Available online at www.ijournalse.org

Emerging Science Journal

(ISSN: 2610-9182)

Vol. 10, No. 2, April, 2026

Enhancing Small Language Models for Code Generation via Strategic Decomposition and Filtering

Yuriy Perezhohin [1, 2], Fabian Collao [1], Mauro Castelli [1*]

[1] *NOVA Information Management School (NOVA IMS), Universidade NOVA de Lisboa, 1070-312 Lisboa, Portugal.*

[2] *Remynd, Alameda Bonifacio Lazaro Lozano, nº 15, 1º C, 2780-125 Oeiras, Portugal.*

Abstract

This study addresses the challenge of enhancing Small Language Models (SLMs) for complex code generation tasks requiring structured planning, which current models struggle with due to their monolithic, single-pass generation approach. A three-stage pipeline architecture is proposed that decouples strategic planning from implementation: (1) an SLM generates diverse natural language strategies at high temperature, (2) a filtering mechanism selects high-quality strategies while removing noise, and (3) refined strategies guide a specialized coding model for final implementation. The approach was evaluated on the ClassEval benchmark for class-level code generation. The pipeline enabled a 1.5B parameter model to achieve 13% class success rate, representing a 30% relative improvement over direct generation (10%) and competitive performance with models 5-8 times larger. Critically, effective strategy filtering proved more important than strategy diversity, with simple pattern-based filters successfully mitigating SLM artifacts like few-shot contamination. This work demonstrates that structured, inference-time computation offers an efficient alternative to parameter scaling, with strategic noise reduction being the key driver of performance gains in resource-constrained models.

Keywords:

Code Generation;
Language Models;
Decomposition;
Small Models;
Benchmarking.

Article History:

Received:	04	November	2025
Revised:	11	March	2026
Accepted:	21	March	2026
Published:	01	April	2026

1- Introduction

The proficiency of Language Models (LMs) in code generation has advanced rapidly, evolving from producing simple snippets to authoring complete software components [1]. Despite this progress, their utility is often constrained when faced with complex programming problems that demand multi-step reasoning, algorithmic design, and strategic foresight [2, 3]. The predominant approach of direct, end-to-end code generation forces models to solve intricate problems in a single pass. This monolithic process is fundamentally at odds with the methodologies of human software development [4] and is a direct consequence of the autoregressive, token-by-token nature of the underlying transformer architecture [5], which lacks inherent mechanisms for high-level planning.

This limitation can be understood through the lens of dual-process cognitive theory [6]. Current LMs excel at System 1 thinking (fast, intuitive, and based on pattern recognition from vast training data). However, complex software engineering is a hallmark of System 2 thinking (slow, analytical, and strategic). Expert human programmers do not begin by writing code; they first decompose the problem, devise mental models, and evaluate multiple potential strategies before implementation [7, 8]. This strategic decomposition is the missing link in modern code generation systems.

Recent empirical studies have demonstrated the limitations of direct code generation approaches. Wang et al. [2] showed that LLMs often fail on problems requiring algorithmic planning, while Abbassi et al. [3] documented systematic inefficiencies in LLM-generated code. These failures are particularly pronounced in Small Language Models, where

* **CONTACT**: mcastelli@novaims.unl.pt

DOI: https://doi.org/10.28991/ESJ-2026-010-02-011

© 2026 by the authors. Licensee ESJ, Italy. This is an open access article under the terms and conditions of the Creative Commons Attribution (CC-BY) license (https://creativecommons.org/licenses/by/4.0/).

resource constraints exacerbate the planning deficit. Liu et al. [9] demonstrated that even surface-level correct code often contains latent functional errors when deeper reasoning is required. Specifically for small models, preliminary experiments with LLaMA-3.2-1B on ClassEval showed near-zero performance (1.6% function success), compared to 25.3% for the specialized Qwen2.5-Coder-1.5B, highlighting the acute planning deficit in general-purpose SLMs. These empirical observations motivated the hypothesis that explicit strategic decomposition could compensate for SLMs' limited reasoning capabilities. Additionally, despite advances in prompting techniques [10-12] and planning-first approaches [2, 13, 14], a critical gap remains: existing methods either require hundreds of model queries (making them impractical for SLMs) or employ complex multi-stage transformations that introduce error cascades. Furthermore, while the importance of intermediate output quality has been recognized [15-17], systematic investigation of strategy filtering mechanisms specifically for SLMs remains unexplored. This gap is significant because SLMs produce qualitatively different artifacts (such as few-shot contamination and repetition anomalies) that require targeted mitigation strategies rather than general-purpose filtering.

To bridge this gap, this paper introduces a novel three-stage pipeline architecture that explicitly separates strategic thinking from implementation, designed specifically to augment the performance of resource-efficient Small Language Models (SLMs). This architecture achieves significant efficiency gains, demonstrating that structured computation can substitute for sheer parameter growth. Our approach consists of three distinct phases:

Strategy Generation: An SLM operating at a high temperature generates multiple, diverse problem-solving strategies in natural language, effectively broadening the exploration of the solution space.

Strategy Selection: A filtering mechanism prunes the generated strategies, removing irrelevant, misleading, or noisy approaches.

Code Generation: The curated set of high-quality strategies then guides a specialized code generation model to produce the final, correct implementation.

This work is motivated by the growing need for efficient and accessible AI systems. While scaling model parameters has been the dominant paradigm for improving performance [18], it comes with prohibitive computational costs. Inference-time scaling, which allocates more computation during generation, offers a promising alternative [19]. The proposed pipeline is a form of inference scaling that structures this additional computation to mimic human cognitive workflows.

A rigorous evaluation is conducted on the ClassEval benchmark [20], which assesses class-level code generation with interdependent methods, a task far more representative of real-world programming than simpler function-level benchmarks. The experiments yield two primary insights. First, our pipeline architecture is highly effective, enabling a 1.5B parameter SLM to improve performance by 30% and compete with models 5-8x larger. Second, a surprising dynamic is uncovered: the key to success is not generating a wide diversity of strategies, but rather the aggressive filtering of strategic noise. The selection stage, which removes artifacts and conceptual contamination from the SLM's output, is the single most critical driver of performance.

