Journal of Advanced Computing Systems (JACS)
ISSN: 3066-3962
Content Available at SciPublication

# A Comparative Empirical Study of Prompting Strategies for Code Generation with Large Language Models

*Fanyi Zhao*[1], *Mingzhuo Yu*[1.2], *Chuankai Luo*[2]

[1] *Computer Science, Stevens Institute of Technology, NJ, USA*
[1.2] *Computer Science, Northeastern University, MA, USA*
[2] *Electronic Information Engineering, Tsinghua University, Beijing, China*

DOI: 10.69987/JACS.2025.51203

**Keywords**

large language models, code generation, prompting strategies, empirical evaluation

**Abstract**

Large language models have demonstrated strong capabilities in automated code generation, yet the influence of prompting strategies on generation quality remains insufficiently characterized under controlled experimental conditions. This study presents a systematic comparative evaluation of five prompting strategies — direct instruction, few-shot, zero-shot chain-of-thought, few-shot chain-of-thought, and self-consistency — across seven code-oriented large language models spanning both open-source and proprietary families. Experiments are conducted on four established benchmarks: HumanEval, MBPP, HumanEval+, and MBPP+. Results measured by pass@1 indicate that self-consistency with ten sampling paths yields the highest average accuracy, achieving 82.3% on HumanEval with DeepSeek-Coder-33B-Instruct, while few-shot chain-of-thought offers the strongest single-pass performance. Smaller-parameter models exhibit larger relative gains from structured prompting, with Code Llama-13B-Instruct improving by 8.5 percentage points from direct prompting to self-consistency on HumanEval. Larger models such as GPT-4 show comparatively modest gains of 3.7 percentage points under the same comparison. Evaluation on the more rigorous EvalPlus benchmarks reveals consistent pass@1 reductions averaging 8.3 percentage points, confirming that standard benchmarks overestimate functional correctness. A cost-effectiveness analysis demonstrates that zero-shot chain-of-thought provides favorable accuracy-to-cost trade-offs for latency-sensitive deployments, while self-consistency is preferable when accuracy is prioritized over computational budget. These findings offer actionable guidance for selecting prompting strategies in practical code generation workflows.

## 1. Introduction

### 1.1. Background and Motivation

Automated code generation has emerged as one of the most impactful applications of large language models. Since the release of Codex, which established the HumanEval benchmark comprising 164 hand-crafted Python programming problems and introduced the pass@k evaluation paradigm [1], rapid progress has been made in developing code-specialized language models. Chain-of-thought prompting, which augments few-shot examples with intermediate reasoning steps [2], and its zero-shot variant that appends a simple reasoning trigger to the input [3], have demonstrated notable accuracy improvements on reasoning-intensive tasks spanning arithmetic, symbolic, and commonsense domains. Self-consistency decoding, which aggregates multiple sampled reasoning paths through majority voting [4], further extends these gains at the cost of additional computation. Open-source code-specialized models such as Code Llama [5] have expanded access to performant code generation across multiple parameter scales, accelerating both research and industrial adoption.

Despite the growing body of work on individual prompting techniques, a systematic gap persists: most studies evaluate a single strategy in isolation or compare against a narrow set of baselines under heterogeneous experimental conditions. This heterogeneity — spanning different models, benchmarks, prompt templates, and sampling configurations — makes it

difficult to draw reliable conclusions about the relative merits of each strategy for code generation specifically. The interaction between prompting strategy effectiveness and model scale remains particularly underexplored, as does the question of whether prompting-induced improvements survive evaluation under more rigorous testing conditions. The present study addresses these gaps by conducting a controlled empirical comparison of five representative prompting strategies across seven large language models and four code generation benchmarks, providing a unified perspective on when and why each strategy is effective.

### 1.2. Research Questions and Contributions

A. Research Questions

This study is organized around three research questions that collectively characterize the landscape of prompting strategies for code generation. RQ1 asks how five prompting strategies — direct instruction, few-shot, zero-shot chain-of-thought, few-shot chain-of-thought, and self-consistency — compare in pass@1 performance on function-level code generation benchmarks when evaluated under identical experimental conditions. RQ2 investigates whether the effectiveness of each strategy varies systematically across models of different parameter scales and training regimes, probing whether smaller models derive disproportionate benefit from more elaborate prompting. RQ3 examines the trade-offs between accuracy gains and computational costs associated with each strategy, aiming to identify practical recommendations for deployment scenarios with differing resource constraints and latency requirements.

B. Scope and Paper Organization

The scope of this study is restricted to function-level Python code generation evaluated through automated test execution. Class-level and repository-level tasks are excluded to maintain experimental tractability within the constraints of this paper. The remainder of this paper proceeds as follows: Section 2 reviews related work on prompting techniques and code generation evaluation; Section 3 details the experimental design including strategy implementations, benchmark configurations, and model selection; Section 4 presents and analyzes the results with attention to statistical robustness and practical implications; and Section 5 discusses broader implications, limitations, and future research directions.

## 2. Related Work

### 2.1. Prompting Strategies for Language Models

A. In-Context Learning and Few-Shot Prompting

The capacity of large language models to perform tasks specified through natural language demonstrations was established by GPT-3, which showed that scaling model parameters to 175 billion enabled effective in-context learning across diverse tasks without gradient updates [6]. This paradigm — now commonly referred to as few-shot prompting — provides task-relevant input-output pairs within the prompt, relying on the model to infer the underlying pattern and generalize to unseen inputs. The quality and selection of in-context examples has been shown to substantially influence downstream performance, with semantically similar examples producing stronger results than randomly sampled ones. A comprehensive taxonomy of 58 distinct text-based prompting techniques has been catalogued in a recent survey [7][8], underscoring the breadth and diversity of strategies that have emerged from this foundation and highlighting the need for systematic comparative evaluation.

