical limitation of existing platforms is the tension between expressiveness and abstraction. Liu et al. [3] demonstrate empirically that even LLM-augmented low-code environments struggle to support complex business logic that deviates from the platform's built-in node vocabulary. Schenkenfelder et al. [4] observe that end-user development tools built on fixed abstraction layers frequently force practitioners back to conventional code for edge cases, negating the productivity benefit. Mishra et al. [5] catalogue modern web development tooling and identify the absence of a general-purpose visual backend builder—one that supports full HTTP API semantics, authentication, database operations, and external integrations within a single composable graph—as an open problem.

This paper addresses that gap with FluxEngine: a visual backend development environment whose central architectural innovation is a pure runtime interpreter that traverses the workflow graph on every

HTTP request. Unlike code-generating platforms, FluxEngine treats the Blueprint graph as the primary deployable artifact; graph edits take effect on the next request without a build or restart cycle. Unlike integration-automation tools such as n8n, FluxEngine models HTTP routing (method, path, parameters, headers) as first-class concerns and provides a Schema Manager for data model authoring within the same environment.

The primary contributions of this paper are as follows:

- A formal graph model and execution semantics for visual API workflows (Section III).
- The FluxEngine interpreter architecture: lazy-loading node registry, execution-context propagation, dot-path expression resolution, and branch-aware graph traversal (Section IV).
- A dual-mode telemetry system that delivers per-node observability to the development canvas via WebSocket without imposing production overhead (Section V).
- An empirical evaluation of interpreter latency, throughput, and memory across five workflow topologies on commodity hardware (Section VI).
- A structured comparison with code-generation and workflow-automation approaches (Section VII).

II. RELATED WORK

A. Low-Code Development Platforms

The academic literature on LCDPs has grown substantially since 2019. Rokis and Kirikova [1] conduct a systematic literature review of 80 primary studies and conclude that the field lacks a unified taxonomy; they propose a classification along dimensions of target domain, abstraction level, and generation strategy. Luo et al. [2] complement this survey with qualitative data from 17 industrial practitioners and identify four recurring challenges: debugging difficulty, platform lock-in, performance unpredictability, and inadequate support for complex logic. FluxEngine directly addresses the last two by (i) exposing per-node execution telemetry on the canvas and (ii) providing an open extensibility model in which new node types are plain TypeScript files.

Liu et al. [3] conduct an empirical comparison of traditional low-code development against LLM-assisted lowcode development across 120 task instances, finding that LLM support reduces completion time by 34% but increases error rate for tasks that require understanding platform-specific constraints. This finding motivates FluxEngine's design choice to make node semantics explicit and composable rather than inferred.

B. End-User and Participatory Development

Schenkenfelder et al. [4] distinguish between low-code development—targeted at professional developers seeking acceleration—and end-user development, targeted at domain experts without programming backgrounds. They argue that effective tools must support a spectrum of users within a single environment. FluxEngine adopts a role-based access model (admin / dev) and a node vocabulary designed to be legible to developers with limited backend experience, positioning it at the lower end of the low-code spectrum rather than targeting nonprogrammers.

C. Web Development Tooling

Mishra et al. [5] survey contemporary web development tools and frameworks, noting the rapid growth of BaaS (Backend-as-a-Service) providers such as Firebase and Supabase that offer instant database APIs but constrain business logic to cloud functions. They observe that no existing tool combines visual API composition, schema design, and self-hostability within a single open-source package—a gap that FluxEngine fills. Ruiz et al. [6] describe a didactic tool for teaching interactive software systems design that uses a node-graph metaphor, demonstrating the pedagogical value of visual representations for software architecture concepts.

D. Graph-Based Execution Models

Workflow automation platforms such as n8n and Zapier represent the dominant deployment of graph execution for software integration. n8n's open-source architecture, which also walks an in-memory workflow graph at execution time, is the closest prior work to FluxEngine. The key distinctions are: n8n targets event-driven service integration rather than synchronous HTTP APIs; it does not expose HTTP method and path as first-class routing configuration; and it provides no integrated schema designer. Dataflow languages such as KNIME (for data analytics) and Kubeflow (for ML pipelines) share the graph execution model but operate in domains with

homogeneous data types and deterministic scheduling, avoiding the asynchronous I/O and branching control flow that characterise API workflows.

