A Low-Cost Lightweight Large Model Code Generation Optimization Framework Based on Structured Constraint Generation and External Verification

Lei Liu
Key Laboratory of
Cyberspace Situation Awareness
of Henan Province
Zhengzhou, China
liulei_94@163.com

Junchao Cui
Key Laboratory of
Cyberspace Situation Awareness
of Henan Province
Zhengzhou, China
cjc20250301@163.com

Xuanzi Ma
Information Engineering University
Zhengzhou, China
mxz20251024@163.com

***Abstract*—Lightweight large language models (LLMs) in resource-constrained scenarios exhibit high reasoning uncertainty, leading to low code generation reliability. Existing zero-shot methods fail to address this systematically. We propose a two-stage "constraint-generation, code-generation" framework that governs process uncertainty. It first extracts structured constraints from problem examples, converting open-domain generation into controlled structured search. Then, code is generated under these constraints, and an external verification mechanism (AST + example execution) creates a model-independent feedback loop to suppress hallucinations. On HumanEval, our framework improves Qwen2.5-1.5B-Instruct's Pass@1 from 40.2% to 80.3%. It is model-agnostic and lightweight, offering a practical solution for edge computing and reliable neural generation.**

***Index Terms*—Lightweight Large Models, Code Generation, Uncertainty Governance, Constraint Generation**

I. Introduction

Code generation, a core task bridging natural language and program logic, has become a key application of large language models (LLMs) [1]. Recent LLM-based systems have advanced in syntax correctness and generalization [2], [3], but these gains rely on exponential model scaling, raising computational, latency, and deployment costs.

In practical scenarios such as edge computing, embedded systems, teaching platforms, and enterprise private deployments, running or frequently calling hundred-billion-parameter models is often infeasible. Lightweight models (2B parameters or fewer) have thus gained attention due to their lower resource requirements. However, extensive empirical studies show that compared to large models, small models are more prone to uncertainty-related issues: syntax errors, variable scope confusion, omission of boundary conditions, and insufficient understanding of complex task requirements [4]. This is because lightweight models must simultaneously perform problem understanding, constraint inference, algorithm design, and code implementation in a single generation process, leading to uncertainty superposition.

Existing research often attributes these problems to insufficient capacity or data, attempting to bridge the gap via parameter-efficient fine-tuning (e.g., LoRA [5]), reinforcement learning (e.g., CodeDPO [6]), or instruction fine-tuning. Although these methods improve performance to some extent, they require additional training stages, annotated data, or complex optimization processes, increasing overall system cost and deployment complexity.

On the other hand, prompt engineering and external process constraint methods try to improve code generation quality without modifying model parameters. However, existing zero-shot methods mostly rely on weak natural language constraints or model self-evaluation mechanisms, which are prone to hallucination accumulation due to insufficient reasoning capabilities of lightweight models, thereby limiting their practical effectiveness.

Notably, in many real-world code generation scenarios (e.g., programming competitions, online judging platforms, teaching tasks), problem descriptions are usually accompanied by explicit input-output examples. These examples contain highly deterministic task constraint information, but are often only used as auxiliary prompts in existing methods and not systematically integrated into the core constraint mechanism of the generation process.

Based on these observations, we propose to govern code generation uncertainty from the process design layer. The core idea is to treat code generation as a composite process with multiple uncertainty sources, externalize high-certainty information first, and control low-certainty generation under explicit constraints. We present the Constraint-Guided Code Generation (CG-CG) two-stage framework and validate it experimentally.

Our contributions include:

- Remodeling the code generation process from the perspective of process uncertainty governance, proposing an explicitly decoupled two-stage framework.

- Designing mechanisms for structured constraint generation and constrained code generation, with a model-independent external verification loop to suppress hallucination accumulation.
- Systematic experiments on HumanEval showing significant performance improvements for multiple lightweight models under zero-shot conditions, with Pass@1 improvement from 40.2% to 80.3% on Qwen2.5-1.5B.
- Analyzing the mechanism and engineering feasibility through ablation and cross-model validation, demonstrating the framework's model-agnostic nature and deployment value.

