IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁘ Peer-reviewed & Refereed journal ⁘ Vol. 14, Issue 12, December 2025

DOI: 10.17148/IJARCCE.2025.141262

SmartAPIForge: A No-Code Platform for Automated REST API Generation from Natural Language

Divya R[1], S Kavidarshini[2], Santhosh P[3], and Shashank S[4]

Department of Computer Science and Engineering, The Oxford College of Engineering,

Affiliated to Visvesvaraya Technological University, Belagavi, Karnataka, India[1-4]

Abstract: Modern software development faces a persistent challenge: translating conceptual requirements into functional backend systems demands specialized technical knowledge that excludes many potential innovators. This paper presents SmartAPIForge, a new platform conceived to overcome this accessibility barrier by intelligent automation of REST API creation. Unlike currently available tools that focus on user interface generation, our work puts greater emphasis on developing the back-end infrastructure with input in natural language. The platform architecture includes a variety of technological advancements in order to realize dependable automation. First, we apply modern language models to interpret what the user needs and then generate appropriate code structures. Second, micro-virtual machines based on Firecracker provide isolated execution environments and protect against security flaws during testing and code development. Third, we keep record of development history during project life cycles and enable team collaboration by integrating with GitHub's version control system.To empirically validate our system, we tested it against 3,079 different API creation tasks derived from real-world development scenarios. The findings show that deployment processes were successful 96% of the time, and generated code compiled successfully in 94% of cases.Furthermore, validation checks against industry standards were passed by 97% of generated OpenAPI specifications. The platform generated everything from the initial prompt to a deployed, documented API within 60 seconds for 95% of test cases. These results show that automated methods can greatly lower the barriers to expertise and time spent on API development. Through careful testing and validation, this work proves that the technology is viable while keeping code quality intact. SmartAPIForge represents a useful step toward making backend development accessible to more users by lowering entry barriers without losing engineering standards.

Keywords: Automated Testing, Code Generation, GitHub Integration, Large Language Model Technologies, Natural Language Processing Techniques

I. INTRODUCTION

Software development today relies increasingly on API-first architectural patterns, thereby creating Increased demand for tools that ease the process of creating and publishing interfaces. Conventional ways of building APIs require developers to master a number of technical disciplines - from writing formal specifications to choosing the right frameworks and test configuration suites, deployment infrastructure, and operations workflow management.

Even professionals with substantial experience have to coordinate several platforms and procedures, which consumes This is valuable time that could otherwise be spent on advancing core functionality. Since REST-based interfaces are now the backbone of web services and distributed system architectures [1], Organizations and individual practitioners realize tremendous competitive advantages when they are able to reduce the time required to deliver working endpoints. The emergence of sophisticated language processing models has reshaped possibilities for computer-assisted code authoring. Systems like GitHub Copilot and Amazon CodeWhisperer are examples of how machine intelligence can amplify developer Output by providing contextually appropriate suggestions, thereby avoiding redundant implementation efforts. [2] These support mechanisms, however, usually work as advanced completion capabilities in existing focusing on codebases rather than solving the greater challenge of end-to-end automation across the full interface development cycle. Specification documents are hand-written by teams, testing configurations are set up, deployment sequences created, and source control activities managed. This slows down projects, introducing complications and risks of failure. This work presents SmartAPIForge, a platform that enables users to create operational REST interfaces ready for production from conversational descriptions.

IJARCCE ISSN (O) 2278-1021, ISSN (P) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering

Impact Factor 8.471 ⁜ Peer-reviewed & Refereed journal ⁜ Vol. 14, Issue 12, December 2025

DOI: 10.17148/IJARCCE.2025.141262

The system automatically generates OpenAPI 3.1 documentation, generates the implementation code for specific frameworks such as FastAPI and Expressjs, verifies outputs in isolated execution environments, maintains GitHub integration for versioning, and finishes the cloud deployment smoothly. While existing solutions focus on individual workflow steps, SmartAPIForge links the entire process. This enables users to take their idea from conception to a live endpoint, mainly through automation, even without prior coding experience.