This paper makes the following contributions:

- It proposes a novel, cognitively-inspired three-stage pipeline that enhances SLM code generation by decoupling planning and implementation.
- It demonstrates that this architecture allows a 1.5B model to achieve results competitive with much larger models, offering an efficient path to improved performance.
- It provides strong empirical evidence that for SLMs, effective strategy filtering is more critical than strategy diversity.
- It analyzes the failure modes of SLMs in a strategic pipeline, revealing how simple mechanisms can correct for artifacts like few-shot contamination.

The remainder of this paper is organized as follows: Section 2 reviews related work on prompting strategies, planning-first approaches, and strategy selection methods. Section 3 presents our three-stage pipeline methodology in detail. Section 4 describes the experimental setup, including benchmark selection, model architecture, and evaluation metrics. Section 5 reports and analyzes our experimental results, with particular focus on the critical role of strategy filtering. Section 6 discusses broader implications and limitations of our findings. Section 7 concludes with key takeaways and future research directions.

2- Related Works

Our research builds upon four primary areas: prompting techniques for enhanced reasoning, explicit planning-first architectures, methods for strategy selection, and small language models.

2-1- Prompting Strategies for Enhanced Reasoning

Early work revealed that the reasoning capabilities of LMs could be "unlocked" through structured prompting. Chain-of-Thought (CoT) prompting [10] showed that instructing a model to generate step-by-step reasoning before the final answer dramatically improves performance on complex tasks. Self-Consistency [11] extended this by sampling multiple reasoning paths and taking a majority vote, demonstrating the value of exploring diverse solutions. Other linear approaches, like Least-to-Most prompting [12], explicitly decompose problems into simpler subproblems to be solved sequentially.

Recent work has formalized these concepts under inference-time compute scaling, demonstrating that allocating additional computation during generation can match or exceed the benefits of parameter scaling [21]. Methods like parallel scaling through multiple answer generation and sequential scaling via iterative refinement have shown particular promise for code generation tasks [22].

While powerful, these methods concentrate the reasoning and generation processes within a single forward pass. Recognizing this limitation, non-linear structures were proposed. The Tree of Thoughts (ToT) [23] allows an LM to explore a tree of reasoning paths, using self-evaluation to guide its search. The Graph of Thoughts (GoT) [24] generalizes this to arbitrary graph structures, enabling more complex operations like thought aggregation and refinement. However, these non-linear methods often require hundreds of LLM queries, making them computationally expensive. Our pipeline differs by creating a clean, one-way separation between a distinct planning phase and an implementation phase, maintaining computational efficiency.

2-2- Planning-First Approaches

A more direct line of research explicitly separates planning from execution. PLANSEARCH [2] generates natural language plans for code, translates them to pseudocode, and then to executable code, using Monte Carlo Tree Search to explore the plan space. While effective, this involves multiple complex transformation steps, each a potential source of error. Other approaches utilize formal planning methods, instructing LMs to generate components for classical search algorithms [13] or to construct directed acyclic graphs of subtasks [14]. These formal methods offer rigor but sacrifice the flexibility of natural language, limiting their applicability to less structured programming problems.

Recent advances in multi-agent frameworks have demonstrated the effectiveness of decomposing code generation into specialized agent roles [25, 26]. MapCoder introduced a four-agent system comprising retrieval, planning, coding, and debugging agents, achieving state-of-the-art results on competitive programming benchmarks including HumanEval and MBPP [26]. Self-Organized multi-Agent frameworks have shown promise for scaling to large codebases through dynamic agent multiplication based on problem complexity [25].

Our approach adopts the natural language planning paradigm but simplifies it into a more streamlined pipeline, avoiding complex intermediate transformations.

2-3- Strategy Selection and Filtering

A critical and often overlooked challenge in multi-strategy generation is how to evaluate and select the best intermediate plans without ground truth. The LLM-as-Judge paradigm [15] has emerged as a viable solution, where a model is used to assess the quality of another model's output [16]. For code, models like CODESCORE [17] can evaluate final outputs along dimensions like functionality and readability.

However, these methods typically evaluate final code rather than intermediate planning artifacts. Our work is novel in that it systematically implements and compares several distinct selection mechanisms (including a baseline, a simple rule-based filter, and an LLM-as-Judge verifier) specifically for the task of filtering natural language strategies before implementation. As experimental results will show, this selection stage is not an auxiliary component but the primary driver of performance in our pipeline. In particular, section 3.2 describes a lightweight pattern-based filter that uses regular expressions over example-specific identifiers (such as class names from the few-shot problems) to remove contaminated strategies.

Recent work by Liu et al. [9] highlighted the importance of rigorous evaluation frameworks for LLM-generated code, demonstrating that surface-level correctness often masks deeper functional failures. This finding reinforces our focus on comprehensive test-based evaluation and the need for quality filtering mechanisms before code generation.

2-4- Small Language Models and Efficiency

The last couple of years have witnessed growing interest in Small Language Models for code generation, motivated by computational efficiency and deployment constraints [27]. Benchmarking studies on competitive programming platforms have revealed that modern SLMs can achieve surprisingly competitive performance, with models like Phi-4-14B reaching 63.6% pass@3 on Codeforces problems [27]. Comprehensive surveys have identified that while SLMs excel at certain code generation tasks, they face systematic challenges in complex reasoning and multi-step planning that require targeted architectural interventions [28].

These findings directly motivate our pipeline architecture, which is specifically designed to compensate for the planning deficits inherent in resource-constrained models through structured decomposition and strategic filtering.

3- Methodology: A Three-Stage Pipeline

This work proposes a three-stage pipeline architecture for strategy-assisted code generation designed to mimic the human cognitive process of planning before acting. The workflow is illustrated in Figure 1.

3-1-Stage 1: Strategy Generation

Given a programming problem *P*, the first stage (see Figure 1, top section) generates a set of N diverse, natural language strategies S = {s_1, s_2, ..., s_N}. This is formalized as:

$$S = f_{gen}(P, T, \theta) \tag{1}$$

where, *T* is our strategy generation prompt template and θ represents the model parameters optimized for exploration.

In terms of prompt design, the prompt template T is structured with three components: T = < $I_{instruction}$, $I_{constraints}$, $I_{examples}$>. In particular, $I_{instruction}$ specifies the high-level task of formulating a solution strategy; $I_{constraints}$ defines output formatting rules, such as using a specific XML structure and, crucially, prohibiting the inclusion of any implementation code; finally, $I_{examples}$ provides several few-shot examples of problems paired with high-quality strategies. A complete prompt template is listed in Appendix I.

