Intention Chain-of-Thought Prompting with Dynamic Routing for Code Generation

Shen Li[1], Li Huang[1], Shaoxiong Zhan[2], Weifeng Sun[1], Tao Yin[1], Zhongxin Liu[3], Meng Yan[1]*

[1]Chongqing University
[2]Tsinghua University
[3]Zhejiang University
{shen.l, yintao}@stu.cqu.edu.cn, zhansx24@mails.tsinghua.edu.cn
liu_zx@zju.edu.cn, {lee.h, weifeng.sun, mengy}@cqu.edu.cn

Abstract

Large language models (LLMs) exhibit strong generative capabilities and have shown great potential in code generation. Existing chain-of-thought (CoT) prompting methods enhance model reasoning by eliciting intermediate steps, but suffer from two major limitations: First, their uniform application tends to induce overthinking on simple tasks. Second, they lack intention abstraction in code generation, such as explicitly modeling core algorithmic design and efficiency, leading models to focus on surface-level structures while neglecting the global problem objective. Inspired by the cognitive economy principle of engaging structured reasoning only when necessary to conserve cognitive resources, we propose RoutingGen, a novel difficulty-aware routing framework that dynamically adapts prompting strategies for code generation. For simple tasks, it adopts few-shot prompting; for more complex ones, it invokes a structured reasoning strategy, termed Intention Chain-of-Thought (ICoT), which we introduce to guide the model in capturing task intention, such as the core algorithmic logic and its time complexity. Experiments across three models and six standard code generation benchmarks show that RoutingGen achieves state-of-the-art performance in most settings, while reducing total token usage by 46.37% on average across settings. Furthermore, ICoT outperforms six existing prompting baselines on challenging benchmarks.

Code — https://github.com/Guai001/RoutingGen

Introduction

Code generation focuses on translating user requirements into executable programs and is regarded as a core task in software engineering (Chen et al. 2021; Austin et al. 2021). LLMs exhibit strong generative capabilities and have shown great potential in this task. This performance stems from their ability to simulate complex reasoning processes in structured problem solving (Wei et al. 2022; Zheng et al. 2025; Tian et al. 2025). However, a critical gap remains between this reasoning capability and its application in code generation: LLMs tend to produce syntactically correct programs that fail to align with the task intention, resulting in functionally incorrect outputs (Cobbe et al. 2021; Jiang et al. 2024; Chen et al. 2023).

To address these challenges, recent studies have explored prompting strategies that guide models to generate intermediate steps. Scratchpads (Nye et al. 2021) introduced the idea of showing intermediate computations. CoT prompting (Wei et al. 2022) then generalized this idea across tasks, followed by self-consistency decoding (Wang et al. 2023b), which enhances consistency through multiple sampled traces. To better align model reasoning with programming tasks, subsequent work has proposed code-centric prompting strategies such as Program-of-Thought (Chen et al. 2023), CodeCoT (Huang et al. 2024), and self-planning (Jiang et al. 2024). Despite their progress, these methods reveal two key limitations. First, their uniform application across functional-level programming problems often induces overthinking on simple tasks, resulting in disorganized logic and reduced accuracy (Figure 1(A)). Second, they lack intention abstraction in code generation, failing to explicitly model core algorithmic design and efficiency considerations. As a result, models tend to focus on structural correctness while neglecting the intended task objective (Figure 1(B)).

Dual-process theories of human cognition describe two complementary systems: System 1 supports rapid and intuitive responses to simple or familiar problems, while System 2 is engaged for deliberate and structured reasoning when tasks are complex (Kahneman 2011). This adaptive mechanism reflects the principle of cognitive economy, which promotes conserving cognitive resources by activating structured deliberation only when necessary (Stanovich and West 2000). Inspired by this principle, we propose RoutingGen, a difficulty-aware dynamic routing framework that employs a classifier to estimate problem difficulty and dynamically selects appropriate prompting strategies. For simpler tasks, RoutingGen generates code directly via few-shot prompting, which mitigates unnecessary reasoning and prevents overthinking. For more complex problems, it invokes a structured reasoning strategy, termed *Intention Chain-of-Thought* (ICoT), which we introduce to guide the model in capturing task intent. Specifically, ICoT comprises two components: a *Specification* element that defines the input-output constraints, and an *Idea* element that captures the core algorithmic logic and estimates time complexity. This decomposi-

*Corresponding author.
Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

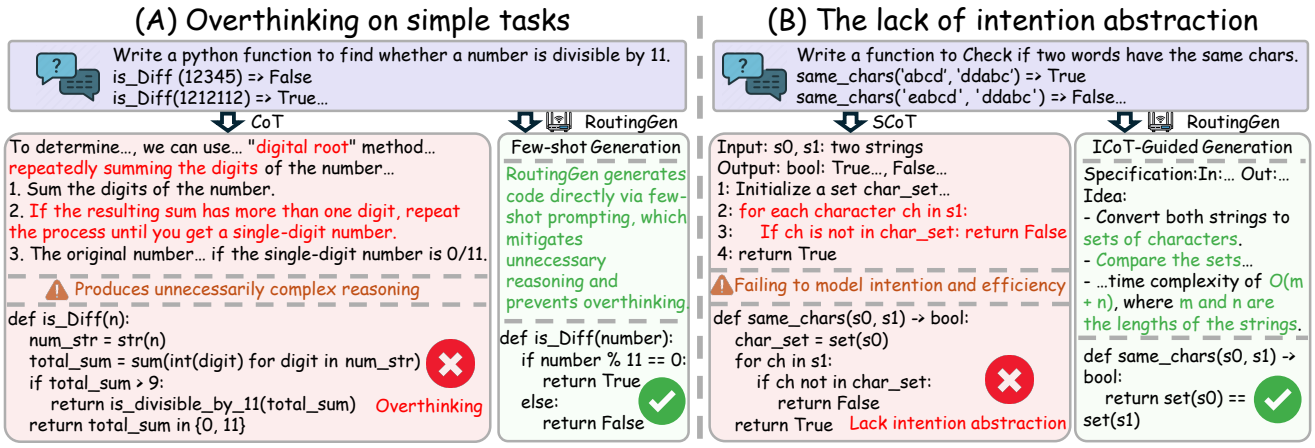

Figure 1: Limitations of existing CoT prompting methods in code generation. (A) Overthinking on simple tasks due to uniform application of structured prompting at the functional code level. (B) The lack of intention abstraction in code generation, such as algorithmic design and efficiency modeling.

tion reflects the classic problem-solving strategy of separating task comprehension from solution design (Polya 1945), whose importance has been further underscored by recent advances in mathematical reasoning with LLMs (Wang et al. 2023a). As a result, this intention representation helps steer code generation toward solutions that preserve structural guidance while explicitly modeling the task's functional requirements.

