Intent↑Coding: Amplifying User Intent in Code Generation

Zheng Fang[1], Yihong Dong[1], Lili Mou[2,3], Dongming Jin[1], Zhi Jin[1], Ge Li[1]*

[1] School of Computer Science, Peking University

[2] Department of Computing Science, University of Alberta

[3] Canada CIFAR AI Chair

fangz@pku.edu.cn dongy

Abstract

Large Language Models (LLMs) have shov strong capabilities in code generation, but th adherence to fine-grained user intent with m tiple constraints remains a significant challen Our empirical analysis reveals two key obs vations: 1) Model performance deteriorat quickly as the number of constraints in t user intent increases, and 2) While user tent does influence the model's logits, su an influence may not be strong enough to fectively steer the decoding process. To tl end, we propose Intent-Amplified Code Gen ation (*Intent↑Coding*), a novel decoding str egy that enhances an LLM's ability to f low user intent. *Intent↑Coding* captures t influence of user intent by masking out t intent, and applies a multi-strength ensemble mechanism to amplify the effect of user intent during generation. *Intent↑Coding* is model-agnostic, requires no additional training, and integrates seamlessly with existing decoding procedures. To enable systematic evaluation, we also construct *CodeConstraints*, a benchmark dataset specifically designed to test user intent compliance under varying numbers of constraints. Experiments on our constructed *CodeConstraints*, as well as popular IFEvalCode, HumanEval and LiveCodeBench datasets, show that our *Intent↑Coding* model significantly improves both constraint satisfaction and functional correctness compared to standard decoding approaches. *Intent↑Coding* achieves up to 71.0% relative improvement on *CodeConstraints*, achieves up to 67.3% relative improvement on IFEvalCode and achieves up to 29.3% relative improvement in pass@1 on HumanEval and LiveCodeBench compared with greedy decoding.

```
def func(n):
    """ Given a positive integer n, return a list of integers.
        The integers should be greater than or equal to n.
        The length of the list should be 41.
    """
    # base case
    if n == 1:
        return [1] * 41
    # recursive case
    else:
        return [n] + func(n-1)
```

Figure 1: An example of generated code given by CodeLlama (Rozière et al., 2023), where the highlighted constraint is not satisfied. CodeLlama successfully handles an edge case ($n = 1$, green ✔). But the code (red ✖) violates the required constraints when considering a broader range of inputs.

1 Introduction

Code generation, a fundamental task in software engineering, refers to the automatic creation of programming code from a certain form of user intent, typically given by a natural language description (Svyatkovskiy et al., 2020; Chen et al., 2021; Li et al., 2022). Code generation can significantly accelerate software development and reduce human efforts (Dong et al., 2024; Jiang et al., 2024a). The recent success of large language models (LLMs) has catalyzed a new paradigm in this area, where LLMs are trained on large amounts of data to generate syntactically and semantically correct code (Li et al., 2022; Rozière et al., 2023). This paradigm is particularly promising for improving developers' productivity and enabling novice programmers to accomplish complex tasks.

Previous work on LLM-based code generation is typically based on autoregressive models, where each token is predicted based on the user intent (natural language description) as well as all previously generated tokens (Jiang et al., 2024b). To translate the model's probabilistic outputs into final code, researchers have explored various decoding methods, e.g., greedy decoding and sampling methods (Freitag and Al-Onaizan, 2017; Zhu et al., 2024).

*Corresponding authors.

Findings of the Association for Computational Linguistics: ACL 2026, pages 33246–33261
July 2-7, 2026 ©2026 Association for Computational Linguistics

However, a user intent may specify a number of constraints for a desired program, but a standard decoding method often fails to fully satisfy the constraints. In Figure 1, for example, the natural language descriptions specify that 1) The code should return a list, 2) The element in list should be integer, 3) The integer in list should be greater than or equal to n, and 4) The length of the list should be 41. However, the code generated by CodeLlama (Rozière et al., 2023) satisfies constraints 1) and 2), but fails to satisfy 3) and 4).

In our work, we make two key observations on how an LLM addresses, or fails to address, the user intent. **Observation 1:** LLM's performance deteriorates quickly as the number of user-specified constraints increases. To interpret the phenomenon, we compare the logits with and without feeding the user intent, leading to **Observation 2:** Although feeding the user intent influences the LLM's logits, this effect is often too weak to guide standard decoding methods effectively, resulting in failures to satisfy certain constraints.

To this end, we propose Intent-Amplified Code Generation (*Intent*$^{\uparrow}$*Coding*), a novel decoding strategy that enhances an LLM's compliance with user intent. At each decoding step, our method generates the original logits using a prefix of the user intent and previously generated tokens, while simultaneously computing intent-masked logits using the same prefix with the user intent masked. Contrasting the original logits and the intent-masked logits captures the influence of user intent. The influence is then scaled and added to the original logits. Our approach is inspired by contrastive decoding (Li et al., 2023b; Kim et al., 2024), but differs in that we do not rely on a single, fixed scale of the contrastive signal. Instead, our method explores multiple scale values at each step and ensembles them during beam search. This allows the decoding process to dynamically adjust the contrastive scale for amplifying user intent. Notably, our method is model-agnostic and training-free.

To facilitate research on multi-constraint user intent modeling, we construct a new benchmark dataset, called *CodeConstraints*. We accomplish this by combining one or more primitive constraints (e.g., data type, return type, length, and value) to form a user intent. We control the task difficulty by varying the number of active constraints. Compared with existing datasets, our constructed *CodeConstraints* have unique features: each primitive constraint is simple and within the capability of current LLMs. Consequently, our dataset focuses on testing LLMs' composability of multiple constraints in user intent. Further, our methodology of data construction is highly extensible and flexible, and can be easily scaled to arbitrary sizes and adapted to include new constraint types.

We evaluated our *Intent*$^{\uparrow}$*Coding* approach on two general benchmarks HumanEval (Chen et al., 2021) and LiveCodeBench (Jain et al., 2025), as well as two constraint-following benchmarks, IFEvalCode (Yang et al., 2025) dataset and our constructed *CodeConstraints* dataset. The experimental results consistently demonstrate the effectiveness of our approach.

In summary, our contributions are as follows: 1) We propose a novel approach, Intent Amplified Code Generation (*Intent*$^{\uparrow}$*Coding*), that enhances the model's ability to follow user intent. 2) We construct *CodeConstraints*, a new benchmark dataset specifically designed to evaluate how well LLMs can satisfy user intent with multiple constraints. 3) We conduct comprehensive experiments on both general and proposed benchmarks, demonstrating the effectiveness and superiority of our method.

2 Related Work

2.1 Code Generation with LLMs

Recent advances in LLMs have driven substantial progress in code generation. Foundational models such as AlphaCode (Li et al., 2022) and CodeGen (Nijkamp et al., 2023) demonstrated the viability of this approach for complex programming tasks. This has been followed by a wave of powerful open-source models, including StarCoder (Li et al., 2023a), CodeLlama (Rozière et al., 2023), DeepSeek-Coder (Guo et al., 2024), and Qwen2.5-Coder (Hui et al., 2024), which have further advanced the state of the art.