To encourage the generation of diverse solution paths, we configure the SLM's sampling parameters for exploration. A high temperature (τ = 1.2) is used to increase the randomness of token selection [29]. The probability distribution over the vocabulary V for the next token x_t is given by:

$$p(x_t \mid x_{<t}) = \frac{\exp(z_i/\tau)}{\sum_{j \in V} \exp(z_j/\tau)} \tag{2}$$

where, z_i is the logit for token i. To maintain coherence while encouraging creativity, Min-P sampling [30] with a threshold p_{base} = 0.1 is employed. This method dynamically creates a reduced vocabulary V_{min} from which to sample:

$$V_{\min} = \{ v \in V \mid P(v \mid x_{1:t-1}) \geq p_{base} \times \max_{v' \in V} P(v' \mid x_{1:t-1})\} \tag{3}$$

For our experiments, N=10 strategies are generated for each problem.

3-2-Stage 2: Strategy Selection

After generating a diverse set of strategies, the pipeline filters this set to produce a refined subset S' = {s'_1, s'_2, ..., s'_k}, where k ≤ N. This stage (see Figure 1, middle section) is critical for removing noise and focusing the subsequent code generation phase. Three distinct selection mechanisms are implemented and compared:

- ***Baseline: Unfiltered Passage.*** In this configuration, no filtering is applied (S' = S). This allows us to isolate the effect of strategy generation alone.
- ***Pattern-Based Filtering.*** Our initial analysis revealed a specific failure mode: the strategy generator sometimes addressed the few-shot examples rather than the target ClassEval problem. This observation led us to develop a pattern-based filtering mechanism using regular expressions. Specifically, the procedure extracts the example-specific identifiers (class names and distinctive tokens) used in the strategy prompt and constructs a set of regular expression patterns based on them. Given the initial strategy set, the procedure discards any strategy whose text matches that of the strategies appearing in the prompt, yielding a filtered set.
- ***Model-Based Verification.*** Our most sophisticated mechanism uses the same SLM as a "judge" to evaluate the quality and relevance of its own generated strategies. The SLM is prompted to identify only the most helpful strategies.

3-3-Stage 3: Code Generation

In the final stage (see Figure 1, bottom section), the filtered set of strategies S' is used to guide a specialized coding model to produce the executable code C:

$$C = f_{code}(P, S', \varphi) \tag{4}$$

where, φ represents the parameters of the code generation model.

The prompt for the coding model includes the original problem P and the selected strategies S'. Crucially, the prompt frames the strategies as optional suggestions rather than strict requirements. This allows the coding model to exercise its own judgment, benefiting from the strategic guidance without being constrained by potentially suboptimal plans.

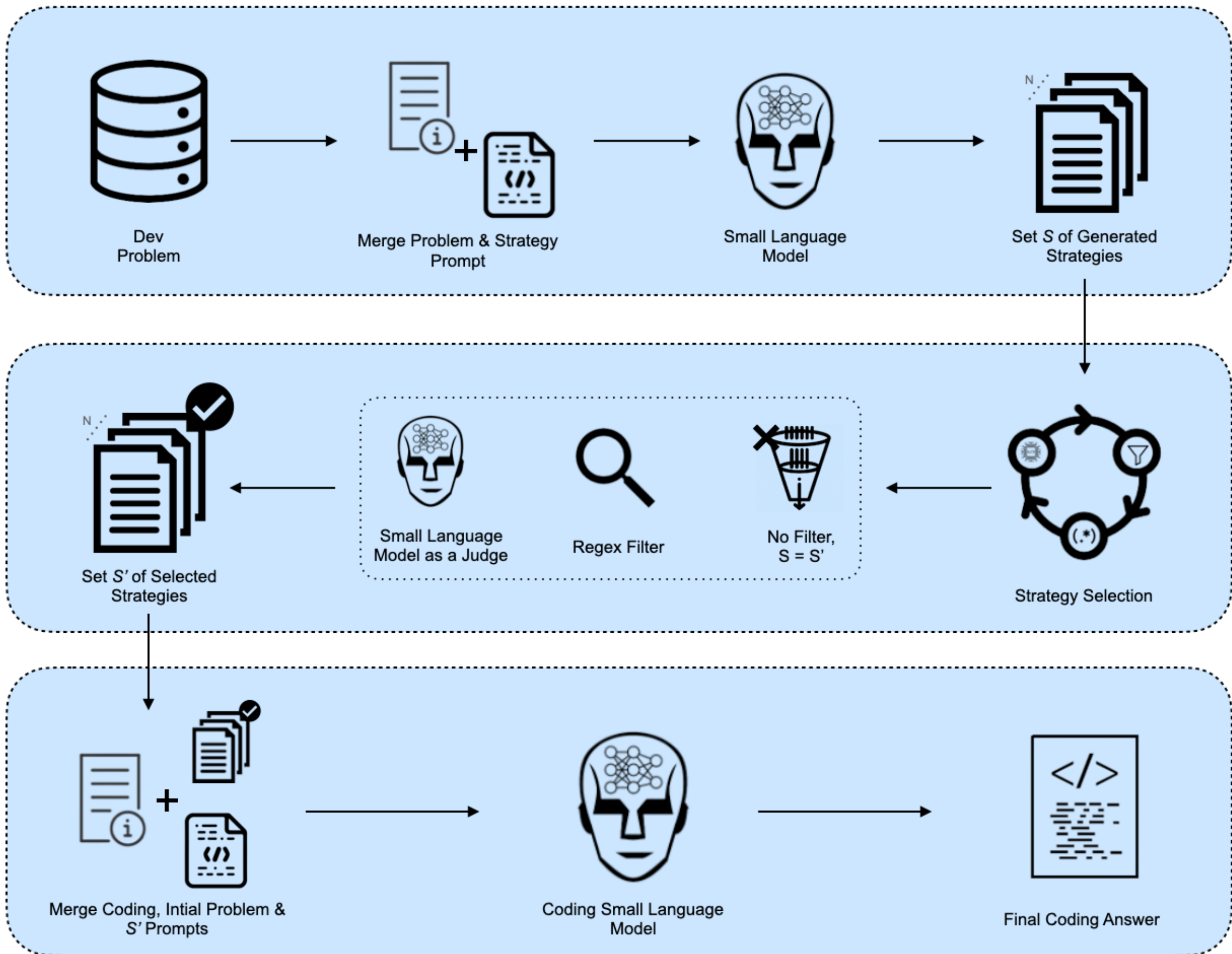

Figure 1. The workflow of our three-stage code generation pipeline. A problem is first used to generate a diverse set of natural language strategies. These strategies are then filtered by a selection mechanism. Finally, the curated strategies guide a coding model to produce the final implementation.