II. Related Work

A. Methods Based on Enhancing Internal Model Capabilities

Enhancing internal model capabilities includes Parameter-Efficient Fine-Tuning (PEFT), reinforcement learning, and instruction fine-tuning. Methods like LoRA [5] significantly reduce the number of fine-tuning parameters by introducing low-rank adapters but still require additional training stages and computational resources. Reinforcement learning-based methods such as CodeDPO [6] and AceCoder [7] guide the model to generate code more aligned with expectations through automated testing or reward functions, but their training processes are complex. Other approaches like hierarchical neural program synthesis [8] and OSS-INSTRUCT [9] also demand extensive computation, limiting their use in resource-constrained settings. Classic inductive program synthesis systems [10] solve synthesis problems through iterative guidance but rely on neural search that is computationally heavy.

B. External Process Constraint and Prompt Engineering Methods

Another line of research constrains model behavior via external process design without parameter modification [11]. Few-shot Chain-of-Thought (CoT) [12] and Expert Prompt guide step-by-step reasoning but often lead to unstable performance in lightweight models due to excessive contextual load. Self-Refine [13] introduces iterative refinement but relies on model self-evaluation, which accumulates hallucinations over multiple iterations.

Recent work introduces external tools to assist code generation. ReAct [18] combines LLMs with actions; CodeJudge explores semantic-level evaluation; AST-based methods [14] and type-constrained decoding [15] reduce compilation errors; execution feedback frameworks [16] enable self-optimization; hybrid approaches combining LLMs with genetic programming [17] show advantages of mixed paradigms. These methods improve quality but do not explicitly decouple constraint construction from code implementation, often still coupling both within the same generation process.

C. Positioning of This Work

Compared to the above works, our key innovation is remodeling code generation at the process level, treating "constraint construction" and "code implementation" as subtasks with significantly different uncertainty characteristics, and systematically controlling generation via structured representation and external verification mechanisms.

III. Constraint-Guided Code Generation (CG-CG) Two-Stage Framework

A. Design Motivation and Problem Modeling

Traditional code generation requires a model to simultaneously perform problem understanding, constraint inference, algorithm design, and code implementation. This highly coupled process imposes high demands on implicit reasoning and long-range modeling. Large models can absorb some constraints internally, but lightweight models suffer from uncertainty superposition, leading to syntax errors and logical flaws.

We formalize code generation as a composite system of sub-processes with different uncertainty characteristics. Information such as function signatures, input-output structures, and examples has high certainty, while specific algorithm implementation has higher uncertainty. We explicitly decompose the process into: (1) Structured Constraint Generation (SCG) for extracting high-certainty information, and (2) Constrained Code Generation (CCG) for low-certainty implementation under explicit constraints. From a neural network system perspective, this reconstruction at the inference stage improves the stability of neural generation systems in uncertain environments. This follows the principle of "externalizing high-certainty information first, controlling low-certainty generation," laying the foundation for external verification and iterative optimization.

B. Core Algorithm Framework Diagram

The framework includes SCG and CCG stages. SCG generates structured constraints from problem description and examples. CCG generates candidate code under constraints and original problem. External validation uses AST static verification and example execution. If verification fails, iterative optimization is triggered.

Verification and feedback rely entirely on external executable rules, not model self-evaluation, avoiding hallucination accumulation.

C. Structured Constraint Generation (SCG) Stage

SCG extracts highly deterministic structural information from natural language and converts it into a structured representation usable by subsequent stages. This is essentially an information extraction and formatting task, aligning well with lightweight models' pattern recognition capabilities.

1) Constraint Content Definition: Structured constraints include:

- Function interface constraints: name, parameter count, return type hints.
- Input-output structure constraints: explicit modeling of composite data structures (lists, dictionaries, nested types).
- Problem-specific example constraints: extracted input-output examples for functional verification.