We evaluate RoutingGen and ICoT across three models and six standard code generation benchmarks. RoutingGen achieves state-of-the-art performance in most settings while reducing total token usage by 46.37% on average. Additionally, ICoT consistently outperforms six prompting baselines on challenging benchmarks. Furthermore, ablation results show that RoutingGen demonstrates robustness to variations in the difficulty classification model and that both the Specification and Idea stages contribute to ICoT's effectiveness.

Our contributions are threefold:

- We focus on the issue of overthinking on simple tasks due to uniform application of structured prompting at the functional code level, and identify a core limitation in existing methods: the lack of intention abstraction in code generation, such as algorithmic design and efficiency modeling.
- We propose *RoutingGen*, a novel difficulty-aware routing framework that dynamically adapts prompting strategies for code generation. For simple tasks, it adopts few-shot prompting; for more complex ones, it invokes a structured reasoning strategy, termed *Intention Chain-of-Thought* (ICoT), which we introduce to guide the model in capturing task intent, including core algorithmic logic and estimated time complexity.
- We empirically validate that RoutingGen achieves state-of-the-art performance in most settings while substantially reducing token usage. Additionally, ICoT consistently outperforms six prompting baselines on challenging benchmarks.

Related Work

The impressive reasoning capabilities exhibited by LLMs have led to extensive research on prompting strategies (Wei et al. 2022; Kojima et al. 2022; Kaplan et al. 2020). A seminal contribution in this direction is the CoT prompting method (Wei et al. 2022; Kojima et al. 2022), which guides models to articulate intermediate reasoning steps. This approach has inspired a range of prompting methods designed to make reasoning more explicit and structured. Representative strategies include enhancing robustness through multi-path sampling as in Self-Consistency (Wang et al. 2023b), applying structured planning in Tree-of-Thoughts (Yao et al. 2023), incorporating step-wise decomposition (Zhou et al. 2023), and enabling self-correction via reflexion (Shinn et al. 2023). However, the uniform application of these methods tends to induce overthinking on simple tasks.

To better align model reasoning with programming tasks, recent work has proposed code-centric prompting strategies, such as leveraging abstract syntax trees (Yin and Neubig 2017), incorporating self-planning (Jiang et al. 2024), employing structured reasoning frameworks (Li et al. 2025), integrating execution-time validation with compiler feedback (Gao et al. 2023; Zelikman et al. 2024), and leveraging in-context learning to organize requirements observed in descriptions and to extrapolate unexpressed requirements (Han et al. 2024). In contrast to their approach, we place greater emphasis on the abstraction of task intention in code generation, which helps steer code generation toward solutions that preserve structural guidance while explicitly modeling the task's functional requirements.

Current studies on routing techniques mainly focus on demonstration selection or cost efficiency. For example, Auto-CoT automates prompt construction through clustering (Zhang et al. 2022), while other studies route queries to different models or adapt sampling strategies (Varangot-Reille et al. 2025; Wang et al. 2025). Our framework differs in that it is motivated by the principle of cognitive econ-

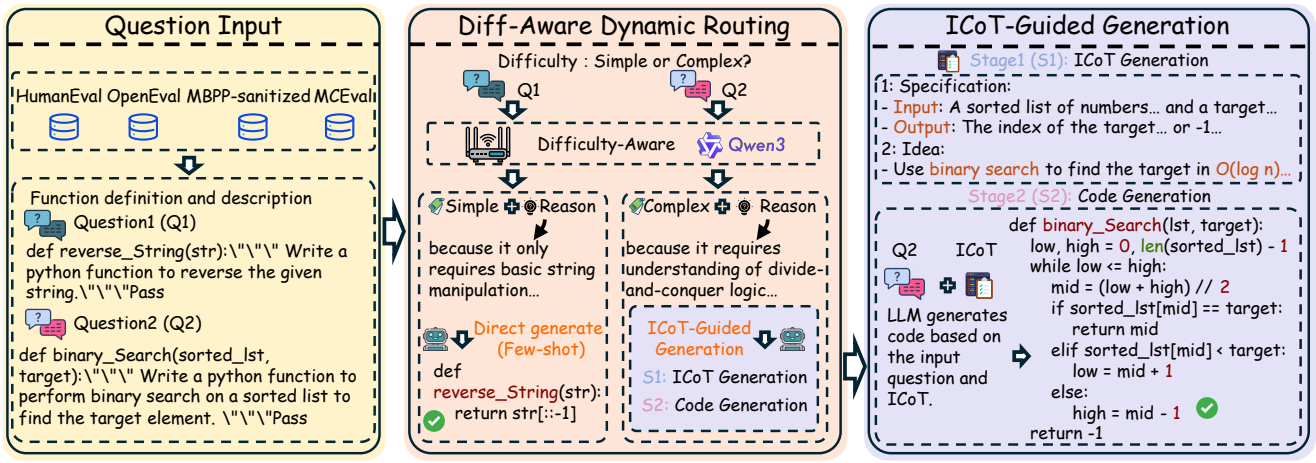

Figure 2: The RoutingGen Framework for Difficulty-Aware Dynamic Routing. For simple problems, it adopts few-shot prompting, while for more challenging cases, it leverages a structured reasoning strategy we propose, termed ICoT, which captures task intention, including the core algorithmic logic and time complexity.

omy and uniquely routes between distinct reasoning strategies based on task complexity.

Methodology

In this section, we introduce **RoutingGen**, a novel difficulty-aware routing framework that dynamically adapts prompting strategies based on problem difficulty in code generation. As illustrated in Figure 2, the overall workflow consists of two key components following the initial input: a Difficulty-Aware Routing module that dynamically assigns each problem to a suitable generation strategy, and an ICoT-Guided Generation process for complex tasks, which guides the model in capturing task intent.

