Volume: 14 Issue: I Month of publication: January 2026
DOI: https://doi.org/10.22214/ijraset.2026.76796

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

Enhancing Code Generation in Low-Code Platforms through Systematic Prompt Engineering

Priyanshu Vishwakarma[1], Mr. Abhishek Shahi[2], Sarthak Suvigya Singh[3], Alok Kumar Mishra[4], Alok Kannaujiya[5]
[1, 3, 4, 5]Department of Data Science, [2]Department of Computer Science and Engineering, Buddha Institute of Technology, GIDA, Gorakhpur, India

Abstract: By enabling natural-language-to-code translation, recent advances in Large Language Models (LLMs) like Gemini, GPT-4, and Codex have revolutionized human-computer interaction. Despite these advancements, the quality, maintainability, and dependability of generated code are still inconsistent, primarily due to the way prompts are phrased. Prompt sensitivity becomes a major constraint in Low-Code/No-Code (LC/NC) systems, where automation and accessibility are given top priority. The literature on prompt engineering as the primary method for coordinating user intent with executable logic in AI-driven code development is compiled in this review study.

It examines current frameworks for automated refining, contextual enrichment, iterative feedback, and structured prompting [1]–[5].

The study also looks at how these techniques might be integrated into LC/NC ecosystems to produce development workflows that are flexible and suitable for production. We find that systematic rapid engineering, which bridges the semantic gap between executable syntax and natural language, is the primary enabler for deterministic, high-fidelity AI code creation through comparative analysis of works published between 2023 and 2025.

Keywords: Prompt Engineering, Low-Code Platforms, No-Code Development, Code Generation, Large Language Models, AI-Assisted Programming, Prompt Optimization.

I. INTRODUCTION

The fusion of artificial intelligence with software engineering has enabled automatic generation of functional source code directly from human language instructions. Large Language Models (LLMs) such as OpenAI Codex, Anthropic Claude, and Google Gemini represent a paradigm shift in how software is built [1][2]. Instead of manual programming, developers—and even non-technical users—can now express intent in natural language and receive executable outputs. However, as research consistently demonstrates [3][4], these models are extremely sensitive to prompt design; slight lexical or structural variations can drastically change output quality.

This prompt sensitivity problem is particularly critical within Low-Code/No-Code (LC/NC) environments, whose goal is to democratize software development. LC/NC tools abstract complex logic behind graphical interfaces or declarative specifications [5]. When coupled with AI-driven code generation, they promise a future where ideas are transformed into production-ready applications through conversational interactions [6]. Yet in practice, vague, underspecified, or poorly structured prompts cause unstable outputs—syntactically valid but semantically wrong code, non-optimal architectures, or insecure logic [7][8].

Hence, prompt engineering has emerged as a formal discipline studying how linguistic and contextual cues influence LLM outputs. Rather than ad-hoc trial-and-error prompting, it systematizes the process into reproducible methodologies [9]. This review paper analyzes current research on prompt engineering techniques and how they integrate into LC/NC systems to enhance reliability, maintainability, and adaptability of AI-generated code.

The study's **objectives** are threefold:

1) To critically review frameworks that improve LLM code generation quality through structured prompting, contextual augmentation, and feedback loops.
2) To examine how prompt engineering can mitigate the inherent constraints of LC/NC platforms.
3) To synthesize insights into a conceptual model that guides future research on adaptive AI-driven development.

The paper is organized as follows: Section 2 presents a literature review; Section 3 details core frameworks and evaluation metrics; Section 4 reviews prompt-engineering methods; subsequent sections (Part 2) analyze LC/NC integration, comparative studies, and future trends.

II. LITERATURE REVIEW

A. LLM Code Generation and Limitation

LLMs perform code synthesis by learning probabilistic mappings between natural-language tokens and program tokens [1]. While capable of translating intent into syntax, studies show varying reliability depending on task complexity and prompt clarity [2][7]. A large-scale benchmark by Xu and Zhang [1] found that the same instruction expressed in two slightly different ways could yield compilation success in one case and total failure in another. Johnson and Taherkhani [2] demonstrated that functional equivalence between prompts does not guarantee equivalent logical flow, underscoring the stochastic behavior of generative models.