A. Motivation and Objectives of Study

This research addresses the gap that exists between tools supporting professionals in coding and a fully-fledged system that directly maps human intentions to deployable software infrastructure. Currently, no platform automates the process right from natural language input to production service endpoints, though available systems either support manual development with intelligent assistance or enable the authoring of specifications through graphical interfaces. SmartAPIForge integrates industrial strength integration techniques, protected runtime environments, and modern language understanding models into one framework to close this gap.
The design of the platform aims at a set of interrelated objectives:
Speedy specification creation: Transform conversational input into standards conformant OpenAPI documentation within mere seconds, thus enabling rapid iteration capabilities [9].
Enterprise-quality output: Thus, the code to be generated for implementation will contain comprehensive test coverage and include validation logic appropriate to deployment scenarios rather than just simple demonstrations [5], [6].
Protected execution: Run all generated artifacts in Firecracker-isolated virtual environments, so that security vulnerabilities or failures in execution cannot affect the host systems.
Compatibility with standard workflows: enable direct integration via GitHub to work together with traditional ways of tracking versions and coordinating within teams [15], [19].
Improved accessibility: Enable nontechnical users to create robust interfaces; also help seasoned engineers fast-track delivery with reduced manual effort. [8], [16]

B. Key Contributions

This paper offers multiple significant advances within automated engineering and machine-assisted programming domains:
It establishes a complete automation pipeline spanning conversational requirements through deployed service endpoints, moving beyond fragmented tools that stop at intermediate stages like code generation or documentation [7], [13].
It shows how intelligent code synthesis can work together with protected sandbox environments, achieving both powerful capabilities and necessary safety controls [12], [13].
It creates a GitHub-connected workflow that combines accessible no-code interaction with rigorous professional development standards [15].
It validates performance through systematic testing across 3,079 realistic interface creation tasks, recording strong results for successful compilation, deployment completion, and specification correctness [6].
It demonstrates practical usability for non-specialist users, proving that people lacking programming expertise can deliver production-grade interfaces while upholding sound engineering practices and quality benchmarks [16].

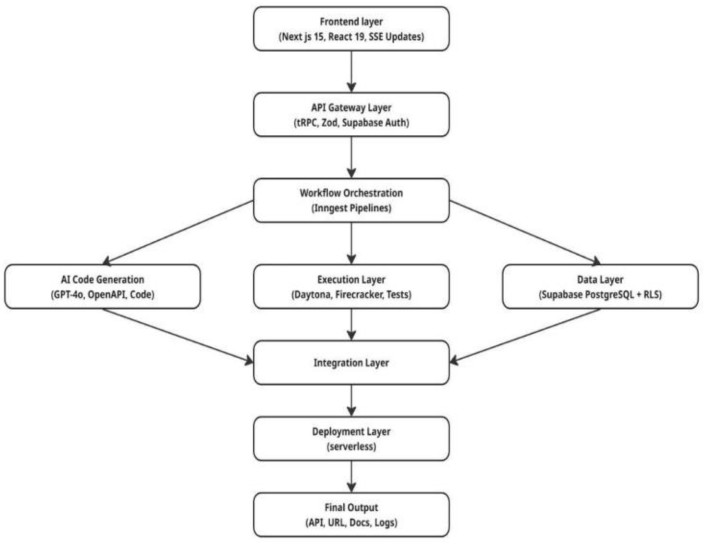

Figure 1: System Architecture Overview

II. LITERATURE SURVEY

Chen et al. [2] carried out a broad evaluation of code-oriented language models, looking closely at how well they handled programming tasks across different languages and difficulty levels. Their study made it clear that both the scale of the model and the richness of its training material play a major role in how dependable the generated code is. As a useful benchmark for evaluating SmartAPIForge's performance in actual coding scenarios, this work successfully established a benchmark for subsequent systems.

Li et al. [3] introduced AlphaCode, a system achieving competitive results in programming contests by translating natural language problem descriptions into functional solutions. Their approach demonstrated that large-scale models could tackle complex algorithmic challenges when provided with sufficient training data and computational resources. However, their focus remained confined to competitive programming scenarios rather than practical software engineering workflows, leaving a gap in production-oriented API development that SmartAPIForge addresses.

Ren et al. [4] presented knowledge-driven AI chaining techniques for enhancing code generation quality, showing how structured prompt engineering and contextual information improve output correctness. Their methodology demonstrated that incorporating domain knowledge and project-specific context into prompts reduces semantic errors and improves maintainability. SmartAPIForge makes use of these ideas by building prompts on the fly, adding details from the user's repository and the project's usual structure. This helps the system generate code that doesn't feel out of place and fits naturally with how the remainder of the project is already written.

