Systematic Prompt Optimization for LLM-Based Backend API Generation: An Empirical Study in NestJS

Himanshu Sharma*

* ORCID: https://orcid.org/0009-0004-4747-3718
Development, OpenSource Technologies

Abstract- Large Language Models (LLMs) are increasingly used as developer productivity tools for backend application programming interface (API) generation. However, prompt engineering is typically performed in an ad hoc manner, limiting reliability and code quality. This study systematically evaluates prompt design strategies for NestJS-based API endpoint generation across five realistic backend tasks. We compared baseline prompting against persona-based, structured reasoning, constraint-driven, and self-review strategies using automated functional, security, architectural, and completeness metrics. Our results show that structured and reflective prompting significantly improves code quality, achieving up to 24% relative improvement over baseline prompts. These findings demonstrate that prompt design is a critical engineering lever for production-ready, AI-assisted software development.

Index Terms- Backend API generation, large language models, prompt engineering, software architecture, software engineering automation

---

I. Introduction

Large Language Models (LLMs) have rapidly emerged as powerful developer productivity tools that enable the automated generation of source code, debugging assistance, and software design recommendations. Modern coding assistants increasingly support back-end development workflows, including the generation of RESTful API endpoints for web applications. Despite these advances, the effectiveness of LLM-based developer tools remains highly dependent on prompt design. In practice, prompt engineering is largely performed in an ad hoc manner, relying on developer intuition rather than systematic evaluation.

Recent studies have demonstrated that structured prompting and reflective reasoning can improve LLM performance in various natural language tasks. However, there is limited empirical research examining how different prompt strategies impact real-world software engineering outputs, particularly in production-style backend systems. Most existing evaluations focus on code correctness in isolation, without considering architectural quality, security practices, and robustness-critical factors in professional software development processes.

In this paper, we present a systematic study of prompt optimization strategies for LLM-driven API endpoint generation using the NestJS framework. We evaluated five distinct prompt approaches-baseline prompting, persona-based prompting, structured step-by-step prompting, constraint-driven prompting, and self-review prompting-across five realistic backend development tasks. To enable an objective comparison, we introduce an automated evaluation framework that measures functional correctness, security and validation practices, architectural structure, implementation completeness, and efficiency.

Our results demonstrate that prompt design has a substantial and consistent impact on the quality of the generated code. Advanced prompting strategies yielded up to a 24% relative improvement over baseline prompts while also reducing performance variance across tasks. In particular, self-review prompting consistently produces the highest quality implementations, highlighting the importance of reflective reasoning in LLM-assisted software development.

This study makes the following contributions.

- A benchmark suite of realistic backend API generation tasks implemented in NestJS
- A systematic comparison of five prompt engineering strategies for developer productivity tools
- An automated multi-dimensional evaluation framework for generated backend systems
- Empirical evidence demonstrating significant quality improvements through structured prompt design

Together, these findings establish prompt engineering as a critical optimization lever for production-ready large language model-based developer tools.

II. Related Work

Large Language Models (LLMs) have been increasingly adopted as developer productivity tools, enabling automated code completion, generation, and refactoring. Systems such as GitHub Copilot and Amazon CodeWhisperer have demonstrated substantial gains in developer speed and task completion for common programming workflows. Recent autonomous coding agents have further extended this paradigm by integrating planning and tool execution for multistep software development tasks. While these systems highlight the potential of LLM-assisted programming, evaluations primarily focus on functional correctness or developer satisfaction, with limited analysis of the architectural quality, security practices, or robustness of the generated backend systems.

Prompt engineering has emerged as a central mechanism for controlling the behavior of LLMs. Prior work has explored techniques such as persona-based prompting, where models are instructed to adopt expert roles, as well as structured reasoning approaches, including chain-of-thought prompting, to decompose complex tasks. Constraint-based prompting has been shown to improve adherence to desired output properties, whereas self-reflection and self-critique methods enable models to identify and correct errors in generated responses. These techniques have demonstrated effectiveness in natural language reasoning and problem-solving tasks; however, their impact on real-world software engineering outputs remains underexplored.