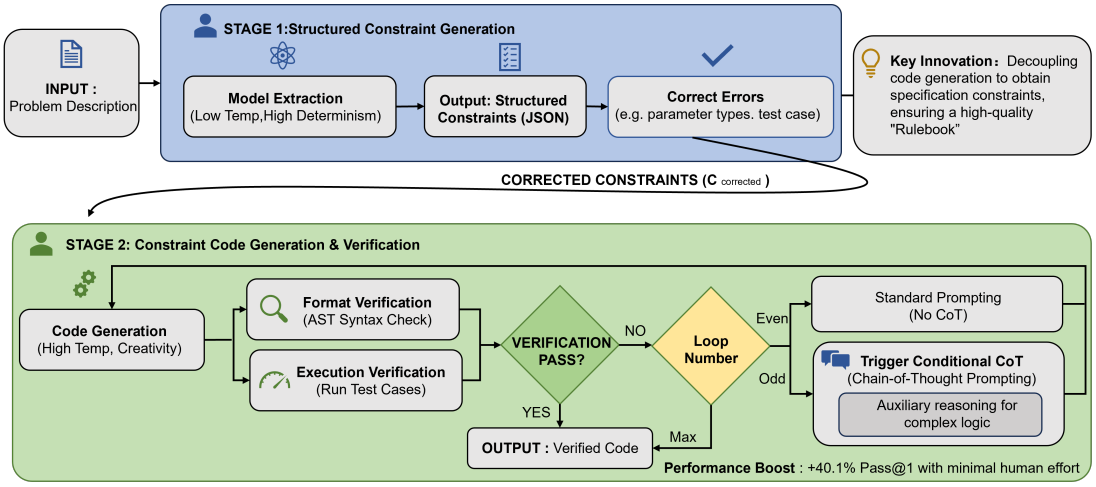

Fig. 1. CG-CG Framework Flowchart

Constraints are uniformly represented using a predefined JSON Schema, compressing open-domain information into a closed, verifiable space.

2) Generation Strategy and Robustness Design: We use low-temperature decoding (temperature = 0.2) and strict format constraints to guide JSON output. Considering that lightweight models may still have issues like bracket mismatch or missing fields in complex format generation, we introduce a rule-based legality check and lightweight post-processing module after the SCG stage to automatically correct format errors. Experimental statistics show that about 64.6% of generated constraints are directly usable; remaining cases involve hallucinations (26.2%), incomplete parsing (6.1%), or content errors.

D. Constrained Code Generation (CCG) Stage

In CCG, the model generates candidate code under the joint guidance of the original problem and the structured constraints from SCG. Unlike using constraints as natural language prompts, we treat structured constraints as an explicit set of rules, imposing strong constraints on the generation space, thereby significantly reducing uncertainty.

1) Prompt Construction and Decoding Strategy: The prompt consists of the original problem, structured constraints, and generation instructions. The instructions require only code output conforming to constraints, no extra text. To balance generation diversity and stability, we use a higher temperature (0.8) in the CCG stage, allowing the model to explore different implementations while remaining within the constraint boundaries.

2) External Dual Verification Mechanism: To avoid model self-evaluation, we introduce a model-independent external verification mechanism:

- Static syntax verification: using Python's built-in AST module to parse generated code; if parsing fails, the result is directly invalid.
- Functional verification: running the generated code in an isolated execution environment and verifying its functional correctness based on the examples extracted from the SCG stage.

Only code passing both verifications is considered valid. This "static blocking + dynamic falsification" strategy eliminates obviously incorrect candidates early, significantly reducing invalid iteration costs.

E. Iterative Optimization Mechanism

If generated code fails external verification, we trigger iterative optimization with maximum iterations N=5. This upper bound is set based on preliminary experiments showing that most generated programs pass verification within a small number of correction rounds, and performance gain saturates beyond 5 rounds. In even-numbered rounds, conditional Chain-of-Thought reasoning is introduced to guide the model in analyzing failure reasons; in odd-numbered rounds, standard re-sampling is performed to introduce new candidate solutions. This design ensures search diversity while leveraging limited reasoning assistance to improve error localization, achieving a high success rate in practice.

F. Algorithm Formal Description

Algorithm 1 provides a formal description. Input: problem description P, example set S, max iterations N. Output: verified code or final candidate. By explicitly separating constraint generation, code generation, and verification, the algorithm systematically models the process, transforming code generation into a controlled structured search.