Beyond scaling base models, another line of research focuses on enhancing code generation at the decoding stage of LLMs. Execution-guided methods leverage runtime feedback to improve code quality. Some of these approaches generate multiple candidate programs and re-rank them based on test case outcomes (Li et al., 2022; Chen et al., 2024). Others employ more sophisticated techniques like Monte Carlo Tree Search to explore promising solution paths informed by execution results (Zhang et al., 2023b). To reduce reliance on sampling, recent methods employ iterative refinement: Self-Edit (Zhang et al., 2023a) introduces an auxiliary model that corrects erroneous outputs based on test feedback, while ROCODE (Jiang et al., 2025) integrates backtracking and static analysis into the decoding process, enabling the model to detect and repair errors during generation. Although effective, these techniques often depend on external feedback, such as test cases or compilers, limiting their applicability in real-world settings where such signals are unavailable.

Code generation benchmarks have evolved from basic function synthesis to more realistic and context-rich tasks. HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) evaluate models on standalone Python functions with unit tests. APPS (Hendrycks et al., 2021) scales up to thousands of real-world problems, emphasizing broader reasoning and domain coverage. LiveCodeBench (Jain et al., 2025) is a continuously updated benchmark that assesses LLMs on recent competitive programming problems from online platforms. These benchmarks enable comprehensive and reliable evaluation of code generation capabilities across diverse scenarios.

2.2 Decoding Methods

Contemporary LLM-based code generation relies fundamentally on decoding strategies to balance precision and expressive flexibility. Traditional techniques can be categorized into deterministic and sampling-based approaches. Greedy decoding selects the single highest-probability token at each time step, while beam search (Freitag and Al-Onaizan, 2017) maintains multiple top-scoring hypotheses to produce a high-probability sequence in a global sense. Sampling-based methods generate the token in a stochastic way, but often the distribution is reshaped before selecting tokens, such as temperature sampling (Renze, 2024), top-k sampling (Fan et al., 2018) and nucleus sampling (Holtzman et al., 2020). Building on temperature sampling, AdapT sampling (Zhu et al., 2024) analyzes token-level loss distributions, classifies tokens into challenging tokens (e.g., block-start) and confident tokens, and applies a higher temperature to challenging tokens and a lower temperature for confident tokens. DoLa (Chuang et al., 2023) improves generation quality by contrasting the logits from premature and mature layers during decoding. Selective Prompt Anchoring (SPA) (Tian and Zhang, 2025) adjusts the next-token logits to emphasize user-specified context. But its effectiveness depends on tuning a hyperparameter on a calibration dataset split. By contrast, our method does not need this kind of hyperparameter search for a single best strength. Instead, we propose a new ensemble method.

3 Methodology

3.1 Construction of the *CodeConstraints*

Researchers have proposed a variety of benchmarks such as HumanEval (Chen et al., 2021) and LiveCodeBench (Jain et al., 2025) to evaluate code generation. While these benchmarks cover diverse tasks and granularities, they primarily assess functional correctness through test case execution. However, some prompts in these benchmarks contain ambiguous user intent (Siddiq et al., 2024), and some include overly complex user intent that exceed the capabilities of current LLMs (Dai et al., 2024). These benchmarks also lack the ability to assess whether specific constraints are satisfied when multiple constraints are present in the prompts. This gap motivates the design of our benchmark.

In our work, we constructed *CodeConstraints*, a new benchmark dataset for code generation that emphasizes the capture of user intent. The dataset provides a controlled environment where the complexity and quantity of constraints can be precisely manipulated.

The data construction process is founded on four core primitive constraints:

- **Data Type:** Specifying the numerical type of the elements to be generated (e.g., `integer`, `float`).
- **Return Format:** Defining the collection type for the output (e.g., `list`, `tuple`, `set`).
- **Length:** Imposing a constraint on the exact number of elements in the returned collection.
- **Value:** Restricting the numerical range of the elements (e.g., must be greater than or less than a given value).

These primitives are combinatorially sampled to construct problems with varying levels of difficulty, where the level number corresponds to the quantity of active constraints in the prompt. For instance, a Level 1 task might only specify the return format, whereas a Level 4 task combines all four primitive types into a single user intent. This hierarchical structure is designed to enable a granular analysis of model performance as the number of constraints increases.

We generate the *CodeConstraints* dataset programmatically using a set of pre-defined constraint

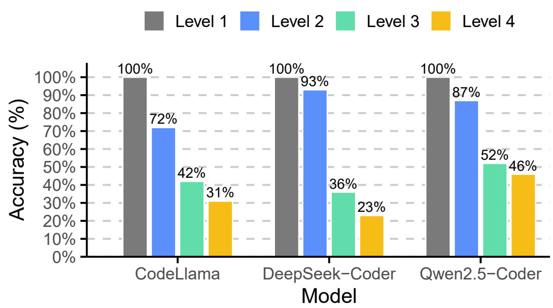

Figure 2: Performance of the 7B models on the *Co Constraints* dataset across various complexity lev using greedy search for all evaluations.

rules. This rule-based approach offers significa advantages. It ensures both scalability and tensibility. The dataset is extensible to arbitra sizes. Our benchmark consists of 300 Level 2 problems, 100 Level 3 problems, and 100 Level 4 problems. Level 1 problems are trivial and can be easily solved by all major LLMs. Its modular design also allows new constraint types to be easily integrated for future work. Despite this potential for expansion, our experiments confirm the current dataset is already highly challenging, proving sufficiently difficult for current LLMs.

For evaluation, we use the accuracy metric, where a program is considered correct if all constraints are satisfied. This follows the rigid nature of program functionality, as opposed to more flexible natural language.

3.2 Empirical Observations

We conduct an empirical analysis on the failure of LLMs to adhere to user intent during code generation. Our analysis leverages the HumanEval and our *CodeConstraints* benchmarks. This investigation yields two key observations.

Observation 1. Our first observation is that the performance of LLMs on code generation deteriorates quickly as the number of user-specified constraints increases. This trend is strongly supported by empirical evidence from our *CodeConstraints* benchmark. As shown in Figure 2, DeepSeek-Coder drops from 93% accuracy with two constraints to 23% with four. Similar declines are observed for other LLMs. The high performance under low-constraint settings indicates that LLMs are capable of understanding and satisfying individual constraints. However, their compliance ability degrades significantly as task complexity increases.

```
def is_equal_to_sum_even(n):
  """Evaluate whether the given number n can be
    written as the sum of exactly 4 positive even numbers
    Example
    is_equal_to_sum_even(4) == False
    is_equal_to_sum_even(6) == False
    is_equal_to_sum_even(8) == True
  """
  if n % 2 != 0:
    return False
  else:
    for i in range(1, n):
      for j in range(1, n):
        for k in range(1, n):
          for l in range(1, n):
            if i + j + k + l == n:
              return True
    return False
```

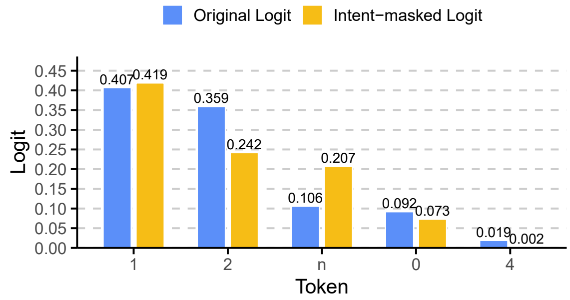

Figure 3: An example of generated code given by DeepSeek-Coder (Guo et al., 2024), where the highlighted token is the wrong token and the top-5 token logits for this position are shown below.