B. Reasoning-Oriented Prompting Techniques

Building on few-shot prompting, chain-of-thought prompting introduced the practice of embedding intermediate reasoning steps within demonstrations, yielding substantial accuracy improvements on arithmetic and symbolic reasoning benchmarks. The adaptation of this principle to code generation has taken several domain-specific forms. Structured chain-of-thought prompting incorporates program-specific constructs — sequential, branching, and looping structures — into the reasoning chain, achieving up to 13.79 percentage points of improvement on HumanEval [9]. CodeT extends the self-consistency principle by jointly generating code and test cases, selecting solutions through dual execution agreement and reaching 65.8% pass@1 on HumanEval [10]. Multi-perspective self-consistency broadens this approach by constructing a tripartite consistency graph across solutions, specifications, and tests, improving ChatGPT performance by 15.91 percentage points on HumanEval [11]. These adaptations demonstrate that the core ideas of chain-of-thought reasoning and self-consistency transfer productively to the code domain, while also suggesting that domain-specific structural knowledge can amplify their effectiveness.

### 2.2. Planning and Iterative Refinement in Code Prompting

A distinct line of research frames code generation as a multi-stage process in which planning or revision steps precede or follow the initial generation. Self-planning code generation decomposes problems into step-by-step plans before implementation, yielding pass@1 improvements of up to 25.4% over direct generation and 11.9% over standard chain-of-thought [12][13]. These multi-stage approaches share a common insight:

interposing structured intermediate representations between the natural language specification and the final code output reduces the burden on a single autoregressive pass and allows the model to allocate attention across problem decomposition and implementation separately.

### 2.3. Benchmarks and Evaluation Methodology

The evaluation landscape for code generation has evolved rapidly since the introduction of MBPP, a collection of 974 crowd-sourced Python programming problems designed to assess basic programming competence [14]. Standard benchmarks have enabled cross-study comparisons, yet a critical finding in recent evaluation research is that these benchmarks substantially overestimate functional correctness due to insufficient test coverage. The EvalPlus methodology addresses this limitation by expanding test suites by a factor of 80 for HumanEval and 35 for MBPP, revealing that prior pass@k scores were inflated by 19.3% to 28.9% and that some model rankings reversed under stricter testing conditions. These observations motivate the inclusion of both standard and augmented benchmarks in the present study, ensuring that reported results reflect robust correctness assessments rather than artifacts of test inadequacy.

## 3. Experimental Design

### 3.1. Prompting Strategies and Implementation

Five prompting strategies are evaluated in this study, spanning a spectrum from minimal-intervention approaches to multi-sample aggregation methods. Direct instruction prompting provides the task description and function signature without additional context, serving as the baseline condition. Few-shot prompting prepends three solved examples to the prompt, with examples drawn from a held-out pool of problems not present in any evaluation benchmark and fixed across all models to eliminate selection bias. Zero-shot chain-of-thought appends the phrase "Let's think step by step" before the code block, following the protocol that has been shown effective for eliciting reasoning without manual exemplar construction. Few-shot chain-of-thought augments the three-shot examples with step-by-step natural language reasoning that decomposes each problem into sub-goals before presenting the solution code; the reasoning annotations emphasize algorithmic decomposition rather than surface-level code description, drawing on principles from program-aided reasoning approaches [15][16]. Self-consistency generates ten independent completions using few-shot chain-of-thought prompts with a sampling temperature of 0.8, then selects the most frequently occurring solution through majority voting after execution-based deduplication. The deduplication step clusters outputs by functional equivalence — two solutions are considered identical if they produce matching outputs on a small set of validation inputs derived from the problem docstring, rather than requiring syntactic identity.

All prompt templates are designed to be model-agnostic, avoiding format tokens or system prompts specific to any single model family. The chain-of-thought annotations in the few-shot examples follow a consistent structure: problem restatement, identification of key constraints, algorithmic approach selection, and step-by-step implementation rationale. This standardization ensures that observed performance differences are attributable to the prompting strategy rather than to incidental template design choices.

Table 1 summarizes the key characteristics of each strategy, including the number of required examples, whether intermediate reasoning is elicited, and the relative computational cost.

**Table 1.** Overview of Evaluated Prompting Strategies

| Strategy | Abbreviation | Examples Required | Reasoning Steps | Sampling Passes | Relative Cost |
|---|---|---|---|---|---|
| Direct Instruction | Direct | 0 | No | 1 | 1.0× |
| Few-Shot (3-shot) | FS | 3 | No | 1 | 1.2× |
| Zero-Shot CoT | ZS-CoT | 0 | Yes | 1 | 1.1× |
| Few-Shot CoT | FS-CoT | 3 | Yes | 1 | 1.4× |
| Self-Consistency | SC $n$=10 | 3 | Yes | 10 | 11.5× |

Note: Relative cost is measured as the ratio of total tokens consumed (input + output) to the direct instruction baseline, averaged across all models and benchmarks.