Difficulty-Aware Dynamic Routing

RoutingGen leverages $\mathcal{M}_{\text{cls}}$ (Qwen3-8B) as a difficulty-aware classifier to steer the selection of prompting strategies for a given input problem q. Conditioned on a carefully designed prompt, the classifier assigns q to one of two difficulty levels from the label space $\mathcal{L} = \{\text{Simple}, \text{Complex}\}$ and generates a textual rationale r explaining its decision. Formally, this classification step is defined as:

$$(d^*, r^*) = \underset{(d,r)\ \text{s.t.}\ d\in\mathcal{L}}{\text{argmax}}\ P_{\mathcal{M}_{\text{cls}}}(d, r \mid q, T_{\text{cls}}) \tag{1}$$

where T_{cls} is the classification prompt. Here, d^* denotes the assigned difficulty label, and r^* is the corresponding rationale produced by the classifier.

Based on the assigned difficulty label d^*, RoutingGen subsequently applies the corresponding generation strategy tailored to the task complexity:

$$\mathcal{G}_{\text{strategy}} = f(d^*) = \begin{cases} \mathcal{G}_{\text{Direct}} & \text{if } d^* = \textit{Simple} \\ \mathcal{G}_{\text{ICoT}} & \text{if } d^* = \textit{Complex} \end{cases} \tag{2}$$

we use a model $\mathcal{M}_{\text{gen}}$ to perform both intention and code generation throughout the framework. For problems classified as *Simple*, RoutingGen applies a direct, low-cost generation strategy $\mathcal{G}_{\text{direct}}$ based on few-shot prompting. The model generates a set of n candidate code solutions by sampling from its conditional distribution:

$$C_i^* \sim P_{\mathcal{M}_{\text{gen}}}(\cdot \mid q, T_{\text{Direct}}) \quad \text{for } i = 1, \ldots, n \tag{3}$$

where T_{direct} is a predefined few-shot prompt template. The resulting set of outputs is denoted as:

$$\mathcal{G}_{\text{Direct}}(q) = \{C_1^*, \ldots, C_n^*\} \tag{4}$$

this constitutes the output of the direct generation strategy for simple problems.

In contrast, for more challenging cases classified as *Complex*, RoutingGen employs a structured reasoning strategy, which we term **ICoT-Guided Generation**. We detail this two-stage process in the following subsection.

ICoT-Guided Generation

The ICoT-guided generation process, illustrated in the right panel of Figure 2, comprises two stages. The first stage, *ICoT Generation*, explicitly models the task intention using a specification-and-idea structure that captures both the functional requirements and the global algorithmic strategy, including core logic and efficiency considerations. The second stage, *Code Generation*, directs the model to generate code conditioned on both the input question and the generated ICoT.

Stage 1: ICoT Generation In this stage, the model is prompted with the input problem to generate a diverse set of n candidate *ICoT* instances. Each instance is a structured pair of a *Specification* and an *Idea*. This generation process leverages stochastic decoding techniques (e.g., nucleus sampling) applied over the model's conditional distribution. The resulting set is denoted as $\mathcal{R}_{\text{ICoT}} = \{\text{ICoT}_i\}_{i=1}^n$, where each instance is sampled as:

$$\text{ICoT}_i \sim P_{\mathcal{M}_{\text{gen}}}(\cdot \mid q, T_{\text{ICoT}}^{(1)}) \quad \text{for } i = 1, \ldots, n \tag{5}$$

here, $T_{\text{ICoT}}^{(1)}$ denotes the prompt used for *Intention generation*, and $P_{\mathcal{M}_{\text{gen}}}$ is the conditional distribution induced by

$\mathcal{M}_{\text{gen}}$. Each ICoT_i is a unified structured output comprising a *Specification* S_i and an *Idea* I_i, i.e., $\text{ICoT}_i = (S_i, I_i)$. These pairs are generated jointly in a single decoding pass.

Stage 2: Code Generation In the second stage, each $\text{ICoT}_i = (S_i, I_i)$ from $\mathcal{R}_{\text{ICoT}}$ is used to generate a corresponding code solution C_i^* via greedy decoding. The model is conditioned on both the input problem q and its associated ICoT_i, guiding *code generation* aligned with the task's functional objective:

$$C_i^* = \text{Greedy}_{\mathcal{M}_{\text{gen}}}(q, \text{ICoT}_i, T_{\text{ICoT}}^{(2)}) \quad \text{for } i = 1, \ldots, n \quad (6)$$

where $T_{\text{ICoT}}^{(2)}$ denotes the prompt template for code generation. Each implementation C_i^* is the unique token sequence deterministically generated by $\mathcal{M}_{\text{gen}}$ under greedy decoding.

The final output is a set of n candidate code completions derived from the ICoT-guided process:

$$\mathcal{G}_{\text{ICoT}}(q) = \{C_1^*, \ldots, C_n^*\} \quad (7)$$

Experiment Setup

Benchmarks

Following recent work in LLM evaluation (OpenAI 2024; Jiang et al. 2024; Yang et al. 2025), we evaluate on six widely used code generation benchmarks. **HumanEval**(Chen et al. 2021) contains 164 Python problems with reference implementations and test cases. **MBPP-sanitized**(Austin et al. 2021) includes 427 verified tasks with three tests per instance. **HumanEval-ET** and **MBPP-ET**(Dong et al. 2025) extend the original sets with around 100 edge-case tests per problem. **OpenEval**(Yang et al. 2024) comprises 178 challenging problems from AVATAR, with manually written test cases. **McEval** (Chai et al. 2024) is a multilingual benchmark; we use its Python subset of 50 problems, adopting the difficulty labels from the original release.

Large Language Models

Difficulty-Aware Classifier. We employ Qwen3-8B as the difficulty classifier, selected for its strong performance on code reasoning tasks and competitive alignment with human preferences (Yang et al. 2025).