Observation 2. Our second observation is that user intent may fail to sufficiently impact the LLMs' logits during decoding, even when the intent appears to be understood. As shown in Figure 3, we analyze a case where DeepSeek-Coder is asked to generate a function that determines whether an integer n can be written as the sum of four positive even numbers. The LLM initially shows correct understanding by checking whether n is even (`if n % 2 != 0:`). But it generates incorrect code `for i in range(1, n):` as it ignores the "even number" constraint in the user intent. We analyze the original logits (with the user intent fed) and the intent-masked logit (with the user intent masked out). We see that, if we feed the user intent to the LLM, the logit for the correct token `2` increases from 0.242 to 0.359. While the logit for the incorrect token `1` decreases from 0.419 to 0.407. However, the logit for `1` remains higher and under greedy decoding, the LLM still selects the incorrect token. This indicates that while the model incorporates some aspects of the intent, the influence is insufficient to overcome the default tendency, leading to constraint non-compliance.

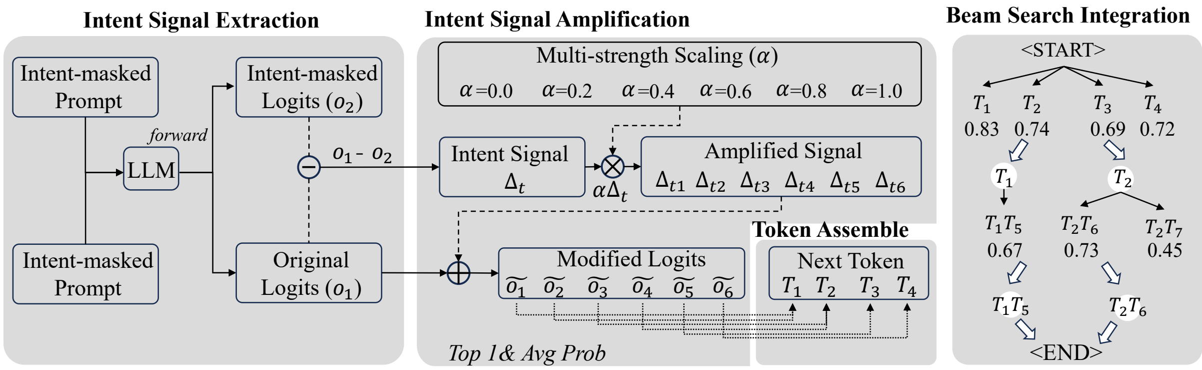

Figure 4: **Overview of *Intent↑Coding*.** We feed the original prompt and intent-masked prompt to the LLM in a same batch (details in Appendix F). We extract the intent signal, scale it with multiple strengths α, and apply a token-level ensemble to obtain a set of candidate next tokens. These candidates are then used to expand each beam hypothesis; the right panel illustrates beam size = 2.

3.3 The Proposed *Intent↑Coding* Method

Based on our observations, we propose Intent-Amplified Code Generation (*Intent↑Coding*), a novel decoding strategy designed to improve LLMs' compliance with user intent. Figure 4 provides an overview of the *Intent↑Coding* pipeline. The core idea is to run an LLM with and without the user intent separately. We enhance user intent in a contrastive way based on the intent-masked logits. Different from traditional contrastive decoding (Li et al., 2023b; Kim et al., 2024; Tian and Zhang, 2025), however, we do not use a fixed contrastive strength but build an ensemble of multiple strengths. Overall, our *Intent↑Coding* consists of four main stages.

Extracting the Intent Signal. As demonstrated in Observation 2, user intent can subtly influence the LLMs' logit distribution during decoding, yet this influence is often too weak to determine the final output. To capture this intent signal, we construct an intent-masked prompt from the original prompt by masking attention of the user intent. This yields two sets of logits at each decoding step t: the original logit $o_t(\cdot|\text{prompt}_{\text{orig}}, x_{<t})$ and the intent-masked logit $o_t(\cdot|\text{prompt}_{\text{masked}}, x_{<t})$, where $\cdot$ represents a token in the vocabulary. The difference between the logits quantifies the influence of the user intent on the LLM's token prediction at each step:

$$\Delta_t(\cdot) = o_t(\cdot|\text{prompt}_{\text{orig}}, x_{<t}) - o_t(\cdot|\text{prompt}_{\text{masked}}, x_{<t}) \quad (1)$$

Amplifying the Signal. To amplify the influence of user intent, we modify the LLM's logits with a scaled intent signal:

$$\tilde{o}_t(\cdot) = o_t(\cdot|\text{prompt}_{\text{orig}}, x_{<t}) + \alpha\Delta_t(\cdot) \quad (2)$$

The choice of the scaling factor α is critical. A small α may fail to sufficiently amplify the influence, while a large α can distort the LLM's original distribution and introduce undesired bias. To balance this, we select six evenly spaced values from the interval $[0, 1]$:

$$A = [0, 0.2, 0.4, 0.6, 0.8, 1.0]$$

where $\alpha = 0$ corresponds to no modification, and $\alpha = 1$ applies the full strength of the intent signal. We will build an ensemble of different values, so our *Intent↑Coding* approach does not rely on a fixed hyperparameter.

Token-Level Ensemble. For each $\alpha \in A$, we compute the logit based on Eqn. (2) and select the top-1 token with the highest probability, yielding a set of candidate tokens. We then perform an ensemble over these tokens: for each unique token, we average its softmax probabilities across all the distributions where it was selected. More implementation details and an example are provided in Appendix E. This ensemble mechanism aggregates evidence across multiple influence strengths, favoring tokens that are robustly supported under varying degrees of intent amplification.

Beam Search Integration. While sampling-based decoding is commonly used for text generation, beam search remains the standard decoding method for code generation, which is a more deterministic task than open-ended text generation (Ippolito et al., 2019; Shi et al., 2024). We