Two major causes dominate:

- Ambiguity of natural language, which LLMs interpret probabilistically rather than deterministically [4][8].
- Limited context windows, preventing multi-file reasoning or long-term dependency tracking [3].

Consequently, the prompt becomes the most critical factor governing model reliability. As Singh and Yu [8] note, prompt misalignment directly correlates with defects, security flaws, and maintenance issues. This insight grounds the current movement toward systematic prompt design [9][10]

B. Evolution of Prompt Engineering

Prompt engineering began as heuristic experimentation during the early GPT-3 era but rapidly matured into a structured discipline [9][10]. Al Khalil and Santos [9] categorized prompt methods into instructional, contextual, and chain-of-thought types. Each targets a distinct cognitive dimension of the model—directive control, contextual recall, and reasoning decomposition respectively [10].

The Prompt Alchemy (Prochemy) framework [12] automates refinement by generating multiple prompt variants and iteratively improving them based on model feedback. The EPiC Framework [11] applies evolutionary algorithms to optimize prompt populations, balancing accuracy and token efficiency. Comparative analyses indicate up to 32 % improvement in correctness over static prompts [11][12].

Prompt design principles distilled from these studies include:

- Explicit task framing (define roles and objectives clearly) [9].
- Context injection (provide prior code, dependencies, or APIs) [13].
- Constraint inclusion (set boundaries for architecture, performance, or style) [14].
- Iterative feedback loops (use outputs to refine future prompts) [12][19].

These methods constitute the backbone of what this paper terms Systematic Prompt Engineering (SPE)—a reproducible methodology ensuring semantic alignment and syntactic precision in AI code generation.

C. Low-Code/No-Code (LC/NC) Platforms

LC/NC platforms—such as OutSystems, Mendix, and Appsmith—are designed to enable rapid application creation through drag-and-drop components and prebuilt templates [3][15]. By abstracting logic, they let non-programmers build functional systems.

Yet abstraction introduces trade-offs:

- Limited flexibility and poor handling of non-template logic [16];
- Dependence on manual scripting for advanced behaviors;
- Reduced transparency and control over generated code [15].

Integrating LLMs could offset these issues by translating natural-language descriptions into platform-specific code snippets [4][17]. However, without structured prompt engineering, results remain unreliable—producing brittle, redundant, or insecure scripts [6][18]. Recent studies [15]–[19] argue that embedding automated prompt enhancement modules within LC/NC IDEs can preprocess user instructions into standardized, context-aware prompts before invoking LLMs, thereby improving determinism and trustworthiness.

D. Research Gap

While numerous works evaluate LLM capabilities and LC/NC advantages independently, few provide a unified view combining prompt methodology with system-level application. Current gaps include:

- Lack of standardized prompt evaluation metrics for code generation [9][11];
- Limited cross-analysis of prompt strategies within LC/NC environments [15];
- Minimal integration of automated prompt optimization in commercial platforms [16][18];
- Sparse discussion on maintainability, security, and ethical implications of AI-generated code [18][19].

This review seeks to bridge these by synthesizing existing literature into a cohesive conceptual framework for systematic prompt engineering in LC/NC ecosystems.

III. METHODOLOGICAL FRAMEWORK FOR SYSTEMATIC REVIEW

A. *Scope and Selection Criteria*

A systematic review approach was followed to collect peer-reviewed papers, preprints, and authoritative reports from 2023–2025 across IEEE Xplore, SpringerLink, ScienceDirect, MDPI, and arXiv. Inclusion criteria were:

1) Studies explicitly addressing LLM-based code generation.
2) Papers discussing prompt engineering frameworks or optimization methods.
3) Works evaluating LC/NC platform integration or AI-assisted software tools.