Code Generation. We evaluate our method on three high-performing models specialized for code generation. **Qwen2.5-Coder-3B-Instruct** (Hui et al. 2024) is a 3B-parameter instruction-tuned model in the Qwen series (formerly CodeQwen), demonstrating strong performance on code generation, mathematical reasoning, and general problem solving. **DeepSeek-Coder-6.7B-Instruct** (Guo et al. 2024) is a state-of-the-art open-source model that demonstrates robust results across multiple programming languages and standard benchmarks. We also include **DeepSeek-V3** (Liu et al. 2025), a Mixture-of-Experts model with 671B total parameters, which achieves performance competitive with or surpassing proprietary LLMs.

Baselines

Self-CoT (Yang et al. 2024) encourages the model to generate natural language reasoning before producing the final output. **Zero-shot-CoT** (Kojima et al. 2022), denoted as ZS-CoT in our results, is a zero-shot prompting approach that guides multi-step reasoning through a simple prefix. **Self-planning** (Jiang et al. 2024), denoted as SP in our results, adopts a two-stage framework, where the model first generates a numbered subtask plan and then uses it to guide code generation. **SCoT** (Li et al. 2025) incorporates sequential, branching, and looping structures into natural language reasoning to align prompts with program logic.

Evaluation Metrics

We evaluate our method along two primary axes: generative effectiveness and computational efficiency.

Effectiveness. We measure effectiveness using the Pass@k metric (Li et al. 2022), which estimates the probability that at least one correct program is included when selecting k candidates (where $k \leq n$) uniformly at random without replacement from a set of n generated candidates, among which c candidates pass all test cases. The unbiased estimator of Pass@k is defined as:

$$\text{Pass@}k := \mathrm{E}_{\text{problem}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \quad (8)$$

Efficiency. To quantify the computational efficiency promised by our framework, we measure the total number of tokens processed per problem. The total cost for a problem q is determined by the routing decision d^* and defined as the sum of its input and output token counts. For problems routed to the direct generator ($\mathcal{G}_{\text{direct}}$), the cost comprises the tokens in the single input prompt and the cumulative tokens of all n generated code candidates. For problems routed to ICoT ($\mathcal{G}_{\text{ICoT}}$), the cost is aggregated across its two-stage process. The first stage cost includes one input prompt and the n resulting intention outputs. The second stage cost includes n separate input prompts (one for each intention) and their corresponding single code outputs. Formally, let $C(\cdot)$ denote the token count of a text sequence. The input and output costs are defined as:

$$\mathcal{C}_{\text{in}}(q) = \begin{cases} C(T_{\text{direct}}(q)), & \textit{Simple} \\ C(T_{\text{ICoT}}^{(1)}(q)) + \sum_{i=1}^{n} C(T_{\text{ICoT}}^{(2)}(\text{In}_i)), & \textit{Complex} \end{cases} \quad (9)$$

$$\mathcal{C}_{\text{out}}(q) = \begin{cases} \sum_{i=1}^{n} C(\text{code}_i), & \textit{Simple} \\ \sum_{i=1}^{n} C(\text{ICoT}_i) + \sum_{i=1}^{n} C(\text{code}_i), & \textit{Complex} \end{cases} \quad (10)$$

where T_{direct} and $T_{\text{ICoT}}^{(1,2)}$ are the respective prompt templates, ICoT_i is the i-th generated ICoT, In_i refers to the pair $(q,\ \text{ICoT}_i)$ and code_i is the final code snippet generated from the corresponding path.

Method	HumanEval	HumanEval-ET	MBPP-sanitized	MBPP-ET	OpenEval	McEval
Qwen2.5-Coder-3B-Instruct						
zero-shot	75.49%	67.29%	61.42%	44.12%	35.06%	26.80%
few-shot	72.80% (-3.56%)	65.91% (-2.05%)	68.74% (+11.92%)	48.20% (+9.25%)	34.75% (-0.88%)	31.10% (+16.04%)
Self-CoT	71.55% (-5.22%)	64.18% (-4.62%)	66.01% (+7.47%)	46.51% (+5.42%)	34.13% (-2.65%)	23.50% (-12.31%)
ZS-CoT	75.70% (+0.28%)	67.99% (+1.04%)	66.73% (+8.65%)	47.24% (+7.07%)	35.42% (+1.03%)	26.20% (-2.24%)
SP	72.84% (-3.51%)	64.02% (-4.86%)	53.69% (-12.59%)	36.93% (-16.30%)	35.65% (+1.68%)	25.30% (-5.60%)
SCoT	65.27% (-13.54%)	58.60% (-12.91%)	61.62% (+0.33%)	41.87% (-5.10%)	33.57% (-4.25%)	35.10% (+30.97%)
RoutingGen	**76.65% (+1.54%)**	**69.02% (+2.57%)**	**68.84% (+13.71%)**	**49.33% (+11.81%)**	**35.76% (+2.00%)**	**35.30% (+31.72%)**
ICoT	**77.10% (+2.13%)**	**69.73% (+3.63%)**	**69.11% (+12.52%)**	**48.58% (+10.11%)**	**35.70% (+1.83%)**	**38.90% (+45.15%)**
DeepSeek-Coder-6.7B-Instruct						
zero-shot	45.95%	39.60%	46.08%	31.10%	16.74%	23.80%
few-shot	72.68% (+58.17%)	64.27% (+62.30%)	**73.40% (+59.29%)**	51.64% (+66.05%)	37.89% (+126.34%)	39.30% (+65.13%)
Self-CoT	68.29% (+48.62%)	60.27% (+52.20%)	36.70% (-20.36%)	23.95% (-22.99%)	35.34% (+111.11%)	38.70% (+62.61%)
ZS-CoT	63.66% (+38.54%)	55.58% (+40.35%)	38.01% (-17.51%)	24.75% (-20.42%)	34.35% (+105.20%)	34.80% (+46.22%)
SP	58.90% (+28.18%)	52.47% (+32.50%)	49.19% (+6.75%)	33.72% (+8.42%)	25.17% (+50.36%)	30.50% (+28.15%)
SCoT	70.79% (+54.06%)	62.87% (+58.76%)	67.70% (+46.92%)	46.58% (+49.77%)	36.91% (+120.49%)	40.90% (+71.85%)
RoutingGen	**73.51% (+59.97%)**	**64.76% (+63.53%)**	72.30% (+56.90%)	**51.73% (+66.33%)**	**38.74% (+131.42%)**	**41.00% (+72.27%)**
ICoT	**73.14% (+59.17%)**	**65.03% (+64.22%)**	71.24% (+54.60%)	50.85% (+63.50%)	**38.23% (+128.38%)**	**42.90% (+80.25%)**
DeepSeek-V3-671B						
zero-shot	85.61%	77.93%	88.99%	63.04%	46.97%	33.20%
few-shot	84.76% (-0.99%)	75.73% (-2.82%)	89.23% (+0.27%)	**63.23% (+0.30%)**	42.81% (-8.86%)	51.60% (+55.42%)
Self-CoT	91.34% (+6.69%)	81.71% (+4.85%)	83.98% (-5.63%)	60.09% (-4.68%)	45.51% (-3.11%)	64.40% (+93.98%)
ZS-CoT	90.85% (+6.12%)	82.32% (+5.63%)	84.92% (-4.57%)	60.42% (-4.16%)	41.01% (-12.69%)	54.00% (+62.65%)
SP	80.37% (-6.12%)	73.17% (-6.11%)	80.52% (-9.52%)	55.93% (-11.28%)	42.81% (-8.86%)	51.60% (+55.42%)
SCoT	90.98% (+6.27%)	81.34% (+4.38%)	79.30% (-10.89%)	54.71% (-13.21%)	47.08% (+0.23%)	60.00% (+80.72%)
RoutingGen	**91.83% (+7.27%)**	82.07% (+5.31%)	**90.21% (+1.37%)**	63.14% (+0.16%)	**47.98% (+2.15%)**	**65.20% (+96.39%)**
ICoT	**92.07% (+7.55%)**	**82.68% (+6.10%)**	80.70% (-9.31%)	56.25% (-10.77%)	**47.30%(+0.70%)**	**67.20% (+102.41%)**