IV. Experimental Design and Result Analysis

A. Experimental Setup

1) Dataset: We use HumanEval [19], containing 164 Python programming problems. Each problem provides a function signature, natural language description, and several input-output examples. The dataset covers various programming patterns such as string processing, array operations,

Algorithm 1 The CG-CG Framework

Input: Problem description P; Original test examples S_{orig}; Max iteration $N = 6$
Output: Verified code C^* or termination state code C_N

Stage 1: SCG
$C \leftarrow$ Call base LLM (temperature=0.2) to extract structured constraints (JSON format) from P
$S_C \leftarrow$ Perform JSON parsing validation on C, calibration possible (e.g., parameter types, example matching)

Stage 2: CCG
Initialize: current iteration round $i = 1$
while $i \leq N$ **do**
  if $i = 1$ **then**
    $C_i \leftarrow$ Call LLM (temperature=0.8) to generate initial candidate code
  else if i is even **then**
    $C_i \leftarrow$ Trigger conditional CoT: generate reasoning steps, then regenerate code
  else
    $C_i \leftarrow$ Re-execute CCG process
  end if
  Verification Barrier (Dual Verification)
  $Valid\_AST \leftarrow$ Perform static syntax check
  $Valid\_Exec \leftarrow$ Perform functional verification in sandbox environment using S_C
  if $Valid\_AST = True \wedge Valid\_Exec = True$ **then**
    Return C_i (Verification passed, output $C^* = C_i$)
  end if
  $i \leftarrow i + 1$
end while
Return C_N (Max iterations reached, output final candidate code)

mathematical calculations, and simple algorithm design, which can comprehensively reflect a model's capability in function-level code generation tasks. This dataset is chosen because: (1) its evaluation standard is clear and the Pass@k metric is widely accepted; (2) the explicitly provided example information offers a naturally suitable scenario for our structured constraint generation and external verification mechanisms; (3) numerous existing works report results on this dataset, facilitating horizontal comparative analysis.

2) Models and Baseline Methods: Primary model: Qwen2.5-1.5B-Instruct (1.5 billion parameters). This model has a relatively small parameter scale, possesses certain instruction-following and code generation capabilities, but still exhibits obvious performance bottlenecks in complex generation tasks, making it representative. To validate model-agnostic nature, we also test DeepSeek-Coder-1.3B-Instruct (1.3B) and StarCoder2-3B (3B). All models use officially released instruction or base versions without any additional fine-tuning. All experiments use a uniform prompt template to ensure comparability, avoiding task-specific prompt engineering that could interfere with results.

3) Evaluation Metrics:

1) **Pass@1**: proportion of problems where the model passes all hidden test cases in a single generation. We adopt this metric because it aligns more closely with real-world "use after single generation" needs and can more realistically reflect changes in generation process stability. To ensure rigor, we use a uniform prompt template across all experiments.
2) **Uncertainty suppression metrics**: To precisely evaluate the governance effect of structured constraints on reasoning uncertainty, we design quantitative metrics from three dimensions:

$$\eta = \frac{E_0 - E_c}{E_0}, \quad \eta_{\text{ast}} = \frac{A_0 - A_c}{A_0}, \quad \eta_{\text{exec}} = \frac{F_0 - F_c}{F_0} \tag{1}$$

where E_0 and E_c are overall invalid generation rates without/with constraints, A_0/A_c syntax error rates, F_0/F_c logical error rates.

B. Experimental Comparison

Table I compares zero-shot code generation enhancement methods on Qwen2.5-1.5B.

The results show significant differences among methods. Static constraint methods (Expert Prompt) actually reduce Pass@1 to 30.5%, indicating that weak natural language constraints alone are insufficient and may even interfere with generation. Methods relying on model self-evaluation like CoT and Self-Refine show substantial improvement (up to 78%) but are prone to accumulating hallucination errors over multiple iterations, as the model cannot reliably judge its own output correctness. ReAct+AST introduces external syntax checking but only verifies syntax, missing logical errors. The CG-CG framework achieves the best performance (80.3% Pass@1). The key lies in decoupling constraint generation from code generation and adopting non-model-dependent external dual verification, separating correctness judgment from model reasoning. This demonstrates that process decoupling and external verification can more systematically govern the uncertainty of lightweight models.