III. FORMAL MODEL

A. Blueprint Graph

A FluxEngine Blueprint B is a four-tuple B = (N, E, T, C) where:

- N is a finite, non-empty set of nodes.
- $E \subseteq N \times N \times L$ is a set of labelled directed edges, where $L = \{\varepsilon\} \cup \{true, false\} \cup S$ and S is a finite set of case strings.
- $T: N \rightarrow \Sigma$ maps each node to a type drawn from the node-type alphabet Σ.
- C: N → Config assigns a (possibly empty) configuration record to each node.

Exactly one node $n_0 \in N$ satisfies $T(n_0) \in$ {trigger.http, trigger.cron, trigger.webhook}; this is the entry point. Nodes satisfying $T(n) \in$ {response.success, response.error, response.redirect} are terminal: the interpreter halts traversal upon executing them. Edges carry a condition label: an unconditional edge (labelled ε) is always traversed; conditional edges are traversed only when the preceding logic node emits a matching branch token in the execution context.

B. Execution Context

The execution context Γ is a mutable record that flows through every node in the graph. It carries: (i) the original HTTP request (method, path, headers, body, params, query)—read-only throughout execution; (ii) a nodes map that accumulates each node's output keyed by node identifier; (iii) optional authentication state populated by auth nodes; (iv) a response handle encapsulating the HTTP response object; and (v) execution metadata (execution-id, workflow-id, mode). Each node executor receives Γ and returns a partial update ΔΓ; the interpreter merges ΔΓ into Γ before proceeding to successor nodes. This design ensures that all prior node outputs are accessible to any downstream node.

C. Expression Resolution

Node configuration values may contain {{path}} expressions where path is a dot-separated accessor into Γ. Resolution proceeds as follows: if the entire configuration string is a single expression, the resolved value preserves its native JavaScript type (enabling structured object forwarding); if the string contains multiple expressions or surrounding literal text, each match is coerced to string and concatenated. This permits both type-safe chaining (e.g., forwarding a db.query result object to a response node) and string interpolation for message templates.

IV. SYSTEM ARCHITECTURE

A deployed FluxEngine instance is packaged as a single Docker image comprising four cooperating subsystems: (1) the Visual Studio, a Next.js single-page application served as a static build directly from the engine's Express server; (2) the Interpreter Engine (Node.js / Express); (3) the Platform Database (SQLite, auto-initialised on first launch); and (4) an optional User Database (PostgreSQL, MySQL, or MongoDB). The following subsections describe the five principal components of the Interpreter Engine.

A. Route Registrar

On startup, the Route Registrar queries the Platform Database for all Blueprint records whose deployed flag is set. For each such record it parses the graph JSON, locates the trigger.http node, and calls app[method](path, handler) on the Express application object, installing a live HTTP handler without restarting the server. The handler constructs an initial execution context Γ_0 from the incoming request and delegates to the Graph Executor. Workflow deployment at runtime is triggered by a REST call to POST /api/workflows/:id/deploy and follows the same registration path. Undeployment splices the corresponding layer from the Express router stack.

B. Node Registry and Lazy Loading

The Node Registry maintains a manifest—a JavaScript object mapping node-type strings to relative file paths— and an in-memory executor cache implemented as a Map<string, NodeExecutor>. When the Graph Executor requests the executor for a node type for the first time, the registry resolves the absolute file path and loads the module via Node.js require (), storing the default export in the cache. Subsequent requests for the same type are O (1) cache lookups. This design ensures that only node types present in deployed workflows consume memory. Adding a new

node type requires one TypeScript file and one manifest entry—no changes to the interpreter.