Table 1: Pass@1 comparison across three models and six code generation benchmarks. Results cover two direct generation baselines, four structured prompting baselines, and the proposed RoutingGen and ICoT. Bold and underline indicate the best and second-best among our proposed methods and all baselines. Parentheses indicate relative improvements over zero-shot.

The total token usage per problem is:

$$\text{Cost}(q) = \mathcal{C}_{\text{in}}(q) + \mathcal{C}_{\text{out}}(q) \tag{11}$$

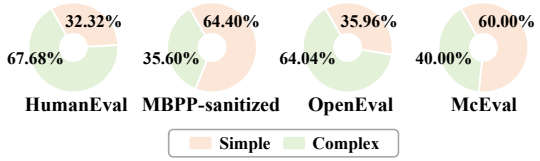

Figure 3: Difficulty perception results across four evaluation benchmarks. These distributions reflect the model's understanding of task complexity and inform the routing decisions in RoutingGen.

Sampling Settings

Following recent work (Jiang et al. 2024; Li et al. 2025), we employ nucleus sampling with top-p filtering to generate candidate programs, ensuring fair comparison across all methods. By default, we generate 20 candidates per problem. For single-stage approaches such as zero-shot and few-shot prompting, we set the sampling temperature to 0.8, top-p to 0.95, and the maximum output length to 300 tokens. In few-shot settings, we select three representative question-code examples, and apply the same configuration in RoutingGen to ensure consistency with the baseline. For Self-CoT, the maximum length is extended to 600 tokens to accommodate longer reasoning chains, while other parameters remain unchanged. For multi-stage approaches including SCoT, ICoT, and Self-Planning, we sample 20 reasoning chains with temperature 0.8, followed by deterministic code generation with temperature 0. Both stages are capped at 300 tokens. An exception is made for DeepSeek-V3, where we generate 5 candidates per problem due to API constraints. All other configurations remain identical to the above.

Experimental Results

Effectiveness and Efficiency of RoutingGen

This section presents a comprehensive evaluation of the proposed RoutingGen framework from three perspectives: overall performance as reported in Table 1, generation cost in terms of token usage in Table 2, and difficulty-aware routing outcomes illustrated in Figure 3.

Main Performance. As presented in Table 1, RoutingGen achieves state-of-the-art Pass@1 performance across most models and benchmarks, demonstrating both accuracy and generality.

MT	HE	MP	OE	ME	Avg
RT	42,872	61,001	35,328	0	34,800
Qwen2.5-Coder-3B-Instruct					
SCoT	4,191,205	12,857,146	4,294,749	1,501,005	5,711,026
ICoT	4,275,896	12,770,188	4,490,805	1,560,068	5,774,239
RG	3,309,539	4,909,856	3,149,950	835,969	3,051,329
R2S	**881,666 (21.04%)**	**7,947,290 (61.81%)**	**1,144,799 (26.66%)**	**665,036 (44.31%)**	**2,659,698 (46.57%)**
R2I	**966,357 (22.60%)**	**7,860,332 (61.55%)**	**1,340,855 (29.86%)**	**724,099 (46.41%)**	**2,722,911 (47.16%)**
DeepSeek-Coder-6.7B-Instruct					
SCoT	5,118,415	14,762,151	5,295,607	1,865,235	6,760,352
ICoT	4,980,141	14,706,320	4,867,249	1,760,589	6,578,575
RG	3,774,494	6,020,323	3,429,603	925,268	3,537,422
R2S	**1,343,921 (26.26%)**	**8,741,828 (59.22%)**	**1,866,004 (35.24%)**	**939,967 (50.39%)**	**3,222,930 (47.67%)**
R2I	**1,205,647 (24.21%)**	**8,685,997 (59.06%)**	**1,437,646 (29.54%)**	**835,321 (47.45%)**	**3,041,153 (46.23%)**
DeepSeek-V3-671B					
SCoT	1,333,100	4,335,969	1,399,146	471,552	1,884,942
ICoT	1,225,360	4,071,784	1,324,069	416,401	1,759,404
RG	993,628	1,907,516	1,050,021	230,027	1,045,298
R2S	**339,472 (25.46%)**	**2,428,453 (56.01%)**	**349,125 (24.95%)**	**241,525 (51.22%)**	**839,644 (44.54%)**
R2I	**231,732 (18.91%)**	**2,164,268 (53.15%)**	**274,048 (20.70%)**	**186,374 (44.76%)**	**714,106 (40.59%)**

Table 2: Total token usage of SCoT, ICoT, and RoutingGen across benchmarks. RT denotes token usage of the routing module; RG indicates the total usage of RoutingGen including inference and routing tokens. R2S and R2I represent RG's token reduction relative to SCoT and ICoT, showing both absolute and percentage decreases. Avg denotes the average of per-benchmark token reductions and the corresponding relative proportion with respect to the total token usage of the baseline method. MT refers to the prompting method. HE, MP, OE, and ME refer to the HumanEval, MBPP-sanitized, OpenEval, and McEval benchmarks, respectively.