Model	Method	CodeConstraints	IFEvalCode	HumanEval	LiveCodeBench	Average
CodeLlama	Greedy	31.0	19.0	31.1	6.8	13.0
	Beam_Search	18.0	15.2	34.2	6.6	11.9
	NS ($p = 0.7$)	35.7±1.2	18.7±0.4	31.3±1.2	7.4±0.1	13.8
	NS ($p = 0.8$)	33.7±1.2	16.2±0.8	31.5±2.2	7.1±0.1	13.2
	NS ($p = 0.9$)	33.7±1.2	15.9±1.2	30.9±1.7	7.0±0.2	13.0
	AdapT	31.3±1.2	18.9±1.1	34.2±0.9	**7.5**±0.1	13.9
	SPA	**44.0**	**21.9**	**37.2**	6.3	**14.7**
	Intent$^{\uparrow}$Coding	**53.0**$_{+71.0\%}$	**24.8**$_{+30.5\%}$	**40.2**$_{+29.3\%}$	**8.4**$_{+23.5\%}$	**17.5**$_{+34.6\%}$
DeepSeek-Coder	Greedy	23.0	17.1	48.2	14.9	20.1
	Beam_Search	28.0	23.8	51.8	**16.4**	**22.6**
	NS ($p = 0.7$)	21.0±2.4	17.1±1.1	46.7±0.3	14.5±0.4	19.5
	NS ($p = 0.8$)	19.3±1.2	16.8±1.2	43.5±1.5	14.4±0.4	18.8
	NS ($p = 0.9$)	19.7±1.2	14.9±1.9	42.4±1.2	14.3±0.5	18.5
	AdapT	20.3±1.9	17.0±1.6	45.3±0.6	14.8±0.3	19.4
	SPA	**29.0**	**24.8**	**53.7**	15.6	22.4
	Intent$^{\uparrow}$Coding	**39.0**$_{+70.0\%}$	**28.6**$_{+67.3\%}$	**59.8**$_{+24.1\%}$	**17.8**$_{+19.5\%}$	**25.9**$_{+28.9\%}$
Qwen2.5-Coder	Greedy	46.0	23.8	60.4	25.1	31.3
	Beam_Search	35.0	27.6	**70.1**	**27.7**	**33.8**
	NS ($p = 0.7$)	47.0±2.9	24.8±0.8	59.2±0.1	23.9±0.1	30.5
	NS ($p = 0.8$)	46.0±1.4	26.0±0.5	56.9±0.2	24.2±0.5	30.4
	NS ($p = 0.9$)	45.3±0.9	25.1±1.2	56.1±0.1	23.8±0.2	29.9
	AdapT	43.3±0.5	26.7±1.8	60.6±2.3	24.9±0.2	31.2
	SPA	**69.0**	**28.6**	64.0	24.2	33.4
	Intent$^{\uparrow}$Coding	**65.0**$_{+41.3\%}$	**33.3**$_{+40.0\%}$	**65.2**$_{+7.9\%}$	**28.5**$_{+13.5\%}$	**36.6**$_{+16.9\%}$

Table 1: The main results of our experiments. **NS** denotes Nucleus Sampling (Holtzman et al., 2020). **SPA** denotes Selective Prompt Anchoring (Tian and Zhang, 2025). All sampling-based methods are run three times, and we report the mean and standard deviation. The subscript values represent the relative improvement of *Intent$^{\uparrow}$Coding* over the greedy search baseline.The top-2 results among all methods are **bold-faced**. Average is computed as a dataset-size weighted average.

integrate our token-level ensemble strategy into a beam search decoding process to enable a more robust search. At each decoding step, we use the candidate tokens, obtained in the previous stage, to expand the current beam hypotheses. This process creates a diverse set of new hypotheses, each reflecting a distinct path under varying degrees of intent amplification. The expanded set of hypotheses is then pruned to the beam size by retaining only those with the highest cumulative log-probabilities. This integration allows the search to actively explore multiple intent-amplified paths, ultimately selecting code that balances overall fluency with precise adherence to user intent.

4 Experiments

4.1 Experimental Setup

We evaluate constraint following ability on our *CodeConstraints* and IFEvalCode (Yang et al., 2025) dataset, and assess general code generation ability on HumanEval (Chen et al., 2021) and LiveCodeBench (Jain et al., 2025) dataset. We conduct experiments with three widely used open-sourced LLMs: CodeLlama (Rozière et al., 2023), DeepSeek-Coder (Guo et al., 2024), and Qwen2.5-Coder (Hui et al., 2024). More experimental details are provided in Appendix A.

4.2 Main Results

As shown in Table 1, our proposed *Intent$^{\uparrow}$Coding* approach consistently and significantly outperforms all baseline decoding strategies across all LLMs and benchmarks, with average relative improvement ranging from 16.9% to 34.6%. These results highlight the general effectiveness and broad applicability of *Intent$^{\uparrow}$Coding* in enhancing user-intent compliance in code generation.

The performance gain is most notable on our *CodeConstraints* benchmark and IFEvalCode benchmark, which are explicitly designed to assess the constraint-following ability of LLMs. Across models and datasets, *Intent$^{\uparrow}$Coding* achieves consistent relative improvements ranging from 30.5% to 71.0%, highlighting its effectiveness in improving models' constraint-following ability.

Intent$^{\uparrow}$Coding not only improves models' constraint-following ability, but also obtains clear gains on general code generation benchmarks such as HumanEval and LiveCodeBench. These results

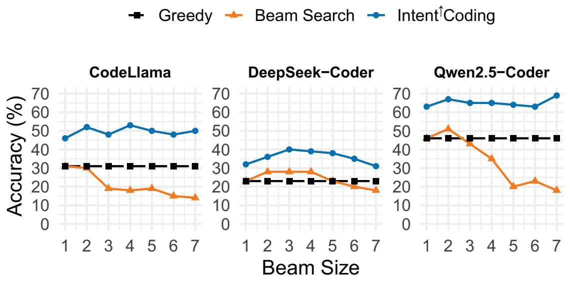

Figure 5: Performance with different beam sizes.

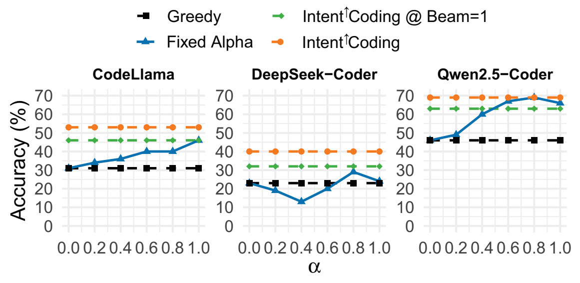

Figure 6: Comparing our ensemble of different α values with a fixed α.

indicate that *Intent$^{\uparrow}$Coding* enhances constraint compliance while preserving and often improving the ability to generate functionally correct code.

Finally, compared to Selective Prompt Anchoring, *Intent$^{\uparrow}$Coding* does not require grid search tuning of an anchoring strength hyperparameter, thereby avoiding extra validation data and compute overhead. Across our experiments, *Intent$^{\uparrow}$Coding* yields larger and more stable gains. Moreover, while beam search can degrade generation quality (Holtzman et al., 2020), *Intent$^{\uparrow}$Coding* remains robust and still yields clear improvements when combined with beam search.

4.3 Comparing Different Beam Sizes

Since *Intent$^{\uparrow}$Coding* is integrated with the beam search decoding process, we analyze its performance across different beam sizes. We conducted the analysis on our constructed *CodeConstraints* dataset. The results in Figure 5 demonstrate that our *Intent$^{\uparrow}$Coding* achieves significant performance gains even with small beam sizes. Notably, in the $beam\_size = 1$ setting where our approach becomes an intent-amplified greedy search, *Intent$^{\uparrow}$Coding* still significantly outperforms greedy search across all LLMs. For instance, when applied to Qwen2.5-Coder, *Intent$^{\uparrow}$Coding* with a beam size of one achieves 63% accuracy, a large improvement over the 46% accuracy from the greedy search. While increasing the beam size in standard beam search yields only modest improvements and may even degrade performance at larger settings, which is consistent with prior findings (Holtzman et al., 2020), *Intent$^{\uparrow}$Coding* maintains high and stable performance at larger settings. This indicates that the performance gains arise not merely from broader search, but primarily from the effectiveness of the intent amplification mechanism in our approach.