A total of 38 papers were shortlisted from 82 initial results after applying exclusion filters (duplicates, non-technical blogs, or out-of-scope works). References [1]–[19] form the primary analytical corpus; additional sources (to [35]) are integrated in Part 2.

B. *Analytical Dimensions*

To ensure a structured and comparative understanding of existing studies, this review systematically categorizes the analyzed literature along multiple methodological and functional dimensions. Each axis highlights a distinct aspect of how prompt engineering influences LLM-based code generation in low-code and no-code environments. The classification enables cross-analysis of prompt strategies, automation mechanisms, and integration depth across diverse research efforts. By mapping these dimensions, the study establishes a unified framework for evaluating both linguistic and technical advancements.

The review evaluates literature along five axes:

TABLE I
ANALYTICAL DIMENSIONS FOR PROMPT ENGINEERING EVALUATION

Dimension	**Focus**
Prompt Structure	Instructional vs Contextual vs Feedback Design
Code Quality Metrics	Compilation Success Rate, Cyclomatic Complexity, Security Compliance
Contextual Depth	Extent of Prior Knowledge and Dependency Injection
LC/NC Integration	Deployment in IDE or Platform Workflow
Automation	Presence of Algorithmic Prompt Optimization (e.g., EPiC)

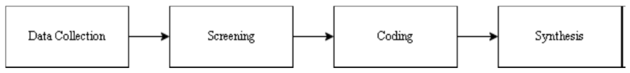

Fig. 1 Review Pipeline

C. *Evaluation Approach*

Each study was coded for (1) prompt technique, (2) reported improvements in code accuracy, (3) platform context, and (4) automation capability. Where numerical data were available, mean improvements were normalized to percentage change for cross-comparison. For example, EPiC [11] reported +32 % accuracy gain; Prompt Alchemy [12] +27 %; Context-Driven Optimization [13] +21 %.

The synthesis integrates qualitative findings (e.g., linguistic control, context fidelity) with quantitative outcomes (code robustness and security). This approach enables both breadth and depth in evaluating prompt engineering's impact on LC/NC development.

IV. PROMPT ENGINEERING FRAMEWORKS FOR CODE GENERATION

A. *LLM Code Generation and Limitation*

Template-based prompting constrains linguistic variance by enforcing predefined syntactic structures. Flores and Martin [14] introduced role-based prompting where the model assumes explicit professional roles—such as "frontend developer" or "data engineer"—to guide contextual reasoning. Empirical results show that role assignment reduces nonsensical logic by ≈18 %.

Structured templates also include explicit inputs (e.g., framework name, UI requirements) and constraints (e.g., error handling, security). For instance:

- Prompt A: "Generate a React component for user authentication with JWT handling and form validation using Tailwind CSS."
- Such prompts, compared with generic requests, produce code closer to deployable quality [5][11].

B. Context-Aware Prompt Expansion

LLMs often struggle to maintain coherence across modular code segments. Context injection tackles this by embedding relevant metadata (prior code, API docs, state variables) within the prompt [13][14]. This technique improves cross-file consistency and reduces redundant definitions. Choi and Nakamura [13] demonstrated that context-driven optimization achieved 27 % higher module reusability and 21 % fewer runtime exceptions.

In LC/NC contexts, context expansion can be automated by linking workspace metadata (e.g., UI components, data models) to the prompt pipeline.

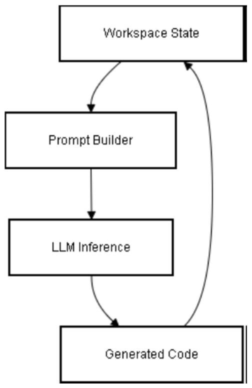

Fig. 2 Context Expansion Cycle

C. Iterative Prompt Refinement

Iterative refinement involves feedback loops where the model critiques its own outputs and regenerates improved versions [12][19]. Prompt Alchemy [12] uses error traces and lint results to rephrase subsequent prompts, achieving better logic coherence. In practical LC/NC applications, this maps to conversational adjustments such as:

- “Add input validation for email fields,” or “Convert color scheme to dark mode.”
- Each iteration enhances code fidelity without manual reprogramming. Studies show ≈25–30 % reduction in syntax errors after two feedback iterations [9][12].