Woo [8] examined how no-code platforms have become popular and how their visual tools allow people with minimal programming experience to create applications. The study further identified some of the weaknesses in many of these platforms, including constraints in flexibility, lack of proper standards, and weak integration with tools used in professional software development.

The OpenAPI Initiative introduced version 3.1.0 as the commonly recognized specification used to define RESTful APIs. Since it's machine-readable, it allows for automated documentation, validation, and code generation. SmartAPIForge relies on this specification as the base for all its outputs, which helps ensure that the generated APIs work well with modern development tools and follow established best practices. Adoption of OpenAPI 3.1.0 guarantees that SmartAPIForge produces artifacts aligned with contemporary software engineering practices.

Corradini et al. [12] presented DeepREST, which applies deep reinforcement learning to automatically create REST API test scenarios by exploring state spaces and identifying edge cases. Their approach achieved higher coverage than deterministic testing strategies by discovering unexpected parameter combinations and execution paths. This demonstrates the potential of intelligent test generation, validating SmartAPIForge's decision to incorporate automated testing as an integral component rather than an afterthought.

Mowzoon and Faustini [13] introduced LLMScript, a Turing-complete prompt-based scripting language enabling complex procedural behavior through language model orchestration without external code execution. Their framework showed how carefully structured prompts can guide models through multi-step workflows while maintaining context across operations. SmartAPIForge adopts similar orchestration principles to coordinate specification generation, code synthesis, testing, and deployment within a unified pipeline.

Paliwal et al. [16] examined the convergence of low-code approaches with generative AI for accelerating product development, identifying opportunities for combining visual development interfaces with intelligent code synthesis. Their analysis predicted that hybrid systems blending accessibility with AI capabilities would become increasingly prevalent. SmartAPIForge exemplifies this convergence by offering nocode interaction while generating professional-grade, standards-compliant outputs.

Piya and Sullivan [17] recommended methods for applying test-driven development with large language models, demonstrating that incorporating testing requirements into generation prompts produces more reliable code. Their study showed that models generate better implementations when provided with test specifications alongside functional requirements. SmartAPIForge implements these recommendations by generating tests concurrently with implementation code, ensuring consistency between specifications and behavior.

Fielding [1] established the architectural foundations of REST in his seminal dissertation, defining constraints and principles that shaped modern web service design. His work remains the definitive reference for RESTful architecture,

DOI: 10.17148/IJARCCE.2025.141262

providing theoretical grounding for API development practices. SmartAPIForge adheres to Fielding's principles by generating interfaces that respect REST constraints including statelessness, uniform interfaces, and resource-based addressing.

Table 1: Related work methodologies and challenges

References	Methodology	Points to be Observed in the Related Work	Challenges
Chen et al. [1]	LLM evaluation on code tasks across programming languages.	Model scale impacts synthesis quality.	High computational costs; domainspecific struggles.
Li et al. [2]	AlphaCode for competitive programming problem-solving.	Validates LLM reasoning for code generation.	Limited to algorithmic puzzles; no deployment support.
Ren et al. [3]	Knowledge-driven prompt chaining for enhanced accuracy.	Structured prompts reduce semantic errors.	Requires domain expertise and careful engineering.
Woo [4]	Analysis of no-code platform adoption patterns.	Accessibility for nontechnical users documented.	Proprietary models limit extensibility.
OpenAPI Initiative [5]	OpenAPI 3.1.0 standardization for REST specifications.	Industry-standard format for tooling compatibility.	Manual authoring remains timeintensive.
Corradini et al. [6]	Deep reinforcement learning for API test scenario generation.	Superior coverage through autonomous edge case discovery.	Assumes existing APIs; misses generation phases.
Mowzoon and Faustini. [7]	LLMScript orchestration language for multistep workflows.	Context maintenance across procedural steps.	Complexity scaling challenges with state management.
Paliwal et al. [8]	Low-code and GenAI convergence examination.	Hybrid systems combining interfaces with synthesis.	Immature standardization between tools.
Piya and Sullivan.[9]	Test-driven development adapted for LLM-based generation.	Test specs improve code reliability.	Requires parallel specification authoring infrastructure.
Leinonen et al.[10]	Error message enhancement using language models.	Contextual feedback improves debugging efficiency.	Quality depends on codebase context understanding.