4.4 Comparing Our Approach with a Fixed α

Our approach dynamically adjusts the contrastive strength by building an ensemble using different α values in Eqn. (2). To validate the effectiveness of such a strategy, we compare *Intent$^{\uparrow}$Coding* with a simpler baseline that uses a fixed α, such as SPA. As shown in Figure 6, a fixed alpha baseline is highly sensitive to the choice of hyperparameter α, and the optimal α value varies significantly across different LLMs. For example, CodeLlama performs best at $\alpha = 1.0$, whereas Qwen2.5-Coder achieves its peak at $\alpha = 0.8$. Moreover, the performance gap between different α values within the same LLM can be substantial. This sensitivity necessitates an expensive, model-specific hyperparameter search, reducing the practicality and generalizability of the fixed-α approach. By contrast, our *Intent$^{\uparrow}$Coding* employs a predefined set of α values and a token-level ensemble method. Thus, it does not require manual tuning, yet consistently outperforms the best α for each LLM. Notably, even under the $beam\_size = 1$ setting, *Intent$^{\uparrow}$Coding* outperforms the fixed-α baseline in almost all cases. These results demonstrate that our dynamic scaling and ensemble strategy are robust and adaptable in amplifying user intent during code generation.

4.5 LLMs of Different Mask Settings

In our standard *Intent$^{\uparrow}$Coding*, the intent-masked prompt is constructed by masking the entire user intent (containing all constraints). However, our approach is flexible and supports a more fine-grained variant, where only one or a subset of constraints is masked, allowing the amplification of the specific constraint(s). To assess this capability, we conducted experiments on the *CodeConstraints* benchmark by masking an individual constraint. As shown in Figure 7, *Intent$^{\uparrow}$Coding* consistently

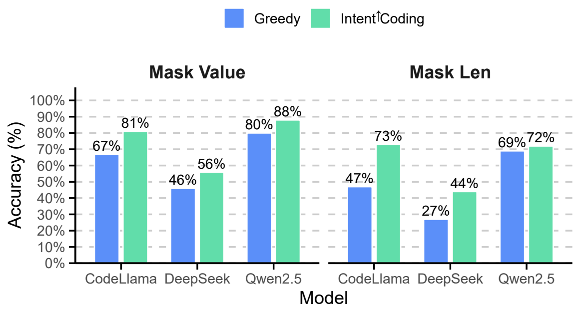

Figure 7: Performance of different constraint-mask settings. Left: Masking the constraint of variable values; Right: Masking the constraint of list/tuple/set length.

improves performance on the targeted constraint across all LLMs and constraint types. For example, when amplifying only the length constraint, all LLMs demonstrate a substantial boost in compliance of the length constraint compared with greedy decoding. The result demonstrates the granular control by our approach: *Intent*$^{\uparrow}$*Coding* offers a robust and adaptable mechanism for enhancing user intent, not only holistically, but also with precision for a specific constraint in code generation.

4.6 Performance with Different LLMs

Instruction-Tuned Models. Instruction-tuning is a widely adopted technique to improve LLMs' alignment with user intent, and instruction-tuned models generally demonstrate stronger compliance with user-specified constraints (Peng et al., 2023). To assess the effectiveness of *Intent*$^{\uparrow}$*Coding* on such LLMs, we conducted experiments using the instruction-tuned versions of CodeLlama, DeepSeek-Coder, and Qwen2.5-Coder. As shown in Table 2, *Intent*$^{\uparrow}$*Coding* provides significant performance gains even in this stronger setting. It boosts the accuracy of CodeLlama-IT from 31% to 53% and DeepSeek-Coder-IT from 52% to 69%, consistently outperforming the best results from optimized greedy or beam search baselines. These results suggest that *Intent*$^{\uparrow}$*Coding* is complementary to instruction-tuning, providing a further layer of capturing and enhancing user intent.

Performance with Different Model Sizes. To assess the scalability and generalizability of our approach, we evaluated *Intent*$^{\uparrow}$*Coding* on a diverse set of LLMs ranging from 1.5B to 33B. As shown in Table 3, *Intent*$^{\uparrow}$*Coding* consistently outperforms the standard decoding baseline across all model sizes. For example, it improves the performance of the smaller Qwen2.5-Coder-1.5B model from 11% to 17% (a 54.5% relative improvement), while yielding a 17-point absolute gain for the much larger Qwen2.5-Coder-32B model (a 34.7% relative improvement). These results demonstrate that the benefits of *Intent*$^{\uparrow}$*Coding* hold across both lightweight and large-scale LLMs, confirming its robustness and model-agnostic nature.

Model	**Baseline**	***Intent*$^{\uparrow}$*Coding***
CodeLlama-IT	31.0	53.0$_{+71.0\%}$
DeepSeek-Coder-IT	52.0	69.0$_{+32.7\%}$
Qwen2.5-Coder-IT	56.0	63.0$_{+12.5\%}$

Table 2: Performance with instruction-tuned models: CodeLlama-7b-Instruct-hf, DeepSeek-Coder-6.7b-Instruct, Qwen2.5-Coder-7b-Instruct. More experimental settings are provided in Appendix A.4.

Model	**Baseline**	***Intent*$^{\uparrow}$*Coding***
CodeLlama		
13B	31.0	39.0$_{+25.8\%}$
34B	35.0	46.0$_{+31.4\%}$
DeepSeek-Coder		
33B	40.0	41.0$_{+2.5\%}$
Qwen2.5-Coder		
1.5B	11.0	17.0$_{+54.5\%}$
3B	19.0	34.0$_{+78.9\%}$
14B	48.0	63.0$_{+31.3\%}$
32B	49.0	66.0$_{+34.7\%}$

Table 3: Performance with different model sizes on the *CodeConstraints* dataset. More experimental settings are provided in Appendix A.4.

5 Conclusion

In this work, we proposed Intent-Amplified Code Generation (*Intent*$^{\uparrow}$*Coding*), a decoding strategy that improves LLMs' ability to follow user intent in multi-constraint code generation tasks. By extracting and amplifying the influence of user intent through a multi-strength ensemble mechanism, our approach enhances constraint satisfaction without requiring additional training or model modification. To support systematic evaluation, we introduced *CodeConstraints*, a benchmark dataset designed to test LLMs under varying levels of constraint complexity. Extensive experiments across multiple LLMs and datasets demonstrate that *Intent*$^{\uparrow}$*Coding* consistently improves both functional correctness and intent compliance, highlighting its generality and effectiveness.

Limitations

We discuss several limitations of our work.

First, *CodeConstraints* covers four constraint types, which are relatively simple; while these constraints are common in practice, this design limits the benchmark's coverage of more complex real-world requirements. In future work, we will extend *CodeConstraints* to include a broader range of constraints, such as stylistic, architectural, and repository-specific constraints observed in real-world repositories.

Second, *Intent*$^{\uparrow}$*Coding* requires batching the original prompt with an intent-masked variant, which introduces additional compute overhead. As analyzed in Appendix B, this overhead is modest in practice.