4- Experimental Setup

4-1- Benchmark Selection and Rationale

The proposed pipeline is evaluated on the ClassEval benchmark [20], a challenging dataset specifically designed to assess class-level code generation capabilities (https://github.com/FudanSELab/ClassEval). This benchmark was selected for several reasons that align with our research objectives. Differently from function-level benchmarks such as HumanEval or MBPP, ClassEval requires models to generate complete Python classes with multiple interconnected methods, mirroring the complexity of actual software development where methods must maintain consistency, share state, and implement coherent interfaces.

The benchmark's structure directly tests our hypothesis that strategic planning becomes crucial for complex, multi-component programming tasks. Each of the 100 handcrafted problems features an average of 5.2 methods per class, with complex interdependencies that cannot be resolved through isolated function generation. This interdependency challenge is precisely what makes strategic decomposition valuable, as models must coordinate multiple related implementations rather than solving independent subproblems.

Furthermore, ClassEval provides comprehensive evaluation through an average of 33.1 test cases per problem, allowing for robust verification across diverse input scenarios. The test suite covers both individual method functionality and class behavior, allowing us to conduct fine-grained analysis of where our pipeline succeeds or fails.

4-2- Model Architecture

Our pipeline employs a carefully designed two-model architecture that leverages complementary strengths while maintaining computational efficiency. For natural language strategy generation and verification, LLaMA-3.2-1B was selected based on several factors. As a general-purpose model, it demonstrates excellent natural language reasoning and planning capabilities, which are essential for generating coherent problem-solving strategies. At 1B parameters, it represents the resource-constrained setting we aim to optimize, making it an ideal testbed for our efficiency-focused

approach. Additionally, its strong performance on instruction-following tasks proves crucial for generating structured strategies that adhere to our specific formatting requirements.

For the final implementation phase, we employ Qwen2.5-Coder-1.5B [31], chosen for its specialized training on code corpora that provides superior syntax and API knowledge compared to general-purpose models. At 1.5B parameters, it maintains our focus on small model efficiency while delivering the coding expertise necessary for high-quality implementations. The model's strong Python generation capabilities make it particularly well-suited for the ClassEval benchmark requirements.

This heterogeneous approach allows each model to operate within its area of expertise while maintaining overall computational efficiency. The strategy generation model can focus on high-level reasoning without the burden of producing syntactically perfect code, while the specialized coding model can rely on the strategic guidance to produce better implementations than it could achieve through direct generation alone.

4-3- Hyperparameter Configuration

The strategy generation phase is configured to maximize exploration of the solution space through carefully tuned sampling parameters. In particular, a high temperature of $\tau = 1.2$ is employed to encourage diverse strategy generation. This setting was empirically determined through preliminary experiments comparing values across $\tau \in \{0.8, 1.0, 1.2, 1.5\}$, where $\tau = 1.2$ provided optimal diversity without semantic degradation. Lower temperatures produced repetitive strategies that failed to explore alternative solution approaches, while higher temperatures introduced excessive noise that compromised strategy coherence.

To maintain coherence while preserving diversity, Min-P sampling with $p_{base} = 0.1$ is implemented. Unlike top-p sampling, Min-P adapts the cutoff threshold relative to the most likely token, preventing the inclusion of extremely low-probability tokens that could harm semantic coherence while still allowing creative exploration. Each problem generates exactly 10 strategies, a number chosen to balance computational cost and the solution space coverage.

For the critical filtering phase, a low temperature of $\tau = 0.4$ is used to ensure consistent and reliable filtering decisions. This conservative approach prioritizes the stability and repeatability of our selection mechanism, which our results demonstrate is the most crucial component of the entire pipeline.

The final code generation stage uses similarly conservative parameters to ensure implementation quality. A temperature of $\tau = 0.4$ is considered to prioritize correctness over creativity, and employ top-p sampling with p = 0.95 to focus on high-probability tokens while allowing minor variations that can improve code quality. These settings reflect the characteristics required for producing functional, testable code.

The decision to generate N=10 strategies was based on preliminary experiments varying N in {5, 10, 15, 20} on a 15-problem development set. Results showed diminishing returns beyond N=10: N=5 achieved 10% class success, N=10 achieved 13%, N=15 achieved 13.3%, and N=20 achieved 13.3%, with no statistically significant difference between N=10, 15, and 20. This plateau effect occurs because strategy diversity saturates quickly: beyond 10 samples, the SLM largely generates semantic variations of existing approaches rather than genuinely novel strategies. Additionally, the filtering stage removes redundant strategies, so generating more provides minimal marginal value while increasing computational cost linearly. The choice of N=10 thus represents an efficient operating point balancing exploration breadth and computational budget. However, this optimal value may differ for other models or problem domains; adaptive N selection based on strategy diversity metrics could be explored in future work.

4-4- Baseline and Evaluation

Our primary baseline is a direct code generation approach using the Qwen2.5-Coder-1.5B model without any strategic pipeline. Code is generated directly from the problem statement using the model's default parameters. By comparing the proposed pipeline variants against this baseline, it is possible to isolate the contribution of our strategic decomposition methodology.

The evaluation relies on the standard evaluation metrics from ClassEval [20]. In particular, the focus is on two primary metrics that capture different aspects of implementation success. The class success rate measures the percentage of problems where the entire generated class passes all test cases, reflecting the strict requirements of real-world deployment where partial functionality is often insufficient. The function success rate provides a more granular view by measuring the percentage of individual methods that pass their respective test cases, offering insight into the pipeline's ability to handle method-level complexity even when overall class integration fails.

5- Results and Analysis

Our experiments reveal that while strategic decomposition can significantly enhance SLM performance, the effectiveness of the pipeline is critically dependent on the selection mechanism used to filter strategies.

5-1- Overall Performance

Table 1 presents the baseline performance of the SLMs and the aggregate results for our pipeline variants on the ClassEval benchmark.