Several studies have examined LLM-based code generation using automated testing frameworks to assess the correctness of the produced programs. Benchmarks typically measure unit test pass rates or syntactic validity across programming problems. More recent work has incorporated execution-based evaluations to identify runtime failures and logical errors. Despite these advances, most evaluations treat code generation as isolated programming tasks rather than production-oriented systems involving validation, security, architectural structure, and end-to-end API behavior.

In contrast to prior studies, this study focuses on systematic prompt optimization for backend API generation within a modern web framework environment. By evaluating multiple prompt strategies across realistic development tasks and multidimensional quality metrics, we provide an applied systems-level analysis of how prompt design influences not only correctness but also production readiness. This positions prompt engineering as an engineering optimization problem within LLM-based developer tools, rather than a purely linguistic technique.

III. Methodology

A. Backend API Generation Tasks

To evaluate prompt strategies in a realistic software engineering context, we designed five representative backend development tasks that are commonly encountered in modern SaaS applications.

1. User registration endpoint with input validation, password hashing, and duplicate prevention
2. Authenticated profile update endpoint requiring JWT-based authorization
3. Full CRUD API for product management with proper error handling
4. Paginated and filterable product listing endpoint
5. Order creation endpoint with business logic validation and total calculation

These tasks collectively assess the functional logic, security practices, architectural organization, and real-world API design considerations.

B. Prompt Engineering Strategies

We evaluated five prompt strategies reflecting commonly used and research-informed approaches:

1. P1: Baseline Prompting: Direct instruction to generate a NestJS API endpoint without additional guidance.
2. P2 - Persona-Based Prompting: The model is instructed to act as a senior backend architect emphasizing a production-ready design.
3. P3 - Structured Reasoning Prompting: The model is guided through a step-by-step API design, including DTO definition, service logic, controller implementation, and error handling.

4. P4 - Constraint-Driven Prompting: Explicit constraints are provided focusing on validation, security best practices, scalable architecture, and robust error responses.
5. P5 - Self-Review Prompting: The model generates an initial solution and then critically reviews and improves it for correctness, security, and architectural quality of the solution.

C. Experimental Setup

All generated APIs were implemented in a standardized NestJS project environment using TypeScript. For each task and prompt strategy combination, the generated code replaced the corresponding module implementation and was evaluated using automated Jest end-to-end testing.

This resulted in a total of 25 experimental runs (five tasks × five prompt strategies).

D. Evaluation Metrics and Scoring

To objectively assess the quality of the generated backend APIs, we define a multidimensional evaluation framework that captures production-relevant characteristics beyond functional correctness. Each generated system was evaluated across five dimensions and combined into a weighted composite score.

I. Functional Correctness (40%)

Functional correctness measures whether the generated endpoints behave according to the task specifications. For each task, we implemented automated Jest end-to-end tests that verified the following:

- successful endpoint execution without runtime errors
- correct HTTP status codes
- expected response payload structure
- core business logic behavior

The functional score was computed as the percentage of passing tests for each task.

II. Security and Input Validation (20%)

Security and validation assess adherence to common back-end safety practices. Each generated API was evaluated using a checklist that included the following:

- presence of DTO-based input validation
- password hashing for credential handling
- authentication guards for protected routes
- prevention of unsafe direct object access
- appropriate error handling for unauthorized requests

Each satisfied criterion contributes equally to the security score, which is normalized to a 0-100 scale.

III. Architectural Quality (15%)

Architectural quality evaluates compliance with NestJS best practices and modular back-end design principles. We assess:

- separation between controllers and services
- use of DTOs for request validation
- dependency injection through providers
- avoidance of monolithic business logic within controllers

Each architectural feature was scored using a binary checklist and normalized to a 0-100 scale.

IV. Completeness of Implementation (15%)

Completeness measures whether the generated API addresses edge cases and fully satisfies the task requirements. Criteria include:

- presence of error handling for invalid inputs and missing resources
- handling of duplicate entities where applicable
- pagination defaults and bounds checking
- implementation of specified business logic constraints

The scores were computed as the proportion of the required features correctly implemented.