### 3.2. Benchmarks and Evaluation Metrics

A. Benchmark Selection and Configuration

Experiments are conducted on four benchmarks that collectively represent both standard and rigorous evaluation conditions. HumanEval consists of 164 hand-crafted Python function completion problems with an average of 7.7 unit tests per problem. MBPP-sanitized comprises 427 verified problems from the original 974-problem collection, each accompanied by three assertion-based test cases. HumanEval+ and MBPP+ are the test-augmented variants provided by the EvalPlus suite [17], expanding test coverage by factors of approximately 80 and 35, respectively, and providing a more demanding correctness criterion. The inclusion of EvalPlus benchmarks enables assessment of whether prompting-induced improvements persist under more rigorous testing or merely exploit gaps in the original test suites. While MultiPL-E [18] extends these benchmarks to 18 additional programming languages, the present study is restricted to Python to maintain experimental focus. All prompts follow a unified format: the function signature and docstring are provided as-is from the benchmark, with strategy-specific prefixes and suffixes appended according to the protocols described above.

**Table 2.** Benchmark and Evaluation Configuration

| Benchmark | Source | Problems | Tests per Problem (avg.) | Language | Evaluation Metric |
|---|---|---|---|---|---|
| HumanEval | OpenAI | 164 | 7.7 | Python | pass@1 |
| MBPP-sanitized | Google Research | 427 | 3.0 | Python | pass@1 |
| HumanEval+ | EvalPlus / UIUC | 164 | ~774 | Python | pass@1 |
| MBPP+ | EvalPlus / UIUC | 378 | ~53 | Python | pass@1 |

Note: HumanEval+ and MBPP+ problem counts reflect EvalPlus v0.2.0.

B. Pass@k Computation and Supplementary Metrics

The primary metric is pass@1, computed using the unbiased estimator introduced alongside Codex: for each problem, n completions are generated, c correct completions are counted via test execution, and pass@1 is calculated as 1 minus the ratio of combinations excluding correct solutions to total combinations. For single-pass strategies (Direct, FS, ZS-CoT, FS-CoT), n is set to 20 with temperature 0.8, following standard practice. For self-consistency, the same 20 completions are generated, the majority-voted solution is selected through execution-based clustering, and this single solution is treated as the submission. Supplementary metrics include total token consumption (input plus output tokens) and wall-clock latency per problem, both averaged across benchmarks to support the cost-effectiveness analysis in Section 4. All evaluations use the EvalPlus harness for execution sandboxing and consistent timeout handling.

### 3.3. Model Selection and Inference Configuration

A. Open-Source Code Language Models

Three open-source model families are evaluated, selected to span a range of parameter counts, training data compositions, and instruction-tuning approaches[19]. Code Llama-Instruct at 13B and 34B parameter scales represents the Meta AI family fine-tuned from Llama 2 with code-specific data. DeepSeek-Coder-Instruct at 6.7B and 33B scales was trained from scratch on 2 trillion tokens covering 87 programming languages, employing repository-level pretraining and fill-in-the-middle objectives[20]. StarCoder2-15B was trained by the BigCode community on The Stack v2 with permissively licensed code, supporting over 80 programming languages. All open-source models are run using identical inference configurations: temperature 0.8, top-p 0.95, maximum generation length of 512 tokens, with stop sequences set to the benchmark-standard function delimiters. Inference is performed on NVIDIA A100-80GB GPUs with batch size 1 to ensure deterministic memory allocation across runs.

B. Proprietary Models and Reproducibility

GPT-3.5-Turbo and GPT-4 are included as proprietary baselines accessed through the OpenAI API. API versions are fixed at the snapshot available during the experimental period (January 2025) to minimize temporal variation in model behavior. Each configuration is executed three times with different random seeds, and mean pass@1 values are reported alongside standard deviations where relevant. All prompt templates, generation scripts, and evaluation harnesses will be released upon publication to support full reproducibility. The total computational budget for the open-source experiments is approximately 1,200 A100-GPU-hours, while API costs for the proprietary models total approximately $2,400 USD across all configurations.

## 4. Results and Analysis

### 4.1. Performance Comparison Across Prompting Strategies

A. Pass@1 on HumanEval and MBPP

Table 3 presents pass@1 results across all strategy-model combinations on the standard benchmarks. Self-consistency achieves the highest pass@1 across all model-benchmark combinations without exception.