Third, *Intent*$^{\uparrow}$*Coding* is model-agnostic and can be applied to any LLM as long as token-level logits are available during decoding. However, popular closed-source APIs do not expose token-level logits or provide only limited information, which makes direct deployment difficult. Therefore, our experiments primarily focus on open-source LLMs.

Acknowledgements

We thank all reviewers and the area chair for their constructive comments. This research is supported by the National Natural Science Foundation of China under Grant No. 62192733, 62192730, 62192731, the National Key R&D Program under Grant No. 2023YFB4503801, the Beijing Major Science and Technology Project under Contract no. Z251100008425005.

References

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *ICLR*.

Yung-Sung Chuang, Yujia Xie, Hongyin Luo, Yoon Kim, James Glass, and Pengcheng He. 2023. Dola: Decoding by contrasting layers improves factuality in large language models. *arXiv preprint arXiv:2309.03883*.

Jianbo Dai, Jianqiao Lu, Yunlong Feng, Rongju Ruan, Ming Cheng, Haochen Tan, and Zhijiang Guo. 2024. MHPP: exploring the capabilities and limitations of language models beyond basic code generation. *arXiv preprint arXiv:2405.11430*.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via chatgpt. *TOSEM*, 33(7):189:1–189:38.

Angela Fan, Mike Lewis, and Yann N. Dauphin. 2018. Hierarchical neural story generation. In *ACL*, pages 889–898.

Markus Freitag and Yaser Al-Onaizan. 2017. Beam search strategies for neural machine translation. In *ACL*, pages 56–60.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *NeurIPS Datasets and Benchmarks Track*.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *ICLR*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Daphne Ippolito, Reno Kriz, João Sedoc, Maria Kustikova, and Chris Callison-Burch. 2019. Comparison of diverse decoding methods from conditional language models. In *ACL*, pages 3752–3762.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *ICLR*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024a. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Xue Jiang, Yihong Dong, Yongding Tao, Huanyu Liu, Zhi Jin, and Ge Li. 2025. ROCODE: integrating backtracking mechanism and program analysis in large language models for code generation. In *ICSE*, pages 334–346.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024b. Self-planning code generation with large language models. *TOSEM*, 33(7):182:1–182:30.

Taehyeon Kim, Joonkee Kim, Gihun Lee, and Se-Young Yun. 2024. Instructive decoding: Instruction-tuned large language models are self-refiner from noisy instructions. In *ICLR*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023a. Starcoder: may the source be with you! *TMLR*, 2023.

Xiang Lisa Li, Ari Holtzman, Daniel Fried, Percy Liang, Jason Eisner, Tatsunori Hashimoto, Luke Zettlemoyer, and Mike Lewis. 2023b. Contrastive decoding: Open-ended text generation as optimization. In *ACL*, pages 12286–12312.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *ICLR*.

Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction tuning with GPT-4. *arXiv preprint arXiv:2304.03277*.

Matthew Renze. 2024. The effect of sampling temperature on problem solving in large language models. In *EMNLP*, pages 7346–7356.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, and 6 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Chufan Shi, Haoran Yang, Deng Cai, Zhisong Zhang, Yifan Wang, Yujiu Yang, and Wai Lam. 2024. A thorough examination of decoding methods in the era of llms. In *EMNLP*, pages 8601–8629.

Mohammed Latif Siddiq, Simantika Dristi, Joy Saha, and Joanna C. S. Santos. 2024. The fault in our stars: Quality assessment of code generation benchmarks. In *SCAM*, pages 201–212.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: code generation using transformer. In *ESEC/SIGSOFT FSE*, pages 1433–1443. ACM.

Yuan Tian and Tianyi Zhang. 2025. Selective prompt anchoring for code generation. In *ICML*. OpenReview.net.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, and 49 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Jian Yang, Wei Zhang, Shukai Liu, Linzheng Chai, Yingshui Tan, Jiaheng Liu, Ge Zhang, Wangchunshu Zhou, Guanglin Niu, Zhoujun Li, and 1 others. 2025. Ifevalcode: Controlled code generation. *arXiv preprint arXiv:2507.22462*.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023a. Self-edit: Fault-aware code editor for code generation. In *ACL*, pages 769–787.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023b. Planning with large language models for code generation. In *ICLR*.

Yuqi Zhu, Jia Li, Ge Li, Yunfei Zhao, Jia Li, Zhi Jin, and Hong Mei. 2024. Hot or cold? adaptive temperature sampling for code generation with large language models. In *AAAI*, pages 437–445.

A Experimental Details

A.1 Datasets

HumanEval (Chen et al., 2021) is a widely-used benchmark dataset for evaluating code generation capabilities of LLMs. It consists of 164 hand-crafted Python programming problems, each defined by a function signature and a natural language description.

LiveCodeBench (Jain et al., 2025) offers a more realistic evaluation of code generation capabilities by sourcing problems directly from programming contests. These tasks are characterized by complex, natural language descriptions and are evaluated against a set of test cases. In our experiments, we utilize Version 5 of the dataset, which comprises 279 easy, 331 medium, and 270 hard problems.

IFEvalCode (Yang et al., 2025) pairs each code generation task with explicit, verifiable instruction constraints and an executable checker that tests whether the generated code satisfies those constraints. In our experiments, we report the instruction compliance metric (Instr.) proposed by this benchmark on its python and english questions.

In addition to existing datasets, our work also constructed a new dataset, ***CodeConstraints***, to evaluate LLMs' compliance with user-specified constraints (detailed in the previous section). In our experiments, we focus on Level 4, which is the most challenging setting, having four constraints in each data sample.

A.2 Base Models

We conducted experiments with popular open-source LLMs, as we need LLM logit during decoding. In particular, we consider the following models.

CodeLlama (Rozière et al., 2023) is an open-source family of code-specialized LLMs, based on the LLaMa model (Touvron et al., 2023) and further pre-trained on permissively licensed code. It has demonstrated strong performance on standard code generation benchmarks. In our experiments, we used the CodeLlama-7B-hf variant.

DeepSeek-Coder (Guo et al., 2024) is a family of open-source LLMs trained from scratch on 2 trillion tokens, comprising 87% code and 13% natural language in both English and Chinese. The LLMs are pre-trained on a project-level code corpus with a 16K context window and an additional fill-in-the-blank objective to support long-range code completion and infilling. In our experiments, we used the DeepSeek-Coder-6.7B-Base model.

Qwen2.5-Coder (Hui et al., 2024) is a series of code-specific LLMs, extending the Qwen2.5 model with specialized training on 5.5 trillion tokens. It supports long-context understanding up to 128K tokens and exhibits strong performance in code generation, reasoning, and fixing. In our experiments, we used the Qwen2.5-Coder-7B model.