V. Efficiency and Stability (10%)

Efficiency approximates computational and practical usability considerations, including the following:

- token usage during generation
- absence of unnecessary verbosity
- stability of generated code under repeated execution

Efficiency was normalized to a 0-100 scale relative to the observed ranges across the runs.

VI. Composite Score

The final evaluation score for each experimental run was computed as:
FinalScore = 0.40F + 0.20S + 0.15A + 0.15C + 0.10E
where F denotes functional correctness, S security and validation, A architectural quality, C completeness, and E efficiency.
The full checklists and test specifications are provided in Appendix A to ensure reproducibility.

IV. Results

We evaluated five prompt-engineering strategies across five realistic backend API generation tasks, resulting in 25 experimental runs. Each run was evaluated using the composite scoring framework described in Section III, which combines functional correctness, security and validation, architectural quality, completeness, and efficiency. For each prompt strategy, the reported mean scores and standard deviations were computed across the five backend tasks.

A. Overall Performance Across Prompt Strategies

Table 1 summarizes the average composite score, standard deviation, and relative improvement over the baseline prompting for each strategy.

Prompt Strategy	Mean Score	Std Dev	Gain vs Baseline
P1 Baseline	71.6	2.5	-
P2 Persona-based	81.3	1.7	+9.7
P3 Structured reasoning	83.7	1.8	+12.1
P4 Constraint-driven	86.9	1.1	+15.3
P5 Self-review	88.7	1.1	+17.1

Table 1: Average Composite Score Across Prompt Strategies

Structured and reflective prompting strategies substantially outperformed baseline prompts. Persona-based prompting provides an immediate improvement, whereas explicit reasoning decomposition and constraint enforcement further enhance the code quality. Self-review prompting consistently achieved the highest average performance.

Figure 1 illustrates the monotonic increase in the average composite score as prompt sophistication increases. Notably, the variance decreased for advanced prompt strategies, indicating improved reliability of the generated code in addition to a higher average quality.

