



International Journal of Research Publication and Reviews

Journal Homepage: www.ijrpr.com ISSN 2582-7421

An LLM-Based Full-Stack Code Generation and Validation Framework

Gourav¹, Arjoo²

^{1,2} Assistant Professor, Guru Nanak Khalsa College Yamunanagar

gauravbawa9898@gmail.com, arjoomalik1@gmail.com

ABSTRACT

Large Language Models (LLMs) have demonstrated strong capability in translating natural-language requirements into working code, yet most existing systems operate on a single layer of an application generating either frontend components or backend services in isolation. This paper proposes an LLM-based full-stack code generation and validation framework that converts a single natural-language requirement into coordinated frontend components (React), backend REST APIs (Node.js), and database schemas and operations (PostgreSQL), and then validates the resulting application as one coherent system rather than as disconnected parts. The central research problem addressed is that independently generated frontend and backend code may each be individually functional yet incompatible when integrated, owing to mismatched data contracts, inconsistent API schemas, and divergent assumptions about state and validation. The proposed framework introduces a cross-layer integration-contract engine that binds the three generated layers to a shared specification, followed by an automated validation pipeline comprising unit testing, integration testing, end-to-end testing, and static security analysis. The framework is evaluated using three criteria: integration success rate, functional correctness, and security defect density. Preliminary evaluation design indicates that explicit contract-level validation substantially reduces integration failures compared to independent, layer-isolated generation. The paper contributes a reusable architecture, an evaluation methodology, and a research agenda for reliable, validated full-stack code generation using LLMs.

Keywords: Large Language Models, Full-Stack Code Generation, Software Integration, Automated Validation, React, Node.js, PostgreSQL, Security Analysis

1. Introduction and Objective

1.1 Background

The rapid advancement of Large Language Models has enabled automated generation of source code directly from natural-language descriptions. Tools built around LLMs are now capable of producing user interface components, server-side logic, and database queries individually with reasonable accuracy. However, real-world software applications are rarely single-layer artifacts; they are composed of interdependent frontend, backend, and data-persistence layers that must agree on data formats, request and response contracts, authentication flows, and error-handling conventions. Current LLM-based code-generation tools largely treat these layers as separate generation tasks, which introduces a structural risk: each layer may be syntactically correct and independently testable, yet the assembled system fails when the layers are connected.

1.2 Motivation

As organizations increasingly explore LLM-assisted development to accelerate prototyping and reduce engineering overhead, the reliability of full-stack generation becomes a practical concern. A frontend component generated to call an endpoint that the backend generator did not produce, or a backend endpoint that returns a payload shape the frontend does not expect, represents exactly the type of integration failure that undermines trust in automated code generation. This motivates a framework that treats full-stack generation as a single, coordinated process governed by shared contracts, rather than three independent generation processes stitched together after the fact.

1.3 Research Objectives

- To design a unified framework that converts a single natural-language requirement into frontend, backend, and database artifacts using an LLM-driven pipeline.
- To introduce a cross-layer integration-contract mechanism that enforces consistency between generated frontend calls, backend endpoints, and database schemas.

- To develop an automated validation pipeline that assesses integration success, functional correctness, and security defect density of the generated application.
- To propose an evaluation methodology suitable for benchmarking full-stack LLM code-generation systems against layer-isolated baselines.

1.4 Scope of the Study

This study focuses on web applications built on the React, Node.js, and PostgreSQL stack, chosen for their prevalence in industry and the availability of mature testing and static-analysis tooling. The framework is presented at the architectural and methodological level; empirical results from a full implementation are left as subsequent experimental work, consistent with the research-gap-driven, framework-proposal nature of this paper.

2. Literature Review

2.1 LLMs for Code Generation

Early work on neural code generation focused on single-function synthesis from natural-language docstrings or comments, evaluated primarily on correctness against unit tests. Subsequent LLM-based systems extended this capability to larger code units, including multi-file modules, by leveraging long-context models and retrieval-augmented prompting. These systems consistently show strong performance on isolated coding tasks but are typically evaluated using benchmarks that do not require cross-component integration.

2.2 Frontend and UI Code Generation

A distinct line of research addresses generation of user-interface code from natural language or visual mock-ups, targeting frameworks such as React and Vue. These approaches emphasize layout fidelity, component reusability, and adherence to design systems, but generally assume that backend services already exist and are called through manually defined or previously documented APIs.