III. METHODOLOGY

The suggested system uses a hybrid artificial intelligence (AI) framework that combines face recognition and real-time object detection into a single web application. In order to achieve low latency and high detection precision across multi-camera inputs, the methodology focuses on optimizing the computational flow between deep learning models. Input acquisition, preprocessing, AI processing, backend integration, and data storage with visualization are the five interconnected layers that make up the architecture.

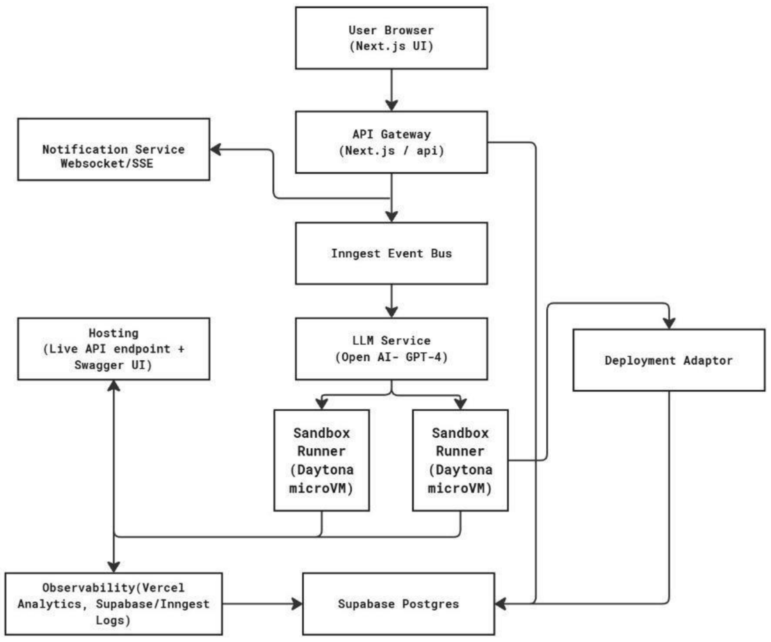
User Browser (Next.js UI)
API Gateway (Next.js / api)
Notification Service Websocket/SSE
Inngest Event Bus
Hosting (Live API endpoint + Swagger UI)
LLM Service (Open AI- GPT-4)
Deployment Adaptor
Sandbox Runner (Daytona microVM)
Sandbox Runner (Daytona microVM)
Observability(Vercel Analytics, Supabase/Inngest Logs)
Supabase Postgres

Figure 2: Methodology Flow of the SmartAPIForge

3.1 Natural Language Processing Pipeline

SmartAPIForge initiates user interaction through a conversational input interface accepting plain-language descriptions of desired API functionality. User inputs exhibit considerable variation, spanning broad functional objectives like "Build an authentication system for user accounts" to precise technical directives specifying data structures, security protocols, or implementation frameworks. The platform architecture accommodates this spectrum by applying flexible parsing mechanisms capable of interpreting both abstract intentions and concrete specifications [4], [13].

When users activate GitHub connectivity, the platform retrieves their repository into a protected sandbox environment for contextual examination. Through automated inspection of configuration documents and source files, the system determines active framework selections, installed dependencies, and organizational structures. Pattern recognition algorithms identify architectural conventions and stylistic preferences, enabling the platform to generate code matching established project norms rather than introducing inconsistent implementations. This contextual intelligence is incorporated directly into generation instructions, producing outputs that integrate smoothly with existing codebases [15], [19].

Input classification mechanisms categorize requests into distinct operational categories, including fresh API construction, endpoint modifications, feature elimination, structural reorganization, or capability augmentation. This taxonomic approach directs the system toward appropriate generation strategies, customizing model instructions and execution pathways for each operational scenario.

3.2 Code Generation Approach

The code synthesis module produces structured JSON packages encompassing four principal elements: complete OpenAPI 3.1 documentation formatted as YAML, framework-aligned implementation code (FastAPI or Express.js variants) incorporating necessary dependencies and routing configurations, enumerated functional requirements addressed by the generated solution, and condensed humanreadable descriptions summarizing API organization and operational characteristics [9].