Table 1. Performance comparison of direct generation baseline and strategy-enhanced pipeline configurations

Approach	Model(s)	Class Success (%)	Function Success (%)
Direct Generation Baselines			
Qwen Direct	Qwen2.5-1.5B	10.0	25.3
LLaMA Direct	LLaMA-3.2-1B	0.0	1.6
Strategy-Enhanced Pipeline			
No Filter	LLaMA + Qwen	11.0	25.1
Regex Filter	LLaMA + Qwen	**13.0**	**27.9**
SLM Verifier	LLaMA + Qwen	**13.0**	25.5

The direct generation baseline for Qwen2.5-1.5B achieves a 10.0% class success rate. The "No Filter" pipeline variant provides only a marginal benefit, increasing success to 11.0%. The transformative results come from the pipeline configurations with strategy selection. Both the Regex Filter and the SLM Verifier achieve a class success rate of 13.0%, representing a 30% relative improvement over the strong Qwen baseline. This demonstrates that the filtering step is the key performance driver.

To contextualize this achievement, Table 2 compares our 1.5B parameter pipeline against previously reported results from much larger models on ClassEval.

Table 2. ClassEval benchmark results across model scales

Model	Parameters	Class Success (%)
GPT-4-Turbo	–	38
deepseek-coder-instruct	6.7B	27
CodeLlama-Instruct	13B	21
Our Pipeline	**1.5B**	**13**
WizardCoder	15B	8
instruct-codegen	16B	5
SantaCoder	1.1B	0

Our pipeline, using a 1.5B model, outperforms several models with 15-16B parameters, demonstrating that architectural enhancements can be a highly effective and efficient alternative to simply scaling up model size.

While the absolute gain (10% → 13%) may appear modest, its practical significance should be evaluated in context. First, ClassEval is specifically designed to be challenging, with even GPT-4-Turbo achieving only 38% success. The 3-percentage point improvement represents solving 3 additional complete class-level problems correctly, each requiring multiple interdependent methods, a meaningful achievement for resource-constrained deployment scenarios. Second, the 30% relative improvement demonstrates a substantial efficiency gain: achieving this performance level without the pipeline would require scaling to 5-8× larger models, with corresponding increases in computational cost, latency, and memory requirements. For edge deployment or high-throughput applications where these resources are constrained, this represents a practically significant optimization. Third, the improvement is achieved with minimal additional inference cost (10-15 model queries vs. single-pass generation), making the cost-benefit ratio favorable. However, the absolute performance ceiling remains a limitation: for applications requiring high reliability (>90% success rates), even the improved pipeline remains insufficient, and larger models or formal verification methods would be necessary. The practical value proposition is thus clearest for applications with moderate quality requirements operating under strict resource constraints.

5-2- The Critical Role of Selection: When Less is More

Our problem-level analysis reveals a striking bimodal performance pattern that explains why filtering is so crucial. Rather than providing consistent modest improvements across all problems, our pipeline creates dramatic transformations in specific cases while potentially harming performance in others. This highlights that the selection mechanism is the most critical component of the pipeline.

A case study of the *DataStatistics* problem illustrates this aspect. This problem requires implementing a class for computing statistical measures including mean, median, and mode with proper error handling for edge cases. The baseline (Qwen Direct) approach achieves 0% test success, producing an incorrect implementation and failing to handle empty dataset edge cases appropriately.

The unfiltered pipeline version achieves 31.25% test success, but analysis of the generated strategies reveals significant contamination. Among the 10 generated strategies, several included relevant statistical computation approaches, but these were mixed with completely irrelevant concepts such as "UI event handling" and "data pipeline operations". This conceptual confusion led to a partially correct but ultimately flawed implementation that incorporated unnecessary complexity and missed critical edge cases.

The regex-filtered pipeline achieves 100% test success by removing five contaminated strategies and retaining five clearly focused on statistical computation. The clean, focused guidance enables the coding model to produce a perfect implementation with comprehensive error checking and mathematically correct algorithms. This improvement from 31.25% to 100% success demonstrates that strategic noise reduction, rather than diversity, drives performance gains.

Conversely, the *ArgumentParser* problem illustrates when strategic decomposition becomes counterproductive. This problem requires implementing a command-line argument parser with support for flags, options, and positional arguments. The baseline achieves 66.67% success while all pipeline variants achieve 0% success, suggesting that strategic decomposition introduced unnecessary complexity that caused the model to over-engineer a solution that worked well with direct generation. This case highlights the importance of understanding when strategic planning helps versus when it creates harmful interference.

These case studies reveal several critical insights into the mechanism of improvement. Concerning the *DataStatistics* problem, first, the contaminated strategies introduced conceptual interference by mixing statistical computation concepts with unrelated software engineering patterns. The coding model, lacking strong discriminative capabilities, attempted to reconcile these conflicting directives, resulting in an overcomplex implementation that failed edge case handling. Second, the regex filter's success demonstrates that even simple, deterministic filtering can dramatically improve performance when targeted at specific, predictable failure modes. Third, the perfect 100% success after filtering indicates that the coding model possesses latent capability to solve this problem correctly, but requires clean strategic guidance to activate this capability. The *ArgumentParser* failure case provides complementary insights into when strategic decomposition becomes counterproductive. Analysis of the generated strategies reveals that they introduced unnecessary abstraction layers for what is fundamentally a straightforward parsing task. The baseline's 66.67% success suggests the coding model has strong prior knowledge for this problem type, likely from extensive training on similar command-line parsing examples. The strategic guidance, rather than clarifying the approach, created cognitive overhead by suggesting multiple implementation paradigms simultaneously (e.g., object-oriented parser design vs. functional parsing). This interference disrupted the model's natural generation patterns, leading to incomplete or incorrect implementations. These contrasting cases highlight the conditions under which strategic planning helps versus hinders: planning provides value when the problem requires coordination across multiple components (*DataStatistics*), but introduces harmful interference when the coding model already has strong, direct solution patterns (*ArgumentParser*). This finding has important implications for adaptive pipeline design, suggesting that future systems should include meta-reasoning to determine when strategic decomposition is appropriate.

5-3- Analysis of SLM Artifacts

Our comprehensive error analysis reveals two primary categories of artifacts that explain why filtering is so crucial for small language model performance. The first issue is few-shot contamination. The LLaMA-1B model systematically confuses target problems with few-shot examples present in its context, generating strategies that reference completely different problems.

The success of the regex filter was largely attributable to its ability to mechanically identify and remove this specific failure mode through simple pattern matching, showing that sophisticated filtering mechanisms may be unnecessary when dealing with predictable failure modes.