2.3 Backend and Database Generation

Parallel research addresses automated generation of REST and GraphQL APIs, ORM models, and database migration scripts from schema descriptions or natural-language specifications. These systems tend to validate correctness against the backend in isolation, using schema-conformance checks and endpoint-level testing, without verifying how a consuming frontend would actually use the generated interface.

2.4 Multi-Agent and Pipeline Approaches to Software Generation

More recent work explores multi-agent LLM pipelines in which specialized agents are assigned to different software-engineering roles, such as requirements analysis, implementation, and review, coordinating through shared intermediate artifacts. This literature demonstrates that role specialization can improve output quality, but integration across generated components is frequently verified only through high-level code review or limited smoke testing rather than systematic contract validation.

2.5 Automated Testing and Security Analysis of Generated Code

A separate body of work evaluates the correctness and security of LLM-generated code using automated test generation, static analysis, and vulnerability scanning. These studies report that LLM-generated code can contain a non-trivial rate of security defects, including injection vulnerabilities and improper input validation, reinforcing the need for automated security validation as a standard stage in any LLM-based generation pipeline.

2.6 Synthesis

Across these strands, the literature converges on two observations relevant to this study. First, LLM-based generation is mature enough to produce individually functional frontend, backend, and database artifacts. Second, systematic validation of cross-layer integration as opposed to per-layer correctness remains underexplored, and security validation is often treated as a secondary concern rather than an integrated pipeline stage. These observations directly motivate the research gap addressed in this paper.

3. Research Gap

Existing LLM-based code-generation systems predominantly address frontend, backend, and database generation as separate problems, each with its own evaluation criteria and validation procedures. Frontend generators are assessed on layout and component correctness; backend generators are assessed on endpoint and schema correctness; database generators are assessed on schema validity and query correctness. This layer-isolated evaluation paradigm produces a critical blind spot: it is possible, and in practice common, for each generated layer to pass its own validation suite while the assembled application fails to function correctly once the layers are connected.

Specifically, three sources of integration failure are insufficiently addressed by current approaches:

- Data-contract mismatches, where the shape or type of data expected by the frontend differs from the shape or type returned by the backend.
- API-schema drift, where the backend generator produces endpoints, methods, or parameter names that the frontend generator does not anticipate, and vice versa.
- Cross-layer security gaps, where input validation implemented on the frontend is not mirrored on the backend, or where generated database operations are vulnerable to injection despite each layer passing isolated testing.

This paper addresses the gap directly by proposing a framework in which frontend, backend, and database generation are driven by a shared specification and reconciled through an explicit integration-contract engine, followed by validation that spans functional correctness across the full stack and security analysis of the integrated system rather than of individual layers alone.

Table 2: Comparison of the Proposed Framework with Existing Approaches

Approach	Scope of Generation	Integration Validation
Single-file code assistants (e.g., inline code completion tools)	Function- or snippet-level generation within one file	None; no cross-layer awareness
Frontend-only generators	UI components from natural language or design mock-ups	Not applicable; backend assumed to exist separately
Backend/API scaffolding tools	REST/GraphQL endpoints and database migrations	Limited; validated in isolation from consuming clients
Proposed Framework	Frontend, backend, and database layers generated jointly from one specification	Explicit integration-contract engine plus automated functional and security validation

4. Proposed Framework

4.1 Overview

The proposed framework accepts a single natural-language requirement as input and produces a validated, deployable full-stack application as output. It is organized into six stages: requirement parsing, parallel layer generation, integration-contract enforcement, automated functional testing, security and static analysis, and final packaging. Figure 1 illustrates the overall architecture.