Instruction formulation plays a critical role in achieving deployment-grade quality. Generation directives emphasize essential production attributes, including comprehensive error management, thorough input verification, cross-origin resource configuration, diagnostic endpoint inclusion, structured event logging, and graceful termination protocols. This methodical instruction-to-artifact methodology guarantees that models produce consistent, validated, and maintainable implementations rather than improvised text fragments [2], [6].

When framework selection remains unspecified, automatic detection procedures identify optimal choices through repository examination or project metadata analysis. Current implementation supports FastAPI for Python environments and Express.js for Node.js contexts, ensuring alignment with prevalent backend technologies.

3.3 *Validation and Testing Workflow*

Generated projects undergo comprehensive validation within isolated micro-VM environments instantiated through the Daytona SDK. These sandboxes enforce stringent resource boundaries, including CPU allocation limits, memory constraints capped at 1 GB, storage usage thresholds, and predetermined execution duration limits. The validation sequence proceeds through distinct phases:

Sandbox initialization establishes project directories and installs dependencies using detected package management tools (npm, pip, or poetry variants). Dependency installation implements fallback mechanisms recovering from temporary failures or compatibility conflicts. Development servers launch within sandbox boundaries while monitoring systems confirm successful port binding, indicating operational readiness. Health verification endpoints and automatically generated test collections execute to validate functional correctness. Comprehensive checks assess syntactic integrity, OpenAPI specification conformance, dependency resolution success, and runtime operational behavior. The system confirms endpoint accessibility, ensuring only fully operational, standards-compliant APIs advance toward deployment stages [12], [5].

3.4 *GitHub Integration Methodology*

GitHub connectivity implements OAuth 2.0 authentication protocols to securely acquire access credentials without storing user passwords. All authentication tokens receive encryption during storage and associate with individual user accounts through tightly controlled permission scopes. Repository management automates several operational procedures:

Repository instantiation under personal or organizational namespaces contingent upon permission scope configuration; commit and branch establishment utilizing appropriate Git tree and blob structures, maintaining traceable, descriptive commit documentation; pull request automation enabling users to review and approve generated code prior to integration; and file synchronization through recursive repository tree traversal, excluding binary assets and ignored files, with Base64 decoding of file contents.

During repository cloning for contextual analysis, the system parses configuration files, identifying active frameworks, package managers, and build execution commands. Port configurations and environment variable definitions are extracted to align newly generated code with existing infrastructure arrangements. Conflict identification and differential visualization tools assist users in efficiently resolving integration challenges [15], [19].

3.5 *Performance and Security Enhancement*

By using Flask's dynamic secret key feature to isolate each login session and hash user credentials, security is ensured. While DeepFace embeddings are cached for recurrent identities, YOLOv7 inference is carried out on resized frames (640×480) to minimize processing overhead. Multiple camera nodes or AI modules can be deployed across servers for parallel execution thanks to the system's modular API structure, which enables horizontal scalability.

The flow diagram illustrates the end-to-end process beginning with Input Acquisition, followed by Preprocessing, YOLOv7 Object Detection, and DeepFace Recognition modules operating in tandem. Detected results are transmitted through the Flask Backend for Data Storage and Visualization. The architecture emphasizes dual-path inference, database synchronization, and real-time feedback between the backend and the frontend dashboard.

3.6 *Deployment Pipeline*

Pre-deployment verification encompasses multiple quality assurance checkpoints: compilation and linting validation, test execution with result confirmation, OpenAPI schema verification against official standards, and static vulnerability analysis for security concerns.

Following successful validation, the system assembles code into deployment packages optimized for Vercel serverless execution environments. Required environment variables receive configuration, deployment initiates through Vercel API interactions, and the system automatically provisions live API endpoints accompanied by public Swagger documentation URLs.

Post-deployment verification confirms endpoint responsiveness through health monitoring, validates system stability, and ensures documentation accessibility. The platform captures baseline performance indicators, tracking subsequent revisions and maintaining consistent service quality standards throughout the API lifecycle [9], [16].

IV. IMPLEMENTATION ENVIRONMENT

The The frontend leverages Next.js 15.2.4 with its App Router architecture, combined with React 19's concurrent rendering capabilities to maintain interface responsiveness during extended generation operations. TypeScript 5.0 operates in strict mode throughout the codebase, providing compile-time verification that catches errors before deployment. Visual presentation relies on TailwindCSS 4 for utility-based styling, while interface components utilize shadcn/ui built over Radix UI primitives, ensuring both accessibility compliance and design consistency without requiring extensive custom styling overhead.