The second category involves repetition anomalies due to implementation bugs in the SLM verifier. In certain cases, such as the *ArgumentParser* problem, the verifier generates identical strategy copies despite our intended limit of 10 strategies. Despite flooding the coding model with valid strategic guidance, performance remains at 0%, suggesting that overwhelming the model with information can be as detrimental as providing poor guidance. This finding reinforces the importance of curated, focused input rather than simply maximizing the quantity of strategic information.

These artifacts reveal that the pipeline's value comes from providing a structured framework to manage and filter the predictable errors and artifacts of a resource-constrained model.

5-4- Comparative Analysis with Previous Approaches

The results position this work within the broader landscape of code generation enhancement methods. Compared to prompting-based approaches like Chain-of-Thought [10] and Self-Consistency [11], which operate within single forward

passes, the pipeline achieves similar improvements (~30%) but with explicit separation of reasoning phases. This separation allows targeted intervention (filtering) that would be impossible in monolithic approaches.

Relative to planning-first methods, the approach offers distinct advantages and trade-offs. PlanSearch [2] achieves higher absolute performance through Monte Carlo Tree Search exploration, but requires orders of magnitude more compute (hundreds of model queries vs. 10-15 in this pipeline). The 13% class success rate achieved here with 1.5B parameters demonstrates that strategic efficiency can partially compensate for reduced search breadth. However, the bimodal performance distribution observed here was not reported in PlanSearch, suggesting that more extensive search may provide robustness against strategic interference effects.

The comparison with Tree of Thoughts [23] and Graph of Thoughts [24] is particularly interesting. These methods achieve strong performance through structured exploration but at substantial computational cost. The pipeline's simpler linear architecture (generate - filter - implement) achieves competitive efficiency by sacrificing exploration breadth for targeted noise reduction. The key insight is that for SLMs, eliminating low-quality reasoning paths (filtering) provides greater value than exploring additional paths (branching), a finding that diverges from results with larger models where exploration typically dominates.

Regarding LLM-as-Judge approaches [15, 16], the results reveal an unexpected pattern: the sophisticated SLM verifier performed no better than the simple regex filter (both 13% class success). This contrasts with findings from CodeScore [17], where learned evaluation metrics outperformed rule-based approaches. The divergence likely stems from the evaluation target: CodeScore evaluates final code (where quality is complex and multidimensional), while this work filters intermediate strategies (where failure modes are simpler and more predictable). This finding suggests that the appropriate filtering mechanism depends critically on the artifact type and the specific failure modes of the model being filtered.

Finally, positioning relative to parameter scaling [18], the results provide empirical support for inference-time compute scaling [19]. The 1.5B pipeline outperforming 15-16B models demonstrates that architectural allocation of compute during inference can substitute for parameter growth, at least within certain problem domains. However, the absolute performance ceiling (13% vs. 38% for GPT-4-Turbo) indicates clear limits to this substitution effect, suggesting that inference-time scaling is a complement to, rather than replacement for, parameter scaling.

6- Discussion

The experimental results provide a clear perspective on enhancing SLMs for complex tasks. Initially, it was hypothesized that mimicking human strategic thinking would unlock deeper reasoning capabilities. What we discovered is more pragmatic: our pipeline succeeds primarily by acting as a compensatory mechanism for the inherent limitations of SLMs.

Our work was framed using the System 1 vs. System 2 thinking paradigm [6]. While our goal was to introduce a System 2-like planning layer, the practical outcome was a system that excels at cleaning up the noisy, artifact-laden outputs of the SLM's System 1-like generation process. The most successful components of our pipeline were those that most effectively filtered out the SLM's failure modes.

The System 1 vs. System 2 framing, while conceptually appealing, requires careful qualification. The analogy provided a useful heuristic for pipeline design but should not be interpreted as a validated cognitive model of LM behavior. Empirical observations align only partially with the framework: the SLM's rapid, pattern-based strategy generation resembles System 1 characteristics, and the deliberate filtering stage adds System 2-like analytical processing. However, critical divergences exist. Human System 2 thinking actively inhibits System 1 responses and generates novel reasoning pathways, whereas the pipeline's filtering stage passively removes artifacts without creating new strategies. Furthermore, the finding that simple regex filtering matches sophisticated model-based verification suggests the "System 2" component operates more mechanistically than the cognitive theory implies. Future work should move beyond conceptual analogies to develop empirically-grounded models of how SLMs actually process strategic information, potentially drawing on mechanistic interpretability techniques to trace information flow through the pipeline stages.

Focusing on the implications for practitioners, our results suggest that when deploying SLMs, performance gains can be achieved through simple, targeted architectural improvements. Rather than pursuing complex reasoning structures, practitioners may find more value in identifying the specific failure modes of their chosen model and implementing simple, rule-based filters to mitigate them.

Despite the relevant results achieved, it is important to highlight that our findings are largely based on the specific artifacts of the LLaMA-1B model. The evaluation was also limited to the ClassEval benchmark, raising important questions about generalizability. ClassEval was selected specifically because its class-level, multi-method problems align with the hypothesis that strategic planning benefits complex, coordinated implementations. However, different benchmark characteristics may yield different results. HumanEval [1, 32] and MBPP [33] focus on standalone functions rather than interconnected classes, potentially reducing the value of high-level strategic decomposition. Future work

should conduct systematic evaluation across multiple benchmarks with varying characteristics (function vs. class level, algorithmic vs. data manipulation focus, with/without API knowledge requirements) to map the conditions under which strategic decomposition provides value. Additionally, real-world coding benchmarks that capture diverse programming paradigms (e.g., BigCodeBench [34] and RepoEval [35]) would provide crucial evidence for practical deployment scenarios.

7- Conclusion

This research addressed the fundamental challenge of enabling Small Language Models to perform complex, multi-step code generation tasks that require strategic planning capabilities. Through the development and rigorous evaluation of a three-stage pipeline architecture (comprising strategy generation, critical filtering, and guided code implementation), this work demonstrates that structured inference-time computation can substantially enhance SLM performance without parameter scaling. Evaluation on the ClassEval benchmark revealed a 30% relative improvement in class-level success rates, allowing a 1.5B parameter model to outperform specialized coding models 5-8 times its size. However, detailed analysis uncovered a complex performance landscape characterized by extreme bimodality: the pipeline transforms complete failures into perfect solutions for certain problems while actively interfering with baseline capabilities on others. This pattern reveals that the pipeline's primary function is not to elicit sophisticated reasoning but to manage and mitigate the specific artifacts and failure modes inherent to small, resource-constrained models.