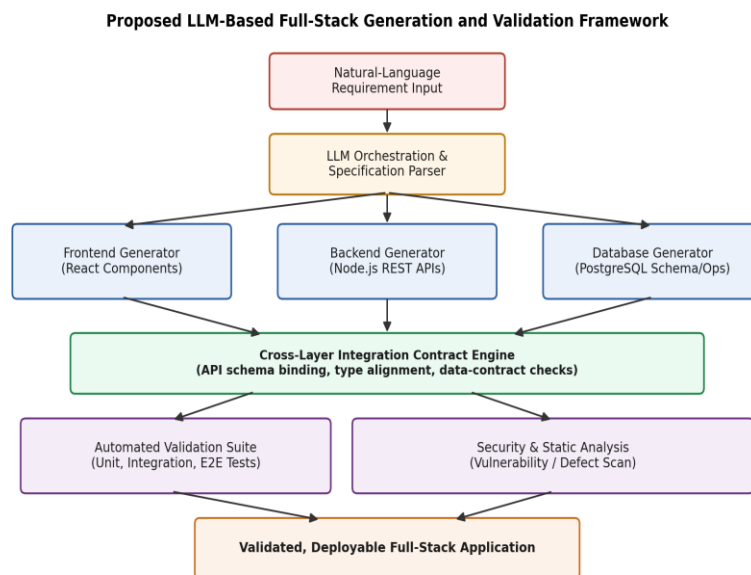


Figure 1: Architecture of the Proposed LLM-Based Full-Stack Generation and Validation Framework

4.2 LLM Orchestration and Specification Parser

The natural-language requirement is first processed by an orchestration layer that extracts a structured intermediate specification, including entities, relationships, required user interactions, and business rules. This specification serves as the single source of truth consumed by all three downstream generators, ensuring that frontend, backend, and database generation begin from identical assumptions rather than independently reinterpreting the original requirement.

4.3 Frontend, Backend, and Database Generators

Three specialized generation modules operate on the shared specification. The frontend generator produces React components, including state management and API-calling logic, expressed against the specification's data entities. The backend generator produces Node.js REST endpoints, including request validation, business logic, and response formatting. The database generator produces PostgreSQL schema definitions, migrations, and parameterized queries corresponding to the same entities. Because all three generators read from one specification, entity names, field types, and relationships are consistent by construction at the specification level; the framework's contract engine then verifies that this consistency is preserved in the generated code itself.

4.4 Cross-Layer Integration Contract Engine

This module is the central contribution addressing the identified research gap. It extracts an API contract (endpoint paths, HTTP methods, request and response schemas) from the generated backend code and compares it against the API usage patterns present in the generated frontend code, using schema-diffing techniques analogous to OpenAPI specification comparison. Discrepancies, such as a frontend call to a non-existent endpoint or a mismatched field name, are flagged and routed back to the relevant generator for regeneration before the pipeline proceeds. The same mechanism checks that database schema fields referenced by backend queries actually exist in the generated database schema.

4.5 Automated Validation Suite

Once contract checks pass, the framework executes unit tests for individual functions and components, integration tests that exercise backend endpoints against the live database, and end-to-end tests that drive the frontend through a headless browser to confirm that complete user workflows succeed against the running backend and database.

4.6 Security and Static Analysis

In parallel with functional validation, generated backend and database code is scanned using static application security testing techniques to detect common vulnerability classes, including SQL injection, insufficient input sanitization, and insecure authentication handling, before the application is considered deployable.

4.7 Output

The framework's final output is a validated, deployable full-stack application accompanied by a validation report summarizing integration success, functional test results, and identified security defects, giving developers actionable visibility into any residual risk before deployment.

5. Research Methodology

5.1 Research Design

This study adopts a design-and-evaluation research methodology, appropriate for proposing and validating a novel software architecture. The methodology proceeds in six sequential steps, illustrated in Figure 2, moving from requirement-dataset collection through to metrics evaluation.

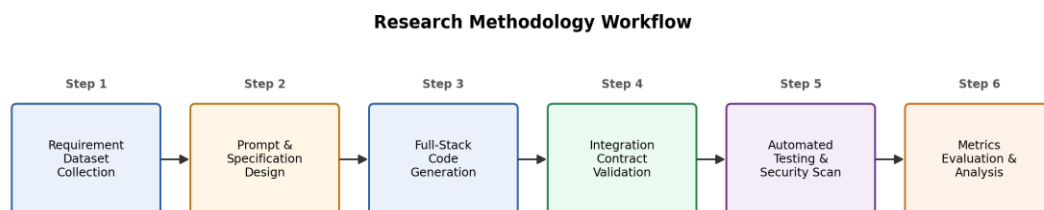


Figure 2: Research Methodology Workflow

5.2 Requirement Dataset Collection

A benchmark set of natural-language application requirements is compiled, spanning common web-application patterns such as CRUD resource management, authentication flows, and multi-entity dashboards, to ensure evaluation covers a representative range of integration complexity.

5.3 Prompt and Specification Design

For each requirement, a structured intermediate specification is derived, and prompt templates for the frontend, backend, and database generators are constructed such that all three generators condition on the same specification fields.