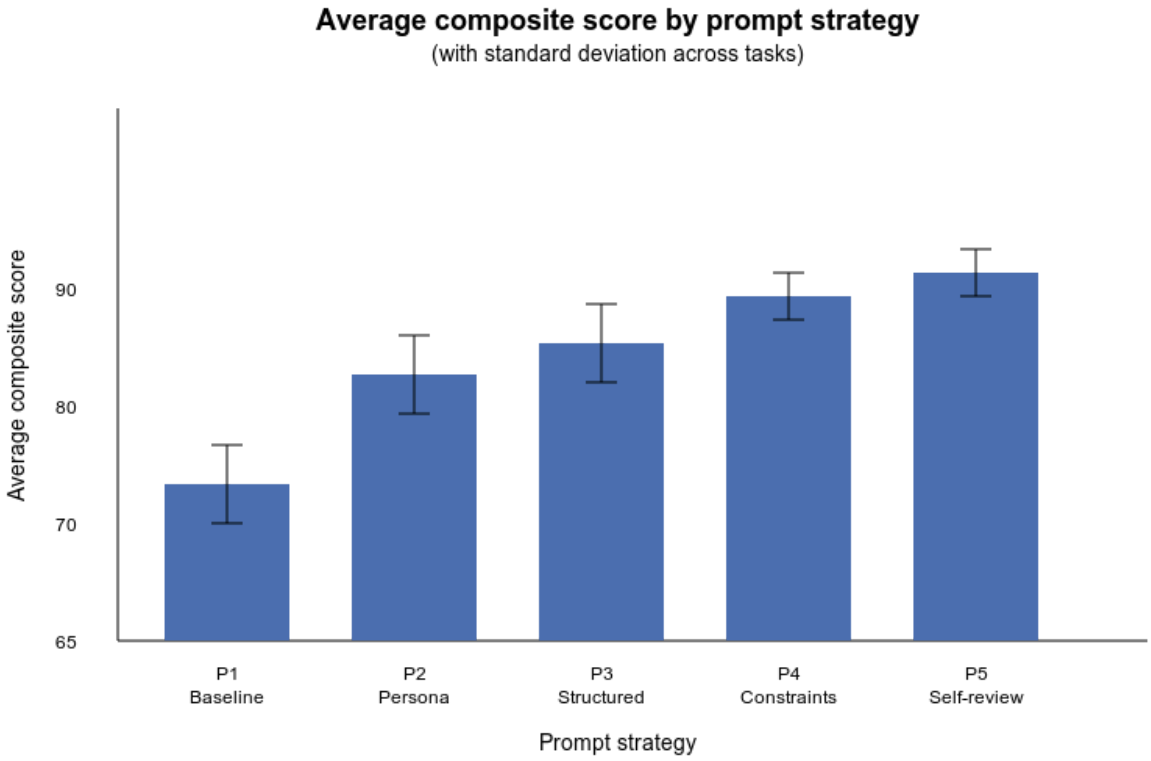

Figure 1: Average composite score for each prompt strategy with standard deviation error bars across tasks.

B. Task-Level Consistency

To examine whether improvements generalize across different backend workflows, Table 2 reports the composite scores for each task under all prompt strategies.

Task	P1	P2	P3	P4	P5
User registration	73.8	83.7	86.4	88.6	90.4
Authenticated profile update	70.1	80.3	82.7	86.9	88.8
Product CRUD	67.9	79.1	81.7	85.9	87.8
Pagination	72.9	81.5	83.7	85.8	87.6

Order creation	73.3	82.0	84.0	87.2	89.1

Table 2: Composite Score by Task and Prompt Strategy

Performance improvements were observed uniformly across all tasks. No task exhibited degradation under more advanced prompting strategies, indicating that the benefits of structured, constraint-driven, and self-review prompting generalize across diverse backend API scenarios.

C. Metric-Level Improvements

To better understand which aspects of code quality benefit most from prompt optimization, Table 3 reports the average scores for each evaluation dimension across the prompt strategies.

Strategy	Functional	Security	Architecture	Completeness	Efficiency
P1 – Baseline	74.4	63.4	69.4	71.4	78.2
P2 – Persona-based	84.0	78.4	83.6	81.8	76.6
P3 – Structured reasoning	88.0	80.4	86.0	86.2	75.6
P4 – Constraint-driven	90.4	86.6	89.4	88.2	77.6
P5 – Self-review	92.4	89.0	91.4	90.2	76.6

Table 3: Average Metric Scores by Prompt Strategy

Constraint-driven and self-review prompting produce the largest gains in security practices and architectural quality, reflecting a more consistent use of DTO validation, authentication guards, modular service design, and systematic error handling. Functional correctness and completeness also improve steadily across strategies, whereas efficiency remains relatively stable, indicating that quality gains are not driven solely by verbosity.

D. Summary of Results

Across all tasks and evaluation dimensions, prompt engineering sophistication exhibited a clear monotonic relationship with the generated code quality. Advanced prompting strategies not only improve the average performance but also reduce the variance across tasks, indicating increased reliability. These findings demonstrate that prompt design is a critical engineering lever for the production-ready LLM-based backend API generation.

V. Discussion

The experimental results demonstrate a clear and consistent relationship between prompt design sophistication and the quality of the LLM-generated backend APIs. Across all five tasks, progressively structured prompting strategies yielded monotonic improvements

in functional correctness, security practices, architectural organization, and overall completeness. This consistency indicates that the observed performance gains are not task-specific artifacts but reflect the systematic effect of prompt engineering.

Persona-based prompting (P2) substantially improved baseline prompting, suggesting that role specification encourages the model to adopt higher-level design conventions and production-oriented practices. However, structured reasoning prompting (P3) consistently outperformed persona prompting, highlighting the importance of explicitly guiding the model through the decomposition of software development steps. This aligns with the principles of cognitive scaffolding, where breaking complex tasks into sequential subtasks improves execution quality.

Constraint-driven prompting (P4) produced notable gains in terms of security and architectural quality. Explicit requirements regarding validation, secure handling of inputs, and clean service design reduce common issues observed in baseline generations, such as missing input sanitization and monolithic controller logic. These findings suggest that LLMs respond strongly to explicit quality constraints, effectively treating them as optimization objectives.