C. Cross-Model Generalization Experiment

To further evaluate the applicability of the CG-CG framework across different lightweight models, we replicate the same experimental procedure on models with different pre-training backgrounds and instruction capabilities. Table II shows the results.

CG-CG stably improves all models without degradation. The improvement is most significant for Qwen2.5-1.5B (40.2%→80.3%), due to its strong instruction-following ability, allowing it to fully utilize the constraint and verification mechanisms. For StarCoder2-3B and DeepSeek-Coder-1.3B, gains are smaller but positive, confirming that CG-CG acts as a process-level amplifier whose effectiveness depends on the base model's capability to understand and respond to constraints. However, stable improvements under zero-shot

TABLE I
PASS@1 RESULTS OF DIFFERENT METHODS ON THE HUMANEVAL DATASET

Model / Method	Pass@1 (%)	Improvement (%)	Key Characteristics
Native Baseline (Qwen2.5-1.5B)	40.2	–	Zero-shot, no constraints
Expert Prompt	30.5	-9.7	Weak constraints, natural language
CoT	70.5	+30.3	Reasoning + generation
Self-Refine (3-iter)	78.0	+37.8	LLM self-evaluation, prone to hallucination
ReAct+AST	70.4	+30.2	External tools, syntax check
CG-CG Framework	**80.3**	**+40.1**	Decoupling, structured constraints, external verification

TABLE II
EFFECTIVENESS OF THE CG-CG FRAMEWORK ON DIFFERENT LLMS

Base Model	Baseline Pass@1	CG-CG Pass@1 (%)	Improvement (%)	Relative Improvement (%)
Qwen2.5-1.5B	40.2	**80.3**	+40.1	99.8
StarCoder2-3b	25.1	39.0	+13.9	55.4
DeepSeek-Coder-1.3b-instruct	20.1	26.8	+6.7	33.3

conditions validate the framework's engineering portability and deployment value in resource-constrained scenarios.

D. Ablation Experiment

To further analyze the contribution of each component module in the CG-CG framework, we design a series of ablation experiments, gradually removing or replacing key components. Table III shows the results.

TABLE III
RESULTS OF KEY MODULES IN THE CG-CG FRAMEWORK

Ablation Group	Pass@1 (%)	Drop Relative(%)
Full Framework	**80.3**	–
No iteration, no external verification, no CoT	40.2	-40.1
No iterative verification (with CoT)	70.1	-10.2
No format validation	57.9	-22.4
No example verification	69.5	-10.8
No correct error	76.8	-3.2
No CoT	68.9	-11.4

Removing all mechanisms (no iteration, no external verification, no CoT) drops Pass@1 to baseline (40.2%), confirming that the framework's core advantage stems from the integrated "constrained generation + verification-driven iteration" mechanism rather than any single technique. Removing iterative verification while retaining CoT (70.1%) or removing example-driven validation (69.5%) both lead to about 10% declines, highlighting the necessity of verification feedback for error correction. Removing format validation significantly reduces Pass@1 to 57.9%, underscoring the critical role of explicit syntax and interface constraints (implicit modeling alone is insufficient for lightweight models). Omitting CoT decreases performance by 11.4%, indicating it benefits complex tasks but is non-essential. Notably, performance only drops by 3.2% without human assistance, demonstrating strong automation potential.

E. Result Discussion

To quantify the governance effect of structured constraints on model reasoning uncertainty, Fig. 2 presents suppression