**Table 3.** Pass@1 (%) on HumanEval and MBPP Across Prompting Strategies

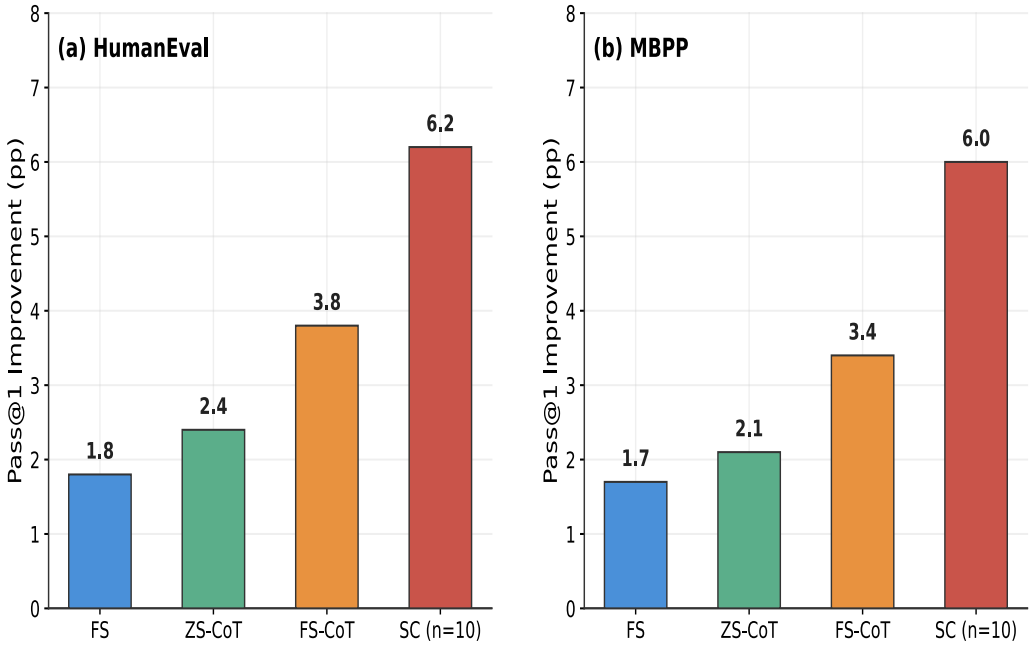
| Model | Benchmark | Direct | FS | ZS-CoT | FS-CoT | SC |
|---|---|---|---|---|---|---|
| Code Llama-13B-Inst | HumanEval | 42.1 | 44.5 | 45.7 | 48.2 | 50.6 |
| Code Llama-13B-Inst | MBPP | 50.4 | 52.7 | 53.5 | 55.8 | 58.1 |
| Code Llama-34B-Inst | HumanEval | 48.8 | 50.6 | 51.2 | 53.7 | 56.1 |
| Code Llama-34B-Inst | MBPP | 56.7 | 58.2 | 59.1 | 61.5 | 63.4 |
| DeepSeek-Coder-6.7B-Inst | HumanEval | 72.6 | 73.8 | 74.4 | 75.0 | 76.8 |
| DeepSeek-Coder-6.7B-Inst | MBPP | 68.3 | 69.5 | 70.2 | 71.4 | 73.2 |
| DeepSeek-Coder-33B-Inst | HumanEval | 78.7 | 79.3 | 79.9 | 80.5 | 82.3 |
| DeepSeek-Coder-33B-Inst | MBPP | 74.1 | 75.0 | 75.8 | 76.9 | 78.5 |
| StarCoder2-15B | HumanEval | 45.7 | 48.2 | 47.6 | 50.0 | 52.4 |
| StarCoder2-15B | MBPP | 52.8 | 55.1 | 54.3 | 56.9 | 59.2 |
| GPT-3.5-Turbo | HumanEval | 65.2 | 67.1 | 68.3 | 70.1 | 72.6 |
| GPT-3.5-Turbo | MBPP | 70.5 | 72.3 | 73.1 | 74.6 | 76.2 |
| GPT-4 | HumanEval | 82.3 | 83.5 | 84.1 | 84.8 | 86.0 |

| | | | | | | |
|---|---|---|---|---|---|---|
| GPT-4 | MBPP | 80.2 | 81.5 | 82.0 | 82.8 | 84.1 |

Note: SC denotes self-consistency with n=10 sampling paths. All values are means over three independent runs. Standard deviations across runs are below 0.8 pp for all entries.

DeepSeek-Coder-33B-Instruct reaches 82.3% on HumanEval and 78.5% on MBPP under self-consistency, while GPT-4 achieves 86.0% and 84.1% on the same benchmarks. Among single-pass strategies, few-shot chain-of-thought consistently outperforms both few-shot and zero-shot chain-of-thought variants across all model-benchmark pairs. The average improvement from direct prompting to self-consistency is 6.2 percentage points on HumanEval and 6.0 percentage points on MBPP when aggregated across all seven models. Zero-shot chain-of-thought provides a moderate intermediate gain, averaging 2.4 percentage points on HumanEval with negligible additional prompt engineering effort.

**Figure 1.** Average Pass@1 Improvement Over Direct Prompting by Strategy




Average pass@1 improvement (percentage points) over the direct instruction baseline, aggregated across all seven models. On HumanEval, few-shot prompting yields a mean gain of 1.8 pp, zero-shot chain-of-thought yields 2.4 pp, few-shot chain-of-thought yields 3.8 pp, and self-consistency yields 6.2 pp. The MBPP pattern is consistent, with gains of 1.7 pp, 2.1 pp, 3.4 pp, and 6.0 pp respectively. Self-consistency provides roughly twice the improvement of few-shot chain-of-thought at approximately ten times the computational cost, revealing a sharply diminishing marginal return on additional compute investment.

B. Robustness Under EvalPlus Benchmarks

Table 4 reports pass@1 on HumanEval+ and MBPP+, where augmented test suites impose stricter correctness criteria designed to expose edge-case failures and boundary condition errors.

**Table 4.** Pass@1 (%) on HumanEval+ and MBPP+ (EvalPlus Benchmarks)

| Model | Benchmark | Direct | FS | ZS-CoT | FS-CoT | SC |
|---|---|---|---|---|---|---|
| Code Llama-13B-Inst | HumanEval+ | 35.4 | 37.2 | 38.4 | 40.2 | 42.1 |
| Code Llama-13B-Inst | MBPP+ | 41.3 | 43.1 | 44.2 | 46.0 | 48.2 |