For instance, it reaches 90.21% on MBPP-sanitized and 91.83% on HumanEval with DeepSeek-V3-671B. Notably, structured prompts underperform compared to simpler few-shot approaches on benchmarks such as MBPP-sanitized. The efficiency of RoutingGen derives from its difficulty-aware routing strategy. As shown in Figure 3, a majority of tasks in MBPP-sanitized and McEval are classified as *Simple*, accounting for 64.40% and 60.00%, respectively, while HumanEval and OpenEval exhibit substantially lower simple-task proportions, both below 36%.

As shown in Table 2, RoutingGen consistently and substantially reduces token usage compared to both SCoT and ICoT across all evaluated models and benchmarks. All reported token usage values include both routing tokens and inference tokens. On average, RoutingGen achieves a 46.37% relative reduction in total token usage across settings. For instance, on the MBPP-sanitized benchmark with DeepSeek-Coder-6.7B-Instruct, RoutingGen reduces token usage by 8.74 million compared to SCoT, corresponding to a 59.22% reduction. Similar trends are consistently observed across other datasets and models, underscoring the efficiency gains introduced by RoutingGen's difficulty-aware routing strategy.

Analysis. The results empirically validate a key limitation: indiscriminate use of complex prompting strategies often leads to overthinking on simple tasks, resulting in reduced performance and increased computational cost. On MBPP-sanitized and MBPP-ET, which contain a higher proportion of simpler problems, several structured prompting methods perform worse than simpler zero-shot or few-shot baselines. For instance, on MBPP-ET, Self-CoT with DeepSeek-Coder-6.7B-Instruct exhibits the largest performance drop, with its Pass@1 score falling 22.99% below the zero-shot baseline.

In contrast, RoutingGen mitigates this limitation by dynamically selecting prompting strategies based on task difficulty, achieving more accurate and efficient code generation. On MBPP-sanitized (64.40% Simple by our classifier), RoutingGen with DeepSeek-Coder-6.7B-Instruct routes most tasks to few-shot generation, achieving a 59.22% token reduction and an accuracy improvement of 4.60 percentage points over the resource-intensive SCoT (72.30% vs. 67.70%). Conversely, on HumanEval (67.68% Complex by our classifier), RoutingGen achieves a comparable performance (73.51%) to ICoT (73.14%), with a moderate token reduction of 24.21%, reflecting the necessary computational investment for harder problems. The consistent performance across three distinct models and six benchmarks demonstrates the robustness of RoutingGen to variations in model architecture and task distribution. In particular, discrepancies between our baseline results and official reports are expected, as our standardized instructions used for fair cross-model comparison differ from the highly optimized model-specific prompts designed to maximize reported performance (Luo et al. 2024). Detailed experimental settings and results are provided in the Appendix.

Effectiveness of ICoT Prompting

To further assess the standalone effectiveness of ICoT within the RoutingGen framework, we evaluate ICoT under a static prompting configuration without dynamic routing.

Results. As shown in Table 1, ICoT consistently outperforms baseline prompting methods across the majority of models and benchmarks. The gains are particularly notable when using Qwen2.5-Coder-3B-Instruct and DeepSeek-Coder-6.7B-Instruct. For instance, with DeepSeek-Coder-6.7B-Instruct, ICoT achieves a Pass@1 score of 38.23% on OpenEval, representing a 128.38% relative improvement over the zero-shot baseline. Similarly, with Qwen2.5-Coder-3B-Instruct, ICoT demonstrates strong performance across

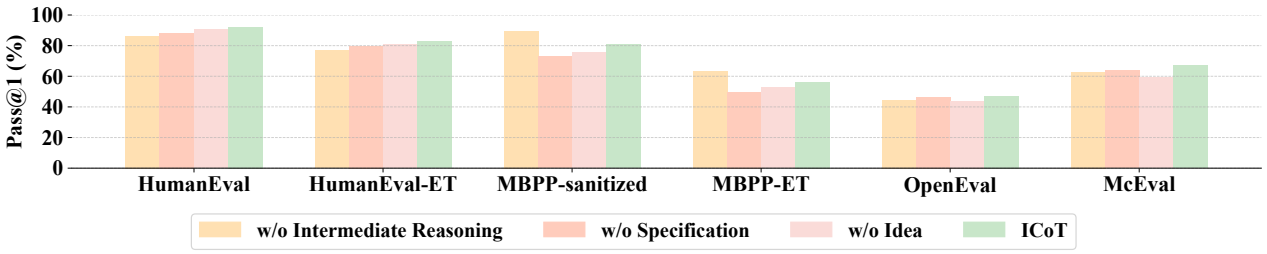

Figure 4: Pass@1 accuracy of ICoT and its ablated variants across six benchmarks using DeepSeek-V3-671B. "ICoT" denotes the model with both specification and idea stages. "w/o Intermediate Reasoning" removes the structured reasoning component entirely. "w/o Specification" omits the specification stage, while "w/o Idea" excludes the idea stage.

all six benchmarks, including a 45.15% relative improvement on McEval. Moreover, ICoT demonstrates strong scalability on DeepSeek-V3-671B, achieving a Pass@1 score of 82.68% on HumanEval-ET and 67.20% on McEval. These results represent relative improvements of 6.10% and 102.41% over the respective zero-shot baselines.

Analysis. The effectiveness of ICoT stems from its two-stage process, which guides the model from explicit modeling of task requirements to core algorithmic design, as illustrated in the "same_chars" example (Figure 1(B)). In the Specification stage, the model grounds the task by defining the inputs (two strings s0 and s1) and the functional requirement of the output (a boolean indicating whether both strings contain the same set of characters). Crucially, the subsequent Idea stage abstracts the core algorithmic logic. Instead of prescribing a surface-level procedural loop as in the SCoT baseline, it formulates an intention abstraction: "convert both strings to sets of characters" and then "compare the sets." This abstraction, which also includes an explicit consideration of time complexity ($O(m+n)$), directly steers code generation toward a concise and correct solution: "return set(s0) == set(s1)".