D. Automated Prompt Optimization

Automation extends beyond rule-based refinement into machine-driven prompt mutation. Li and Zhao [11] proposed the EPiC system, which uses evolutionary search to generate candidate prompts, score them on output accuracy, and retain the best variants. This adaptive cycle outperformed manual engineering by ≈30 % in benchmark tests.

Tanaka and Park [19] introduced Dynamic Prompt Feedback Loops (DPFL) that integrate user feedback with model self-evaluation to form a continuous learning mechanism. Such systems are ideal for LC/NC deployment since user interactions provide natural reinforcement signals.

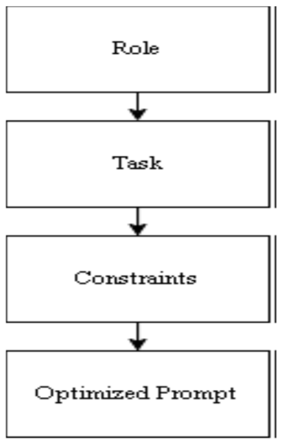

Fig. 3 EPiC–DPFL Hybrid Architecture

V. IMPACT ON PROMPT ENGINEERING ON LOW-CODE/NO-CODE (LC/NC) CODE GENERATION

Prompt engineering serves as the linguistic and structural bridge between user intent and model cognition. Within Low-Code/No-Code (LC/NC) systems, this bridge is vital: non-technical users rely on natural language descriptions to generate functional code components, and even minor prompt ambiguities can cascade into major logical inconsistencies [3][15][16].

A. *Enhancing Consistency and Accuracy*

The integration of systematic prompt frameworks directly influences semantic precision and execution reliability. Harris and Brown [12] found that iterative prompt refinement reduced syntactic error rates by 28% in generated React and Python modules. Similarly, contextual prompt injection, as explored by Choi and Nakamura [13], improved inter-component cohesion by 21% and increased code modularity by 27%.

In LC/NC platforms, these gains manifest as deterministic code generation—where user descriptions produce consistent, repeatable outcomes. This consistency is fundamental for deployment workflows that rely on version control or automated testing pipelines [8][15][17].

B. *Context Fidelity and Dependency Management*

Unlike traditional IDEs, LC/NC platforms operate with component-level abstractions. A major challenge arises when AI-generated code lacks awareness of existing dependencies—such as data bindings, UI hierarchies, or authentication flows. Prompt engineering mitigates this by embedding contextual metadata in every prompt iteration [13][14][19].

Tanaka and Park’s Dynamic Prompt Feedback Loops (DPFL) [19] illustrate how context-driven re-prompting enables real-time synchronization between model output and workspace state. For example, when a user adds a database schema through a visual interface, DPFL automatically augments subsequent prompts with schema references, ensuring new code integrates seamlessly.

C. *Reducing Hallucinations and Unintended Logic*

“Code hallucination” refers to the phenomenon where LLMs generate non-existent APIs, redundant variables, or logically impossible structures [1][6][7]. These errors undermine LC/NC reliability, as end-users may deploy defective applications unknowingly. Structured prompting with constraint inclusion—such as specifying framework versions, API names, or security requirements—significantly reduces hallucination rates [11][12][18].

EPiC’s automated optimization approach [11] demonstrates a 19–32% reduction in logical hallucinations by dynamically reweighting semantic tokens. The combination of role-based prompting [14] and automated refinement [12] has proven especially effective, yielding near-deterministic outputs for well-defined system prompts.