C. Graph Executor

The Graph Executor pre-processes the Blueprint into an adjacency map (node-id → outgoing edges) for O (1) successor lookup, then executes the graph depth-first from the trigger node. For each node it: (1) retrieves the executor from the registry; (2) resolves all {{}} expressions in the node's configuration against the current Γ; (3) invokes the executor with resolved configuration and Γ, receiving ΔΓ; (4) merges ΔΓ into Γ, recording the output at Γ.nodes[nodeId]; (5) emits a node:output telemetry event in development mode; and (6) determines successor nodes by filtering outgoing edges against the branch token in Γ.nodes[nodeId].__branch. Multiple successors are dispatched concurrently via Promise.all and their results merged.

Error handling is configurable per node via an error Config record. The default terminate policy propagates exceptions to the top-level execute () call, which logs the error and marks the execution as failed. A continue policy records the error in the context and resumes traversal. A retry policy re-invokes the executor up to retryCount times with configurable delay between attempts.

D. Platform Database

FluxEngine uses SQLite as its default Platform Database, auto-initialised on first launch via a migration runner that applies numbered SQL files in lexicographic order. Five tables are maintained: users (bcrypt-hashed credentials, roles), workflows (Blueprint JSON), error_logs (production failure records), settings (key-value pairs), and _migrations (applied filenames). The Platform Database is strictly isolated from the User Database: all platform queries target the SQLite instance; db.* workflow nodes target a separately configured external database connection.

E. Schema Manager

For projects that combine workflow logic with data model design, FluxEngine includes a Schema Manager that accepts an entity-relationship diagram expressed in the Blueprint's schema section and provisions the User Database at deploy time. For SQL targets, the Schema Manager synthesises CREATE TABLE statements via Drizzle ORM; for MongoDB it registers Mongoose model definitions. This allows practitioners to author both the data model and the API logic within a single visual environment.

V. DUAL-MODE TELEMETRY

A key usability requirement for any visual development IDE is that practitioners be able to observe execution in real time without parsing log files or inserting debugging code. FluxEngine addresses this via a dual-mode telemetry system implemented over Socket.IO.

A. Development Mode

In development mode, the Graph Executor emits a structured node:output event after each node completes, carrying the node identifier, node type, execution duration in milliseconds, and the output object. The Visual Studio canvas subscribes to these events and updates each node's visual state: idle → running → success (green border) or error (red border). The configuration panel renders the node's output JSON, enabling developers to inspect intermediate data and compose {{expression}} references for downstream nodes. The full execution log—including branch decisions and timing—is displayed in an execution panel beneath the canvas.

B. Production Mode

In production mode, all telemetry emission is suppressed at the call site within the Graph Executor via a single conditional guard on the execution mode setting. The Socket.IO server remains active for canvas-collaboration events (node:property) but emits no execution data. Failures are persisted to the SQLite error_logs table, capturing node identifier, error message, and the serialised request body. A configurable retention policy (default 30 days) prevents unbounded growth. The mode can be toggled at runtime via POST /api/settings/mode; a mode:changed WebSocket event propagates the transition to all connected canvas clients without a page reload.

VI. EVALUATION

A. Experimental Setup

All experiments were conducted on an Ubuntu 22.04 LTS virtual machine provisioned with 4 vCPUs (Intel

Xeon E5-2686 v4 @ 2.30 GHz) and 8 GB RAM, representative of a low-cost cloud instance. FluxEngine was run from its compiled TypeScript distribution under Node.js v20 LTS in production mode (telemetry disabled). HTTP load was generated with wrk using 8 threads and 64 concurrent connections over 30-second windows. For database-intensive workflows, a local PostgreSQL 16 instance served as the User Database. All latency and throughput figures are the mean of five independent runs.

B. Workflow Topologies

Five workflow topologies of increasing complexity were evaluated, as summarised in Table I.

TABLE I. Workflow Topologies Used in Evaluation

ID	Description	Nodes	Key Operations
W1	Hello World	2	HTTP trigger → JSON response
W2	Validate+Respond	3	+input validation
W3	Auth Flow	5	+JWT verify + role check
W4	Register User	6	+bcrypt hash + DB insert + JWT sign
W5	Conditional Route	5	+if/else branch

C. Interpreter Overhead

Table II reports round-trip p50 latency for each workflow against a bare Express handler returning a static JSON object. Interpreter overhead is decomposed into route lookup (O (1) Map access), graph construction (O(|E|)), and node dispatch (O (1) per node after registry warm-up). The median per-node overhead across W1-W5, excluding I/O operations, is 3.1 ms.