A.3 Implementation Details

In our experiment, we explored multiple decoding strategies as baselines to ensure a comprehensive comparison. For both standard beam search and our *Intent*$^{\uparrow}$*Coding* method, the beam size was set to 4. In nucleus sampling (Holtzman et al., 2020), we used top-p with a p value chosen from {0.7, 0.8, 0.9} and a fixed temperature of 0.2. For the AdapT sampling method (Zhu et al., 2024), we fixed top-p at 0.95, and performed a grid search over its temperature parameters T_1 and T_2; we report the results from the best configuration. All sampling-based methods are run three times, and we report the mean and standard deviation. For the Selective Prompt Anchoring method, we performed a grid search over its anchoring strength ω; we report the results from the best configuration. The maximum generation length is set to 512 for HumanEval and IFEvalCode, 1024 for LiveCodeBench, and 256 for our *CodeConstraints* dataset. For the HumanEval and LiveCodeBench datasets, we report results in the standard $\mathrm{pass@1}$ metric, whereas for the CodeConstraints benchmark, we use the accuracy metric, where the code is considered accurate if all constraints are satisfied.

A.4 Performance with Different LLMs Settings

Baseline denotes the best performance achieved by either greedy search or beam search. For both methods, we report the best performance from a grid search over beam sizes $\{1, 2, 3, 4\}$. The subscript values represent the relative improvement of *Intent*$^{\uparrow}$*Coding* over the baseline.

B Inference Efficiency

We analyze the inference efficiency of our *Intent*$^{\uparrow}$*Coding* approach. We considered the *CodeConstraints* dataset and used a single NVIDIA A100-40G GPU. For a fair comparison, all methods, greedy, beam search, and *Intent*$^{\uparrow}$*Coding*, were run with the `early_stop="never"` configuration, ensuring that generation continues until all

Model	best-of-2	*Intent*$^{\uparrow}$*Coding*	Δ
HumanEval			
CodeLlama-7B	34.2	40.2	+6.0
DeepSeek-Coder-6.7B	51.2	59.8	+8.6
Qwen2.5-Coder-7B	60.4	65.2	+4.8
CodeConstraints			
CodeLlama-7B	42.0	53.0	+11.0
DeepSeek-Coder-6.7B	25.0	39.0	+14.0
Qwen2.5-Coder-7B	55.0	65.0	+10.0

Table 1: Results on HumanEval and CodeConstraints.

Model	Greedy	*Intent*$^{\uparrow}$*Coding*	Δ
Level3			
CodeLlama-7B	42.0	56.0	+14.0
DeepSeek-Coder-6.7B	35.0	54.0	+19.0
Qwen2.5-Coder-7B	52.0	81.0	+29.0
Level4			
CodeLlama-7B	31.0	53.0	+22.0
DeepSeek-Coder-6.7B	23.0	39.0	+16.0
Qwen2.5-Coder-7B	46.0	65.0	+19.0

Table 2: Results on Level3 and Level4 of CodeConstraints.

beams produce an EOS token. Each experiment was repeated five times to reduce measurement variance, and we report the average and standard deviation in Table 3.

Results show that our *Intent*$^{\uparrow}$*Coding* is slightly slower than greedy and beam search in terms of time per token. This is expected because our approach involves an ensemble of multiple contrastive strengths. However, our method often outputs fewer tokens than beam search (which tends to generate code snippets that are less related to the user intent); consequently, our approach is more efficient than beam search in terms of inference time. Overall, our *Intent*$^{\uparrow}$*Coding* is an efficient decoding method for code generation.

C More Results

We compare our *Intent*$^{\uparrow}$*Coding* with the best-of-two approach under a controlled setting with a fixed budget. As shown in Table 1, our performance is much higher than the best-of-two approach.

Table 2 reports the results of *Intent*$^{\uparrow}$*Coding* and Greedy decoding across Level 3 and Level 4 on the CodeConstraints benchmark. Across all three LLMs, *Intent*$^{\uparrow}$*Coding* obtains large gains at both difficulty levels. These results show that *Intent*$^{\uparrow}$*Coding* consistently improves performance as task complexity increases.

D Case Studies

Table 4 presents an example involving four user-specified constraints: (1) The code should return a set, (2) The element in set should be integer, (3) The integer in set should be greater than or equal to n, and (4) The length of the set should be 99.

The code generated by greedy decoding satisfies the first two constraints but fails to ensure the correct value and set size. Beam search decoding improves upon this by correctly constraining the value, but it still fails to guarantee the exact set length. Additionally, it produces unrelated code after finishing the function body. In contrast, our *Intent*$^{\uparrow}$*Coding* approach generates code that fully satisfies all four constraints and terminates precisely at the end of the intended logic, without producing additional tokens. Overall, this example highlights the strength and efficiency of *Intent*$^{\uparrow}$*Coding* in capturing fine-grained constraints.

E Token-Level Ensemble Details

As described in Section 3.3, at each decoding step we obtain six logit distributions $\{\tilde{o}^{(k)}\}_{k=1}^{6}$ and the corresponding probabilities $p^{(k)} = \mathrm{softmax}(\tilde{o}^{(k)})$. For each k, we take the top-1 token

$$t^{(k)} = \arg\max_{v} p_v^{(k)}, \qquad s^{(k)} = p_{t^{(k)}}^{(k)}. \quad (3)$$

Since multiple $t^{(k)}$ may share the same token id v, we group candidates by id and assign an ensemble score

$$\bar{p}(v) = \frac{1}{|S(v)|} \sum_{k \in S(v)} s^{(k)}, \qquad S(v) = \{k \mid t^{(k)} = v\}. \quad (4)$$

We return each unique token id v with $\bar{p}(v)$. For example, if the six top-1 tokens $T_1, \ldots, T_6$ have probabilities $P_1, \ldots, P_6$, where T_1, T_2 map to token id A and T_3, T_4, T_5, T_6 map to token id B, then we return A and B with $\bar{p}(A) = (P_1 + P_2)/2$ and $\bar{p}(B) = (P_3 + P_4 + P_5 + P_6)/4$.

F Prompt

In *Intent*$^{\uparrow}$*Coding*, we construct an intent-masked prompt from each dataset-provided original prompt. We start from the raw prompt as released by the benchmark, then use Python regular expressions to automatically locate the natural-language intent span and mark it with a mask token while preserving the rest of the context. This masking pipeline is fully automated. Moreover, attention masks are modified at the transformer level. After tokenization, we set attention masks to 0 for the positions corresponding to the span. The prompt templates for each dataset are shown below. We use <mask> to indicate the masked text span for demonstration purposes.

F.1 HumanEval Prompt

```
from typing import List


def has_close_elements(numbers: List[
    float], threshold: float) -> bool:
    """ Check if in given list of
    numbers, are any two numbers closer
    to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0,
    3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8,
    3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

Listing 1: HumanEval Original Prompt

```
from typing import List


def has_close_elements(numbers: List[
    float], threshold: float) -> bool:
    <mask>
    """ Check if in given list of
    numbers, are any two numbers closer
    to each other than
    given threshold.
    >>> has_close_elements([1.0, 2.0,
    3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8,
    3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
    <mask>
```

Listing 2: HumanEval Intent-Masked Prompt

F.2 LiveCodeBench Prompt

```
### Question
You are given an integer sequence A=(A_1
    ,\ldots,A_N) of length N. Here, A_1,
     A_2, \ldots, A_N are all distinct.
Which element in A is the second largest
    ?

Input

The input is given from Standard Input
    in the following format:
N
A_1 A_2 \ldots A_{N}

Output

Print the integer X such that the X-th
    element in A is the second largest.