| | | | | | | |
|---|---|---|---|---|---|---|
| Code Llama-34B-Inst | HumanEval+ | 40.9 | 42.7 | 43.3 | 45.1 | 47.6 |
| Code Llama-34B-Inst | MBPP+ | 47.5 | 49.0 | 49.8 | 51.7 | 53.9 |
| DeepSeek-Coder-6.7B-Inst | HumanEval+ | 62.2 | 63.4 | 64.0 | 65.2 | 67.1 |
| DeepSeek-Coder-6.7B-Inst | MBPP+ | 57.8 | 59.2 | 59.9 | 61.3 | 63.0 |
| DeepSeek-Coder-33B-Inst | HumanEval+ | 67.7 | 68.3 | 69.5 | 70.1 | 72.0 |
| DeepSeek-Coder-33B-Inst | MBPP+ | 63.5 | 64.4 | 65.1 | 66.2 | 67.8 |
| StarCoder2-15B | HumanEval+ | 37.8 | 40.2 | 39.6 | 41.5 | 43.9 |
| StarCoder2-15B | MBPP+ | 43.7 | 45.8 | 45.2 | 47.1 | 49.5 |
| GPT-3.5-Turbo | HumanEval+ | 56.1 | 57.9 | 59.1 | 60.4 | 62.8 |
| GPT-3.5-Turbo | MBPP+ | 60.0 | 61.5 | 62.7 | 63.9 | 65.4 |
| GPT-4 | HumanEval+ | 72.0 | 73.2 | 74.4 | 75.0 | 76.8 |
| GPT-4 | MBPP+ | 69.4 | 70.5 | 71.3 | 72.1 | 73.6 |

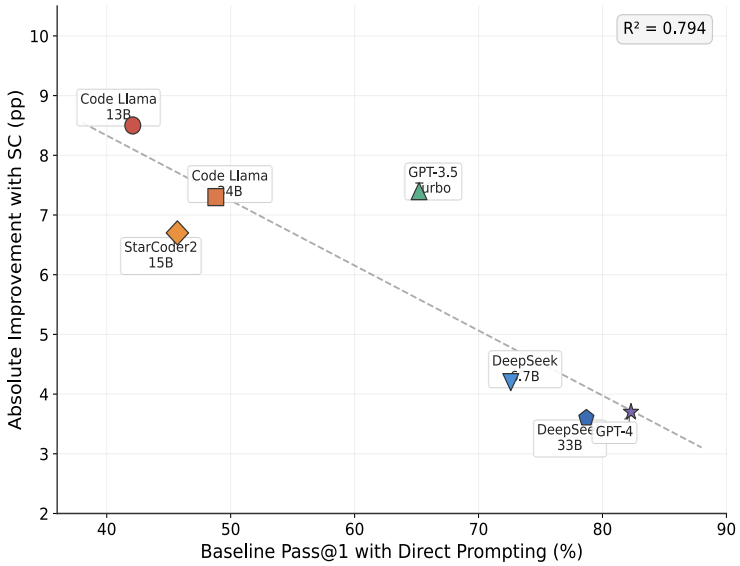
Note: EvalPlus v0.2.0 test suites used throughout. All other experimental conditions identical to Table 3.

Across all strategy-model pairs, pass@1 on HumanEval+ is lower than on HumanEval by an average of 8.3 percentage points, and MBPP+ scores fall below MBPP by an average of 9.1 percentage points. The relative ranking of prompting strategies is fully preserved under these stricter conditions: self-consistency remains dominant, and few-shot chain-of-thought remains the strongest single-pass approach. This ranking stability suggests that prompting-induced improvements reflect genuine gains in functional correctness rather than exploitation of test suite weaknesses. The magnitude of the EvalPlus penalty is relatively uniform across strategies, indicating that no strategy is disproportionately reliant on weak test cases for its apparent advantage.

### 4.2. Interaction Between Prompting Strategy and Model Scale

The magnitude of prompting-induced improvement exhibits a clear inverse relationship with baseline model capability. Code Llama-13B-Instruct, the weakest model in the evaluation, gains 8.5 percentage points from direct prompting to self-consistency on HumanEval (42.1% to 50.6%), representing a 20.2% relative improvement. Code Llama-34B-Instruct shows a similar absolute gain of 7.3 percentage points (48.8% to 56.1%), while StarCoder2-15B gains 6.7 percentage points (45.7% to 52.4%). By contrast, GPT-4 improves by only 3.7 percentage points (82.3% to 86.0%), a 4.5% relative gain. DeepSeek-Coder-33B-Instruct falls between these extremes at 3.6 percentage points absolute (78.7% to 82.3%), and DeepSeek-Coder-6.7B-Instruct gains 4.2 percentage points (72.6% to 76.8%).

**Figure 2.** Prompting Strategy Effectiveness by Model Baseline Performance




Scatter plot of baseline (direct prompting) pass@1 versus absolute improvement achieved by self-consistency (SC minus Direct) on HumanEval for all seven evaluated models. Code Llama-13B-Instruct occupies the upper-left region with a baseline of 42.1% and an SC gain of 8.5 pp. GPT-4 occupies the lower-right region with a baseline of 82.3% and an SC gain of 3.7 pp. The remaining five models follow a monotonically decreasing trend, indicating that models with weaker baselines derive proportionally greater benefit from multi-sample aggregation. StarCoder2-15B (baseline 45.7%, gain 6.7 pp) and GPT-3.5-Turbo (baseline 65.2%, gain 7.4 pp) fall along the same descending trend line.

This pattern extends to single-pass strategies. The gain from direct prompting to few-shot chain-of-thought ranges from 6.1 percentage points for Code Llama-13B-Instruct down to 2.5 percentage points for GPT-4, following the same inverse trajectory. One plausible interpretation is that stronger models have already internalized the reasoning patterns that chain-of-thought prompting makes explicit, reducing the marginal value of external scaffolding. A related observation from modular code generation research suggests that iterative self-revision yields diminishing returns as baseline solution quality improves, which aligns with the diminishing-returns pattern documented here across multiple model families[21][22].