W4's total latency (41.2 ms) is dominated by the PostgreSQL INSERT and bcrypt hash operations (~38 ms combined); the net interpreter cost is 2.8 ms. This confirms that for I/O-bound API workflows, interpreter overhead is negligible relative to external service latency.

TABLE II. Latency Breakdown (p50, ms)

Workflow	p50 Latency	Interpreter Overhead	I/O Contribution
Bare Express	0.8 ms	0 ms	0 ms
W1 (2 nodes)	1.7 ms	0.9 ms	0 ms
W2 (3 nodes)	2.9 ms	2.1 ms	0 ms
W3 (5 nodes)	5.3 ms	4.5 ms	0 ms
W4 (6 nodes)	41.2 ms	2.8 ms	38.4 ms
W5 (5 nodes)	5.1 ms	4.3 ms	0 ms

D. Throughput

Table III reports sustained throughput under full load. W1 achieves 4,210 req/s (61% of bare handler baseline). W4 is bounded by database connection pool concurrency at 820 req/s. These figures are sufficient for small-tomedium backend deployments, which the practitioner literature identifies as the primary adoption context for LCDP tooling [2].

TABLE III. Throughput Under Full Load

Workflow	Throughput (req/s)	vs. Baseline
Bare Express	6,870	100%
W1 (2 nodes)	4,210	61%
W2 (3 nodes)	3,150	46%
W3 (5 nodes)	2,480	36%
W4 (6 nodes)	820	12%
W5 (5 nodes)	2,560	37%

E. Memory Profile

Cold-start RSS after boot was 87 MB for a deployment with three registered workflows. After warming the executor cache across all five topology types (12 unique node types loaded), RSS rose to 112 MB. No measurable growth was observed over a 30-minute sustained load test, confirming the absence of executor cache leaks. Node executor functions for a 30-type

registry would consume approximately 150 KB—negligible on any modern server.

F. WebSocket Telemetry Cost

To quantify the impact of development-mode telemetry, the W3 throughput test was repeated with one connected canvas client and telemetry enabled. Throughput decreased from 2,480 to 2,290 req/s (7.7% reduction) and p50 latency increased from 5.3 to 5.9 ms. This cost is negligible for a development workload; production mode is entirely unaffected.

VII. DISCUSSION

A. Runtime Interpretation vs. Code Generation

The principal argument for code generation in LCDP tooling is performance: a pre-compiled handler incurs no graph traversal cost at request time. Our evaluation demonstrates that the absolute interpreter cost is 2-4 ms per request for realistic workflows, which is below the resolution of standard HTTP SLAs and imperceptible to human users. The advantages of runtime interpretation are: (i) zero-latency deployment updates—graph changes propagate on the next request with no build or restart cycle; (ii) a clean separation between the visual model and the runtime, eliminating the impedance mismatch that makes generated code hard to debug; and (iii) uniform extensibility—new node types require no changes to a code generator. We conclude that for API workflow graphs dominated by I/O operations, the interpreter architecture is preferable on all criteria other than raw throughput on compute-bound tasks, which are not the target workload class.

B. Addressing LCDP Practitioner Challenges

The four challenges identified by Luo et al. [2]—debugging difficulty, platform lock-in, performance unpredictability, and inadequate support for complex logic—are addressed by FluxEngine as follows. Debugging difficulty is mitigated by the real-time per-node telemetry system (Section V). Platform lock-in is avoided through self-hostability (Docker), an open extensibility model, and the absence of proprietary runtime dependencies. Performance unpredictability is addressed by the empirical characterisation in Section VI, which shows that interpreter overhead is bounded and predictable for I/O-bound workloads. Complex logic is supported by a compositional node vocabulary including conditional branching, loops, authentication, and LLM integration.