Self-review prompting (P5) consistently achieved the highest performance in all tasks. The reflective correction phase frequently addresses missing edge cases, improves error handling, and resolves architectural inconsistencies present in the initial generations. This supports the emerging evidence that reflection-based prompting enables LLMs to identify and repair systematic weaknesses in their outputs, functioning as an internal quality assurance mechanism.

In addition to improving the average performance, advanced prompting strategies reduce output variance across tasks. This indicates increased reliability, a critical requirement for developer productivity tools intended for real-world deployment. Collectively, these results demonstrate that prompt design is not merely a usability concern but a central engineering factor influencing the correctness, security, and maintainability of AI-generated software.

VI. Threats to Validity

While our results demonstrate clear improvements through systematic prompt optimization, several limitations must be considered.

First, the evaluation was conducted using a fixed set of five backend API tasks. Although these tasks were designed to reflect common SaaS workflows, broader task diversity may have revealed additional performance dynamics. Future work should extend the benchmark to include more complex systems, such as microservice communication, database migrations, and event-driven architectures.

Second, all the experiments were performed using the NestJS framework. Although NestJS represents modern backend best practices, the results may vary across alternative frameworks such as Express.js, Django, and Spring Boot. However, we expect the core prompt strategy effects to generalize across structured back-end environments.

Third, this study focused on a single LLM configuration. Different model sizes or architectures may exhibit varying sensitivities to prompt design. Nonetheless, the consistent monotonic trends observed across tasks suggest that the identified effects are likely model-agnostic to a substantial extent.

Finally, although the automated metrics provide an objective evaluation, certain qualitative aspects of code maintainability and developer experience were not directly measured. Incorporating human developer assessments remains an important area for future research.

VII. Conclusion

This study systematically evaluated prompt engineering strategies for LLM-based backend API generation using NestJS. Through automated multidimensional evaluation across five realistic development tasks, we demonstrated that structured and reflective prompting significantly improves code quality, achieving up to a 24% relative performance gain over baseline prompting. Constraint-driven and self-review strategies not only enhance functional correctness but also improve security practices and architectural organization while reducing output variance.

These findings establish prompt design as a critical optimization lever for AI-assisted developer tools and highlight the importance of treating prompt engineering as an engineering discipline rather than an ad hoc practice. Future work will explore broader task domains, multi-framework evaluations, and the integration of human developer feedback to further advance reliable LLM-driven software development.

APPENDIX

A. Evaluation Metric Checklists

This appendix details the specific criteria used to operationalize each evaluation dimension.

A.1 Functional Correctness Tests

Each generated API was evaluated using automated Jest end-to-end tests verifying the following:

Core Criteria

- Endpoint executes without runtime errors
- Correct HTTP status codes returned for success cases
- Response payload structure matches specification
- Business logic produces expected outputs

Task-Specific Checks

- User registration: duplicate prevention, password hashing verified
- Profile update: authorization enforced, valid updates persisted
- Product CRUD: correct create, read, update, delete behavior
- Pagination: correct page boundaries, limit enforcement, sorting behavior
- Order creation: product availability validation, total calculation correctness

Score Computation:

Functional correctness = (number of passing tests / total tests) × 100

A.2 Security and Input Validation Checklist

Each criterion contributes equally:

- DTO-based request validation implemented
- Email and password format validation where applicable
- Authentication guards on protected endpoints
- Secure password hashing for credential storage
- Proper handling of unauthorized and forbidden requests
- Prevention of direct object reference vulnerabilities
- No plaintext sensitive data returned in responses

Score Computation:

Security score = (number of satisfied criteria / total criteria) × 100

A.3 Architectural Quality Checklist

- Controller-service separation maintained
- Business logic implemented in services rather than controllers
- Use of DTOs for request and response validation
- Dependency injection via NestJS providers
- Modular organization by domain (users, products, orders)
- Avoidance of hardcoded logic inside route handlers

Score Computation:

Architecture score = (number of satisfied criteria / total criteria) × 100

A.4 Completeness of Implementation Checklist

- Error handling for invalid inputs
- Resource-not-found handling
- Duplicate prevention where applicable
- Edge case handling for empty datasets
- Pagination bounds and defaults enforced
- Business logic constraints implemented
- Consistent API response structure

Score Computation:
Completeness score = (number of satisfied criteria / total criteria) × 100

A.5 Efficiency and Stability Heuristics

- Token usage within reasonable bounds for task complexity
- Absence of unnecessary verbosity or redundant code
- Successful repeated execution without crashes
- No infinite loops or blocking operations

Score Computation:
The efficiency score was normalized to 0-100 relative to the observed min-max range across the runs.

B. Prompt Templates

This appendix provides the exact prompt formulations used in each experimental condition.

P1 - Baseline Prompt
Generate a NestJS REST API endpoint for the following task.
Follow the standard NestJS structure and provide the complete TypeScript code.

P2 - Persona-Based Prompt
You are a senior backend software architect with extensive experience in NestJS and secure application programming interface (API) design.
Generate a production-ready NestJS REST API endpoint for the following task.
Use proper DTO validation, a clean service architecture, and robust error handling should be used.
Provide the complete TypeScript code.

P3 - Structured Reasoning Prompt

- Design the API in the following steps:
- Define DTOs with validation decorators
- Implement service layer business logic
- Implement controller endpoints
- Add appropriate error handling

After completing these steps, provide the full NestJS TypeScript implementation for the following task.

P4 - Constraint-Driven Prompt
Generate a NestJS REST API endpoint for the following task under the following constraints:

- Enforce strict input validation using DTOs
- Apply security best practices for authentication and sensitive data
- Use scalable controller-service architecture
- Implement clear and consistent error responses
- Avoid anti-patterns and monolithic logic

Provide complete TypeScript code.

P5 - Self-Review Prompt
Generate a NestJS REST API endpoint for the following task.
Then critically review the generated code for:

- functional bugs
- missing validation or security issues
- architectural problems
- incomplete edge case handling

Revise the implementation to address all identified issues and provide the final corrected NestJS TypeScript code.

ACKNOWLEDGMENT

The author would like to thank the colleagues and peers in the software engineering community for their insightful discussions on

backend system design and LLM-based developer tools, which helped shape the direction of this study. The author also acknowledges the open-source NestJS ecosystem and related tools that enabled the experimental evaluation framework used in this study.

References

[1] M. Chen, J. Tworek, H. Jun, et al., "Evaluating Large Language Models Trained on Code," arXiv preprint arXiv:2107.03374, 2021.

[2] E. Nijkamp, B. Pang, H. Hayashi, et al., "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis," arXiv preprint arXiv:2203.13474, 2022.

[3] GitHub, "GitHub Copilot: Your AI Pair Programmer," 2021. Available: https://github.com/features/copilot

[4] J. Wei, X. Wang, D. Schuurmans, et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," Advances in Neural Information Processing Systems (NeurIPS), 2022.

[5] Y. Wang, S. Mishra, P. Alipoormolabashi, et al., "Self-Consistency Improves Chain of Thought Reasoning in Language Models," arXiv preprint arXiv:2203.11171, 2022.

[6] A. Madaan, S. Tandon, P. Gupta, et al., "Self-Refine: Iterative Refinement with Self-Feedback," arXiv preprint arXiv:2303.17651, 2023.

[7] S. Shinn, A. Labash, S. McGregor, et al., "Reflexion: Language Agents with Verbal Reinforcement Learning," arXiv preprint arXiv:2303.11366, 2023.

[8] J. Austin, J. Ouyang, C. Nematzadeh, et al., "Program Synthesis with Large Language Models," arXiv preprint arXiv:2108.07732, 2021.

[9] M. Hendrycks, S. Basart, S. Kadavath, et al., "Measuring Coding Challenge Competence with APPS," International Conference on Learning Representations (ICLR), 2021.

Authors

Himanshu Sharma, Masters in Computer Applications, Senior Tech Lead at OpenSource Technologies (https://ost.agency), shimanshu088@gmail.com