The work's central and most practically significant finding challenges conventional assumptions about multi-strategy approaches: for SLMs, performance gains arise primarily from strategic noise reduction rather than solution diversity. Simple, targeted filtering mechanisms, such as the regex filter that identifies few-shot contamination, proved as effective as sophisticated model-based verifiers, demonstrating that understanding a model's specific failure modes enables efficient compensatory interventions. This insight has important implications for practitioners deploying SLMs in production environments, suggesting that investment in model-specific error analysis and targeted architectural interventions may yield greater returns than pursuing general-purpose reasoning frameworks. While the approach has limitations (evaluation on a single benchmark, artifacts specific to the LLaMA-1B model, and absolute performance still substantially below frontier models), it establishes a foundational principle: progress in AI systems can emerge from deeply understanding and working with existing model limitations rather than solely attempting to impose idealized cognitive frameworks. Future work should extend this paradigm across model scales and task domains, develop automated methods for failure mode identification, and create adaptive pipelines that dynamically determine when strategic decomposition enhances versus interferes with generation.

8- Declarations

8-1- Author Contributions

Conceptualization, Y.P., F.C., and M.C.; methodology, Y.P., F.C., and M.C.; software, F.C.; validation, Y.P. and F.C.; formal analysis, Y.P. and F.C.; investigation, Y.P., F.C., and M.C.; resources, M.C.; data curation, Y.P. and F.C.; writing—original draft preparation, F.C.; writing—review and editing, Y.P. and M.C.; visualization, Y.P.; supervision, M.C.; project administration, Y.P.; funding acquisition, M.C. All authors have read and agreed to the published version of the manuscript.

8-2- Data Availability Statement

The data presented in this study are openly available in: https://github.com/FudanSELab/ClassEval (accessed on March 2026).

8-3- Funding and Acknowledgments

This work was supported by national funds through FCT (Fundação para a Ciência e a Tecnologia), under the project - UID/04152/2025 - Centro de Investigação em Gestão de Informação (MagIC)/NOVA IMS - https://doi.org/10.54499/UID/04152/2025 (2025-01-01/2028-12-31) and UID/PRR/04152/2025 https://doi.org/10.54499/UID/PRR/04152/2025 (2025-01-01/ 2026-06-30) This work was funded by the European Union throught the project 101084013 - DIGITAL4Business. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency (HADEA). Neither the European Union nor the granting authority can be held responsible for them.

8-4- Institutional Review Board Statement

Not applicable.

8-5- Informed Consent Statement

Not applicable.

8-6- Conflicts of Interest

The authors declare that there is no conflict of interest regarding the publication of this manuscript. In addition, the ethical issues, including plagiarism, informed consent, misconduct, data fabrication and/or falsification, double publication and/or submission, and redundancies have been completely observed by the authors.

9- References

[1] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv Preprint, arXiv:2107.03374. doi:10.48550/arXiv.2107.03374

[2] Wang, E., Cassano, F., Wu, C., Bai, Y., Song, W., Nath, V., Han, Z., Hendryx, S., Yue, S., & Zhang, H. (2025). Planning in Natural Language Improves LLM Search for Code Generation. 13th International Conference on Learning Representations, ICLR 2025, 7230–7276.

[3] Abbassi, A. A., Da Silva, L., Nikanjam, A., & Khomh, F. (2025). A Taxonomy of Inefficiencies in LLM-Generated Python Code. IEEE International Conference on Software Maintenance and Evolution (ICSME2025), 393-404. doi:10.1109/ICSME64153.2025.00043

[4] Wirth, N. (1983). Program Development by Stepwise Refinement. Communications of the ACM, 26(1), 70–74. doi:10.1145/357980.358010.

[5] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In Advances in Neural Information Processing Systems, 5999–6009. doi:10.1201/9781003561460-19.

[6] Kahneman, D. (2011). Fast and slow thinking. Allen Lane and Penguin Books, New York, United States.

[7] Soloway, E., & Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. IEEE Transactions on Software Engineering, SE-10(5), 595–609. doi:10.1109/TSE.1984.5010283.

[8] Song, T., & Becker, K. (2014). Expert vs. novice: Problem decomposition/recomposition in engineering design. International Conference on interactive collaborative learning (ICL2014), 181-190. doi:10.1109/ICL.2014.7017768.

[9] 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. Advances in Neural Information Processing Systems, 36, 21558–21572.

[10] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., & Zhou, D. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. Advances in Neural Information Processing Systems, 35, 24824–24837.

[11] Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E. H., Narang, S., Chowdhery, A., & Zhou, D. (2023). Self-Consistency Improves Chain of Thought Reasoning in Language Models. 11th International Conference on Learning Representations, ICLR 2023, 1-24.

[12] Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O., Le, Q., & Chi, E. (2023). Least-To-Most Prompting Enables Complex Reasoning in Large Language Models. In 11th International Conference on Learning Representations, ICLR 2023, 1-61.

[13] Katz, M., Kokel, H., Srinivas, K., & Sohrabi, S. (2024). Thought of Search: Planning with Language Models Through the Lens of Efficiency. Advances in Neural Information Processing Systems, 37, 138491–138568. doi:10.52202/079017-4395.

[14] Hernández-Gutiérrez, S., Alakuijala, M., Nikitin, A. V., & Marttinen, P. (2025). Recursive decomposition with dependencies for generic divide-and-conquer reasoning. arXiv Preprint, arXiv:2505.02576. doi:10.48550/arXiv.2505.02576.

[15] Zheng, L., Chiang, W. L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., Zhang, H., Gonzalez, J. E., & Stoica, I. (2023). Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. Advances in Neural Information Processing Systems, 36, 46595–46623.

[16] Liu, Y., Iter, D., Xu, Y., Wang, S., Xu, R., & Zhu, C. (2023). G-EVAL: NLG Evaluation using GPT-4 with Better Human Alignment. EMNLP 2023 - 2023 Conference on Empirical Methods in Natural Language Processing, Proceedings, 2511–2522. doi:10.18653/v1/2023.emnlp-main.153.

[17] Dong, Y., Ding, J., Jiang, X., Li, G., Li, Z., & Jin, Z. (2025). CodeScore: Evaluating Code Generation by Learning Code Execution. ACM Transactions on Software Engineering and Methodology, 34(3), 1–22. doi:10.1145/3695991.