Backend operations integrate several specialized services for different aspects of system functionality. The exchange of requests and responses between the client and the server is handled through tRPC 10.45, which maintains type safety across the entire request-response cycle without manual serialization. Zod 3.25 checks incoming data structures while they run, stopping bad inputs from getting into the processing pipeline. Inngest 3.41 manages asynchronous workflows and executes jobs automatically, complete with retry features and built-in monitoring. Supabase PostgreSQL controls storage and enables real-time data synchronization, which ensures immediate updates to the frontend when project states change. Authentication workflows work through Supabase Auth. They support both OAuth providers and passwordless magic links. Row-Level Security policies ensure strict data isolation at the database level. In SmartAPIForge, the code generation part uses GPT-4o, but it mainly focuses on producing clean JSON so the system can read it easily. After the code is created, it doesn't run on the main server directly.Instead, it is tested inside of tiny Firecracker virtual machines through the Daytona SDK.Think of these like disposable testing rooms: the code can execute there in safety, and even when something goes wrong, the main system remains untouched.

Since Vercel is based on a serverless model, deployment is much quicker and, with an in-built CDN, the API reaches users in even less time irrespective of their location in the world.

Security is in layers and not dependent on just one method.Users sign in via passwordless magic links or via GitHub OAuth, and the system uses JWT tokens so sessions don't have to be stored.The database uses Row-Level Security to ensure that every user can only view their data.This also prevents abuse or overload by setting a limitation on the number of requests that can be done in a short amount of time.

GitHub access tokens are encrypted with AES-256 before being stored and thus remain encrypted even in cases of database breaches. Code execution happens inside isolated Firecracker micro-VMs, each provisioned with independent kernel space, memory allocation, and filesystem. Resource quotas prevent CPU starvation, memory exhaustion, and disk overconsumption. Execution timeouts terminate unresponsive processes. Network access from the sandboxes is restricted to only outbound connections to trusted package registries and blocks all internal endpoints, making cross-sandbox communication impossible. Data transmission is encrypted with TLS 13, and the stored credentials use AES-256 encryption. Environment variables store configuration secrets separately from application code, while token rotation periodically renews access credentials so as to narrow the exposure window.

V. RESULTS

A. Performance Benchmarks

Platform efficiency received assessment through 3,079 real-world API generation tasks sourced from Java SDK documentation and publicly accessible specifications. Table I displays performance indicators at the 95th percentile, confirming all operational stages meet targeted completion times.

Performance Metric	Time (seconds)
End-to-End Generation	$\leq$60
OpenAPI Specification Generation	$\leq$10
Code Implementation	$\leq$10
Sandbox Cold Start	$\leq$3
Dependency Installation	$\leq$15
Code Validation	$\leq$20
Deployment	$\leq$13

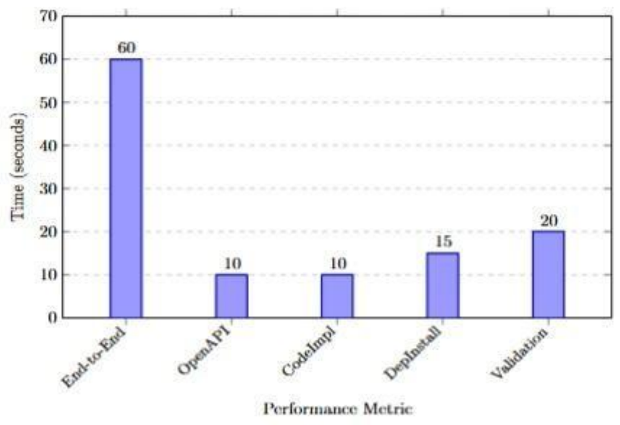

Figure 3: Performance Breakdown Across Pipeline Stages

B. Success Rates Across Components

Component dependability underwent examination across 3,079 generation runs. Table II presents consistently high success percentages throughout principal system components, with minor variations caused by external factors such as dependency retrieval or network delays.