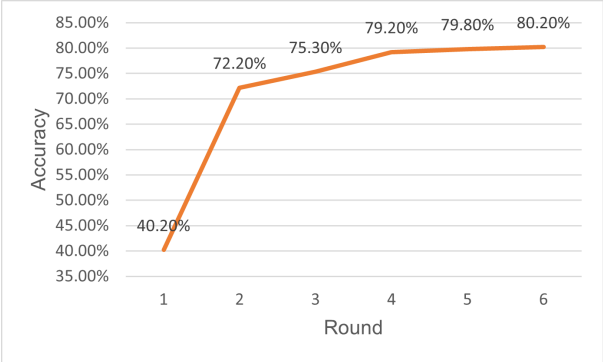

Fig. 2. Uncertainty Suppression Rates of the CG-CG Framework

rates from four dimensions. The overall uncertainty suppression efficiency η reaches 67.1%. The suppression effect on syntax uncertainty is most significant (83.3%), benefiting from the explicit constraints and format validation on function interfaces and data structures in the SCG stage, which directly blocks structural errors prone to occur in lightweight models. The suppression rate for logical uncertainty is 50.7%, indicating that problem-specific example constraints can guide the model to focus on core logical derivation, reducing issues like boundary condition omission. These quantitative results corroborate with the ablation findings (22.4% drop without format validation, 10.8% drop without example verification), validating the effectiveness of the design principle of "externalizing high-certainty information." This shows that structured constraints are not merely supplementary prompts but represent a systematic dimensionality reduction and uncertainty governance of the generation process.

F. Limitations Analysis

Despite significant experimental performance, the framework still has several limitations. First, the method relies on example information available for verification in the problem description; when examples are scarce or coverage is insufficient, the filtering capability of dynamic verification is limited. Second, the current framework primarily targets function-

level code generation; for program generation tasks requiring cross-file collaboration or complex state management, its structured constraint expression capability needs expansion. Third, external verification and multiple iterations inevitably increase inference latency. Although still within a controllable range compared to large-scale multi-sampling methods, further optimization is needed for latency-sensitive real-time systems.

G. Future Work Directions

Future research can extend the CG-CG framework in the following directions: (1) exploring the introduction of automatically generated test cases or symbolic execution techniques to enhance external verification capability when example information is insufficient; (2) extending structured constraints from function-level to module-level or project-level code generation tasks; (3) combining lightweight static analysis and program synthesis techniques to further improve the robustness and security of generated code.

V. Conclusion

This paper proposes an uncertainty governance framework for lightweight neural generation systems. Through structured constraint generation and external deterministic verification, it explicitly decouples and controls generation uncertainty. Experiments on code generation show that the framework systematically improves reliability. On HumanEval, it boosts Qwen2.5-1.5B-Instruct's Pass@1 from 40.2% to 80.3%. Its model-agnostic nature and engineering value are validated across multiple lightweight models. The CG-CG framework governs inherent model instability via process reconstruction and external determinism injection, offering a practical paradigm for deploying reliable neural generation in resource-constrained scenarios such as edge computing. It also supports research on lightweight resource agents and reliable software engineering.

VI. Acknowledgment

During the preparation of this manuscript, the authors used an AI-assisted tool (ChatGPT) to help correct syntax errors and optimize the implementation of the code framework. The AI tool was employed solely for code-level formatting and error correction, and did not contribute to the core algorithmic design or experimental analysis. The authors have reviewed and taken full responsibility for all code and experimental results presented in this paper.

References

[1] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A Survey on Large Language Models for Code Generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 35, no. 2, pp. 1–72, Feb. 2026.

[2] X. Hou et al., "Large Language Models for Software Engineering: A Systematic Literature Review," Apr. 10, 2024, *arXiv*: arXiv:2308.10620.

[3] N. Tao, A. Ventresque, V. Nallur, and T. Saber, "Enhancing Program Synthesis with Large Language Models Using Many-Objective Grammar-Guided Genetic Programming," *Algorithms*, vol. 17, no. 7, p. 287, Jul. 2024.

[4] X. Lian et al., "Uncovering Weaknesses in Neural Code Generation," Jul. 17, 2024, *arXiv*: arXiv:2407.09793.

[5] E. J. Hu et al., "LoRA: Low-Rank Adaptation of Large Language Models," Oct. 16, 2021, *arXiv*: arXiv:2106.09685.

[6] K. Zhang et al., "CodeDPO: Aligning Code Models with Self Generated and Verified Source Code," Jun. 03, 2025, *arXiv*: arXiv:2410.05605.

[7] H. Zeng, D. Jiang, H. Wang, P. Nie, X. Chen, and W. Chen, "ACECODER: Acing Coder RL via Automated Test-Case Synthesis," May 24, 2025, *arXiv*: arXiv:2502.01718.