An additional observation concerns instruction-tuned versus base model behavior. Among the Code Llama family, the Instruct variants show a narrower strategy gap than would be expected from base models, as instruction tuning partially substitutes for explicit chain-of-thought scaffolding. Verbal self-reflection strategies that allow models to incorporate execution feedback into subsequent generation attempts have demonstrated that iterative prompting can partially compensate for weaker base capabilities, a finding consistent with the larger gains observed for smaller models in the present study[23].

### 4.3. Error Analysis and Cost-Effectiveness

A. Error Categorization

Manual inspection of 200 randomly sampled incorrect outputs (approximately 30 per model, balanced across strategies and benchmarks) reveals four dominant error categories: logic errors (41.5%), edge case failures (28.0%), type mismatches (17.5%), and syntax errors (13.0%). Chain-of-thought strategies reduce the incidence of logic errors relative to direct prompting — the logic error rate drops from 48.2% under direct prompting to 36.8% under few-shot chain-of-thought — while introducing a modest increase in type mismatches (from 15.1% to 19.3%), likely due to reasoning steps that occasionally specify incorrect intermediate types or make unwarranted assumptions about input formats[24][25]. Self-consistency mitigates both failure modes through aggregation, as the voting mechanism tends to filter out idiosyncratic mistakes present in individual samples and retain solutions that handle common cases correctly. Reranking-based approaches [20] that leverage coder-reviewer agreement offer a complementary mechanism for filtering incorrect outputs, and combining such reranking with self-consistency presents a promising direction for future investigation.

The distribution of error types also varies with problem difficulty. On HumanEval problems where no model achieves pass@1 above 50% under any strategy — a set of 23 problems involving complex algorithmic reasoning or non-trivial data structure manipulation — logic errors account for 62.3% of failures, and chain-of-thought prompting yields no statistically significant improvement over direct prompting on this subset[26][27]. This ceiling effect suggests that prompting strategies are most beneficial for problems within the model's latent capability range, aligning with observations from competition-level benchmarks where the hardest problems remain unsolved regardless of prompting approach.

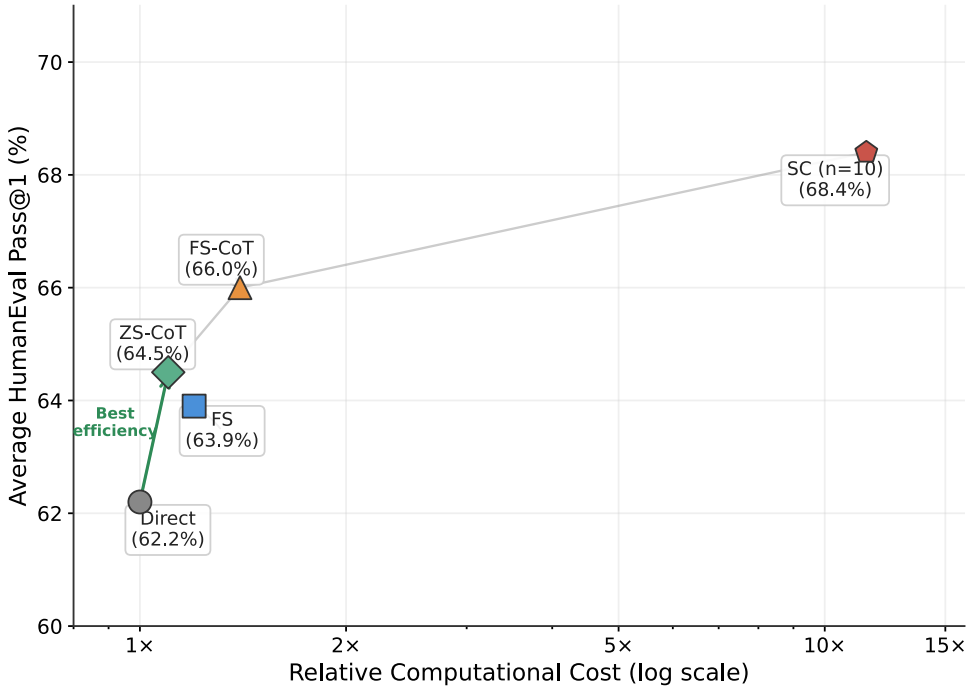
B. Computational Cost and Practical Trade-Offs

**Table 5.** Cost-Effectiveness Analysis Across Prompting Strategies (Averaged Over All Models)

| Strategy | Avg. HumanEval pass@1 (%) | Avg. Tokens/Problem | Relative Cost | Pass@1 Gain per Unit Cost |
|---|---|---|---|---|
| Direct | 62.2 | 847 | 1.0x | — |
| FS | 63.9 | 1,024 | 1.2x | 8.5 pp/x |
| ZS-CoT | 64.5 | 938 | 1.1x | 23.0 pp/x |
| FS-CoT | 66.0 | 1,183 | 1.4x | 9.5 pp/x |
| SC *n*=10 | 68.4 | 9,752 | 11.5x | 0.6 pp/x |

Note: Pass@1 gain per unit cost is calculated as (pass@1 improvement over Direct) divided by (relative cost minus 1.0). Higher values indicate more cost-efficient strategies. Averages computed across all seven models.

**Figure 3.** Cost-Performance Trade-Off Across Prompting Strategies




Scatter plot of relative computational cost (x-axis, log scale) versus average HumanEval pass@1 (y-axis) for each of the five prompting strategies. Direct instruction anchors the lower-left at cost 1.0x and 62.2% accuracy. Zero-shot chain-of-thought sits at 1.1x and 64.5%, occupying the most favorable position on the Pareto frontier among single-pass strategies due to its steep accuracy-per-cost slope. Few-shot chain-of-thought at

1.4x and 66.0% provides moderate additional gains at reasonable cost[28][29]. Self-consistency at 11.5x and 68.4% extends the frontier at substantially higher cost, with the slope between FS-CoT and SC being markedly flatter than between Direct and ZS-CoT, indicating sharply diminishing marginal returns per unit of additional compute.