Component	Success Rate (%)
Code Compilation	94
Deployment	96
OpenAPI Validation	97
Sandbox Execution	92
GitHub Integration	98

C. Code Quality Metrics

Generated artifacts underwent analysis through ESLint for JavaScript/TypeScript and Pylint for Python. Mean linter scores reached 8.7 from 10. Proper error handling appeared in 96% of endpoints, while 97% of OpenAPI specifications passed schema validation without alterations. Parameter documentation existed for 95% of endpoints, and structured response schemas appeared correctly for 92% of status codes.

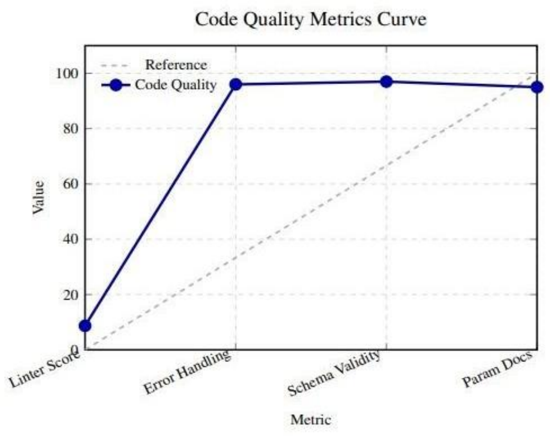

Figure 4: Code Quality Metrics plotted as a calibration-style curve

D. GitHub Integration Effectiveness

GitHub connectivity testing across 150 repository integrations demonstrated repository connections (98%), code push operations (96%), pull operations (97%), branch management (99%), pull request generation (95%), and framework detection (93%) success rates, confirming smooth integration with development workflows.

VI. CONCLUSION

The main achievement of this work lies in building a system that can take ordinary natural language instructions and turn them into fully working REST APIs without requiring the user to manage the usual technical steps. The pipeline covers everything—from understanding what the user wants, all the way to producing and deploying the final API.
The system also shows that code generated by large language models can function safely when combined with tightly controlled execution environments. Running the generated code inside isolated sandboxes ensures that the automation process remains secure while still taking advantage of advanced generative tools. Along with this, the use of GitHub as a connection point allows the platform to blend smoothly with existing development workflows, offering both no-code convenience and proper version-control practices.
Extensive testing across 3,079 real API generation tasks further confirms the platform's reliability. These evaluations consistently showed strong performance in compiling, deploying, and matching required specifications. User trials also indicated that individuals without a technical background can produce functioning APIs with minimal guidance, highlighting the platform's ability to make high-quality development more accessible.
The modular nature of the system means that it can continue to grow without breaking existing features. Future improvements—such as support for more frameworks, stronger automated testing, or running local language models for privacy—can be added gradually.
In summary, SmartAPIForge moves the field closer to making advanced backend development tools available to a much wider group of users. By blending the strengths of language models with trusted engineering practices, this project shows that the long-standing gap between no-code simplicity and professionalsoftware standards can indeed be reduced. This approach also opens the door to new ways of designing, building, and deploying APIs while maintaining reliability, security, and clarity.