Dataset	Method	Simple	Complex	Total	Pass@1
HumanEval	Qwen3-8B	53	111	164	73.51%
	GPT-4o	78	86	164	72.99%
	Different pairs: 37		*Different rate*: 22.56%		
MBPP-sanitized	Qwen3-8B	275	152	427	72.30%
	GPT-4o	363	64	427	72.79%
	Different pairs: 100		*Different rate*: 23.42%		
OpenEval	Qwen3-8B	64	114	178	38.74%
	GPT-4o	69	109	178	38.51%
	Different pairs: 15		*Different rate*: 8.43%		

Table 3: Comparison of difficulty-aware routing and code generation across models. This table compares task-level difficulty labels Simple and Complex classified by Qwen3-8B and GPT-4o, and reports Pass@1 scores based on DeepSeek-Coder-6.7B-Instruct. The "Different pairs" indicates the number of tasks with conflicting difficulty labels between the two models, while "Different rate" denotes their proportion relative to the total.

Ablation Analysis

Robustness to Difficulty Classifier Variants. We evaluate the robustness of RoutingGen under different difficulty classifiers by comparing Qwen3-8B and GPT-4o on three benchmarks. As shown in Table 3, the two models produce conflicting difficulty labels for 22.56% of HumanEval, 23.42% of MBPP-sanitized, and 8.43% of OpenEval tasks. Despite this variation, RoutingGen consistently outperforms all baseline methods under both classifiers. On MBPP-sanitized, it achieves Pass@1 scores of 72.30% and 72.79% with Qwen3-8B and GPT-4o, respectively. Similar trends are also observed when applying self-routing. The consistent gains across classifier variants show that RoutingGen generalizes well under different difficulty estimation conditions.

Effectiveness of ICoT Components. As shown in Figure 4, the full ICoT method consistently outperforms all ablated variants on the more challenging benchmarks, confirming the synergistic contribution of both the *Specification* and *Idea* stages. For example, on HumanEval, removing the Specification stage reduces Pass@1 from 92.07% to 88.17%, while on McEval, removing the Idea stage leads to a drop from 67.20% to 59.20%. Concurrently, the variant without intermediate reasoning achieves the strongest results on the simpler MBPP-sanitized and MBPP-ET datasets. This supports our prior finding that elaborate prompt structures can be counterproductive for simple problems.

Conclusion

In this work, we address two key limitations in code generation: overthinking caused by uniformly applying structured prompting on simple tasks and the lack of intention abstraction in existing methods. We propose RoutingGen, a difficulty-aware routing framework that dynamically adapts prompting strategies using a classifier to direct simple problems to direct generation and complex ones to ICoT, a two-stage reasoning process that formulates specifications and algorithmic ideas. Experiments on three models and six benchmarks demonstrate that RoutingGen achieves state-of-the-art performance with significantly reduced token usage, and ICoT consistently outperforms other prompting baselines on challenging benchmarks.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. 62372071), the Chongqing Technology Innovation and Application Development Project (No. CSTB2022TIAD-STX0007 and No. CSTB2023TIAD-STX0025), and in part by the Fundamental Research Funds for the Central Universities under Grant 2023CDJKYJH013.

References

Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; and Sutton, C. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732.

Chai, L.; Liu, S.; Yang, J.; Yin, Y.; Jin, K.; Liu, J.; Sun, T.; Zhang, G.; Ren, C.; Guo, H.; Wang, Z.; Wang, B.; Wu, X.; Wang, B.; Li, T.; Yang, L.; Duan, S.; and Li, Z. 2024. McEval: Massively Multilingual Code Evaluation. arXiv:2406.07436.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; 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.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; and Zaremba, W. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

Chen, W.; Ma, X.; Wang, X.; and Cohen, W. W. 2023. Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks. *Transactions on Machine Learning Research*.

Cobbe, K.; Kosaraju, V.; Bavarian, M.; Chen, M.; Jun, H.; Kaiser, L.; Plappert, M.; Tworek, J.; Hilton, J.; Nakano, R.; Hesse, C.; and Schulman, J. 2021. Training Verifiers to Solve Math Word Problems. arXiv:2110.14168.

Dong, Y.; Ding, J.; Jiang, X.; Li, G.; Li, Z.; and Jin, Z. 2025. Codescore: Evaluating code generation by learning code execution. *ACM Transactions on Software Engineering and Methodology*, 34(3): 1–22.

Gao, L.; Madaan, A.; Zhou, S.; Alon, U.; Liu, P.; Yang, Y.; Callan, J.; and Neubig, G. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, 10764–10799. PMLR.

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.; and Liang, W. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196.

Han, H.; Kim, J.; Yoo, J.; Lee, Y.; and Hwang, S.-w. 2024. ArchCode: Incorporating Software Requirements in Code Generation with Large Language Models. In Ku, L.-W.; Martins, A.; and Srikumar, V., eds., *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 13520–13552. Bangkok, Thailand: Association for Computational Linguistics.

Huang, D.; Bu, Q.; Qing, Y.; and Cui, H. 2024. CodeCoT: Tackling Code Syntax Errors in CoT Reasoning for Code Generation. arXiv:2308.08784.

Hui, B.; Yang, J.; Cui, Z.; Yang, J.; Liu, D.; Zhang, L.; Liu, T.; Zhang, J.; Yu, B.; Lu, K.; Dang, K.; Fan, Y.; Zhang, Y.; Yang, A.; Men, R.; Huang, F.; Zheng, B.; Miao, Y.; Quan, S.; Feng, Y.; Ren, X.; Ren, X.; Zhou, J.; and Lin, J. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186.

Jiang, X.; Dong, Y.; Wang, L.; Fang, Z.; Shang, Q.; Li, G.; Jin, Z.; and Jiao, W. 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7): 1–30.

Kahneman, D. 2011. *Thinking, fast and slow*. macmillan.