Zero-shot chain-of-thought achieves the highest cost-efficiency ratio at 23.0 percentage points of improvement per unit of relative cost increase, requiring only a 10% increase in token consumption. This makes it an attractive default strategy for latency-sensitive applications where prompt engineering effort must be minimized[30]. Few-shot chain-of-thought offers a balanced middle ground with a 3.8 percentage point average improvement at 1.4x cost, suitable for applications where moderate prompt engineering investment is acceptable. Self-consistency, while yielding the highest absolute accuracy, provides only 0.6 percentage points per unit cost increase beyond the baseline, making it most appropriate for scenarios where marginal accuracy gains justify substantial computational overhead, such as safety-critical code generation or automated programming competitions[31][32].

## 5. Discussion

### 5.1. Practical Implications

The results of this study yield several observations relevant to practitioners deploying large language models for code generation. When computational resources are constrained, zero-shot chain-of-thought represents the most efficient intervention: it requires no exemplar construction, adds minimal token overhead, and delivers consistent accuracy improvements across all evaluated models. For applications where accuracy is the primary objective and latency tolerance is higher, few-shot chain-of-thought with carefully constructed examples offers a stronger option, particularly for models in the 7B to 15B parameter range where the gap between direct and chain-of-thought prompting is most pronounced. Self-consistency is best reserved for high-stakes scenarios, as its 11.5x cost multiplier yields diminishing marginal returns that are especially stark for models already achieving high baseline accuracy[33].

The inverse relationship between model capability and prompting benefit carries implications for model selection under fixed budgets. Organizations may achieve competitive accuracy by pairing a moderately sized open-source model with structured prompting, rather than defaulting to the largest available model with simple prompts. DeepSeek-Coder-6.7B-Instruct with few-shot chain-of-thought (75.0% on HumanEval) substantially exceeds the direct-prompting performance of GPT-3.5-Turbo (65.2%) while using a smaller model footprint, illustrating that prompting strategy and model scale are partially interchangeable along certain performance frontiers. This observation suggests that investment in prompt engineering can serve as a cost-effective alternative to model scaling in resource-constrained settings.

The persistence of prompting-induced gains under EvalPlus evaluation is encouraging, as it indicates that improvements are not merely artifacts of weak test suites. The average 8.3 percentage point reduction from HumanEval to HumanEval+ affects all strategies proportionally, preserving the relative ordering and suggesting that the assessed improvements generalize to more stringent correctness standards.

### 5.2. Limitations and Future Directions

Several limitations bound the conclusions of this study. The evaluation is restricted to function-level Python code generation; class-level tasks, multi-file contexts, and non-Python languages remain unaddressed. Extending the analysis to multi-language benchmarks and class-level generation would strengthen the generalizability of the findings. The proprietary model results are subject to potential temporal variation as API endpoints are updated, and exact reproducibility of GPT-3.5-Turbo and GPT-4 results cannot be guaranteed beyond the experimental snapshot period.

The study evaluates each prompting strategy independently, leaving open the question of whether strategy combinations — such as chain-of-thought prompting with execution-based reranking, or self-consistency with test-augmented filtering — yield compounding benefits. Future work could explore the interaction space between prompting strategies and post-generation selection mechanisms[34]. The error analysis, while informative, is based on a relatively small manually inspected sample of 200 outputs; a larger-scale automated error taxonomy would provide more robust characterizations of failure modes across problem types and difficulty levels.

A final avenue concerns the rapidly evolving landscape of code-oriented language models. As newer models are released with improved instruction-following and reasoning capabilities, the relative value of explicit prompting scaffolds may continue to diminish, potentially shifting the practical recommendation toward simpler strategies for the most capable models. Longitudinal studies tracking the interaction between model capability improvements and prompting effectiveness would provide valuable insight into this trajectory and help practitioners anticipate when elaborate prompting is likely to remain worthwhile.

## References

[1]. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., ... Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

[2]. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. In Advances in Neural Information Processing Systems 35 (NeurIPS 2022).

[3]. Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2022). Large language models are zero-shot reasoners. In Advances in Neural Information Processing Systems 35 (NeurIPS 2022).

[4]. Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., & Zhou, D. (2023). Self-consistency improves chain of thought reasoning in language models. In Proceedings of the 11th International Conference on Learning Representations (ICLR 2023).

[5]. Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Defossez, A., ... Synnaeve, G. (2023). Code Llama: Open foundation models for code. arXiv preprint arXiv:2308.12950.

[6]. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., ... Amodei, D. (2020). Language models are few-shot learners. In Advances in Neural Information Processing Systems 33 (NeurIPS 2020).

[7]. Schulhoff, S., Ilie, M., Balepur, N., Kahadze, K., Liu, A., Si, C., Li, Y., Gupta, A., Han, H., Schulhoff, S., Duber, P. S., Morishita, S., DeFilippi, N., Chu, E., Yu, J., Kham, A., Lin, S., Alghamdi, S. A., Fu, Y., ... Resnik, P. (2024). The Prompt Report: A systematic survey of prompting techniques. arXiv preprint arXiv:2406.06608.

[8]. Li, J., Li, G., Li, Y., & Jin, Z. (2024). Structured chain-of-thought prompting for code generation. ACM Transactions on Software Engineering and Methodology, 33(7), Article 178.