5.4 Full-Stack Code Generation

Each benchmark requirement is passed through the full pipeline to produce a complete set of frontend components, backend APIs, and database artifacts, under both the proposed integrated framework and a layer-isolated baseline configuration for comparison.

5.5 Integration Contract Validation

Generated outputs are passed through the cross-layer integration-contract engine, and the number and type of contract violations detected are recorded for each benchmark case and each configuration.

5.6 Automated Testing and Security Scanning

Each generated application is deployed in an isolated containerized environment and subjected to unit, integration, and end-to-end test suites, followed by static security analysis of backend and database code.

5.7 Evaluation Metrics

Three primary evaluation dimensions are used to assess the framework, detailed in Table 1: integration success, functional correctness, and security defect density. These metrics are computed per benchmark case and aggregated across the full requirement dataset to compare the proposed integrated framework against the layer-isolated baseline.

Table 1: Evaluation Metrics for the Proposed Framework

Metric	Definition	Measurement Method
Integration Success Rate	Percentage of generated applications where frontend, backend, and database layers interoperate without contract mismatches	Automated integration test suite executed against the deployed stack
Functional Correctness	Degree to which generated functionality satisfies the original natural-language requirement	Unit tests and requirement-to-behaviour traceability scoring
Security Defect Density	Number of security vulnerabilities per thousand lines of generated code	Static application security testing (SAST) and dependency scanning
API Contract Consistency	Alignment between backend endpoint schemas and frontend request/response expectations	Schema-diff validation using OpenAPI specification comparison
End-to-End Task Success	Completion rate of representative user workflows across the full stack	Automated end-to-end (E2E) browser-driven test execution

5.8 Threats to Validity

Anticipated threats include the representativeness of the benchmark requirement set relative to real-world application diversity, variability introduced by non-deterministic LLM outputs across repeated generations, and the coverage limits of automated security scanning relative to manual security review. These threats are mitigated through repeated-trial averaging, a diverse benchmark set, and use of established static-analysis tooling.

6. Conclusion

This paper identified a critical gap in current LLM-based code-generation research: frontend, backend, and database code may each be individually functional yet fail to operate correctly once integrated, due to unresolved contract, schema, and security inconsistencies across layers. To address this gap, the paper proposed a full-stack code-generation and validation framework built on React, Node.js, and PostgreSQL, centered on a cross-layer integration-contract engine and an automated validation pipeline spanning functional testing and security analysis. A corresponding research methodology was outlined, defining a benchmark-driven evaluation using integration success rate, functional correctness, and security defect density as primary metrics.

The framework offers a structural response to a problem that is likely to grow more consequential as LLM-based code generation moves from single-file assistance toward full-application scaffolding. Future work includes implementing the complete pipeline, executing the proposed benchmark-driven evaluation empirically, extending the framework to additional technology stacks, and investigating adaptive regeneration strategies that allow the framework to automatically resolve detected contract violations rather than merely flagging them for manual correction.

References

- [1] Chen, M., Tworek, J., Jun, H., et al. (2021). Evaluating large language models trained on code. arXiv preprint.
- [2] Austin, J., Odena, A., Nye, M., et al. (2021). Program synthesis with large language models. arXiv preprint.
- [3] Nijkamp, E., Pang, B., Hayashi, H., et al. (2023). CodeGen: An open large language model for code with multi-turn program synthesis. International Conference on Learning Representations.
- [4] Qian, C., Cong, X., Yang, C., et al. (2023). Communicative agents for software development. arXiv preprint.
- [5] Hong, S., Zhuge, M., Chen, J., et al. (2023). MetaGPT: Meta programming for a multi-agent collaborative framework. arXiv preprint.
- [6] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. IEEE Symposium on Security and Privacy.
- [7] Fried, D., Aghajanyan, A., Lin, J., et al. (2023). InCoder: A generative model for code infilling and synthesis. International Conference on Learning Representations.
- [8] Li, Y., Choi, D., Chung, J., et al. (2022). Competition-level code generation with AlphaCode. *Science*, 378(6624), 1092–1097.
- [9] Ross, S. I., Martinez, F., Houde, S., Muller, M., & Weisz, J. D. (2023). The programmer's assistant: Conversational interaction with a large language model for software development. Proceedings of the 28th International Conference on Intelligent User Interfaces.
- [10] Fowler, M. (2018). Patterns of Enterprise Application Architecture (2nd ed.). Addison-Wesley.