[18] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... & Amodei, D. (2020). Scaling laws for neural language models. arXiv PREPRINT, arXiv:2001.08361. doi:10.48550/arXiv.2001.08361.

[19] Snell, C., Lee, J., Xu, K., & Kumar, A. (2025). Scaling Llm Test-Time Compute Optimally Can Be More Effective Than Scaling Parameters for Reasoning. 13th International Conference on Learning Representations, ICLR 2025, 7595–7629.

[20] Du, X., Liu, M., Wang, K., Wang, H., Liu, J., Chen, Y., ... & Lou, Y. (2023). Classeval: A manually-crafted benchmark for evaluating LLMs on class-level code generation. arXiv Preprint, arXiv:2308.01861. doi:10.48550/arXiv.2308.01861.

[21] Balachandran, V., Chen, J., Chen, L., Garg, S., Joshi, N., Lara, Y., ... & Yousefi, S. (2025). Inference-time scaling for complex tasks: Where we stand and what lies ahead. arXiv Preprint, arXiv:2504.00294. doi:10.48550/arXiv.2504.00294.

[22] Li, D., Cao, S., Cao, C., Li, X., Tan, S., Keutzer, K., Xing, J., Gonzalez, J. E., & Stoica, I. (2025). S*: Test Time Scaling for Code Generation, 15964–15978. doi:10.18653/v1/2025.findings-emnlp.865.

[23] Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., & Narasimhan, K. (2023). Tree of Thoughts: Deliberate Problem Solving with Large Language Models. Advances in Neural Information Processing Systems, 36, 11809–11822.

[24] Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Podstawski, M., Gianinazzi, L., Gajda, J., Lehmann, T., Niewiadomski, H., Nyczyk, P., & Hoefler, T. (2024). Graph of Thoughts: Solving Elaborate Problems with Large Language Models. Proceedings of the AAAI Conference on Artificial Intelligence, 38(16), 17682–17690. doi:10.1609/aaai.v38i16.29720.

[25] Ishibashi, Y., & Nishimura, Y. (2024). Self-organized agents: A LLM multi-agent framework toward ultra large-scale code generation and optimization. arXiv Preprint, arXiv:2404.02183. doi:10.48550/arXiv.2404.02183.

[26] Islam, M. A., Ali, M. E., & Parvez, M. R. (2024). MapCoder: Multi-Agent Code Generation for Competitive Problem Solving. Proceedings of the Annual Meeting of the Association for Computational Linguistics, 4912–4944. doi:10.18653/v1/2024.acl-long.269.

[27] Souza, D., Gheyi, R., Albuquerque, L., Soares, G., & Ribeiro, M. (2025). Code Generation with Small Language Models: A Codeforces-Based Study. arXiv Preprint, arXiv:2504.07343. doi:10.48550/arXiv.2504.07343.

[28] Jiang, J., Wang, F., Shen, J., Kim, S., & Kim, S. (2026). A survey on large language models for code generation. ACM Transactions on Software Engineering and Methodology, 35(2), 1-72. doi:10.1145/3747588.

[29] Renze, M., & Guven, E. (2024). The Effect of Sampling Temperature on Problem Solving in Large Language Models. EMNLP 2024 - 2024 Conference on Empirical Methods in Natural Language Processing, Findings of EMNLP 2024, 7346–7356. doi:10.18653/v1/2024.findings-emnlp.432.

[30] Minh, N. N., Baker, A., Neo, C., Roush, A., Kirsch, A., & Shwartz-Ziv, R. (2025). Turning Up the Heat: Min-P Sampling for Creative and Coherent LLM Outputs. 13th International Conference on Learning Representations, ICLR 2025, 34593–34626.

[31] Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., ... & Lin, J. (2024). Qwen2.5-coder technical report. arXiv Preprint, arXiv:2409.12186. doi:10.48550/arXiv.2409.12186.

[32] Liu, X., Sun, X., Bo, L., Hu, Y., Liu, X., & Ye, Z. (2025). Evaluating the test adequacy of benchmarks for LLMs on code generation. Journal of Software: Evolution and Process, 37(7), e70034. doi:10.1002/smr.70034.

[33] Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., ... & Sutton, C. (2021). Program synthesis with large language models. arXiv Preprint, arXiv:2108.07732. doi:10.48550/arXiv.2108.07732.

[34] Zhuo, T. Y., Chien, V. M., Chim, J., Hu, H., Yu, W., Widyasari, R., ... & Von Werra, L. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. The Thirteenth International Conference on Learning Representations, ICLR 2025, 1-55.

[35] Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J., Zan, D., Mao, Y., Lou, J. G., & Chen, W. (2023). RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. EMNLP 2023 - 2023 Conference on Empirical Methods in Natural Language Processing, Proceedings, 2471–2484. doi:10.18653/v1/2023.emnlp-main.151.

Appendix I: Prompt Templates

A-1- Strategy Generation Prompt Template

```
<instruction>
You are an expert software architect. Given a programming problem, your task is to formulate
a high-level solution strategy. Focus on the conceptual approach, key algorithms, and design
considerations. Do NOT include any implementation code.
</instruction>

<constraints>
- Output must be wrapped in <strategy></strategy> XML tags
- Describe the approach in natural language only
- Identify key data structures and algorithmic patterns
- Consider edge cases and potential challenges
- Maximum 200 words
</constraints>

<examples>
[Few-shot examples would be inserted here - 3 examples of problem-strategy pairs]
</examples>

<target_problem>
{problem_description}
</target_problem>
```

A-2- Strategy Verification Prompt Template

```
<instruction>
You are evaluating solution strategies for quality and relevance. Given a programming problem
and a set of proposed strategies, identify which strategies are actually addressing the target
problem (not few-shot examples) and are conceptually sound.
</instruction>

<problem>
{problem_description}
</problem>

<strategies>
{strategy_list}
</strategies>

<output_format>
Return a JSON array of strategy indices that should be kept: [1, 3, 5, ...]
</output_format>
```

A-3- Code Generation Prompt Template

```
<instruction>
Implement the following Python class according to the problem specification. You may use the
provided solution strategies as optional guidance, but you should exercise your own judgment
and are not required to follow them if you have a better approach.
</instruction>

<problem>
{problem_description}
</problem>

<strategies>
{selected_strategies}
</strategies>

<output_format>
Provide complete, executable Python code with proper error handling and edge case management.
</output_format>
```