[9]. Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., & Chen, W. (2023). CodeT: Code generation with generated tests. In Proceedings of the 11th International Conference on Learning Representations (ICLR 2023).

[10]. Huang, B., Lu, S., Wan, X., & Duan, N. (2024). Enhancing large language models in coding through multi-perspective self-consistency. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL 2024).

[11]. Jiang, X., Dong, Y., Wang, L., Fang, Z., Shang, Q., Li, G., Jin, Z., & Jiao, W. (2024). Self-planning code generation with large language models. ACM Transactions on Software Engineering and Methodology, 33(7), Article 190.

[12]. Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., & Sutton, C. (2021). Program synthesis with large language models. arXiv preprint arXiv:2108.07732.

[13]. Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., & Neubig, G. (2023). PAL: Program-aided language models. In Proceedings of the 40th International Conference on Machine Learning (ICML 2023).

[14]. Li, Y., & Long, L. (2026). Lightweight AI-Driven Stress Testing for Small and Medium Financial Institutions: A Variational Autoencoder Approach with Extreme Value Theory for Macroeconomic Scenario Generation. Artificial Intelligence and Machine Learning Review, 7(1), 108-119.

[15]. Zhao, F., Zhang, M., Zhou, S., & Lou, Q. (2024). Application of deep reinforcement learning for cryptocurrency market trend forecasting and risk management.

[16]. Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In Advances in Neural Information Processing Systems 36 (NeurIPS 2023).

[17]. Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S., Phipps-Costin, L., Pinckney, D., Yee, M.-H., Zi, Y., Anderson, C. J., Feldman, M. Q., Guha, A., Greenberg, M., & Jangda, A. (2023). MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. IEEE Transactions on Software Engineering, 49(7), 3456-3472.

[18]. Chen, Y., Chen, Z., & Zou, D. (2025). CarbonShift: Harnessing Grid Carbon Variability for Geo-Distributed Workload Scheduling. Artificial Intelligence and Machine Learning Review, 6(4), 18-31.

[19]. Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y. K., Luo, F., Xiong, Y., & Liang, W. (2024). DeepSeek-Coder: When the large language model meets programming - the rise of code intelligence. arXiv preprint arXiv:2401.14196.

[20]. Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., ... von Werra, L. (2023). StarCoder: May the source be with you! Transactions on Machine Learning Research.

[21]. Le, H., Chen, H., Saha, A., Gokul, A., Sahoo, D., & Joty, S. (2024). CodeChain: Towards modular code generation through chain of self-revisions with representative sub-modules. In Proceedings of the 12th International Conference on Learning Representations (ICLR 2024).

[22]. Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. In Advances in Neural Information Processing Systems 36 (NeurIPS 2023).

[23]. Han, M., & Lai, J. (2026). Temporal Feature Engineering and Threshold Optimization for Early Warning in Healthcare Claims Anomaly Detection. Journal of Advanced Computing Systems, 6(4), 27-49.

[24]. Chen, Y., & Lai, J. (2026). Multi-Metric Trustworthiness Evaluation of AI-Assisted Medical Imaging Diagnosis: Integrating Confidence Calibration and Distribution Shift Detection. Journal of Global Engineering Review, 4(1), 113-126.

[25]. Long, L., & Hu, J. (2026). Multi-Objective Particle Swarm Optimization for Site Selection and Policy Subsidy Maximization of Foreign Renewable Energy Enterprises in the United States. Artificial Intelligence and Machine Learning Review, 7(2), 54-69.

[26]. Cao, H., & Long, L. (2026). Empirical Evaluation of Multi-Source Monitoring Signal Effectiveness and Lead Time for Performance Degradation Prediction in Kubernetes-Based Microservices. Journal of Advanced Computing Systems, 6(4), 15-26.

[27]. Zhang, T., Yu, T., Hashimoto, T., Lewis, M., Yih, W.-T., Fried, D., & Wang, S. (2023). Coder reviewer reranking for code generation. In Proceedings of the 40th International Conference on Machine Learning (ICML 2023).

[28]. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Huber, T., Choy, P., de Masson d'Autume, C., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., ... Vinyals, O. (2022). Competition-level code generation with AlphaCode. Science, 378(6624), 1092-1097.

[29]. Li, Y. (2026). Enhancing Financial Compliance Transparency through Automated Data Governance and Intelligent Risk Reporting. Journal of Science, Innovation & Social Impact, 2(1), 299-313.

[30]. Long, L., Zou, D., & Shi, W. (2026). NLP-Driven Psychological Contract Risk Detection in Cross-Cultural Teams: An XGBoost Approach with Cultural Adaptation. Artificial Intelligence and Machine Learning Review, 7(2), 43-53.

[31]. Cao, H., & Shi, W. (2026). Statistical Anomaly Detection Approach for Field Mapping Validation in Enterprise Payroll Data Migration. Journal of Computing Innovations and Applications, 4(1), 137-153.

[32]. Li, Z., & Chen, Z. (2025). Performance Evaluation of Prompt Generation Strategies for AI Agents in Online Programming Education. Journal of Advanced Computing Systems, 5(9), 14-27.

[33]. Zhang, H., & Shi, W. (2026). Comparative Evaluation of Automated Detection Approaches for Identifying Implicit Compliance Violations in Cross-border Commercial Contract Clauses. Artificial Intelligence and Machine Learning Review, 7(2), 1-22.

[34]. Long, X., Hu, J., & Ling, Z. (2026). A Comparative Analysis of Telemetry-Driven Anomaly DetectionApproaches for Dual-Purpose Operational and SecurityOptimization in Edge Computing Infrastructure. Journal of Computing Innovations and Applications, 4(1), 79-88.