Constraints


- 2 \leq N \leq 100
- 1 \leq A_i \leq 10^9
- A_1, A_2, \ldots, A_N are all distinct.

- All input values are integers.

Sample Input 1

4
8 2 5 1

Sample Output 1

3

The second largest element in A is A_3,
    so print 3.

Sample Input 2

8
1 2 3 4 5 10 9 11

Sample Output 2

6

### Answer
```

Listing 3: LiveCodeBench Original Prompt

```
### Question
<mask>
You are given an integer sequence A=(A_1
    ,\ldots,A_N) of length N. Here, A_1,
     A_2, \ldots, A_N are all distinct.
Which element in A is the second largest
    ?
<mask>

Input

The input is given from Standard Input
    in the following format:
N
A_1 A_2 \ldots A_{N}

Output

Print the integer X such that the X-th
    element in A is the second largest.

Constraints


- 2 \leq N \leq 100
- 1 \leq A_i \leq 10^9
- A_1, A_2, \ldots, A_N are all distinct.

- All input values are integers.

Sample Input 1
```

```
4
8 2 5 1

Sample Output 1

3

The second largest element in A is A_3,
    so print 3.

Sample Input 2

8
1 2 3 4 5 10 9 11

Sample Output 2

6

### Answer
```

Listing 4: LiveCodeBench Intent-Masked Prompt

F.3 *CodeConstraints* Prompt

```
def func(n):
    '''
    Given a positive integer n,
    return a tuple of floats.
    The floats should be less than n.
    The length of the tuple should be 2.
    '''
```

Listing 5: *CodeConstraints* Original Prompt

```
def func(n):
    <mask>
    '''
    Given a positive integer n,
    return a tuple of floats.
    The floats should be less than n.
    The length of the tuple should be 2.
    '''
    <mask>
```

Listing 6: *CodeConstraints* Intent-Masked Prompt

F.4 IFEvalCode Prompt

~~~~
Given a string, calculate the minimum
    cost to complete character escaping.
     The escaping rules are as follows:
- '<' escapes to '&lt;'
- '>' escapes to '&gt;'
- '&' escapes to '&amp;'
- '\"' escapes to '&quot;'
- '\'' escapes to '&#39;'

There are two types of escape operations:

1. Single character escape: Replace a
    single escapable character with its
    corresponding HTML entity, cost is 1
     unit
2. Substring escape: Replace a
    continuous substring containing only
     escapable characters with their
    corresponding HTML entities, cost is
     2 units (regardless of substring
    length)

Constraints:
1. Internal variable names must use
    snake_case
2. Must use list comprehension
3. Must use one ternary operator
4. Total code lines must not exceed 15
    lines (including empty lines)
5. Must include at least one lambda
    expression

Function signature:```
def min_escape_cost(input_str: str) ->
    int:
     pass
```

Example input output:
```
Input: 'Hello <World> & \"Everyone\"'
Output: 5
Explanation: One optimal solution:
1. Escape '<' and '>' as substring, cost
     = 2
2. Escape '&' as single character, cost
    = 1
3. Escape two '\"' as substring, cost =
    2
Total cost = 2 + 1 + 2 = 5
```

Please return all the complete code in
    the first code block.
~~~~

Listing 7: IFEvalCode Original Prompt

~~~~
Given a string, calculate the minimum
    cost to complete character escaping.
     The escaping rules are as follows:
- '<' escapes to '&lt;'
- '>' escapes to '&gt;'
- '&' escapes to '&amp;'
- '\"' escapes to '&quot;'
- '\'' escapes to '&#39;'

There are two types of escape operations:

1. Single character escape: Replace a
    single escapable character with its
    corresponding HTML entity, cost is 1
     unit
2. Substring escape: Replace a
    continuous substring containing only
     escapable characters with their
    corresponding HTML entities, cost is
     2 units (regardless of substring
    length)

Constraints:
<mask>
1. Internal variable names must use
    snake_case
2. Must use list comprehension
3. Must use one ternary operator
~~~~

```
4. Total code lines must not exceed 15
    lines (including empty lines)
5. Must include at least one lambda
    expression
<mask>

Function signature:```
def min_escape_cost(input_str: str) ->
    int:
     pass
```

Example input output:
```
Input: ’Hello <World> & \"Everyone\"’
Output: 5
Explanation: One optimal solution:
1. Escape ’<’ and ’>’ as substring, cost
     = 2
2. Escape ’&’ as single character, cost
    = 1
3. Escape two ’\"’ as substring, cost =
    2
Total cost = 2 + 1 + 2 = 5
```

Please return all the complete code in
    the first code block.
```

Listing 8: IFEvalCode Intent-Masked Prompt
```

| Model | Method | Avg. Time (s) | Avg. Tokens | Time/Token (ms) |
|---|---|---|---|---|
| CodeLlama | Greedy | 7.84±0.04 | 172.48±0.00 | 45.30±0.04 |
| | *Intent*$^{\uparrow}$*Coding*@1 | 10.45±0.04 | 211.62±0.00 | 49.34±0.04 |
| | Beam Search@4 | 12.65±0.06 | 253.53±0.00 | 50.21±0.21 |
| | *Intent*$^{\uparrow}$*Coding*@4 | 10.82±0.05 | 210.41±0.00 | 51.24±0.11 |
| DeepSeek-Coder | Greedy | 8.65±0.04 | 192.58±0.00 | 44.83±0.03 |
| | *Intent*$^{\uparrow}$*Coding*@1 | 10.12±0.07 | 206.47±0.00 | 49.04 ±0.06 |
| | Beam Search@4 | 12.65±0.09 | 255.04±0.00 | 49.72±0.14 |
| | *Intent*$^{\uparrow}$*Coding*@4 | 12.12±0.04 | 235.69±0.00 | 51.26±0.08 |
| Qwen2.5-Coder | Greedy | 8.61±0.06 | 185.51±0.00 | 45.86±0.03 |
| | *Intent*$^{\uparrow}$*Coding*@1 | 5.13±0.06 | 104.44±0.00 | 49.09±0.06 |
| | Beam Search@4 | 12.44±0.04 | 254.23±0.00 | 49.06±0.13 |
| | *Intent*$^{\uparrow}$*Coding*@4 | 7.38±0.05 | 146.09±0.00 | 50.82±0.16 |

Table 3: Average inference time (in seconds), maximum generated token count per problem and average inference time per token (in milliseconds) on the *CodeConstraints* benchmark, using a single NVIDIA A100-40G GPU. Results are reported for 7B base models with beam sizes of one (@1) and four (@4).

```python
def func(n):
    '''Given a positive integer n, return a set of integers.
    The integers should be greater than or equal to n.
    The length of the set should be 99.
    '''
```

```python
    # base case
    if n == 1:
        return {n}

    # recursive case
    else:
        return {n} | func(n-1)
```

Greedy

```python
        return set(range(n, n + 100))


    if __name__ == '__main__':
        import doctest
        doctest.testmod()
```

Beam Search

```python
    set_ = set()
    while len(set_) < 99:
        set_.add(n)
        n += 1
    return set_
```

Intent$^{\uparrow}$Coding

Table 4: Case studies.