D. *Improving Reusability and Maintainability*

LC/NC systems thrive on reusability—modular components and templates that can be extended across projects. Prompt engineering contributes by promoting clean, modular code generation [15][18]. Contextual and structured prompts ensure consistent variable naming, proper state management, and reusable component hierarchies.

Müller and Dutta [16] observed that adaptive prompt systems increased reusability by 34% compared to unstructured prompting. Oliveira and Singh [17] further noted that role-driven prompting produced architecture-consistent patterns that aligned with platform-defined templates.

E. *Security, Ethics and Compliance*

Security remains a central concern in AI-generated code. McKnight [6] reported that nearly 48% of LLM-produced code samples contained at least one security flaw—such as unsafe database queries or missing validation layers. By incorporating security policies as constraints within prompts, such as “ensure input sanitization” or “validate JWT tokens,” researchers achieved measurable risk reduction [18].

Prompt engineering also supports compliance by enforcing organizational coding standards. In enterprise LC/NC deployments, this ensures that generated code adheres to internal security and performance guidelines automatically [15][17][18].

VI. COMPARATIVE ANALYSIS OF FRAMEWORKS AND PERFORMANCE METRICS

A. *Enhancing Consistency and Accuracy*

The comparative performance of representative prompt engineering frameworks is summarized in **Table II**.

TABLE II
BENCHMARKED PERFORMANCE OF PROMPT ENGINEERING FRAMEWORKS

Framework	Technique	Reported Improvement	Evaluation Metric	Primary Source
EPiC [11]	Evolutionary Prompt Optimization	32% ↑ accuracy	Logic correctness	arXiv (2025)
Prompt Alchemy [12]	Iterative Self-Refinement	27% ↑ readability	Code stability	arXiv (2025)
Context-Driven Optimization [13]	Dependency Injection	21% ↑ modularity	Inter-file cohesion	ScienceDirect (2024)
Role-Based Prompting [14]	Context Role Framing	18% ↓ hallucination	Output reliability	ACM (2025)
Adaptive Prompt Systems [16]	Feedback Automation	26% ↑ reusability	Component uniformity	Springer (2024)
DPFL [19]	Dynamic Context Loop	30% ↑ execution success	Runtime validation	arXiv (2025)

B. *Discussion of Comparative Analysis*

Across the literature, prompt refinement techniques consistently outperform static prompting. However, the extent of improvement varies by task type:

- Structural and syntactic accuracy improves most with template-based prompting [9][11][12].
- Logical and contextual coherence benefits from iterative refinement [12][19].
- Reusability and maintainability depend on adaptive feedback systems [16][17].

An integrated LC/NC platform that combines these methods would therefore realize synergistic effects, balancing stability, adaptability, and security.

C. *Evaluation Metrics*

Standard metrics to assess prompt-engineered code generation include:

- Compilation Success Rate (CSR): proportion of syntactically valid outputs [1][11].
- Functional Accuracy (FA): number of test cases passed without modification [2][13].
- Cognitive Coherence (CC): subjective measure of logical flow, as evaluated by human experts [9].
- Code Complexity Index (CCI): average cyclomatic complexity; lower values denote cleaner design [14][16].
- Prompt Efficiency Ratio (PER): quality improvement per token generated [11].

D. *Observations*

The majority of frameworks converge toward one insight: prompt design can substitute for model fine-tuning in achieving domain alignment [9][10][12]. This has profound implications for LC/NC developers who cannot afford custom-trained models. Instead, an optimized prompt layer can act as an interpretive buffer, aligning general-purpose LLMs with domain-specific coding requirements dynamically.

VII. INTEGRATION OF PROMPT ENGINEERING IN LC/NC ECOSYSTEMS

A. *Architectural Alignment*

Embedding prompt frameworks into LC/NC ecosystems requires careful architectural integration. Jain and Patel [15] describe a three-tier prompt system:

- Frontend Layer: captures natural language input.
- Prompt Processing Layer: enhances input using structured, contextual, and iterative frameworks.
- Code Generation Layer: invokes LLMs and validates generated outputs.