[8] L. Zhong, R. Lindeborg, J. Zhang, J. J. Lim, and S.-H. Sun, "Hierarchical Neural Program Synthesis," Mar. 09, 2023, *arXiv*: arXiv:2303.06018.

[9] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Empowering Code Generation with OSS-Instruct," Jun. 07, 2024, *arXiv*: arXiv:2312.02120.

[10] K. Ellis et al., "DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Virtual Canada: ACM, Jun. 2021, pp. 835–850.

[11] Z. Huang et al., "Learning to Guarantee Type Correctness in Code Generation through Type-Guided Program Synthesis," Oct. 11, 2025, *arXiv*: arXiv:2510.10216.

[12] J. Wei et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," Jan. 10, 2023, *arXiv*: arXiv:2201.11903.

[13] A. Madaan et al., "SELF-REFINE: Iterative Refinement with Self-Feedback."

[14] S. Haug, C. Böhm, and D. Mayer, "Automated Code Generation and Validation for Software Components of Microcontrollers," Feb. 26, 2025, *arXiv*: arXiv:2502.18905.

[15] N. Mündler, J. He, H. Wang, K. Sen, D. Song, and M. Vechev, "Type-Constrained Code Generation with Language Models," *Proc. ACM Program. Lang.*, vol. 9, no. PLDI, pp. 601–626, Jun. 2025.

[16] Y. Ding, M. J. Min, G. Kaiser, and B. Ray, "CYCLE: Learning to Self-Refine the Code Generation," Mar. 27, 2024, *arXiv*: arXiv:2403.18746.

[17] N. Tao, A. Ventresque, V. Nallur, and T. Saber, "Enhancing Program Synthesis with Large Language Models Using Many-Objective Grammar-Guided Genetic Programming," *Algorithms*, vol. 17, no. 7, p. 287, Jul. 2024.

[18] S. Yao et al., "ReAct: Synergizing Reasoning and Acting in Language Models," Mar. 10, 2023, *arXiv*: arXiv:2210.03629.

[19] M. Chen et al., "Evaluating Large Language Models Trained on Code," Jul. 14, 2021, *arXiv*: arXiv:2107.03374.

[20] C. Ziems, W. Held, O. Shaikh, J. Chen, Z. Zhang, and D. Yang, "Can Large Language Models Transform Computational Social Science?," *Comput. Linguist.*, vol. 50, no. 1, pp. 237–291, Mar. 2024.

[21] E. Nijkamp et al., "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis," Feb. 27, 2023, *arXiv*: arXiv:2203.13474.

[22] W. Tong and T. Zhang, "CodeJudge: Evaluating Code Generation with Large Language Models," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Miami, Florida, USA, 2024, pp. 20032–20051.

[23] A. Gurfinkel and V. Ganesh, Eds., *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part II*, vol. 14682. Cham: Springer, 2024.

[24] M. Zhang, X. Shen, J. Cao, Z. Cui, and S. Jiang, "EdgeShard: Efficient LLM Inference via Collaborative Edge Computing," *IEEE Internet Things J.*, vol. 12, no. 10, pp. 13119–13131, May 2025.

[25] Z. Wang, J. Jiang, T. Qiu, H. Liu, X. Tang, and H. Yao, "Efficient Long CoT Reasoning in Small Language Models," Jun. 18, 2025, *arXiv*: arXiv:2505.18440.

[26] G. Feng et al., "How Numerical Precision Affects Arithmetical Reasoning Capabilities of LLMs," Jun. 21, 2025, *arXiv*: arXiv:2410.13857.

[27] I. Ma, A. K. Martins, and C. V. Lopes, "Integrating AI Tutors in a Programming Course," Jul. 14, 2024, *arXiv*: arXiv:2407.15718.

[28] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is Your Code Generated by ChatGPT Really Correct?".

[29] A. Hosseini, A. Sordoni, D. Toyama, A. Courville, and R. Agarwal, "Not All LLM Reasoners Are Created Equal," Oct. 02, 2024, *arXiv*: arXiv:2410.01748.

[30] R. Shin, M. Allamanis, M. Brockschmidt, and O. Polozov, "Program Synthesis and Semantic Parsing with Learned Code Idioms," Nov. 05, 2019, *arXiv*: arXiv:1906.10816.