C. Limitations and Future Work

Three limitations warrant attention. First, the Blueprint graph is re-parsed from JSON on every request; caching the parsed adjacency map would reduce per-request overhead by approximately 0.3 ms for a six-node graph. Second, parallelism is currently exploited only at branch-merge points; supporting concurrent sibling nodes in linear sequences (e.g., two independent DB reads) is left for future work. Third, the node library currently covers approximately 30 types; expansion to message queues, event streaming, and additional LLM providers is ongoing. The LLM integration dimension is particularly promising in light of Liu et al.'s finding [3] that LLM assistance reduces low-code task completion time by 34%, suggesting that an AI-assisted node configuration panel could further reduce the authoring overhead identified by Luo et al. [2].

VIII. CONCLUSION

We have presented FluxEngine, a self-hostable visual backend development environment built around a graphdriven runtime interpreter. By eliminating code generation and treating the Blueprint graph as the primary deployable artifact, FluxEngine achieves zero-latency deployment updates, a clean extensibility model, and realtime per-node observability—properties that directly address the practitioner challenges documented in the low-code development literature [1, 2, 3]. Empirical evaluation on commodity hardware demonstrates a median per-node interpreter overhead of 3.1 ms and sustained throughput of 820-4,210 req/s depending on workflow complexity, meeting the performance requirements of small-to-medium backend deployments. The dual-mode telemetry system provides development-time observability with less than 8% throughput impact and no production cost. FluxEngine is released as open-source software and is intended to serve both as a practical tool and as a reference design for runtime-interpretation approaches to visual backend development.

REFERENCES

[1] K. Rokis and M. Kirikova, "Exploring Low-Code Development: A Comprehensive Literature Review," *Complex Systems Informatics and Modeling Quarterly (CSIMQ)*, no. 36, pp. 68–86, 2023.

[2] Y. Luo, P. Liang, C. Wang, M. Shahin, and J. Zhan, "Characteristics and Challenges of Low-Code Development: The Practitioners' Perspective," in *Proc. 43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 280–289.

[3] Y. Liu, J. Chen, T. Bi, J. Grundy, Y. Wang, J. Yu, T. Chen, Y. Tang, and Z. Zheng, "An Empirical Study on Low-Code Programming Using Traditional vs. Large Language Model Support," *IEEE Transactions on Software Engineering*, vol. 50, no. 2, pp. 312–329, 2024.

[4] B. Schenkenfelder, U. Brandstatter, H. Kirchtag, and M. Wimmer, "Low-Code Development and End-User Development," in *Proc. First International Workshop on Participatory Design and End-User Development (PDEUD 2024)*, CEUR Workshop Proceedings, pp. 1–10, 2024.

[5] P. Mishra, K. K. Rout, and S. R. Salkuti, "Modern Tools and Current Trends in Web Development," *International Research Journal of Engineering and Technology (IRJET)*, vol. 8, no. 11, pp. 1425–1432, 2021.

[6] J. Ruiz, E. Serral, and M. Snoeck, "A Fully Implemented Didactic Tool for the Teaching of Interactive Software Systems," in *Proc. 10th International Conference on Computer Supported Education (CSEDU)*, vol. 1, 2018, pp. 445–452.

[7] R. Wieringa, N. Maiden, N. Mead, and C. Rolland, "Requirements Engineering Paper Classification and Evaluation Criteria: A Proposal and a Discussion," *Requirements Engineering*, vol. 11, no. 1, pp. 102–107, 2006.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Pearson Education, 2006.

[9] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to Build High-Performance Network Programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, Nov. 2010.

[10] wg/wrk, "A Modern HTTP Benchmarking Tool," 2023. [Online]. Available: wrk GitHub Repository

[11] Socket.IO, "Real-Time Bidirectional Event-Based Communication," 2024. [Online]. Available: Socket.IO Official Website

[12] J. Dragovich, "Drizzle ORM: TypeScript ORM for SQL Databases," 2024. [Online]. Available: Drizzle ORM Official Website

[13] R. T. Fielding, "Architectural Styles and the Design of Network-Based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[14] Deissenboeck et al., "Tool Support for Continuous Quality Control," *IEEE Software*, vol. 25, no. 5, pp. 60–67, Sep./Oct. 2008.

[15] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.