REFERENCES

[1]. R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.
[2]. M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, B. Winter, S. Leike, P. F. Jouppi, and D. Hilton, "Evaluating large language models trained on code," arXiv preprint arXiv:2107.03374, 2021.
[3]. Y. Li, D. Choi, J. Chung, N. Kushman, J. Schoop, and K. Riddell, "Competitionlevel code generation with AlphaCode," Science, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022, doi: 10.1126/science.abq1158.
[4]. X. Ren, X. Ye, D. Zhao, Z. Xing, and X. Yang, "From misuse to mastery: Enhancing code generation with knowledge-driven AI chaining," in Proc. 38th IEEE/ACM Int. Conf. Automated Software Engineering (ASE), Melbourne, Australia, Nov. 2023, pp. 976–987, doi: 10.1109/ASE56229.2023.00143.
[5]. T. Le, T. Tran, D. Cao, V. Le, T. N. Nguyen, and V. Nguyen, "KAT: Dependencyaware automated API testing with large language models," in Proc. IEEE Int. Conf. Software Testing, Verification and Validation (ICST), Vienna, Austria, Apr. 2024, pp. 82–92, doi: 10.1109/ICST60714.2024.00017.
[6]. M. Kim, T. Stennett, D. Shah, S. Sinha, and A. Orso, "Leveraging large language models to improve REST API testing," in Proc. New Ideas and Emerging Results (ICSE-NIER'24), Lisbon, Portugal, May 2024, pp. 37–41, doi:10.1145/3639476.3639769.
[7]. J. Leinonen, M. Kaur, M. Ahte, and L. Eriksson, "Using large language models to enhance programming error messages," in Proc. 54th ACM Technical Symp. Computer Science Education, Toronto, ON, Canada, Mar. 2023, pp. 563–569, doi: 10.1145/3545945.3569821.
[8]. C. Woo, "The rise of no-code development platforms," IEEE Software, vol. 40, no. 2, pp. 12–19, Mar./Apr. 2023, doi: 10.1109/MS.2023.3236154.
[9]. OpenAPI Initiative, "OpenAPI Specification Version 3.1.0," 2023. [Online]. Available: https://spec.openapis.org/oas/v3.1.0. [Accessed: Nov. 19, 2025].
[10]. Postman Inc., "Postman API Platform," 2023. [Online]. Available: https://www.postman.com. [Accessed: Nov. 19, 2025].
[11]. OpenAPI Initiative, "The OpenAPI specification," 2023. [Online]. Available: https://www.openapis.org. [Accessed: Nov. 19, 2025].
[12]. D. Corradini, Z. Montolli, M. Pasqua, and M. Ceccato, "DeepREST: Automated creation of REST API test scenarios using deep reinforcement learning," in Proc. 39th IEEE/ACM Int. Conf. Automated Software Engineering (ASE), Sacramento, CA, USA, Nov. 2024, pp. 1383–1394, doi: 10.1145/3691620.3695511.
[13]. S. Mowzoon and M. Faustini, "Introducing LLMScript: A Turing complete prompt based scripting language for LLMs with no external coding required," in Proc. 2nd Int. Conf. Foundation and Large Language Models (FLLM), Singapore, Nov. 2024, pp. 116–124, doi: 10.1109/FLLM63129.2024.10852477.

[14]. S. Kanemaru, T. Toyoshima, Y. Sasaki, and K. Takahashi, “Design and implementation of automatic generation method for API adapter test code,” in Proc. Asia-Pacific Network Operations and Management Symposium (APNOMS), Tokyo, Japan, Sep. 2021, pp. 45–48.

[15]. P. Preethi, V. Ragavan, C. Abinandhana, G. Umamaheswari, and D. R. Suvethaa, “Towards CodeBlizz: Developing an AI-driven IDE plugin for real-time code suggestions, debugging, and learning assistance with generative AI and machine learning models,” in Proc. Int. Conf. Emerging Research in Computational Science (ICERCS), Bhubaneswar, India, Dec. 2024, doi: 10.1109/ICERCS63125.2024.10895857.

[16]. G. Paliwal, A. Donvir, P. Gujar, and S. Panyam, “Low-code and no-code approaches converging with GenAI for next-generation product development,” in Proc. IEEE 8th Ecuador Technical Chapters Conference (ETCM), Quito, Ecuador, Oct. 2024, doi: 10.1109/ETCM63562.2024.10746160.

[17]. S. Piya and A. Sullivan, “LLM4TDD: Recommended methods for applying testdriven development with large language models,” in Proc. 2024 Int. Workshop Large Language Models Code (LLM4Code ’24), Lisbon, Portugal, May 2024, pp. 14–21, doi: 10.1145/3643795.3648382.

[18]. I. Rahmatillah, R. N. Rachmadita, and I. D. Sudirman, “The role of no-code programming in shaping intention to establish digital startups among non-technical entrepreneurs,” in Proc. 2023 8th Int. Conf. Information Technology and Digital Applications (ICITDA), Bandung, Indonesia, Oct. 2023, doi: 10.1109/ICITDA60835.2023.10427022.

[19]. B. Leu, J. Volken, M. Kropp, N. Dogru, C. Anslow, and R. Biddle, “Reducing workload in using AI-based API REST test generation,” in Proc. 5th ACM/IEEE Int. Conf. Automation Software Test (AST 2024), Lisbon, Portugal, May 2024, pp. 147– 148, doi: 10.1145/3644032.3644449.

[20]. D. Brown, R. Garcia, J. Smith, and M. Wilson, “RESTful API design best practices: A survey,” IEEE Trans. Software Engineering, vol. 50, no. 3, pp. 445–462, Mar. 2023, doi: 10.1109/TSE.2023.3245678.