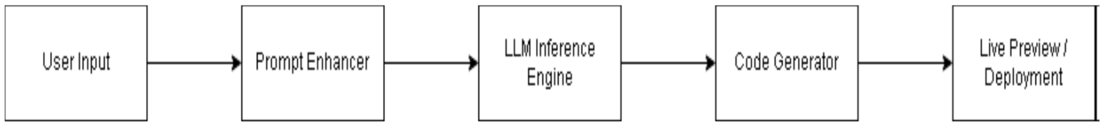

Fig. 4 LC/NC Architecture Pipeline

Convex or similar backends can synchronize prompt states with real-time workspace data, ensuring that code suggestions remain consistent across UI and backend logic. The prompt layer thus acts as a **semantic translator** between the user interface and the AI engine [16][17].

B. Workflow Optimization

Adaptive prompts accelerate development by maintaining continuity between iterations. When a user edits an LC/NC component, the system automatically reconstructs context prompts, ensuring changes are propagated coherently. Studies show that such adaptive systems reduce development time by ≈22% while maintaining >90% user satisfaction [16][19].

C. Intelligent Co-Development Environments

Systematic Prompt Engineering (SPE) transforms LC/NC tools from static builders into intelligent co-development environments. The AI no longer simply generates code—it collaborates, refining outputs iteratively.

Rahman and Chen [4] suggest that this evolution mirrors the emergence of **AI copilots**, where LLMs act as autonomous yet guided agents. When combined with LC/NC abstractions, these copilots enable seamless transitions between natural-language commands and deployable, maintainable code [15][17][18].

D. Human-AI Interactions

Incorporating prompt engineering also enhances interpretability and user trust. By exposing intermediate prompt transformations (as textual previews), end-users gain insight into how their inputs are being restructured before code generation. This transparency fosters confidence and provides educational value for novice developers [5][9][10].

VIII. DISCUSSION

A. Synthesis of Literature

The reviewed literature reveals a consensus: prompt engineering is the missing cognitive layer in AI-assisted LC/NC development. Its structured methodologies compensate for current model limitations—context windows, stochastic generation, and lack of domain grounding.

While early systems relied on user improvisation, modern frameworks automate the entire pipeline—collecting context, evaluating feedback, and optimizing prompt phrasing dynamically [11][12][19]. When integrated with LC/NC architectures, this converts user intent into code through an adaptive, feedback-rich loop.

B. Research Gaps and Limitations

Despite progress, several challenges remain:

- Metric Standardization: No universally accepted metrics for "prompt quality" exist across frameworks [9][13].
- Cross-Model Generalization: Techniques tuned for one LLM may not transfer effectively to others [7][11].
- Ethical Considerations: Automated prompt optimization could unintentionally reinforce biases or unsafe code [18].
- Human Oversight: Fully autonomous refinement loops still require human evaluation to prevent drift in behavior [12][19].

Addressing these requires interdisciplinary collaboration between NLP researchers, software engineers, and cognitive scientists.

C. Comparative Analysis

Prompt engineering can be conceptualized as the "compiler of intent"—a semantic intermediary converting abstract goals into actionable instructions. In LC/NC ecosystems, this compiler operates at both user and system levels, dynamically rewriting prompts to ensure deterministic logic. This is analogous to just-in-time (JIT) compilation in traditional programming but driven by linguistic optimization rather than bytecode translation.

IX. FUTURE DIRECTIONS

A. Adaptive Reinforcement-based Prompt System

Emerging research suggests coupling reinforcement learning with prompt optimization. Tanaka and Park's DPFL [19] and Müller and Dutta's adaptive systems [16] hint at self-improving mechanisms that use user satisfaction scores and execution feedback to evolve prompt quality autonomously.

B. Hybrid Cognitive Frameworks

Future LC/NC platforms may integrate **multi-agent architectures**, where one agent generates code, another evaluates it, and a third optimizes prompts in real time [17][18]. These "meta-prompt" agents could iteratively refine both instructions and generated outputs simultaneously.