Kaplan, J.; McCandlish, S.; Henighan, T.; Brown, T. B.; Chess, B.; Child, R.; Gray, S.; Radford, A.; Wu, J.; and Amodei, D. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Kojima, T.; Gu, S. S.; Reid, M.; Matsuo, Y.; and Iwasawa, Y. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213.

Li, J.; Li, G.; Li, Y.; and Jin, Z. 2025. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2): 1–23.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; Hubert, T.; Choy, P.; de Masson d'Autume, C.; Babuschkin, I.; Chen, X.; Huang, P.-S.; Welbl, J.; Gowal, S.; Cherepanov, A.; Molloy, J.; Mankowitz, D. J.; Sutherland Robson, E.; Kohli, P.; de Freitas, N.; Kavukcuoglu, K.; and Vinyals, O. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624): 1092–1097.

Liu, A.; Feng, B.; Xue, B.; Wang, B.; Wu, B.; Lu, C.; Zhao, C.; Deng, C.; Zhang, C.; Ruan, C.; et al. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437.

Luo, Z.; Xu, C.; Zhao, P.; Sun, Q.; Geng, X.; Hu, W.; Tao, C.; Ma, J.; Lin, Q.; and Jiang, D. 2024. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. In *The Twelfth International Conference on Learning Representations*.

Nye, M.; Andreassen, A. J.; Gur-Ari, G.; Michalewski, H.; Austin, J.; Bieber, D.; Dohan, D.; Lewkowycz, A.; Bosma, M.; Luan, D.; Sutton, C.; and Odena, A. 2021. Show Your Work: Scratchpads for Intermediate Computation with Language Models. arXiv:2112.00114.

OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774.

Polya, G. 1945. *How to solve it: A new aspect of mathematical method*. Princeton university press.

Shinn, N.; Cassano, F.; Gopinath, A.; Narasimhan, K.; and Yao, S. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36: 8634–8652.

Stanovich, K. E.; and West, R. F. 2000. Individual differences in reasoning: Implications for the rationality debate? *Behavioral and Brain Sciences*, 23(5): 645–665.

Tian, Y.; Yan, W.; Yang, Q.; Zhao, X.; Chen, Q.; Wang, W.; Luo, Z.; Ma, L.; and Song, D. 2025. Codehalu: Investigating code hallucinations in llms via execution-based verification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 25300–25308.

Varangot-Reille, C.; Bouvard, C.; Gourru, A.; Ciancone, M.; Schaeffer, M.; and Jacquenet, F. 2025. Doing More with Less: A Survey on Routing Strategies for Resource Optimisation in Large Language Model-Based Systems. arXiv:2502.00409.

Wang, L.; Xu, W.; Lan, Y.; Hu, Z.; Lan, Y.; Lee, R. K.-W.; and Lim, E.-P. 2023a. Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models. arXiv:2305.04091.

Wang, X.; Feng, S.; Li, Y.; Yuan, P.; Zhang, Y.; Tan, C.; Pan, B.; Hu, Y.; and Li, K. 2025. Make Every Penny Count: Difficulty-Adaptive Self-Consistency for Cost-Efficient Reasoning. In Chiruzzo, L.; Ritter, A.; and Wang, L., eds., *Findings of the Association for Computational Linguistics: NAACL 2025*, 6904–6917. Albuquerque, New Mexico: Association for Computational Linguistics. ISBN 979-8-89176-195-7.

Wang, X.; Wei, J.; Schuurmans, D.; Le, Q.; Chi, E.; Narang, S.; Chowdhery, A.; and Zhou, D. 2023b. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le, Q. V.; Zhou, D.; et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35: 24824–24837.

Yang, A.; Li, A.; Yang, B.; Zhang, B.; Hui, B.; Zheng, B.; Yu, B.; Gao, C.; Huang, C.; Lv, C.; Zheng, C.; Liu, D.; Zhou, F.; Huang, F.; Hu, F.; Ge, H.; Wei, H.; Lin, H.; Tang, J.; Yang, J.; Tu, J.; Zhang, J.; Yang, J.; Yang, J.; Zhou, J.; Zhou, J.; Lin, J.; Dang, K.; Bao, K.; Yang, K.; Yu, L.; Deng, L.; Li, M.; Xue, M.; Li, M.; Zhang, P.; Wang, P.; Zhu, Q.; Men, R.; Gao, R.; Liu, S.; Luo, S.; Li, T.; Tang, T.; Yin, W.; Ren, X.; Wang, X.; Zhang, X.; Ren, X.; Fan, Y.; Su, Y.; Zhang, Y.; Zhang, Y.; Wan, Y.; Liu, Y.; Wang, Z.; Cui, Z.; Zhang, Z.; Zhou, Z.; and Qiu, Z. 2025. Qwen3 Technical Report. arXiv:2505.09388.

Yang, G.; Zhou, Y.; Chen, X.; Zhang, X.; Zhuo, T. Y.; and Chen, T. 2024. Chain-of-thought in neural code generation: From and for lightweight language models. *IEEE Transactions on Software Engineering*.

Yao, S.; Yu, D.; Zhao, J.; Shafran, I.; Griffiths, T.; Cao, Y.; and Narasimhan, K. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36: 11809–11822.

Yin, P.; and Neubig, G. 2017. A Syntactic Neural Model for General-Purpose Code Generation. arXiv:1704.01696.

Zelikman, E.; Lorch, E.; Mackey, L.; and Kalai, A. T. 2024. Self-taught optimizer (stop): Recursively self-improving code generation. In *First Conference on Language Modeling*.

Zhang, Z.; Zhang, A.; Li, M.; and Smola, A. 2022. Automatic Chain of Thought Prompting in Large Language Models. arXiv:2210.03493.

Zheng, K.; Decugis, J.; Gehring, J.; Cohen, T.; Negrevergne, B.; and Synnaeve, G. 2025. What Makes Large Language Models Reason in (Multi-Turn) Code Generation? arXiv:2410.08105.

Zhou, D.; Schärli, N.; Hou, L.; Wei, J.; Scales, N.; Wang, X.; Schuurmans, D.; Cui, C.; Bousquet, O.; Le, Q.; and Chi, E. 2023. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. arXiv:2205.10625.