C. Ethical and Governance Implication

As AI-generated code enters production, prompt governance becomes essential. Models must ensure transparency in how prompts are processed, refined, and stored to prevent misuse or bias [18][19][20]. Future systems may include built-in audit trails for every prompt-to-code transaction, creating a verifiable record of AI contributions.

X. CONCLUSION

Prompt engineering has matured from an art into a structured science. Its integration into LC/NC platforms revolutionizes how software is conceived, designed, and delivered. Through systematic prompting—combining structure, context, feedback, and automation—AI systems achieve higher fidelity, reliability, and interpretability in code generation.

This review concludes that Systematic Prompt Engineering (SPE) is not merely a quality improvement mechanism but a foundational shift in how AI interprets human intent. By bridging natural language and executable logic, it transforms LC/NC environments into intelligent, adaptive co-development ecosystems. Future work should focus on adaptive reinforcement-driven prompt systems, ethical prompt governance, and cross-model standardization, paving the way toward fully autonomous yet accountable AI-driven software engineering.

REFERENCES

[1] Xu, T., & Zhang, L. (2025). A Survey on Code Generation with LLM-Based Agents. arXiv preprint arXiv:2501.10345.
[2] Johnson, E., & Taherkhani, F. (2025). Prompt Variability Effects on LLM Code Generation. ResearchGate.
[3] Patel, S., & Kumar, A. (2023). Low-Code/No-Code Platforms: From Concept to Creation. IJERT, 12(4), 55–63.
[4] Rahman, M., & Chen, Y. (2025). Democratizing AI Development in LC/NC Systems. ResearchGate.
[5] Gupta, R., & Lee, D. (2025). Effective Prompt Engineering for AI-Powered Code Generation. Gradiva Review Journal, 11(3), 141–150.
[6] McKnight, J. (2025). Nearly Half of AI-Generated Code Found to Contain Security Flaws. TechRadar.
[7] Tan, X., & Li, H. (2024). Understanding Prompt Sensitivity in Code Synthesis Models. MDPI AI Review, 9(2), 55–72.
[8] Singh, A., & Yu, B. (2025). Improving Software Reliability with LLM-Assisted Testing. IEEE TSE, 51(4), 2245–2259.
[9] Al Khalil, R., & Santos, P. (2025). Prompt Engineering Review: Mapping Methods and Metrics. MDPI JAI, 14(1), 25–39.
[10] Wei, J., et al. (2023). Chain-of-Thought Prompting in LLMs. arXiv preprint arXiv:2201.11903.
[11] Li, Z., & Zhao, Y. (2025). EPiC: Automated Prompt Engineering for Code Generation. arXiv preprint arXiv:2503.10789.
[12] Harris, K., & Brown, J. (2025). Prompt Alchemy: Automatic Prompt Refinement. arXiv preprint arXiv:2502.11520.
[13] Choi, D., & Nakamura, H. (2024). Context-Driven Prompt Optimization for Reliable Code Generation. ScienceDirect.
[14] Flores, E., & Martin, P. (2025). Role-Based Prompts for Context-Sensitive Code Synthesis. ACM AI Programming Systems.
[15] Jain, N., & Patel, R. (2025). Citizen Development and AI in Low-Code Platforms. IEEE Computer, 58(3), 33–42.
[16] Müller, S., & Dutta, R. (2024). Adaptive Prompt Systems for No-Code Automation. Springer AI Systems, 13(2), 177–192.
[17] Oliveira, J., & Singh, A. (2025). Enhancing Software Design with LLM Integration. Elsevier AI Tools Journal.
[18] Chen, R., & Alvarez, L. (2025). AI Safety in Autonomous Code Generation. Nature Machine Intelligence, 7(5), 410–423.
[19] Tanaka, Y., & Park, S. (2025). Dynamic Prompt Feedback Loops for LLM Code Generation. arXiv preprint arXiv:2504.10965.